Critical MASS: Performance and Programmability Evaluation of MASS (Multi-Agent Spatial Simulation) and Hybrid OpenMP/MPI

Zachary J Brownell

A thesis
submitted in partial fulfillment of the
requirements for the degree of:

Master of Science in Computer Science & Software Engineering

University of Washington
2015

Committee:

Munehiro Fukuda, Ph.D. (Faculty Advisor)
Michael Stiber, Ph.D.
Hazeline Asuncion, Ph.D.
William Erdly, Ph.D.

Program Authorized to Offer Degree:

School of Science, Technology, Engineering & Mathematics
Computing & Software Systems

ABSTRACT

In this paper, we explore the relationship between programmability and performance within the context of two C++ parallel/distributed programming approaches: Hybrid OpenMP/MPI & MASS (Multi-Agent Spatial Simulation).

Our study begins by working with the following hypothesis: programmers in big data analysis and Agent-Based Models (ABM) will find MASS easier to use than hybrid OpenMP/MPI, despite its slower performance.

We then detail the planned experiments and criteria used for testing this hypothesis, which include a mixture of broadly-accepted characteristics for programmability within parallel/distributed frameworks, survey application, line of code counting, and actual performance testing.

During our research, we found that MASS offered more of a global view of computation than hybrid OpenMP/MPI and that programmers typically took 39 minutes less to write corresponding applications using MASS. When writing these applications, MASS required around 8.17% less parallel/distributed-specific lines of code. In addition, we learned that applications written in MASS were approximately 4.4% easier to debug than corresponding ones based on OpenMP/MPI.

While there were promising results for MASS, our data showed that OpenMP/MPI slightly outperformed MASS in general characteristics of programmable parallel/distributed frameworks and received more favorable assessments across most surveyed questions related to time, effort, and programmability. We also found that the same application written in OpenMP/MPI typically had an execution time that was 25.82% better (lower) than corresponding applications built using MASS.

Overall, even though we found that the programmability results were quite close between the two frameworks, we were unable to accept the alternative hypothesis presented. It is worth noting, however, that the C++ version of MASS is around 3 years old and is actively being developed by a small handful of students and faculty at the University of Washington Bothell. Whereas, OpenMP/MPI has nearly two decades of development and support from major hardware/software corporations across the world.

ACKNOWLEDGEMENTS

First and foremost, I want to acknowledge the incredible patience, understanding, and support of my wonderful wife, Koriel Jock. This would not have been possible without her being there along the way - pretending to understand what I was talking about, listening to me explain problems anyway, and removing so many of the obstacles in everyday life to make this journey as smooth as possible.

I would also like to extend my deepest gratitude for the support of the faculty and professors at the University of Washington Bothell. I probably would not have enrolled in this program if it was not for the great people involved with this program. From some of the most kind, helpful faculty members like Megan Jewell to the knowledgable, friendly, and caring professors like Dr Fukuda, Dr Erdly, Dr Stiber, and Dr Asuncion (whom I've been equally blessed to have had on my faculty committee), words can not possibly express the profound gratitude and respect I have for this program (although, I've tried anyway here - I guess they can not unteach stubborness).

Finally, I'd like to thank Adobe Systems Inc and the folks at the Seattle office who have covered on-call duties, put up with my last minute PTO requests, and accommodated an ever-changing and flexible work schedule to support my education. I'd specifically like to call out James Boag and the rest of the SET Seattle family. You've been my home away from home for years and an incredibly talented, creative, and caring group of people that continue to inspire me to greater heights.

From the bottom of my heart, thank you one and all.

Contents

1 Overview

1.1 What is MASS

MASS is an acronym for Multi-Agent Spatial Simulation. It is a paradigm-oriented, parallel/distributed framework that allows programmers to write applications that can make use of multi-core, connected computational resources such as those on a grid or cloud. What really sets it apart from other frameworks, though, is that it was designed specifically with users in agent-based modeling in mind. Over the course of its limited development, it has also been extended to allow for big data analysis using its agents paradigm.

MASS was originally developed in Java, with an initial port to C++ occurring in late 2012 by Narayani Chandrasekaran. Chandrasekaran managed the initial implementation of the Places paradigm in MASS [4] and over time, several other students have worked and are continuing to work on making the MASS C++ framework a viable option for parallel/distributed application development, including:

1. Chris Rouse
   In 2014 Rouse [28] added the initial agent implementation to MASS
2. Cherie Lee Wasous
   Also in 2014 Wasous [30] added distributed agent management to MASS, which became the focus/topic of her Master's thesis
3. Jennifer Kowalsky
   Currently, Kowalsky [24] is working on updating the documentation and functionality of MASS, adding additional logic to encompass an idea of neighbors and inter-neighbor communication
4. Hung Ho
   Ho [19] is in the process of adding asynchronous and automatic migration of agents in MASS

MASS was originally created to help address a perceived shortfall in many parallel/distributed frameworks, at the time (OpenMP, Open MPI, MapReduce, etc). While the number of cores in computing hardware and the interconnectedness of machinery was growing - moving away from the continued pursuit of higher clock speeds, in favor of more cores, grids, and cloud frameworks, the libraries/languages that existed to support such parallelization in applications were not keeping up. By and large, it could be said that these existing frameworks:

1. Were Tied to a Specific Data Model
   Which, was often times hard to adapt applications to make use of (e.g. - MapReduce's key/value pair [4])
2. Required Deep Developer Understanding
   Aside from learning these new frameworks or languages, developers had to be very careful when using them to ensure that effective use of computational resources was actually occurring (good cache usage, reduced chance of thrashing, protection/synchronization around critical sections, etc)

MASS was developed to try to address these concerns by providing [4]:

1. Automatic Parallelization
   Instead of having to carefully divide and conquer, or take a bag of tasks approach to decomposing data in your application, you could simply rely on MASS to take care of parallel and distributed execution, resource allocation, and efficiency for you
2. Utilization of Symmetric Multi-Processor Cluster
   MASS has the ability to not only distribute work across a cluster/grid, but it also has the ability of further parallelizing execution across cores on each machine within the cluster
3. Abstraction of Parallelization Constructs
   Using MASS, programmers no longer have to be aware of processes, threads, or communication approaches in parall/distributed computing. Of course, it helps to have an idea of what you are doing, but the over head of having to be intrinsically involved with the maintenance of these tasks has been abstracted away from programmers in MASS

4. Single Programming Paradigm
   Through providing both a distributed and shared memory model, MASS allows for individual resources to work together, making efficient use of dispersed memory across cooperating hardware on common data sets (shared network storage, etc)

So, as you can probably tell by now, there is something a bit different about MASS. This difference can be considered as the places/agents paradigm. It is a blessing and a curse, in that it helps address some common pain points in other distributed frameworks by abstracting away the minutiae of parallel/distributed design/coding, but it also forces users to reconsider problem spaces within the context of either:

1. Places
   Places are implmented as a distributed array. Using this approach, programmers can simply concentrate on breaking down their application to use a series of Place objects to accomplish goals - while, under the hood, MASS will divide the total number of Places used across hosts provided; slicing the data to work on it independently across machines (distributed/parallelized computation). Examples of this approach are: Wave2D, Heat2D, and computational fluid dynamics (CFD).
2. Agents
   Agents are mobile objects in MASS, divided among Threads available to processes on each corresponding place/host machine. The Agents approach is a similar one to Places, with the difference being that the Place objects are generally mobile agents (but, can be stationary/static) and the really interesting activity is occurring with the interaction between these moving Agents and between the Places that they inhabit. Examples of this approach include artificial lives and swarms.
3. Places and Agents
   This is a more complex way of modeling a system available in MASS. You could use active Places and Agents (that change state, share data, etc) to model truly complex interactions, some of which could be quite prescient for society (e.g. - how will people in low-lying areas of the World move/travel to different Places as climate continues to change, or - as Osmond Gunarso [17] studied - how does influenza spread across people in different communities, neighborhoods, and settings, and how do different treatment methods help manage infection)

1.2 What is OpenMP/MPI

OpenMP and Open MPI are also acronyms (or contain acronyms) that stand for Open Multi Processing [2] and Open Message Passing Interface, respectively. Both frameworks are general-purpose computing libraries. Used in conjunction, these tools allow programmers to take advantage of multiple cores on an individual machine (OpenMP) and distibuting work across connected machines (Open MPI).

MPI was originally conceived in 1991 [10] and very quickly became a joint endeavor to come to full fruition. “The MPI effort involved about 80 people from 40 organizations, mainly in the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI along with researchers from universities, government laboratories, and industry.” [10].

Following on the heels of MPI, OpenMP’s first specification came into being in 1997 [11]. The first C++ port of OpenMP came out the following year, with subsequent new versions of the specification released, as follows:

1. 2.0: 2000
2. 3.0: 2008
3. 3.1: 2011
4. 4.0: 2013

Like MPI, it enjoys support from major technology companies, that includes a “group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and more.” [11].

Delving into the exact specifications of these frameworks is beyond the scope of this paper. Suffice it to say that combined they offer a well-maintained, well-defined, and well-supported method for communicating between machines and dividing up execution tasks/data to make efficient use of available cores on individual machines participating in a group computation.

1.3 Research Goals

It is a combination of rooting for the "underdog" and really believing in the merits of MASS's paradigm-oriented approach (using Agents/Places model) that really got us interested in investigating how these two frameworks stacked up against one another. In so many situations, MASS's paradigm just makes a lot of sense for the application. From modeling spatial relations like heat transfer or wave dissemination to complex agent interactions like war simulations, population growth, traffic patterns, or weather modeling/forecasting, the paradigm-oriented approach that MASS takes seems to offer an easier method than the classic general-purpose programming environment of a hybrid OpenMP/MPI solution.

Hybrid OpenMP/MPI enjoys nearly a two decade head start, wide support, documentation, and a large user base with active forums, examples, and questions/answers to be found online. But, with such wide support comes the challenge of being general enough for a variety of applications. On the other hand, MASS has a unique way of simplifying and abstracting away a lot of the pain involved with parallel/distributed code development. A trait that was built in to its design to specifically target agent-based models, spatial simulations, and big data analysis.

Over the course of this paper, we will talk about these two frameworks and how we have chosen to evaluate them. We will discuss some general parallel/distributed framework programmability characteristics, how we have designed tests to survey users, and how we have approached gauging each frameworks' performance.

The paper will then move on to discuss the actual results of our testing, before wrapping up with our conclusions and ideas for further research in this area.

1.3.1 Goals

1. Provide Further Support for Programmability Claims
   There have been many papers written and published that relate to programmability within MASS. Examples include:
   (a) Design and Qualitative/Quantitative Analysis of Multi-Agent Spatial Simulation Library [6]
   (b) A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems, [7]
   (c) A multi-process library for multi-agent and spatial simulation. [12]
   However, upon deeper inspection, you can find the results of the original paper (Design and Qualitative/Quantitative Analysis of Multi-Agent Spatial Simulation Library) were simply repeated in each of the following IEEE conference proceedings listed. So, while you can find three articles that discuss programmability, they're all based on the same study.
2. Provide First Programmability Assessment of C++ Implementation
   Previous papers have only focused on the Java implementation of MASS. This paper will be the first to consider programmability in MASS, using the C++ implementation.
3. Track User Assessment of MASS
   We can also consider the current state of user involvement in MASS programmability assessments. Previous papers discussing programmability within MASS have been qualitative in nature, but this is the first paper to actually quantitatively measure this attribute through the use of surveys.
4. Provide Insight into Effort and Time Using MASS
   The survey data included in this paper not only provides actual quantifiable insight into programmability within MASS, but it also records characteristics of MASS related to effort and time - which, have been previously ignored in evaluations of the MASS framework.

5. Provide Further Support for Performance Claims
   Like programmability, performance in MASS is a topic that has already been presented in previous research. Excluding MASS CUDA - GPU-enabled versions - of the library, these papers include:
   (a) Design and Qualitative/Quantitative Analysis of Multi-Agent Spatial Simulation Library [6]
   (b) A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems [7]
   (c) A multi-process library for multi-agent and spatial simulation [12]
   (d) Dynamic load balancing in MASS [27]
   (e) Field-Based Job Dispatch and Migration [22]
   (f) A parallelization of orchard temperature predicting programs [25]

   While some of the performance analyses focus on more than the esoteric subject that the paper is based on, none of them actually include general performance data - removed from practical, application-specific implementations. There are instances where data on applications discussed in this paper (Wave2D and Sugarscape) are compared. However, our paper is unique for a couple of reasons, in regards to performance:
   (a) First Benchmarked Baseline MASS Performance Data
   This study is the only one published that contains baseline performance data. We used a benchmarking application, specifically developed to exercise and track various Place/Agent methods offered through MASS - offering graphical representations and raw accounts of the data collected through these tests.
   (b) First Analysis of FluTE Performance in MASS
   In addition to the benchmarking performed, this study will introduce a new application into the mix - FluTE. FluTE is unique and interesting to academia due to its non-trivial nature and possession of emergent, interesting qualities as an outcome of its execution. It is also an established application that has been parallelized using OpenMP. So, the corresponding MASS implementation not only offers a view into how MASS compares, in this regard, but a unique glimpse into how an existing OpenMP application can be easily converted into the agent-based paradigm of MASS.

2 Hypothesis

Considering the differences between MASS and hybrid OpenMP/MPI applications, we naturally wondered how they would stack up against one another. After all, MASS would seem to offer a much easier method of modeling data - allowing a more object-oriented approach to managing parallelization, compared to hybrid OpenMP/MPI's more general, "hands-on" approach. To examine this intersection, we developed the following hypothesis to guide our study of the two frameworks.

2.1 Hypothesis Statement

Programmers in big data analysis and Agent-Based Models (ABM) will find MASS easier to use than hybrid OpenMP/MPI, despite its slower performance.

This hypothesis will allow us to not only consider the relative programming difficulty or ease between our two approaches (MASS & hybrid OpenMP/MPI), but also allow us to consider the performance difference between the two systems. While we expect the performance to lag using MASS, we also expect that it will be much easier to model many applications due to its paradigm-oriented approach.

2.2 Formal definition of null hypothesis

Programmers in big data analysis and ABM will not find MASS easier to use than hybrid OpenMP/MPI

Phrasing the null hypothesis in this manner yields the following mathematical equivalent:

$$H_0 = \mu \textit{ MASS Ease-of-Use} \leq \mu \textit{ Hybrid OpenMP/MPI Ease-of-Use}$$

There is also an orthogonal null hypothesis nested in our original statement, being:

$$H_0 = \mu \textit{ MASS Performance} \geq \mu \textit{ Hybrid OpenMP/MPI Performance}$$

2.3 Formal definition of alternative hypothesis

Programmers in big data analysis and ABM will find MASS easier to use than hybrid OpenMP/MPI

Stating the alternative hypothesis this way, we are able to formally define the following mathematical equivalent:

$$H_A = \mu \textit{ MASS Ease-of-Use} > \mu \textit{ Hybrid OpenMP/MPI Ease-of-Use}$$

There is also an orthogonal alternative hypothesis nested in our original statement, being:

$$H_A = \mu \textit{ MASS Performance} < \mu \textit{ Hybrid OpenMP/MPI Performance}$$

2.4 Operationalization of Hypothesis Variables

The hypotheses, as written, are easy to read and understand, on the surface. They use language that people typically take advantage of when talking with one-another - which, is great. However, it does leave on thing to be desired: ensured clear understanding of the topics being considered.

One of the fallbacks to using language that is easy to understand, is that it leaves some of that understanding up to the individual doing the reading. The following table takes each of the terms that we have used in these hypotheses and offers concise definitions for them, reducing the potential for confusion.

Term	Definition
Agent-Based Models	This term refers to a method of modeling an application that uses a pattern of representatives (agents) that interact with each other or the environment that they are based in, to study a desired effect/outcome. Examples of Agent-Based Models would be applications like traffic simulations that allow users to study the effect of altering signal synchronicity/timing on the effect of vehicles in the city or a reforestation application that allows users to study the effect of climate change on tree growth, dispersal, movement, etc over time. In each case, you can see that an agent does not necessarily have to be a person. You can also see how ABMs are useful in discovering emergent, collective group behavior (traffic behaviors, forest movement) of simulation entities (vehicles, trees) that can not be covered with mathematical models alone.
Big Data Analysis	Decomposing the terms, we can intuit that bid data analysis deals with large amounts of data (big) and it also deals with how to organize, make sense of, or use that data (analysis). Some examples would be stock-ticker applications that track trends in the market or real-time weather applications that track large amounts of data (temperature, humidity, barometric pressure, wind speed, wind direction, cloud cover, etc) for weather forecasting. Using the paradigm-oriented approach of MASS, it might actually be more intuitive and easier to move agents rather than data in these types of simulations (think of dealing with weather systems, instead of dealing with coordinating tables of representational data).
Ease-of-Use	In the context of this paper, when we refer to "ease-of-use" of use the term "easier," we are considering this concept from the point-of-view of a programmer. As such, well generally be using the notion of "programmability" to catalog (quantify) and compare characteristics of MASS and hybrid OpenMP/MPI application frameworks.
Hybrid OpenMP/MPI	This is a common approach in parallel/distributed development in which developers use MPI to handle distributing work to multiple, connected computers, while also using OpenMP to parallelize the work occurring on each computing node. This approach allows for nested parallelization and distribution of work.
MASS	Multi-Agent Spatial Simulation. For more information on MASS, please see Section 3.1 "What is MASS" (above).
MPI	Message Passing Interface. For more information on MPI, please see Section 3.2 "What is OpenMP/MPI" (above).
OpenMP	Open Multi Processing. For more information on OpenMP, please see Section 3.2 "What is OpenMP/MPI" (above).
Performance	In the context of this paper, when we refer to performance, we are really talking about execution time. We do not consider CPU cycles, memory-usage, net electricity drain, etc in our analysis. Instead, youll find that, while the unit of measurement may change (hour, minute, second, millisecond, microsecond, etc), the subject of the measurement is always execution time for a given scenario.
Programmer	In our paper, the term programmer is pretty much synonymous with developer, coder, or software engineer. It is a person who creates software designed to take advantage of a particular computing environment.

3 Test Design

In this study, there are a couple of factors that need to be quantified and measured, which will influence the ability to determine whether or not the null hypothesis can be rejected. These factors are: programmability and performance. This section will not only define these terms, but detail how these definitions were developed and how these factors will be tested.

3.1 Programmability

The term programmability is one that is used often, but seldom well-defined using quantifiable metrics. In order to get a better idea of the history and use, a search was performed for the term "programmability" across the following databases:

1. Compendex
2. EBSCO
3. IEEE
4. Inspec
5. Web of Science

Results from this search were exported into various formats (CSV, Tab-delimited, XML), depending on the academic database being used. These results were then normalized and aggregated to remove redundancies between searches, resulting in a decrease from the original 11,346 documents retrieved down to 5,494 unique documents (after removing non-alphabetical characters from titles and transforming to lowercase prior to hash generation and comparison).

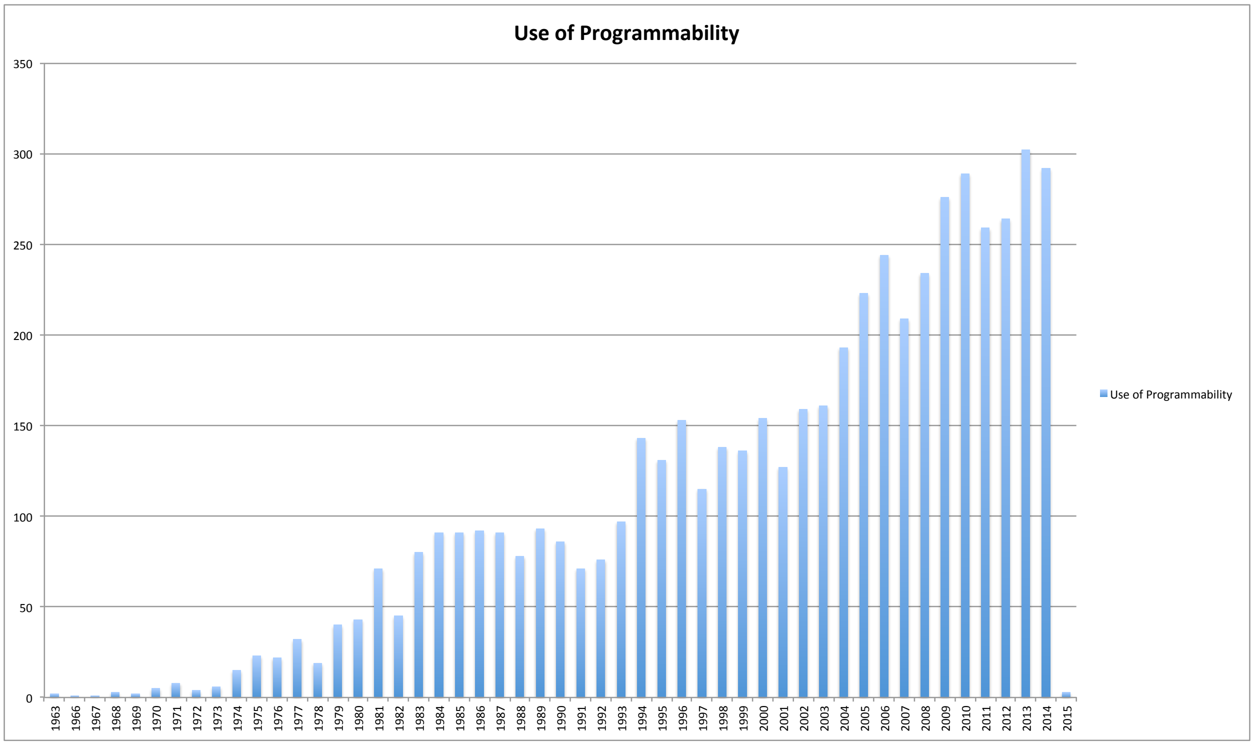

Figure 1: Use of programmability

Taking this high-level view of the term helped to visualize not only the genesis of its use in computing literature, but to also see how its use has grown over the years. The first time that this term was used was in 1963, in reference to "programming ability" [29]. It was used as a binary characteristic of the system under consideration - or, to put it another way, you either had the ability to program something, or you did not.

This definition continues in use today, especially in papers concerning bio-medical fields, hardware, and software research. While functional and pretty descriptive, this idea doesnt capture degrees (variation) in the ability to program something, which makes it difficult to measure variation in this property.

For this paper, we will build on the definition or programmability, as it was used in "Parallel Programmability and the Chapel Language" [3]. Section 2.2 of this article defines "Productive Parallel Language Desiderata", that include:

1. A global view of computation
2. Support for general parallelism
3. Separation of algorithm and implementation
4. Broad-market language features
5. Data abstractions
6. Performance
7. Execution model transparency
8. Portability
9. Interoperability with existing codes
10. Bells and whistles

This provides a framework for defining characteristics of parallel languages, but still fails to track the the ease or difficulty faced by users (programmers, in this case of this paper).

Key metrics to track programmability of MASS:

1. Time needed to learn and use parallel/distributed framework
2. Lines of code necessary to write parallel/distributed applications using framework
3. Developer assessment of parallel/distributed framework

For our study, the dependent variables will be:

1. Effort (LOC)
   Variable type: Ratio (continuous)
2. Time (Hours)
   Variable type: Ratio (continuous)
3. Programmability (Likert 1-5)
   Variable type: Ordinal (discrete)

The independent variable will be the "Framework Used," which is split into two groups:

1. Hybrid OpenMP/MPI
2. MASS

3.1.1 Sampling Technique

For this study, we used a convenience sampling of students at the University of Washington Bothell. Our sample was a non-random survey provided to students enrolled in CSS 534: Parallel Programming in the Grid and Cloud [15]. Students were asked to complete a programming assignment using hybrid OpenMP/MPI and, later in the course, were asked to use MASS to recreate the same program. After completing the assignment using MASS, our survey was administered to these same students.

3.1.1.1 Population Characteristics

There were two classes whose results were aggregated to form the basis of this thesis:

1. CSS 534 - Parallel Programming in Grid and Cloud/Spring 2014
2. CSS 534 - Parallel Programming in Grid and Cloud/Winter 2015

Each of these classes was a graduate level course. However, the make up of each class was slightly different. The first course (Spring 2014) consisted of students that had previous experience in programming and a desire to specifically learn parallel/distributed programming that targeted the grid/cloud. The second course (Spring 2015) contained a few undergraduate students and a large number of students that were experiencing their first graduate-level programming course. Since this was the first programming course available, many students opted to take it, not necessarily coming into the course with a strong desire to learn parallel/distributed programming in the grid/cloud, so much as just wanted to fulfill program requirements.

The number of students enrolled in each course differed, as you could guess - given that the second offering was a way of fulfilling program requirements earlier. As such, the number of students in the first class was 16, while the number taking the second was 30. It is worth noting that the second class contained 28 first year students and three undergraduate students.

3.1.2 Survey Design

We designed our survey to gather metrics around programmability for each framework (hybrid OpenMP/MPI and MASS). Specifically, we were interested in learning about the following points:

1. Time needed to...

   (a) Learn the library
   (b) Design the program
   (c) Write the program
   (d) Debug the program

2. Lines of code needed to...
   (a) Write entire application
   (b) Perform parallel-specific tasks

3. Developer assessment of...
   (a) Learning curve
   (b) Application suitability
   (c) Difference between sequential and parallel programs
   (d) Debugging difficulty
   (e) Call All methods between frameworks
   This is the act of accessing each Place or Agent (distributed array element) and performing a discrete, at times non-trivial, task at that location
   (f) Exchange All methods between frameworks
   This describes the ability to access each Place (distributed array element) and exchange data (interact) with other Places relative to the location of this element (neighbors) within a simulation
   (g) Manage All methods between frameworks
   The manageAll method refers to the task of actually updating each Agent's status within a simulation, based on a variety of tasks that this Agent could have performed (e.g. - move, spawn, kill, or suspend/resume).

Survey questions related to time were measured in hours and relied on students to remember or correctly estimate the amount of time different tasks actually took.

Survey questions measuring lines of code were also left up to students' ability to gather the information independently (no automated system to gather this information was provided).

Survey questions that measured degrees of satisfaction/disatisfaction (assessment) were measured using a Likert scale of 1 - 5, offering textual cues to help guide answers. In each question that used this type of measurement, the lowest option always corresponded with a negative evaluation (quite hard, not useful at all), whereas the highest option always corresponded with a positive evaluation (excellent, quite easy, quite useful).

Appendix A shows the actual survey that was provided to students.

3.2 Performance

Since the hypothesis relies on this notion of a performance difference between frameworks, we will need to quantify this difference. To accomplish this, we will take a look at the general performance charactistics of MASS. We will also take a more in-depth look at how MASS performs against the same application written using hybrid OpenMP/MPI across the following domains:

1. Agent-Based Models
2. Spatial Simulations

When evaluating each aspect of performance (general, agent-based models, spatial simulations), researchers have made an effort to adjust the resources available to the application. This ends up providing us with a sense of how each framework (OpenMP/MPI vs MASS) performs at different variations of the following characteristics:

1. Number of Processes
2. Number of Threads
3. Simulation Size
4. Iterations (Simulated Time)

Our performance results will present the various matrices that were collected, showing how each framework performed at various levels ad providing for a more in-depth look into how these frameworks stack up against each other.

3.2.1 General Performance

The general performance of MASS, calculated by Jennifer Kowalsky [24], was obtained by running a test program created by a previous student, Jay Hennan [18]. This application exposes several different tests that users can run by passing various command line arguments to the application. A detailed list of these arguments can be found in Appendix B.

We wanted to include this data to provide readers with an overall profile of how the agent-based approach of MASS performed at various levels - independent of an actual simulation. Our hope is to provide a sense of how MASS's built-in distribution and parallelization of work actually performs at various time and computational effort levels. We will then aggregate this data and provide three-dimensional graphs to help illustrate how execution time changes.

3.2.1.1 Test Types

There are two programs that we have created to obtain performance results for MASS. One program is configured to target the performance of Place calls within the MASS library, while the other program targets Agents.

3.2.1.1.1 Places Test Program

There are four test types that we have identified for gathering baseline performance characteristics of Places within MASS. These tests are given a numerical identifier, which is used on the command line when running the performance application to target a specific scenario. A detailed list, illustrating each test type can be found in Appendix C.

The main idea behind these tests is to exercise the built-in functionality (methods) that Place Objects in MASS expose to programmers. By isolating this functionality and varying hardware resources, computational effort, and simulation length, we begin to build a picture of MASS Place performance.

3.2.1.1.2 Agents Test Program

There are seven test types that we have identified for gathering baseline performance characteristics of Agents within MASS. These tests are given a numerical identifier, which is used on the command line when running the performance application to target a specific scenario. Once again, a detailed list that illustrates each test type can be found in Appendix D.

We wanted to exercise Agents in a similar manner to how we targeted our Place tests. So, you will see a variety of tests that target methods germane to Agents in MASS. This is important because terms like "agent migration" bring to mind non-trivial computing tasks for systems to handle. Being able to provide a picture of how these tasks actually perform as time, load, and resources are varied provides a more accurate picture of the overhead incurred through various Agent operations.

3.2.2 Agent-Based Models

Agent-Based Models are applications that model relationships between agents (objects) within a space. A classic example of an agent-based model would be a war strategy game - where troops are moved across a playing field. We will examine two agent-based models to gather practical performance profiles between MASS and OpenMP/MPI.

The first application, FluTE, is a large complex simulation that serves to model real-world scenarios related to influenza/epidemics, mitigation strategies, and emergent behaviors (almagamated from individual data points within model) of the overall population as an effect of varying simulation parameters. It is also notable in that a parallelized OpenMP/MPI-compliant version of this application already exists, which we were able to reuse with some slight modifications (bug fixing).

The other applications, Sugarscape, is a smaller proof-of-concept sort of application that serves to simulate a far less complex scenario. This application differs significantly from FluTE in that the general runtime cost (execution time) is much lower, the simulation is much less complex, it is generally easier to maintain/track inter-related simulation variables, and the emergent properties of the simulation are not very interesting to users.

3.2.2.1 FluTE

FluTE is "an individual-based simulation model of influenza epidemics" [5] and fits our understanding of an agent-based model quite nicely. In this model, the simulation space is broken up into census tracts, communities, and households. Within these constructs, agents represent individuals that each have a possibility of contracting an infection (either through initial seeding of the population, or through subsequent transmission from someone already infected).

The implementation of FluTE that we used for performance testing was developed by Osmond Gunarso. Osmond [17] described several interactions (processes) that are taking place during each iteration: Agent Interactions, State and Places Interactions, and Master Interactions. For simplicity, we have condensed these into one general process that is repeated for each iteration of the simulation, as follows:

1. Agents Interact with One Another
   - (a) If I am not sick
     - i. Interact with every sick person in the community
       - A. If I get sick stop and change my state
   - (b) Save current state
   - (c) Migrate to my next location
2. Move to Night
3. Agents Interact with One Another
   - (a) If I am not sick
     - i. Interact with every sick person in the community
       - A. If I get sick stop and change my state
   - (b) Save current state
   - (c) Migrate to my next location
4. Move to Day
5. Update State and Places
   - (a) Start vaccines
     - i. Open schools as appropriate
     - ii. Count ascertained cases of infection
     - iii. If there is an epidemic

A. Adjust migration policies

iv. Close or open schools

v. Take stock of vaccines

vi. Distribute vaccine

(b) Repeat for antivirals omitting schools

3.2.2.2 SugarScape

Sugarscape is a broad term used to describe social models that all, "include the agents (inhabitants), the environment (a two-dimensional grid) and the rules governing the interaction of the agents with each other and the environment." [9] As such, it is also quite suited as a representational algorithm for evaluating performance characteristics between hybrid OpenMP/MPI and MASS.

In the implementation used for comparing OpenMP/MPI and MASS application performance, the focus is on monitoring how agents survive within the simulation, given limited space, resources (sugar), and their individual metabolism. Abdulhadi Ali Alghamdi [1], the researcher who developed/collected these results, describes the general algorithm, as follows:

> "In the simulation, I allocate the sugar and the places, followed. Then, I create the agents in the number specified by the user. Each process then is allocated a chunk of the agents, given their relative position and metabolism to the rest of the agents handled by other processes. After that begins the traversal procedure: some agents find sugar to consume, all agents have their metabolism changed accordingly, and all agents relocate randomly to survive."

This implies the following steps taking place during each iteration of the simulation:

1. Some Agents Find Sugar to Consume
2. All Agents Have Their Metabolism Changed Accordingly
3. All Agents Relocate Randomly to Survive

3.2.3 Spatial Simulations

Spatial Simulations are applications that model the relationship of a space with its given neighbors. Spatial simulations differ from agent-based models in that there is no concept of an "agent" (object) that needs to be modeled across a given simulation space. Instead, these models track the behavior of the space itself. Good examples of spatial simulations are problems like modeling the heat transfer across a known medium or modeling wave dynamics. In each of these cases, the model can exist and run without needing to add additional logic outside of the characteristics of the simulation space itself.

The application We will use to gather spatial simulation performance data is called Wave2D. Wave2D is very similar to Sugarscape, in that it is a smaller proof-of-concept sort of application and it represents a far less complex scenario. This application also differs significantly from FluTE along the same lines that Sugarscape did: significantly smaller execution times, far less complex simulation, reduced inter-related simulation variables (Places only, no Agents), and the emergent properties of the simulation are not very interesting to users.

3.2.3.1 Wave2D

Wave2D is a wave dissemination simulation, based on Schroedingers wave formula [14], that models how a column of water (wave) disperses within a two-dimensional space over time. This type of simulation can be modeled as a spatial simulation by considering the simulation space as the water itself. Using this metaphor, the solution fits perfectly with our idea of a spatial simulation, as each section of the simulation space only needs to know the characteristics of its neighbors to influence its own characteristics over time.

The basic algorithm for our test application, follows the guidelines set up in the second homework assignment for CSS 543: Parallel Programming in Grid and Cloud - Multithreaded Schroedingers Wave Simulation [14].

After the simulation space is set-up, an initial amount of water is added to the center of the simulation space. Then, during each iteration of the simulation each Place ends up calculating its new (current) height. This calculation is based off of the following factors, which lead to the complexity in the design:

1. Previous height of Place over last two iterations
2. Previous height of neighboring Places over last two iterations

This implies that our Place objects either have to store historical wave height data as attributes, or the simulation space has to be three dimensional to account for different heights at different times (current time, previous iteration, and previous previous iteration). In our case, Abdulhadi explains his implementation, as follows "[We] used an object with the previous states stored within them. It [is] relatively simple: basically, [we] made a struct 'Cell' with doubles t, t-1, and t-2, created an MPI_Datatype for the struct, declared it (MPI_Type_create_struct), then committed it (MPI_Type_commit)." [1]

This approach represents one of the fundamental differences between MASS and OpenMP/MPI. As you can see, MPI is unable to pass complex Objects around, instead relying on custom structs to model data needed in the simulation. MASS, on the other hand, has a much more familiar Object-oriented approach to modeling data, allowing custom Place and Agent Objects to not only store their own data attributes, but to also contain their own functions/methods - which, are accessible through the base callAll() method.

4 Results

4.1 Programmability

4.1.1 General Programmability

There have been studies into the programmability of various parallel/distributed frameworks in the past. Instead of redefining the playing field each time a new paper is written, we are going to reuse the assessment criteria codified by B. L. Chamberlain, D. Callahan, and H. P. Zima(2007) [3] in their paper, "Parallel programmability and the chapel language," applying the same measures against both hybrid OpenMP/MPI and MASS applications.

Each section below will discuss the merits/drawbacks of each approach for the given assessment category. When applicable, sample code will be provided to illustrate the real difference, as it applies to writing applications in each of these frameworks. At the end, we will present a roll-up summary of the findings, offering a concise view of how hybrid OpenMP/MPI and MASS stack up against one another.

4.1.1.1 Global View of Computation

The idea of a global view of computation, is one "in which programmers express their algorithms and data structures as a whole, mapping them to the processor set in orthogonal sections of code, if at all. These models execute the programs entry point with a single logical thread, and the programmer introduces additional parallelism through language constructs" [3]. To put this another way, it is the idea that a framework allows for clean parallelization without having to significantly alter the data structures and logic to support parallel execution.

4.1.1.1.1 Hybrid OpenMP/MPI Support

Hybrid OpenMP/MPI applications provide a "mixed" support of a global view of computation.

On the OpenMP side of things, you are presented with a very simple, easy-to-use set of compiler directives that enable programmers to quickly parallelize simple code constructs (loops). However, OpenMP also forces programmers to consider shared data within their parallel sections, in order to obtain efficient memory use and scalability in their code.

Using MPI, you are presented with a fragmented view of the computation - meaning, pretty much the exact opposite of a global view. Programmers are forced to split data into chunks that correspond with how many machines will be simultaneously operating on the computation, then they must handle non-trivial problems related to things like cross-boundary communication and synchronization of distributed tasks.

To illustrate this concept in use, let us consider the case of setting up MPI for a parallel/distributed application. Leaving out the set-up and initialization of MPI, one of the first things that programmers will need to do is to break their data/problem space up into "chunks" that each MPI rank (machine) can work on in parallel.

The following code blocks are examples from a Heat2D application that synthesizes heat transfer across a space over time. For simplicity, much of the logic for cross-boundary communication has been removed. Starting from the top, you can see what it would look like to just set up for a 200 x 200 unit simulation space:

```
int size = 200;                          // simulation space
int mpi_size = 4;                        // # of mpi processes
int mpi_total_elements = size * size;    // # elements to process
int mpi_buffer_size = mpi_total_elements / mpi_size; // # elements per rank
double heat[mpi_total_elements];         // 1d representation of space
double rank_heat[mpi_buffer_size];       // rank-specific section of array
```

Next, programmers would need to send this data out to each rank (machine) in participating in the computation. While there are many methods of communication available, we will illustrate one of the simpler ones here:

```
MPI_Scatter(heat, mpi_buffer_size, MPI_DOUBLE, rank_heat, mpi_buffer_size,
    MPI_DOUBLE, MPI_MASTER, MPI_COMM_WORLD);
```

Finally, programmers need to create complex logic to differentiate what machine is working on what section of the simulation space and handle cross-boundary communication appropriately:

```
if (mpi_rank == 0) {    // master MPI machine
  MPI_Send( send_lookbehind_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE,
      (mpi_rank + 1), mpi_tag, MPI_COMM_WORLD );
  MPI_Recv( lookahead_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE, (mpi_rank + 1),
      mpi_tag, MPI_COMM_WORLD, &mpi_status );
} else if (mpi_rank == mpi_size - 1) {    // last ranked MPI machine
  MPI_Send( send_lookahead_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE,
      (mpi_rank - 1), mpi_tag, MPI_COMM_WORLD );
  MPI_Recv( lookbehind_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE, (mpi_rank - 1),
      mpi_tag, MPI_COMM_WORLD, &mpi_status );
} else {    // middle two machines (rank 1 & 2)
  MPI_Send( send_lookahead_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE,
      (mpi_rank - 1), mpi_tag, MPI_COMM_WORLD );
  MPI_Recv( lookbehind_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE, (mpi_rank - 1),
      mpi_tag, MPI_COMM_WORLD, &mpi_status );
  MPI_Send( send_lookbehind_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE,
      (mpi_rank + 1), mpi_tag, MPI_COMM_WORLD );
  MPI_Recv( lookahead_buffer, LOOKUP_BUF_SIZE, MPI_DOUBLE, (mpi_rank + 1),
      mpi_tag, MPI_COMM_WORLD, &mpi_status );
}
```

We have left out some of the other hurdles that programmers will have to overcome. Suffice it to say, coordinating messaging, maintaining the integrity of shared data, and managing the partitioning/aggregation process in an MPI-driven application is not simple.

So, when we are talking about a “fragmented view” of computation, this is exactly what we mean. On the other hand, we can look to OpenMP for a good example of a global view of computation.

Using OpenMP, developers can add compiler directives that will compile their existing code into applications that are optimized to take advantage of multiple cores on a single machine. Here is an example, using the same Heat2D application, of how OpenMP can quickly optimize a loop for parallel execution:

```
#pragma omp parallel for default( none ) firstprivate( p, p2, NUM_COLUMNS, size,
lookahead_buffer, lookbehind_buffer ) private( east, west, north, south ) shared(
rank_heat, send_lookbehind_buffer )
```

This is what is meant by a global view of computation. Programmers do not need to spend time splitting a data structure apart to have OpenMP use it. There is some knowledge of the visibility and sharing of memory that OpenMP imposes on programmers, but it does not force major changes to the data or algorithm to fragment it into easily-parallelized computational units.

4.1.1.1.2 MASS Support

MASS supports a global view of computation. Programmers are not required to split their data or algorithm apart in order to “bake in” parallelization. Instead, the challenge comes in the form of adapting their needs to MASS’s agents and places paradigm. To illustrate this, we will continue to use the Heat2D scenario (from the OpenMP/MPI discussion; above), but adapted toward MASS.

The first thing developers will need to do is set up their simulation space. This is much cleaner - no slicing of data/manually partitioning arrays:

```
// distribute places over computing nodes
int size = 200;           // simulation space
char *msg = "start\0";    // arbitrary start message
Places *sections = new Places( 1, "Section", msg, 7, 2, size, size );
sections->callAll( Section::init_ );    // initialize places
```

The problem of communicating with each place in the simulation is also much simpler in MASS:

```
sections->callAll( Section::calculateDispersal_ );
```

4.1.1.2 Support for General Parallelism

General parallelism, in this context, is the notion that a framework can support multiple approaches for parallelizing applications. There are two main types that we consider here, when comparing hybrid OpenMP/MPI and MASS:

1. Task vs Data Parallelism
   Are frameworks only able to achieve parallel computation by breaking down data into operable chunks (data), or are they also able to break apart processes that operate on the same data into operable chunks (task)?
2. Support for Nested Parallelism
   Are frameworks able to support multiple layers of parallelism (nested), or are they simply able to parallelize top-level constructs, leaving nested opportunities for simultaneous computation left up to serial execution?

4.1.1.2.1 Hybrid OpenMP/MPI Support

By itself, MPI is only able to achieve top-level parallelization. However, a hybrid implementation that also uses OpenMP is able to nest parallelizable code and achieve a greater degree of distributed work at runtime.

MPI is well-suited toward data decomposition, but could be used for task parallelization, too - with a bit of effort. However, thanks to OpenMP's inclusion of constructs for defining "sections" within a "parallel" directive, a hybrid OpenMP/MPI application can enjoy both forms of parallelization quite simply. Each "section" defined can encompass a discrete task, which can also benefit from nested parallelization, as described above.

The following (very basic, psuedo-code) example shows how we can use these constructs to obtain task parallelization and nested parallelization within the same application:

```
JsonObject userHistory;
#pragma omp parallel default(none) shared(userHistory)
{
    #pragma omp sections
    {
        #pragma omp section
        {
            // parse/store user ID from userHistory
        }
        #pragma omp section
        {
            #pragma omp for
            for( i = 0; i < numMovieTitles; i++ )
            {
                // parse/store viewing history from userHistory
            }
        }
        #pragma omp section
        {
```

```
            // parse/store related titles/suggestions from userHistory
        }
    }
}
```

So, a hybrid OpenMP/MPI application fully supports the concept of general parallelization.

4.1.1.2.2 MASS Support

MASS is capable of supporting the concept behind nested parallelism - being able to make full use of the cores available to a simulation, but it does not offer the same kind of fine-grained control that OpenMP presents to users. Instead, MASS handles communication and distribution of work behind the scenes - allowing an easier entry into utilizing system resources than hybrid OpenMP/MPI.

If the initial goal of "General Parallelism" was to ease the burden of having to fragment and coordinate tasks to achieve parallel execution, then it would follow that hiding the low-level breakdown of nested parallelizable sections of an application would actually be a bonus. So, while the explicit commands may be missing, the fact is that MASS automatically breaks your Places across processors on machines and further breaks down Agents to run on Threads per machine process (corresponding to Place they exist within). So, in order to achieve nested parallelism in MASS, you really have to consider the overall application design and how parallel tasks or nested parallelization can take advantage of the Places/Agents metaphor.

Using the same example, as above (in simple pseudo-code), we could achieve task parallelization in MASS with the following code:

```
    Places *places = new Places( 1, "ExamplePlace", msg, sizeof( msg ), 2, size, size );   // create grid
    Agents *workers = new Agents( 2, "ExampleWorker", ( void * )args, sizeof( args ), places, Nrequested );    // distribute workers

    workers->callAll( ExampleWorker::parseData_ );
```

```
void *Agent::parseData( void *argument ) {
    switch (agentId % 3)
    {
        case 0:
            // parse/store user ID from userHistory
            break;
        case 1:
            // parse/store viewing history from userHistory
            break;
        case 2:
        default:
            // parse/store related titles/suggestions from userHistory
            break;
    }
}
```

Unfortunately, we'd have to adjust the logic in our "parse/store viewing history from userHistory" method in each worker to decompose the data set being worked on to support true nested parallel execution. While possible using similar patterns (e.g. - use "floor(agentID / 3)" to translate back to numeric series, then use the value to correspond to nested JSON array element to process, etc), it is not the easiest thing to create, test, and maintain these types of complex structures (e.g. - you would need to apply extra logic in the case that the user history is longer than the number of Agents running in your simulation divided by three - a very real possibility, considering my Netflix binging habits).

4.1.1.3 Separation of Algorithm and Implementation

The idea of separating the algorithm from the implementation really boils down to the ability to express "algorithms in a manner that is independent of their data structures implementation in memory." [3] In most cases, it seems like the major concerns are with language support for different parallel frameworks and inconsistencies or esoteric requirements that could force the algorithm to change in order to suite the programming language.

In our case, both of the frameworks being compared have been limited to their C++ versions. So, there will not be any differences here that are imposed by the language. However, there are some concerns worth mentioning, as they relate to data structures, memory, and fine-tuning algorithms to take advantage of these factors.

4.1.1.3.1 Hybrid OpenMP/MPI Separation

Hybrid OpenMP/MPI applications do not have to explicitly adjust their implementation of an alogorithm to meet esoteric demands of C++. However, it is highly-recommended that programmers understand the limits of the architecture being used (especially as it relates to memory, cache-size, etc) in order to obtain optimal performance of their application.

Both OpenMP and Open MPI require programmers to correctly understand memory being used, in order to make the most efficient use of cache. This can translate into having to implement an algorithm differently, in the case of using an optimal slicing approach and iteration approach for data arrays.

The following example shows how an implementation may have to change, to make better use of cache and avoid cache misses:

```
double score[size * size];     // previous score
double new_score[size * size]; // new score

#pragma omp parallel for
for (int y = 0; y < size; y++) {
    #pragma omp parallel for
    for (int x = 0; x < size; x++) {
      new_heat[(x*size) + y] = heat[(x*size) + y] + 1.0;
    }
}
```

To correct this, you would want to adjust the for loops, as follows:

```
double score[size * size];     // previous score
double new_score[size * size]; // new score

#pragma omp parallel for
for (int x = 0; x < size; x++) {
    #pragma omp parallel for
    for (int y = 0; y < size; y++) {
      new_heat[(x*size) + y] = heat[(x*size) + y] + 1.0;
    }
}
```

While it is difficult to think of a real-world algorithm that would specifically call for this kind of cherry-picking through a sequential array, but it is conceivable that you could have an algorithm that was something like:

1. Compile Alphabetical List of All World Cities

2. Update Current City Weather Details By Timezone

Using an approach like this, you would run the distinct possibility of hitting cache misses that could seriously impact performance. Depending on the size of the city Objects in your code, it may become unavoidable. Still, you could end up with better runtime performance and fewer cache misses, using an adjustment to the algorithm like:

1. Compile Alphabetical List of All World Cities

2. Update Current City Weather Details Alphabetically By Timezone

4.1.1.3.2 MASS Separation

MASS is also free from having to specifically account for shortfalls in the C++ programming language, as it relates to having to adjust your original algorithm. Due to the nature of MASS imposing its Places/Agents paradigm on algorithms and managing the underlying parallelization that takes place, a lot of the concern here is mitigated.

The downside to this is that it does not allow the same fine-grained optimization choices that OpenMP/MPI application will expose to programmers. However, in the context of this category (Separation of Algorithm and Implementation), this is probably a benefit.

The other downside to this, is that MASS imposes its own structure around an application. While this structure is actually quite useful and easier to work with than OpenMP/MPI for certain application domains (spatial simulations, agent-based models), it does provide a challenge when adapting other types of problems to its underlying model (big-data analysis).

So, while the algorithm may not need to change to fit the implementation, the algorithm itself may have to change to fit the model that MASS presents (Agents/Places).

4.1.1.4 Broad-market Language Features

This category has to do with the idea that newer programming languages contain features that are quite useful and are well-known to newer programmers. To evaluate how OpenMP/MPI applications compare to MASS in regards to broad-market (current) language features, we will examine their support for concepts described by Chamberlain, et al (2007) [3], such as:

1. Object-Oriented Programming
2. Function/Operator Overloading
3. Garbage Collection
4. Generic Programming
5. Latent Types
6. Support for Programming In-The-Large
7. Support for Support Routines (Libraries, etc)

4.1.1.4.1 Hybrid OpenMP/MPI Support

1. Object-Oriented Programming
   The main limitation here is MPI support of custom types. Users can declare their own custom types and pass the Objects along to remote machines using MPI. However, these can not be considered true Objects, since they do not contain functions/methods. Instead, they should be considered more as "structs" - complex data structures.

2. Function/Operator Overloading
   According to Hughes & Hughes(2004) [20], the C++ ability to provide operator overloading can be leveraged to simplify sending and receiving using MPI functions. However, given that OpenMP is governed by compiler directives, this same sort of functionality is not available to all aspects of a hybrid OpenMP/MPI application.
3. Garbage Collection
   Not supported by hybrid OpenMP/MPI applications.
4. Generic Programming
   While there have been proposals [23] [32] for extensions to OpenMP that would allow for support of C++ generics, as of this writing, they have not been included in the standard. Similarly, due to MPI's use of pre-defined or custom-defined datatypes in communication, generics are not supported.
5. Latent Types
   Avoiding a deep discussion into the nuances of C++ that support dynamic typing through inheritance and pointer assignment in C++, we will instead focus directly on factors within OpenMP/MPI that allow for latent typing (or not). As the case stands, MPI enforces types when sending/receiving messages. As discussed earlier, custom types for MPI can be created/used, but are more like structs than true Objects. However, the idea that we could possibly use latent types within MPI is not supported, since MPI will need to know the size and structure of the data being sent in order to correctly marshall/unmarshall these Objects for remote communication. OpenMP provides the same support for latent-like metphors in C++ (inheritance, type casting, implicit type conversion), but true latent typing is also not supported within the C++ OpenMP framework.
6. Support for Programming In-The-Large
   The C++ language offers support for creating modularized code that can be organized, developed in tandem, and combined to form large enterprise-scale applications. Using hybrid OpenMP/MPI within these applications can add to the complexity of managing computing resources - network traffic with MPI and thread management with OpenMP. However, there is nothing inherent in each framework that specifically targets large scale development.
7. Support for Support Routines (Libraries, etc)
   OpenMP and MPI are essentially libraries that provide routines to allow applications to perform message-passing (distribution of work) and division of work between threads (parallelization of work). As such, they fundamentally provide support for routines.

4.1.1.4.2 MASS Support

1. Object-Oriented Programming
   MASS fundamentally supports Object-Oriented programming with their Places and Agents paradigm. Programmers must extend these classes and provide their own implementation for MASS simulations. However, there is no scope or limitation placed on functions/methods that can be created and used within these classes.
2. Function/Operator Overloading
   MASS applications can similarly take advantage of the ability to overload operators in C++. However, due to the way that functions are referenced/called within custom Place/Agent classes (mapped to numerical ID), the same can not be true for function overloading.
3. Garbage Collection
   Much of the inner-workings of MASS is out of control for programmers using this system. So, a great deal of memory management is already taken care of within a MASS application. However, the framework itself still leaves the possibility of memory leaks if user-implemented Place/Agent classes end up creating dynamic memory that is not cleaned up in their destructors (or neglecting to create a proper destructor, in the first place). It is also worth noting that MASS does not currently make use of smart pointers - a potential area for future improvement.

4. Generic Programming
   MASS currently relies on inheritance (extending parent/base Place/Agent classes) to provide users a method to customize MASS for their own applications. Within these classes, it is possible to use C++ templates, but for greater flexibility, it would be a nice improvement to translate this paradigm into an actual template interface.
5. Latent Types
   Through C++ inheritance and polymorphism, dynamic typing (in terms of runtime pointer assignment, implicit casting, etc) is possible within MASS, but it does not offer the same type of latent typing available in other languages like Python or Javascript (types are still tied to variables, instead of to values).
6. Support for Programming In-The-Large
   MASS is really designed to facillitate agent-based models and spatial simulations. Large-scale projects can use MASS within individual modules, or as a center-piece to the overall application code (depending on the scope/objective of the application). However, in the sense of providing language to specically target integration of modular code into an enterprise-level deliverable, MASS is limited to the constructs available in C++.
7. Support for Support Routines (Libraries, etc)
   MASS is essentially a library that can be integrated into projects and manipulated (through the creation of custom Agent/Place Objects) to suit an application's needs. It provides an API for interacting with Agents/Places within the simulation space and pulls in its own library to provide support for SSH communication (libssh2). It is flexible enough to allow other libraries to be integrated into the codebase, but does not currently provide recommended libraries to plug-and-play various add-on routines to the core framework.

4.1.1.5 Data Abstractions

Data abstractions has to do with a framework's support of different data structures to model the application. It tends to be the case that parallel/distributed frameworks are written to support a single data structure, often forcing programmers to adapt their model to fit the framework's needs. For instance, a website dealing with family trees may prefer to keep their data structured in a binary tree - easing the readability/maintainability of their code, as the abstraction lends itself well toward the subject domain. On the other hand, in order to parallelize a search across this tree, they may be forced to flatten the hierarchical structure of their data into a one-dimensional array.

4.1.1.5.1 Hybrid OpenMP/MPI Support

MPI supports the passing of arbitrary bits of data to remote machines for processing. As such, there is not really a notion of a forced data structure on application programmers.

Along these same lines, OpenMP does not force a particular structure on programmers to be able to parallelize tasks that operate on these structures. While, on the surface, this seems like a good thing, it can end up causing a litany of problems with the actual performance of the application (thrashing, cache misses, thread starvation, locking issues, etc). So, while you can traverse a binary tree, the results could be terrible as the random spots in memory could span cache lines and result in very poor performance as multiple threads spend cycles swapping in data from RAM (or worse - disk).

4.1.1.5.2 MASS Support

MASS provides its own data abstractions for programmers. So, on the surface, it does not support different methods of modeling the data. Instead, it forces programmers to analyze their algorithms and adjust them to fit the paradigm set up by MASS (Agents distributed over Places within a simulation space).

4.1.1.6 Performance

Performance deals with the actual runtime characteristics (usually actual time, or computing cycles) of frameworks. Other aspects of performance could be CPU utilization and cache efficiency. For the purpose of this study, we have conducted our own performance evaluation of each framework - operating on similar applications, across a variety of domains, using similar hardware. More details can be found in section 4.2.2.4. Details on how these tests were designed can be found in section 3.2.

4.1.1.7 Execution Model Transparency

This section deals with the ability for programmers to know how communication and parallelization is achieved within the framework. Having access to this knowledge (transparency) allows programmers to fine tune applications to make better use of computing resources and improve the runtime characteristics of the programs.

4.1.1.7.1 Hybrid OpenMP/MPI Support

Out of the box, OpenMP and MPI provide a good level of transparency. This is because they are, in themselves, a fairly low-level method of obtaining parallel/distributed functionality within your code. Instead of providing a high-level framework that masks how data is decomposed, how/when remote calls are made, etc - the hybrid OpenMP/MPI model forces programmers to make these decisions themselves, using methods from these libraries to control and achieve parallelism as needed.

4.1.1.7.2 MASS Support

MASS provides a higher-level framework that masks much of the parallel/distributed nature of the underlying application. There is documentation that serves to illustrate "how" code distribution and parallel computation is mapped to actual hardware resources [16] that can be found by searching on the web. Figure 2 illustrates how processes and threads are mapped to hardware to support the Agents/Places paradigm in MASS. However, detailed information and illustrations of the underlying functionality in MASS are either very hard to find or non-existent - making it quite difficult to tune applications built using MASS.

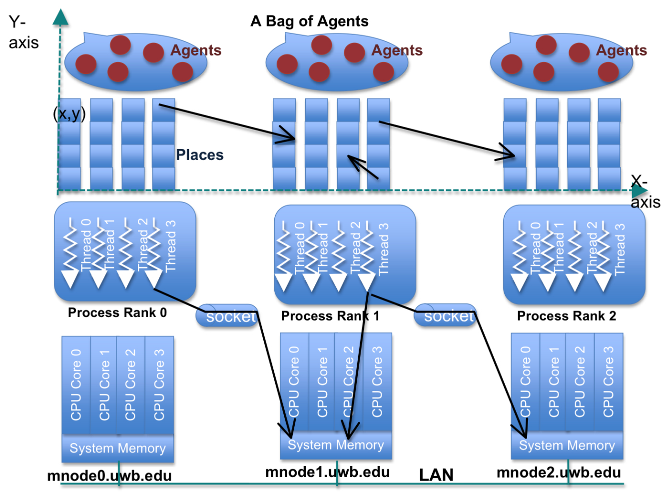

Figure 2: Parallel Execution with MASS Library

4.1.1.8 Portability

Portability deals with the concept of being able to move an application from one architecture to another, with relative ease. If a parallel/distributed framework is highly portable, then that means that it is widely supported across platforms (Mac OSX, Windows, Linux, etc). On the other hand, if frameworks are not portable, then this means that much work is required to translate from one architecture to another, or that certain architectures are just not supported.

4.1.1.8.1 Hybrid OpenMP/MPI Support

Hybrid OpenMP/MPI applications built on C++ enjoy a high degree of portability, as they are supported frameworks across many architectures. Since we are considering the C++ flavor of these frameworks, it is worth noting that applications will have to be compiled on/for the architecture that they are targeted to run on. But, this is a common limitation to C++ and is not specific to OpenMP or MPI.

4.1.1.8.2 MASS Support

MASS has a dependency on the libssh library - but, this does not directly limit its portability. Instead, the only limiting factor would seem to be whether or not a platform supports C++ compilation (i.e. - a C++ compiler exists to compile source code into object/machine code capable of running on that platform). So, the portability of MASS would appear to possess the same constraints as hybrid OpenMP/MPI applications. It is worth noting, though, that MASS has currently only been used on grids supporting a Linux kernel. It would be worth the effort to test and debug this framework on varying architectures to identify/address any potential issues and improve the portability of the system.

4.1.1.9 Interoperability with Existing Codes

The two frameworks in this study, hybrid OpenMP/MPI and MASS, are written in C++. They both support other languages, but the idea of interoperability is really kind of a moot point. This idea is based around frameworks that may have their own language/constructs that programmers would have to learn, and that may not integrate well with existing languages. Since this is not the case for either of the frameworks we are considering in this study, they both get a pass on this category (both support).

4.1.1.10 Bells and whistles

This section takes a look at add-ons that could increase the usability of a framework, or the productivity of developers programming within a framework. Things like built-in debuggers, IDE extensions, performance monitors, GUIs (Graphical User Interfaces), or other tooling built around the framework will be considered in this section.

4.1.1.10.1 Hybrid OpenMP/MPI Support

Hybrid OpenMP/MPI applications offer the following benefits to programmers, in this category:

1. Visual Studio Integration
   Visual Studio has integrated support for MPI development [26]. This includes things like:
   (a) MPI Cluster Debugger for C++ MPI applications
   (b) Project templates for C/C++ MPI programs
2. Eclipse Integration
   The popular Eclipse IDE has also integrated tooling to support parallel application development [13]. Their tooling provides:
   (a) Support for the MPI, OpenMP and UPC programming models, as well as OpenSHMEM and OpenACC
   (b) Support for a wide range of batch systems and runtime systems, including PBS/Torque, LoadLeveler, GridEngine, Parallel Environment, Open MPI, and MPICH2
   (c) A scalable parallel debugger
   (d) Support for the integration of a wide range of parallel tools
3. Performance Monitoring Solutions
   (a) POMP [21] for OpenMP
   (b) IDE tooling (above) for MPI
4. Several Data Visualization Tools
   We will leave it up to the reader to Google this phrase. Suffice it to say, we immediately found three promising solutions right off the bat.
5. Extensive Documentation
   It is not so much a bell or a whistle, but it really was not covered in other categories, so we wanted to call attention to it here. OpenMP and MPI have been around for over 20 years and have been widely-adopted by all of the big names in computing and high-performance computing. So, with a few quick presses of some keys and a click or two of the mouse, it is pretty easy to find:
   (a) Journal Articles
   (b) Entire Books
   (c) Code Examples
   (d) FAQ Sections
   (e) Message Boards

4.1.1.10.2 MASS Support

MASS has many irons in the fire that they are currently working on:

1. Integrating a Debugger
2. Improving Source Documentation
3. Improving User Documentation
4. Improving Usability of Framework
   (a) Identifying Pain Points in Current Workflows
   (b) Adding More Helpful Methods for Programmers
   (c) Addressing Bugs
5. Adding Asynchronous Support
6. Implementing Complete End-to-End Project Lifecycle Tooling

However, compared to the nearly two decade head start and support of nearly all technology companies during its continued development, OpenMP/MPI are just crushing MASS in this category.

4.1.1.11 Results of General Programmability Comparison

To ease the comparison of General Programmability between hybrid OpenMP/MPI and MASS applications, we have included a graphic that summarizes the textual comparison that was just presented. In Figure 3, we can see how each framework measured up side-by-side for each category. To aid in quantifying, we have scored each category on a 0 - 2 scale, indicating 0 for “no support”, 1 for “partial supported”, and 2 for “full support”.

	Hybrid OpenMP/MPI	MASS
Global View of Computation	1	2
Support for General Parallelism	2	1
Separation of Algorithm and Implementation	2	2
Broad-Market Language Features	1	1
Data Abstractions	2	0
Performance	2	1
Execution Model Transparency	2	0
Portability	2	2
Interoperability with Existing Codes	2	2
Bells and Whistles	2	0
Total Score	18	11

Figure 3: General Programmability Comparison

Looking at these results, it appears that OpenMP/MPI has the clear advantage over MASS. However, it is also worth noting that the measurement criteria is skewed in favor of "general" parallel/distributed frameworks and often overlooks or fails to take into consideration advantages that many paradigm-oriented frameworks like MapReduce, GlobalArray, UPC, or MASS offer to programmers.

We can see in Figure 4 what it would look like if categories that exhibited an inherent bias toward general programmability were removed from the comparison. You will notice the caption for this uses the word "impartial" - this is to indicate that categories with inherent bias have been removed.

	Hybrid OpenMP/MPI	MASS
Global View of Computation	1	2
Separation of Algorithm and Implementation	2	2
Broad-Market Language Features	1	1
Performance	2	1
Portability	2	2
Interoperability with Existing Codes	2	2
Bells and Whistles	2	0
Total Score	12	10

Figure 4: Impartial General Programmability Comparison

In this chart, the programmability between the two frameworks is much, much closer aligned, with the main difference being the support for "Bells & Whistles" - an advantage of OpenMP/MPI having been on the market for decades.

The justification for removing some of the categories from the intial comparison is:

1. Support for General Parallelism
   MASS focuses on agent-based models. If we included a category that also tracked how well a parallel/distributed framework handled agent-based modeling, then we'd see hybrid OpenMP/MPI underperform just as MASS failed to perform in this category. Instead of adding additional categories to the original criteria, we opted to simply remove this one from the comparison.
2. Data Abstractions
   This category is set up with the idea that being able to define and use your own data absractions is a bonus to programmers. However, it does not consider being able to use established abstractions already set up by paradigm-oriented libraries. Instead of creating a counter measure to track the ease in using pre-established abstractions for paralleling data in your application, we opted to remove this category, too.

3. Execution Model Transparency MASS and other paradigm-oriented frameworks have an unfair disadvantage right out of the gate in this category. It assumes that in-depth knowledge and exposure to the precise details of your execution model is a good thing. However, paradigm-oriented libraries take a different approach, working under the idea that abstracting platform details and execution model details from users is more beneficial. Instead of adding an additional category to represent both sides of this argument, we have also opted to remove this category from our comparison.

We believe that this is a more accurate comparison of general programmability features between MASS and OpenMP/MPI. Removing the bias against paradigm-oriented languages provides a more even playing field to perform our evaluation against. While the results still favored the hybrid OpenMP/MPI approach, we must remember that this framework has been around for much, much longer and has enjoyed support from a wide variety of organizations throughout its life.

4.1.2 Surveyed Programmability

In this section, we will present the results from our surveys. The results will typically be broken down to provide descriptive statistics for each question, before providing an overall comparison of the values discovered. For a more in-depth look at the actual survey results, please see Appendix E.

We would also like to take a moment to acknowledge the contributions of the students that took these courses and allowed us to use the data collected their surveys for this quantitative analysis. Without their work, none of this would have been possible [19].

4.1.2.1 Time

We tracked the time it took to complete several tasks while programming in OpenMP/MPI and MASS applications. This section details the results gathered from our research.

4.1.2.1.1 Time to Learn Library

Looking at Figure 5, we can see that it typically took programmers around 6 hours to learn how to use OpenMP and MPI to develop their applications.

Count	47	Range	30	Skewness Standard Error	0.33912	Fourth Moment	9,005.76163
Mean	5.98936	Sum	281.5	Kurtosis	8.98188	Median	4
Mean LCL	3.98968	Sum Standard Error	38.99484	Kurtosis Standard Error	0.63715	Median Error	0.15168
Mean UCL	7.98904	Total Sum Squares	3,174.25	Alternative Skewness (Fisher's)	2.35581	Percentile 25% (Q1)	2.5
Variance	32.35315	Adjusted Sum Squares	1,488.24468	Alternative Kurtosis (Fisher's)	6.8101	Percentile 75% (Q2)	8
Standard Deviation	5.68798	Geometric Mean	4.29797	Coefficient of Variation	0.94968	IQR	5.5
Mean Standard Error	0.82968	Harmonic Mean	3.27608	Mean Deviation	3.96514	MAD	2
Minimum	0E+00	Mode	4	Second Moment	31.66478		
Maximum	30	Skewness	2.27995	Third Moment	406.24728	Alpha value (for confidence interval)	0.02

Figure 5: Time to Learn Library: OpenMP/MPI

On the other hand, Figure 6, shows that it typically took programmers around 7 hours to learn how to use MASS when developing the same application.

Count	46	Range	15	Skewness Standard Error	0.3424	Fourth Moment	723.9028
Mean	7.03261	Sum	323.5	Kurtosis	2.53915	Median	6
Mean LCL	5.55507	Sum Standard Error	28.17732	Kurtosis Standard Error	0.64246	Median Error	0.11319
Mean UCL	8.51015	Total Sum Squares	3,051.75	Alternative Skewness (Fisher's)	0.62769	Percentile 25% (Q1)	4
Variance	17.26002	Adjusted Sum Squares	776.70109	Alternative Kurtosis (Fisher's)	-0.37246	Percentile 75% (Q2)	10
Standard Deviation	4.15452	Geometric Mean	5.74438	Coefficient of Variation	0.59075	IQR	6
Mean Standard Error	0.61255	Harmonic Mean	4.44905	Mean Deviation	3.38469	MAD	3
Minimum	1	Mode	#N/A	Second Moment	16.88481		
Maximum	16	Skewness	0.60704	Third Moment	42.11721	Alpha value (for confidence interval)	0.02

Figure 6: Time to Learn Library: MASS

This means that on average, programmers took 1.04 hours less time to learn the libraries for creating hybrid OpenMP/MPI applications.

4.1.2.1.2 Time to Design the Program

Figure 7 illustrates that on average, programmers took 5 hours, using OpenMP and MPI, to design their applications.

Count	47	Range	20	Skewness Standard Error	0.33912	Fourth Moment	2,763.0878
Mean	4.85319	Sum	228.1	Kurtosis	5.69787	Median	3
Mean LCL	3.18559	Sum Standard Error	32.51918	Kurtosis Standard Error	0.63715	Median Error	0.12649
Mean UCL	6.52079	Total Sum Squares	2,142.01	Alternative Skewness (Fisher's)	1.79302	Percentile 25% (Q1)	2
Variance	22.49994	Adjusted Sum Squares	1,034.99702	Alternative Kurtosis (Fisher's)	3.14792	Percentile 75% (Q2)	6.5
Standard Deviation	4.74341	Geometric Mean	3.15262	Coefficient of Variation	0.97738	IQR	4.5
Mean Standard Error	0.6919	Harmonic Mean	1.63139	Mean Deviation	3.50421	MAD	1.5
Minimum	0E+00	Mode	2	Second Moment	22.02121		
Maximum	20	Skewness	1.73528	Third Moment	179.32104	Alpha value (for confidence interval)	0.02

Figure 7: Time to Design the Program: OpenMP/MPI

When we consider the same task in MASS, Figure 8 shows that it typically took programmers around 6 hours to design their applications.

Count	46	Range	29	Skewness Standard Error	0.3424	Fourth Moment	10,135.50705
Mean	5.58696	Sum	257	Kurtosis	8.83367	Median	4
Mean LCL	3.4942	Sum Standard Error	39.90962	Kurtosis Standard Error	0.64246	Median Error	0.16032
Mean UCL	7.67971	Total Sum Squares	2,994	Alternative Skewness (Fisher's)	2.46474	Percentile 25% (Q1)	2
Variance	34.6256	Adjusted Sum Squares	1,558.15217	Alternative Kurtosis (Fisher's)	6.66397	Percentile 75% (Q2)	6
Standard Deviation	5.88435	Geometric Mean	3.83634	Coefficient of Variation	1.05323	IQR	4
Mean Standard Error	0.8676	Harmonic Mean	2.81499	Mean Deviation	3.90359	MAD	2
Minimum	1	Mode	4	Second Moment	33.87287		
Maximum	30	Skewness	2.38363	Third Moment	469.91294	Alpha value (for confidence interval)	0.02

Figure 8: Time to Design the Program: MASS

This means that on average, programmers took approximately 0.73 less hours (43 minutes and 48 seconds) to design their hybrid OpenMP/MPI applications.

4.1.2.1.3 Time to Write the Program

On average, programmers using hybrid OpenMP/MPI to write their applications took 8 hours, as evidenced in Figure 9.

Count	46	Range	49	Skewness Standard Error	0.3424	Fourth Moment	69,604.69852
Mean	8.22826	Sum	378.5	Kurtosis	14.0836	Median	5
Mean LCL	5.21336	Sum Standard Error	57.49536	Kurtosis Standard Error	0.64246	Median Error	0.23097
Mean UCL	11.24316	Total Sum Squares	6,348.25	Alternative Skewness (Fisher's)	3.0299	Percentile 25% (Q1)	4
Variance	71.86341	Adjusted Sum Squares	3,233.85326	Alternative Kurtosis (Fisher's)	12.53267	Percentile 75% (Q2)	10
Standard Deviation	8.47723	Geometric Mean	5.60604	Coefficient of Variation	1.03026	IQR	6
Mean Standard Error	1.2499	Harmonic Mean	3.72635	Mean Deviation	5.51323	MAD	3
Minimum	1	Mode	5	Second Moment	70.30116		
Maximum	50	Skewness	2.93019	Third Moment	1,727.18677	Alpha value (for confidence interval)	0.02

Figure 9: Time to Write the Program: OpenMP/MPI

Using MASS, we can see in Figure 10 that programmers spent around 7.5 hours writing their applications.

Count	45	Range	49	Skewness Standard Error	0.34578	Fourth Moment	79,356.99468
Mean	7.57778	Sum	341	Kurtosis	13.23533	Median	4
Mean LCL	4.37521	Sum Standard Error	59.69649	Kurtosis Standard Error	0.64791	Median Error	0.24785
Mean UCL	10.78034	Total Sum Squares	6,068.5	Alternative Skewness (Fisher's)	3.04247	Percentile 25% (Q1)	2.25
Variance	79.19268	Adjusted Sum Squares	3,484.47778	Alternative Kurtosis (Fisher's)	11.61701	Percentile 75% (Q2)	8
Standard Deviation	8.89903	Geometric Mean	4.88308	Coefficient of Variation	1.17436	IQR	5.75
Mean Standard Error	1.32659	Harmonic Mean	3.40591	Mean Deviation	5.70716	MAD	2
Minimum	1	Mode	#N/A	Second Moment	77.43284		
Maximum	50	Skewness	2.9401	Third Moment	2,003.31798	Alpha value (for confidence interval)	0.02

Figure 10: Time to Write the Program: MASS

The actual difference in means here (.65) shows that it took programmers, on average, 39 minutes less to write their corresponding applications using MASS.

4.1.2.1.4 Time to Debug the Program

Programmers debugging their applications written using hybrid OpenMP/MPI took about 8.5 hours, as shown in Figure 11.

Count	47	*Range*	51	*Skewness Standard Error*	0.33912	*Fourth Moment*	79,806.79287
Mean	8.40426	*Sum*	395	*Kurtosis*	17.15904	*Median*	6
Mean LCL	5.46959	*Sum Standard Error*	57.22762	*Kurtosis Standard Error*	0.63715	*Median Error*	0.2226
Mean UCL	11.33892	*Total Sum Squares*	6,525	*Alternative Skewness (Fisher's)*	3.47073	*Percentile 25% (Q1)*	4
Variance	69.68085	*Adjusted Sum Squares*	3,205.31915	*Alternative Kurtosis (Fisher's)*	15.92887	*Percentile 75% (Q2)*	10
Standard Deviation	8.34751	*Geometric Mean*	6.20182	*Coefficient of Variation*	0.99325	*IQR*	6
Mean Standard Error	1.21761	*Harmonic Mean*	4.6387	*Mean Deviation*	5.12902	*MAD*	2
Minimum	1	*Mode*	4	*Second Moment*	68.19828		
Maximum	52	*Skewness*	3.35896	*Third Moment*	1,891.7573	*Alpha value (for confidence interval)*	0.02

Figure 11: Time to Debug the Program: OpenMP/MPI

Using MASS, we can see in Figure 12 that the debugging time for their applications was also around 8.5 hours.

Count	46	*Range*	20	*Skewness Standard Error*	0.3424	*Fourth Moment*	2,977.6845
Mean	8.72826	*Sum*	401.5	*Kurtosis*	1.99793	*Median*	6
Mean LCL	6.49409	*Sum Standard Error*	42.60653	*Kurtosis Standard Error*	0.64246	*Median Error*	0.17116
Mean UCL	10.96243	*Total Sum Squares*	5,280.25	*Alternative Skewness (Fisher's)*	0.51915	*Percentile 25% (Q1)*	4
Variance	39.46341	*Adjusted Sum Squares*	1,775.85326	*Alternative Kurtosis (Fisher's)*	-0.97747	*Percentile 75% (Q2)*	13.5
Standard Deviation	6.28199	*Geometric Mean*	6.37455	*Coefficient of Variation*	0.71973	*IQR*	9.5
Mean Standard Error	0.92623	*Harmonic Mean*	5.14445	*Mean Deviation*	5.39036	*MAD*	4
Minimum	0E+00	*Mode*	5	*Second Moment*	38.60551		
Maximum	20	*Skewness*	0.50206	*Third Moment*	120.42945	*Alpha value (for confidence interval)*	0.02

Figure 12: Time to Debug the Program: MASS

Though the approximate debugging time for both frameworks was nearly identical, we can see (taking a closer look at the actual results) that, on average, it took programmers 19 minutes and 12 seconds (.32 hours) less to debug their corresponding applications using OpenMP/MPI.

4.1.2.1.5 Summary of Time Difference

To aid in visualizing how these frameworks stack up against one another, in terms of time taken to complete similar tasks, we have created Figure 13 - which shows the mean time it took programmers to complete various phases of the development process in each framework. As a helpful measure, the total average time has also been calculated, along with the difference between the time taken for hybrid OpenMP/MPI and MASS applications (represented as a numerical and percentage difference).

	OpenMP/MPI (Baseline)	MASS	Difference	Percent Difference
Learn the Library	5.99	7.03	1.04	17.42%
Design the Program	4.85	5.59	0.73	15.12%
Write the Program	8.23	7.58	-0.65	-7.91%
Debug the Program	8.40	8.73	0.32	3.86%
Total	27.48	28.93	1.45	7.12%

Figure 13: Time Summary Between OpenMP/MPI and MASS

As we can see from Figure 13, programmers using MASS typically took 7.12% longer to complete the various phases of application development. In terms of time, this translates to 1 hour and 27 minutes (1.45 hours).

4.1.2.2 Effort (Lines of Code)

The test design relied on students to gather precise measurements for the lines of code in their applications - measuring not only the total lines of code, but more importantly (for this study) the parallel/distributed-specific lines of code.

After collecting and reviewing survey results, we suspected that there had been a tendency to estimate these numbers (data appeared to be rounded) and we also began to understand that the method of determining what constituted parallel/distributed-specific code was ultimately left to individuals' understanding of this term - an understanding that could vary between respondents.

To address these issues, we went through the data and source code submitted to double-check and update actual values, using a consistent method for parallel/distributed-specific inclusion.

4.1.2.2.1 Inclusion Criteria: Hybrid OpenMP/MPI

For hybrid OpenMP/MPI applications, we considered parallel/distributed-specific code to be confined to the actual OpenMP or Open MPI statements/directives.

For OpenMP, examples of this include lines like:

Setting Number of Threads for OMP

```
omp_set_num_threads(numthreads);
```

Setting Compiler Directives for OMP Parallel Sections

```
#pragma omp parallel for default( none ) firstprivate(start, stop, size, r, p, p2)
 shared (z)
```

While, for Open MPI, we have more complex commands, such as:

Initializing MPI

```
MPI_Init(&argc, &argv);
MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
MPI_Comm_size( MPI_COMM_WORLD, &mpi_size );
```

Receiving Messages

```
MPI_Status status;
MPI_Recv( &(z[p][nodePointers[rank]][0]), nodeLength[rank]*size, MPI_DOUBLE, rank,
 tag, MPI_COMM_WORLD, &status);
```

Sending Messages

```
MPI_Send(&(z[p][nodePointers[my_rank]][0]), nodeLength[my_rank]*size, MPI_DOUBLE,
0, tag, MPI_COMM_WORLD );
```

Shutting Down MPI

```
MPI_Finalize( ); // shut down MPI
```

4.1.2.2.2 Inclusion Criteria: MASS

For MASS applications, we took a similar approach as hybrid OpenMP/MPI applications - concentrating on the specific calls to set up and use MASS functionality within the application to support parallel/distributed operation.

Examples of the types of lines we included are:

MASS initialization

```
MASS::init( arguments, nProc, nThr );
```

Places/Agents Construction/Initialization

```
Places *places = new Places( 1, "ExamplePlace", msg, 7, 2, size, size );
places->callAll( ExamplePlace::init_ );
Agents *agents = new Agents( 2, "ExampleAgent", msg2, sizeof( char* ), places,
PLACES_SIZE );
agents->callAll( ExampleAgent::agentInit_, msg2, sizeof( char* ) );
```

Places/Agents Calls

```
places->exchangeAll( 1, ExamplePlace::setupEdges_, &neighbors );
places->callAll( ExamplePlace::copyEdges_ );
agents->manageAll();
```

MASS Termination

```
MASS::finish( );
```

4.1.2.3 Actual Lines of Code

After reviewing the source code submitted for the course and applying the inclusion rules listed above, we found that the mean difference in values between reported and actual lines of code were pretty significant. As you can see in Figure 14 (below), the actual values were on average 65.91% less than the ones reported, with the most significant differences represented by the number of parallel/distributed lines of MASS code.

	Hybrid OpenMP/MPI Total LOC	Hybrid OpenMP/MPI Parallel-Only LOC	MASS Total LOC	MASS Parallel-Only LOC
Reported Mean	526	167	325	110
Actual Mean	191	24	247	11
Difference	336	143	78	99
% Difference	63.75%	85.66%	24.10%	90.16%
Avg % Difference	65.91%			

Figure 14: Difference in Lines of Code

Taking a closer look at these differences, there is a particularly (in terms of numbers, not necessarily percentage) glaring difference found between hybrid OpenMP/MPI reported lines of code, and actual lines of code. This is most likely a result of a combination of factors:

1. Lack of Effort
   While programs written in MASS were just recently developed/implemented by students, their corresponding hybrid OpenMP/MPI applications were finished several weeks prior. We are assuming that a large part of this overestimation is a result of being fixated on the current assignment and not doing due dilligence to go back and count only "applicable" lines of code individually (for instance, merely opening the file and looking at the number of the last line that appears - inadvertently including comments and whitespace in the reported value)
2. Overestimation
   Along this same line of thining, it is not unreasonable to assume that if students are finding a hard time exhaustively counting lines of code from a previous assignment (while trying to finish the current one), that they may simply "estimate" this value - potentially inflating if they remembered hybrid OpenMP/MPI being more difficult, or inherently having to include more code because of the fine-grained control required by this general-purpose parallelization approach (i.e. - having to distribute with MPI and then further parallelize using OpenMP; more frameworks, more code)
3. Counting Comments in with Total Even if students went through the additional work of manually counting lines of code and breaking on whitespace (easy enough), they may still have included lines that contain only comments in their total (or lines that contain only control characters that have been moved to their own line for legibility - closing braces, etc)

Due to the discrepancies uncovered from manually verifying the actual lines of code, the remainder of the study will be using the actual (corrected) lines of code, instead of the lines of code presented in survey results.

4.1.2.3.1 Hybrid OpenMP/MPI Applications

On average, hybrid OpenMP/MPI applications could be written in around 190 lines of code. We can see this data in Figure 15.

Count	40	*Range*	387	*Skewness Standard Error*	0.36432	*Fourth Moment*	53,081,397.8203
Mean	190.875	*Sum*	7,635	*Kurtosis*	5.77999	*Median*	166
Mean LCL	156.53367	*Sum Standard Error*	566.25841	*Kurtosis Standard Error*	0.67721	*Median Error*	2.80533
Mean UCL	225.21633	*Total Sum Squares*	1,769,963	*Alternative Skewness (Fisher's)*	1.88407	*Percentile 25% (Q1)*	141
Variance	8,016.21474	*Adjusted Sum Squares*	312,632.375	*Alternative Kurtosis (Fisher's)*	3.32802	*Percentile 75% (Q2)*	214
Standard Deviation	89.53332	*Geometric Mean*	175.94436	*Coefficient of Variation*	0.46907	*IQR*	73
Mean Standard Error	14.15646	*Harmonic Mean*	164.99306	*Mean Deviation*	61.525	*MAD*	35.5
Minimum	104	*Mode*	#N/A	*Second Moment*	7,815.80938		
Maximum	491	*Skewness*	1.81267	*Third Moment*	1,252,505.65547	*Alpha value (for confidence interval)*	0.02

Figure 15: Total Lines of Code: OpenMP/MPI

When considering the lines of code that are parallel/distributed-specific within the application, Figure 16 provides us with a value close to 24 lines of code for OpenMP/MPI applications.

Count	40	Range	50	Skewness Standard Error	0.36432	Fourth Moment	69,619.92284
Mean	23.975	Sum	959	Kurtosis	7.33497	Median	20.5
Mean LCL	20.1409	Sum Standard Error	63.22102	Kurtosis Standard Error	0.67721	Median Error	0.31321
Mean UCL	27.8091	Total Sum Squares	26,889	Alternative Skewness (Fisher's)	2.01652	Percentile 25% (Q1)	18
Variance	99.92244	Adjusted Sum Squares	3,896.975	Alternative Kurtosis (Fisher's)	5.09645	Percentile 75% (Q2)	27
Standard Deviation	9.99612	Geometric Mean	22.43997	Coefficient of Variation	0.41694	IQR	9
Mean Standard Error	1.58053	Harmonic Mean	21.23268	Mean Deviation	7.11875	MAD	4
Minimum	12	Mode	19	Second Moment	97.42438		
Maximum	62	Skewness	1.9401	Third Moment	1,865.63184	Alpha value (for confidence interval)	0.02

Figure 16: Parallel/Distributed-Specific Lines of Code: OpenMP/MPI

4.1.2.3.2 MASS Applications

As shown in Figure 17, applications built on MASS typically took around 247 lines of code.

Count	41	*Range*	569	*Skewness Standard Error*	0.36037	*Fourth Moment*	546,494,661.127
Mean	246.7561	*Sum*	10,117	*Kurtosis*	5.34074	*Median*	203
Mean LCL	196.77496	*Sum Standard Error*	845.64981	*Kurtosis Standard Error*	0.67105	*Median Error*	4.03715
Mean UCL	296.73723	*Total Sum Squares*	3,194,113	*Alternative Skewness (Fisher's)*	1.77277	*Percentile 25% (Q1)*	162
Variance	17,442.03902	*Adjusted Sum Squares*	697,681.56098	*Alternative Kurtosis (Fisher's)*	2.81542	*Percentile 75% (Q2)*	276.5
Standard Deviation	132.06831	*Geometric Mean*	221.75155	*Coefficient of Variation*	0.53522	*IQR*	114.5
Mean Standard Error	20.62561	*Harmonic Mean*	203.88522	*Mean Deviation*	96.05711	*MAD*	49
Minimum	117	*Mode*	#N/A	*Second Moment*	17,016.62344		
Maximum	686	*Skewness*	1.70724	*Third Moment*	3,789,699.15361	*Alpha value (for confidence interval)*	0.02

Figure 17: Total Lines of Code: MASS

Looking at Figure 18, we see that the lines of parallel/distributed-specific code for MASS applications were about 11.

Count	41	*Range*	27	*Skewness Standard Error*	0.36037	*Fourth Moment*	12,593.49782
Mean	10.82927	*Sum*	444	*Kurtosis*	5.5343	*Median*	9
Mean LCL	8.18296	*Sum Standard Error*	44.77388	*Kurtosis Standard Error*	0.67105	*Median Error*	0.21375
Mean UCL	13.47558	*Total Sum Squares*	6,764	*Alternative Skewness (Fisher's)*	1.96289	*Percentile 25% (Q1)*	7.25
Variance	48.89512	*Adjusted Sum Squares*	1,955.80488	*Alternative Kurtosis (Fisher's)*	3.03484	*Percentile 75% (Q2)*	11.75
Standard Deviation	6.9925	*Geometric Mean*	9.34042	*Coefficient of Variation*	0.6457	*IQR*	4.5
Mean Standard Error	1.09205	*Harmonic Mean*	8.27656	*Mean Deviation*	4.59607	*MAD*	2
Minimum	3	*Mode*	8	*Second Moment*	47.70256		
Maximum	30	*Skewness*	1.89034	*Third Moment*	622.80385	*Alpha value (for confidence interval)*	0.02

Figure 18: Parallel/Distributed-Specific Lines of Code: MASS

4.1.2.3.3 Summary of Lines of Code Difference

As with our time data, we are including a summary section here to help illustrate how these frameworks stack up against one another, in terms of the lines of code needed to write similar applications. We have created Figure 19 to illustrate this breakdown. There is a "Percent Difference" column added to this chart that shows the differences in LOC between frameworks. There is also a "Percent Difference" row added to this chart that shows the percentage of the overall code written that has to do with parallel/distributed-specific functionality (per framework). To aid in this final view (row), we've also added a "Ratio (Parallel-specific : Regular LOC)" row that shows how many standard lines of code will be written before parallel-specific code has to be put in place (on average for an application).

	OpenMP/MPI (Baseline)	MASS	Difference	Percent Difference
Total Lines of Code	190.88	246.76	55.88	29.28%
Parallel/Distributed-Specific Lines of Code	23.98	10.83	-13.15	-54.83%
Percent Difference	12.56%	4.39%	-8.17%	
Ratio (Parallel-specific : Regular LOC)	1 : 8.00	1 : 23.00		

Figure 19: Lines of Code Summary Between OpenMP/MPI and MASS

Looking at this data, we see two things of interest:

1. MASS requires around 29.28% more lines of code than comparable applications built using hybrid OpenMP/MPI
2. MASS applications require 54.83% less lines of parallel-specific code than hybrid OpenMP/MPI counterparts

Overall, this means that applications based on MASS will write 15 more lines of code than comparable hybrid OpenMP/MPI applications, prior to having to deal with parallel-specific code sections.

4.1.2.4 Developer Assessment

4.1.2.4.1 Learning Curve

We can see from Figure 20 that programmers developing applications based on a hybrid OpenMP/MPI approach, generally found that the learning curve was pretty fair - not hard, but not easy.

Count	44	Range	4	Skewness Standard Error	0.34926	Fourth Moment	2.89692
Mean	3.11364	Sum	137	Kurtosis	2.84089	Median	3
Mean LCL	2.74336	Sum Standard Error	6.74278	Kurtosis Standard Error	0.65348	Median Error	0.02895
Mean UCL	3.48391	Total Sum Squares	471	Alternative Skewness (Fisher's)	-0.09769	Percentile 25% (Q1)	3
Variance	1.0333	Adjusted Sum Squares	44.43182	Alternative Kurtosis (Fisher's)	-0.02897	Percentile 75% (Q2)	4
Standard Deviation	1.01651	Geometric Mean	2.91836	Coefficient of Variation	0.32647	IQR	1
Mean Standard Error	0.15325	Harmonic Mean	2.67206	Mean Deviation	0.74587	MAD	1
Minimum	1	Mode	3	Second Moment	1.00981		
Maximum	5	Skewness	-0.09433	Third Moment	-0.09572	Alpha value (for confidence interval)	0.02

Figure 20: Learning Curve: OpenMP/MPI

On the other hand, Figure 21 reflects that the same programmers, developing the same applications, found that learning the MASS library was generally hard - not ‘quite hard’, but definitely closer to hard than average. This could have been for a number of reasons: students were given more lectures and lab time that dealt with MPI and OpenMP, there are not online resources readily-available to answer questions (searching internet for answers/explanations will not help yield results), and there is an additional difficulty over from their previous general-parallel thinking to a new paradigm-oriented approach.

Count	44	Range	4	Skewness Standard Error	0.34926	Fourth Moment	3.48769
Mean	2.38636	Sum	105	Kurtosis	1.97763	Median	2
Mean LCL	1.96174	Sum Standard Error	7.73244	Kurtosis Standard Error	0.65348	Median Error	0.0332
Mean UCL	2.81099	Total Sum Squares	309	Alternative Skewness (Fisher's)	0.38042	Percentile 25% (Q1)	1
Variance	1.35888	Adjusted Sum Squares	58.43182	Alternative Kurtosis (Fisher's)	-0.99901	Percentile 75% (Q2)	3
Standard Deviation	1.16571	Geometric Mean	2.09684	Coefficient of Variation	0.48849	IQR	2
Mean Standard Error	0.17574	Harmonic Mean	1.82446	Mean Deviation	1.00207	MAD	1
Minimum	1	Mode	2	Second Moment	1.328		
Maximum	5	Skewness	0.36733	Third Moment	0.56215	Alpha value (for confidence interval)	0.02

Figure 21: Learning Curve: MASS

4.1.2.4.2 Application Suitability

Figure 22 displays the data gathered around asking how programmers found the hybrid OpenMP/MPI framework suited toward their application. The results show that programmers typically found that it was on the easy side of fairly suited toward their needs.

Count	43	Range	4	Skewness Standard Error	0.35285	Fourth Moment	2.1618
Mean	3.69767	Sum	159	Kurtosis	3.65503	Median	4
Mean LCL	3.37041	Sum Standard Error	5.81869	Kurtosis Standard Error	0.65919	Median Error	0.02586
Mean UCL	4.02494	Total Sum Squares	621	Alternative Skewness (Fisher's)	-0.63684	Percentile 25% (Q1)	3
Variance	0.78738	Adjusted Sum Squares	33.06977	Alternative Kurtosis (Fisher's)	0.89176	Percentile 75% (Q2)	4
Standard Deviation	0.88734	Geometric Mean	3.56501	Coefficient of Variation	0.23997	IQR	1
Mean Standard Error	0.13532	Harmonic Mean	3.37696	Mean Deviation	0.70525	MAD	1
Minimum	1	Mode	4	Second Moment	0.76906		
Maximum	5	Skewness	-0.6144	Third Moment	-0.41438	Alpha value (for confidence interval)	0.02

Figure 22: Application Suitability: OpenMP/MPI

If we look at Figure 23, we can see that programmers also found MASS to be on the easy side of fairly suitable for their application - though, slightly less suited than OpenMP/MPI.

Count	43	Range	4	Skewness Standard Error	0.35285	Fourth Moment	3.25113
Mean	3.60465	Sum	155	Kurtosis	2.04036	Median	4
Mean LCL	3.18538	Sum Standard Error	7.45462	Kurtosis Standard Error	0.65919	Median Error	0.03313
Mean UCL	4.02393	Total Sum Squares	613	Alternative Skewness (Fisher's)	-0.27136	Percentile 25% (Q1)	3
Variance	1.29236	Adjusted Sum Squares	54.27907	Alternative Kurtosis (Fisher's)	-0.92769	Percentile 75% (Q2)	5
Standard Deviation	1.13682	Geometric Mean	3.39807	Coefficient of Variation	0.31538	IQR	2
Mean Standard Error	0.17336	Harmonic Mean	3.15018	Mean Deviation	0.98107	MAD	1
Minimum	1	Mode	#N/A	Second Moment	1.2623		
Maximum	5	Skewness	-0.2618	Third Moment	-0.37129	Alpha value (for confidence interval)	0.02

Figure 23: Application Suitability: MASS

4.1.2.4.3 Difference Between Sequential and Parallel Programs

Programmers were asked to rate their experience moving their sequential algorithms into a parallel suitable equivalent for hybrid OpenMP/MPI and MASS applications. Figure 24 shows that for hybrid OpenMP/MPI applications, programmers found this task to be pretty fair, with a slight bent toward being hard.

Count	43	*Range*	4	*Skewness Standard Error*	0.35285	*Fourth Moment*	4.3899
Mean	2.90698	*Sum*	125	*Kurtosis*	1.88458	*Median*	3
Mean LCL	2.44595	*Sum Standard Error*	8.19698	*Kurtosis Standard Error*	0.65919	*Median Error*	0.03643
Mean UCL	3.368	*Total Sum Squares*	429	*Alternative Skewness (Fisher's)*	-0.20008	*Percentile 25% (Q1)*	2
Variance	1.56257	*Adjusted Sum Squares*	65.62791	*Alternative Kurtosis (Fisher's)*	-1.10323	*Percentile 75% (Q2)*	4
Standard Deviation	1.25003	*Geometric Mean*	2.58077	*Coefficient of Variation*	0.43001	*IQR*	2
Mean Standard Error	0.19063	*Harmonic Mean*	2.21269	*Mean Deviation*	1.04705	*MAD*	1
Minimum	1	*Mode*	4	*Second Moment*	1.52623		
Maximum	5	*Skewness*	-0.19303	*Third Moment*	-0.36397	*Alpha value (for confidence interval)*	0.02

Figure 24: Difference Between Sequential and Parallel Programs: OpenMP/MPI

Considering Figure 25, we see that programmers found this task to be hard, but leaning toward fair. The difference between the two frameworks being slight.

Count	42	Range	4	Skewness Standard Error	0.35656	Fourth Moment	5.95278
Mean	2.69048	Sum	113	Kurtosis	1.86798	Median	2.5
Mean LCL	2.18534	Sum Standard Error	8.76384	Kurtosis Standard Error	0.66505	Median Error	0.04035
Mean UCL	3.19561	Total Sum Squares	379	Alternative Skewness (Fisher's)	0.28859	Percentile 25% (Q1)	2
Variance	1.82869	Adjusted Sum Squares	74.97619	Alternative Kurtosis (Fisher's)	-1.12164	Percentile 75% (Q2)	4
Standard Deviation	1.35229	Geometric Mean	2.33136	Coefficient of Variation	0.50262	IQR	2
Mean Standard Error	0.20866	Harmonic Mean	1.98425	Mean Deviation	1.16667	MAD	1.5
Minimum	1	Mode	2	Second Moment	1.78515		
Maximum	5	Skewness	0.27817	Third Moment	0.66348	Alpha value (for confidence interval)	0.02

Figure 25: Difference Between Sequential and Parallel Programs: MASS

4.1.2.4.4 Debugging Difficulty

Figure 26 shows that programmers generally found it difficult (hard) to debug hybrid OpenMP/MPI applications.

Count	44	Range	3	Skewness Standard Error	0.34926	Fourth Moment	1.17661
Mean	2.43182	Sum	107	Kurtosis	2.40196	Median	2
Mean LCL	2.12355	Sum Standard Error	5.61352	Kurtosis Standard Error	0.65348	Median Error	0.02411
Mean UCL	2.74008	Total Sum Squares	291	Alternative Skewness (Fisher's)	-0.01757	Percentile 25% (Q1)	2
Variance	0.71617	Adjusted Sum Squares	30.79545	Alternative Kurtosis (Fisher's)	-0.52219	Percentile 75% (Q2)	3
Standard Deviation	0.84627	Geometric Mean	2.26664	Coefficient of Variation	0.348	IQR	1
Mean Standard Error	0.12758	Harmonic Mean	2.07874	Mean Deviation	0.72417	MAD	1
Minimum	1	Mode	#N/A	Second Moment	0.6999		
Maximum	4	Skewness	-0.01696	Third Moment	-0.00993	Alpha value (for confidence interval)	0.02

Figure 26: Debugging Difficulty: OpenMP/MPI

If we look at Figure 27, they also found it difficult (hard) to debug within the MASS framework. However, there is a slight tendency toward the average (fair) here, giving MASS a slight leg up in the comparison.

Count	44	Range	4	Skewness Standard Error	0.34926	Fourth Moment	6.1711
Mean	2.64773	Sum	116.5	Kurtosis	1.74307	Median	2
Mean LCL	2.14229	Sum Standard Error	9.20408	Kurtosis Standard Error	0.65348	Median Error	0.03952
Mean UCL	3.15317	Total Sum Squares	391.25	Alternative Skewness (Fisher's)	0.32869	Percentile 25% (Q1)	1.5
Variance	1.92534	Adjusted Sum Squares	82.78977	Alternative Kurtosis (Fisher's)	-1.26258	Percentile 75% (Q2)	4
Standard Deviation	1.38757	Geometric Mean	2.2729	Coefficient of Variation	0.52406	IQR	2.5
Mean Standard Error	0.20918	Harmonic Mean	1.92701	Mean Deviation	1.22934	MAD	1
Minimum	1	Mode	2	Second Moment	1.88159		
Maximum	5	Skewness	0.31738	Third Moment	0.81914	Alpha value (for confidence interval)	0.02

Figure 27: Debugging Difficulty: MASS

4.1.2.4.5 Summary of Developer Assessment

Through the administration of this survey, programmers were asked several questions that compared similar tasks in the development process between OpenMP/MPI and MASS applications. While we have presented very detailed results of each individual response to these questions (above), we find that it is much easier to get a clear overall picture of the differences in these frameworks by putting all of the data side-by-side.

Figure 28 provides this view into the data - giving a side-by-side comparison of responses across all four questions, averages across the four questions, and a difference summary to illustrate the degree to which one framework surpasses the other.

	OpenMP/MPI (Baseline)	MASS	Difference	Percent Difference
Learning Curve	3.11	2.39	-0.73	-23.36%
Application Suitability	3.70	3.60	-0.09	-2.52%
Difference Between Sequential and Parallel Programs	2.91	2.69	-0.22	-7.45%
Debugging Difficulty	2.43	2.65	0.22	8.88%
Average Rating	3.04	2.83	-0.21	-6.76%

Figure 28: Programmability Summary Between OpenMP/MPI and MASS

On average, it appears as if hybrid OpenMP/MPI applications have a slight advantage over MASS when considering common tasks in the development process. The difference in means here (-0.21) points to MASS being slightly more difficult to use. Translated into a percentage difference, we see that it is about 6.76% more difficult for programmers.

4.1.2.5 Comparison of Like Functionality

4.1.2.5.1 Call All

As shown in Figure 29, programmers tended to find the process of calling all (all places, agents, etc) in their applications slightly easier within MASS than using hybrid OpenMP/MPI.

Count	40	*Range*	4	*Skewness Standard Error*	0.36432	*Fourth Moment*	5.17852
Mean	3.5625	*Sum*	142.5	*Kurtosis*	2.25668	*Median*	4
Mean LCL	3.08441	*Sum Standard Error*	7.88336	*Kurtosis Standard Error*	0.67721	*Median Error*	0.03906
Mean UCL	4.04059	*Total Sum Squares*	568.25	*Alternative Skewness (Fisher's)*	-0.4529	*Percentile 25% (Q1)*	3
Variance	1.55369	*Adjusted Sum Squares*	60.59375	*Alternative Kurtosis (Fisher's)*	-0.67893	*Percentile 75% (Q2)*	5
Standard Deviation	1.24647	*Geometric Mean*	3.28408	*Coefficient of Variation*	0.34989	*IQR*	2
Mean Standard Error	0.19708	*Harmonic Mean*	2.91616	*Mean Deviation*	1.05938	*MAD*	1
Minimum	1	*Mode*	5	*Second Moment*	1.51484		
Maximum	5	*Skewness*	-0.43573	*Third Moment*	-0.8124	*Alpha value (for confidence interval)*	0.02

Figure 29: OpenMP/MPI vs MASS Comparison: Call All

4.1.2.5.2 Exchange All

Figure 30, shows that trying to exchange all (agents, data across places, etc) was slightly harder in MASS than in corresponding applications written on hybrid OpenMP/MPI.

Count	32	Range	4	Skewness Standard Error	0.4013	Fourth Moment	4.5625
Mean	3	Sum	96	Kurtosis	1.8688	Median	3
Mean LCL	2.44932	Sum Standard Error	7.18421	Kurtosis Standard Error	0.73273	Median Error	0.04974
Mean UCL	3.55068	Total Sum Squares	338	Alternative Skewness (Fisher's)	0E+00	Percentile 25% (Q1)	2
Variance	1.6129	Adjusted Sum Squares	50	Alternative Kurtosis (Fisher's)	-1.11634	Percentile 75% (Q2)	4
Standard Deviation	1.27	Geometric Mean	2.69666	Coefficient of Variation	0.42333	IQR	2
Mean Standard Error	0.22451	Harmonic Mean	2.36162	Mean Deviation	1.0625	MAD	1
Minimum	1	Mode	#N/A	Second Moment	1.5625		
Maximum	5	Skewness	0E+00	Third Moment	0E+00	Alpha value (for confidence interval)	0.02

Figure 30: OpenMP/MPI vs MASS Comparison: Exchange All

4.1.2.5.3 Manage All

In Figure 31 we see that it was slightly easier, using MASS, to manage all resources (agents, places, etc) within an application.

Count	11	*Range*	4	*Skewness Standard Error*	0.59761	*Fourth Moment*	6.09808
Mean	3.54545	*Sum*	39	*Kurtosis*	2.10392	*Median*	4
Mean LCL	2.40509	*Sum Standard Error*	4.53872	*Kurtosis Standard Error*	0.92118	*Median Error*	0.15592
Mean UCL	4.68582	*Total Sum Squares*	157	*Alternative Skewness (Fisher's)*	-0.68817	*Percentile 25% (Q1)*	2.75
Variance	1.87273	*Adjusted Sum Squares*	18.72727	*Alternative Kurtosis (Fisher's)*	-0.66013	*Percentile 75% (Q2)*	5
Standard Deviation	1.36848	*Geometric Mean*	3.2186	*Coefficient of Variation*	0.38598	*IQR*	2.25
Mean Standard Error	0.41261	*Harmonic Mean*	2.79661	*Mean Deviation*	1.12397	*MAD*	1
Minimum	1	*Mode*	4	*Second Moment*	1.70248		
Maximum	5	*Skewness*	-0.59053	*Third Moment*	-1.3118	*Alpha value (for confidence interval)*	0.02

Figure 31: OpenMP/MPI vs MASS Comparison: Manage All

4.1.2.5.4 Summary of Comparison of Like Functionality

The following table, Figure 32, provides a summary of the overall average ratings when comparing similar functions between hybrid OpenMP/MPI and MASS applications. This table also displays an average across all similar function comparisons.

	MASS Equivalent Rating	Difference	Percent Difference
Call All	3.56	0.56	28.13%
Exchange All	2.45	-0.55	-27.53%
Manage All	3.55	0.55	27.27%
Average	3.19	0.19	9.29%

Figure 32: Comparison Summary Between Like Functionality in OpenMP/MPI and MASS

Looking at the average, we can see that corresponding functions in MASS were fractionally easier to use than the corresponding functions in hybrid OpenMP/MPI applications. If we convert this value (0.19) to a percentage, we end up with MASS being a slight 9.29% easier to use.

4.1.2.6 Comparison Between Surveyed Classes

So far, we have been focusing on the combined results from both class surveys. However, in order to take a further look at how the differences in class make up (programming experience, interest in parallel/distributed computing) could have affected the results of our study, we compiled the data in Figure 33 (below).

Question 1: State your time (in hours) needed to complete your HW2 and HW4 respectively							
OpenMP/MPI Hours				MASS Hours			
To learn the library	To design the program	To write the program	To debug the program	To learn the library	To design the program	To write the program	To debug the program
2.35	2.15	-0.32	2.51	1.43	2.50	2.77	3.26

Question 2: State the code size (in lines) of your HW2 and HW4 respectively			
Homework 2 (Hybrid OpenMP/MPI)		Homework 4 (MASS)	
Total lines	Parallelization-specific code	Total lines	Parallelization-specific code
-58.15	-7.89	26.11	4.26

Question 3: State the programmability of HW2 and HW4: 1 quite hard, 2: hard, 3: fair, 4: good, 5: excellent							
Hybrid MPI/OpenMP version				MASS version			
Learning curve	Application Suitability	Difference Between Sequential and Parallel Programs	Debugging difficulty	Learning curve	Application Suitability	Difference Between Sequential and Parallel Programs	Debugging difficulty
0.48	-0.55	0.42	0.32	-0.17	-0.48	0.00	0.01

Question 4: State the degree of easiness of the following MASS functions when you wrote your program, as compared to MPI/ OpenMP functions: 1: quite hard, 2: hard, 3: fair, 4: easy, 5: quite easy, (blank): not used		
Existing MASS Functions		
Places/Agents callAll	Places exchangeAll	Agents manageAll
-0.62	0.40	-0.32

Figure 33: Difference Summary Between Spring 2014 & Winter 2015 Results

In this figure, we can see that students enrolled in the second course (Winter 2015):

1. Generally took more time to develop using both frameworks
2. Wrote less lines of code for their hybrid OpenMP/MPI applications, but more lines of code using MASS
3. Found the learning curve to be easier for OpenMP/MPI and harder for MASS
4. Found the application suitability to be harder for OpenMP/MPI and MASS
5. Found the difference between sequential and parallel programs to be easier for OpenMP/MPI, but the same for MASS
6. Found the debugging difficulty to be easier for OpenMP/MPI and also just slightly easier for MASS
7. Found the corresponding callAll functionality more difficult in MASS
8. Found the corresponding exchangeAll functionality easier using MASS
9. Found the corresponding manageAll functionality more difficult in MASS

So far, this data fits with what we would expect and shows that programmers with (assumed) more interest in parallel/distributed computing and (assumed) more programming experience generally found MASS easier to use, took less effort (lines of code) to create, and took less time to create than OpenMP/MPI.

However, it is also very important to consider that we only had 16 responses from the first quarter that we could use, which represents a small sample size. We are not able to really say, with a high degree of certainty that the reponses from the second class are significantly different from the results from the first. After all, it could turn out that with more students enrolled in the first class, and more survey results to add to our data set, that the initial results collected fall within the low end of a normal distribution. Of course, the opposite is true - and, the differences we can observe with this small sample size could be significantly greater if there were more data points collected from the first class.

It is possible to use further statistical analysis to see if the data between classes is significantly different. To do so, we will use a Student's t-test. Specifically, we will use a two-sample t-test, assuming equal variances (homoscedastic) in our data sets. The idea here being that we are comparing observations of like data between two classes. So, while the average (mean) may differ, if we collect enough data, the variance should begin to coalesce around a common value. If we were comparing different types of data or data that could strongly vary between two samples, then we'd want to use a heteroscedastic version of the t-test.

4.1.2.6.1 Sample Mean Comparison

Note: An exhaustive look at how each question performed under the t-Test can be found in Appendix F. If you are interested in taking an in-depth look into additional data around each response (t Critical Value (5%), Pooled Variance, Degrees Of Freedom, etc), please see Appendix F. In this section, we have taken an in-depth look into each

survey question and evaluated whether or not the difference between results between the Spring 2014 & Winter 2015 classes represent a statistically significant difference. The null hypothesis in each case is that both of the sampled means are identical, or:

$$H_0 = \mu \textit{ Spring 2014 Response} = \mu \textit{ Winter 2015 Response}$$

While we know that they all differ (from Figure 33; above), the question that remains is, "Given the smaller sample size of the first class, can we say that the difference is large enough to account for the lack of degrees of freedom (data points) in this group?"

Since the value of each survey question can either be higher or lower than the value from the other class, we will need to use the p-value of the two-tailed test to evaluate significance. A p-value less than 0.05 indicates that the results are significantly different between the two samples. We have highlighted this value, when encountered in Figure 34.

Question 1: State your time (in hours) needed to complete your HW2 and HW4 respectively							
OpenMP/MPI Hours				MASS Hours			
To learn the library	To design the program	To write the program	To debug the program	To learn the library	To design the program	To write the program	To debug the program
0.18188	0.14339	0.90431	0.33451	0.27755	0.17905	0.33123	0.09974

Question 2: State the code size (in lines) of your HW2 and HW4 respectively			
Homework 2 (Hybrid OpenMP/MPI)		Homework 4 (MASS)	
Total lines	Parallelization-specific code	Total lines	Parallelization-specific code
0.05872	0.02004	0.56241	0.06915

Question 3: State the programmability of HW2 and HW4: 1 quite hard, 2: hard, 3: fair, 4: good, 5: excellent							
Hybrid MPI/OpenMP version				MASS version			
Learning curve	Application Suitability	Difference Between Sequential and Parallel Programs	Debugging difficulty	Learning curve	Application Suitability	Difference Between Sequential and Parallel Programs	Debugging difficulty
0.14581	0.05385	0.31979	0.24869	0.66395	0.19776	0.9954	0.98753

Question 4: State the degree of easiness of the following MASS functions when you wrote your program, as compared to MPI/OpenMP functions: 1: quite hard, 2: hard, 3: fair, 4: easy, 5: quite easy, (blank): not used		
Existing MASS Functions		
Places/Agents callAll	Places exchangeAll	Agents manageAll
0.13655	0.39721	0.72857

Figure 34: Students t-Test of Results Between Spring 2014 & Winter 2015 Surveyed Questions (p-levels)

4.1.2.6.2 Class Difference Summary

We found that there were statistically significant differences between the Spring 2014 and Winter 2015 survey results for the following surveyed question (highlighted green in Figure 34):

1. OpenMP/MPI - Parallel-Specific Lines of Code
   The Spring 2014 course wrote approximately 8 more lines of parallel/distributed code in the hybrid OpenMP/MPI applications

We also found that there were a few survey results that were very nearly statistically different between the Spring 2014 and Winter 2015 classes (highlighted orange in Figure 34). These were:

1. OpenMP/MPI - Total Lines of Code
   The Spring 2014 course generally wrote around 58 more lines of code in their hybrid OpenMP/MPI applications than the Winter 2015 course

2. OpenMP/MPI - Application Suitability
   The Spring 2014 course generally found hybrid OpenMP/MPI applications to be 11% more suitable for their applications

Given the close nature of these values, we feel that further research would really help solidify the validity of some of these trends. We discuss this idea more during the conclusion of this paper, in Section 6.3.

All of this additional research into the data brings up a new question: "Now that we have established a statistically significant difference in one aspect between classes, how should we handle the interpretation of results?"

For the course of this paper, we are choosing to remain neutral between courses. This means that we will consider the entirety of surveyed results, without adjusting our findings in favor of one class over the other. The data does suggest that the first course found OpenMP/MPI more suitable for their applications and, interestingly enough, this resulted in them writing more overall lines of code and parallel-specific lines of code than the second class.

We have outlined suggestions to remove the future potential for this type of bias in Section 5.1.3. In this section, we also suggest ways to improve the quality of the data being surveyed in order to draw more clear correlations by collecting additional information about the individual filling out the survey. More information on these details can be found in the "Future Work" section of the paper, Section 6.

4.2 Performance

4.2.1 General MASS Performance

Before diving into comparisons between hybrid OpenMP/MPI and MASS application performance, we'd like to spend a bit of time just documenting the general performance characteristics of MASS itself.

4.2.1.1 Agents Performance

Section 3.2.1.1.2 details the particulars of the various tests that were run to get Agent performance within MASS. In this section, we will present the results of these tests varying both the *iterations* and *max_time* values for simulations - providing a view into "computationally heavy" Agent performance and "simulation time heavy" Agent performance, respectively.

It should be noted that tests that varied the value of *iterations* were performed using 256 Place Objects and a constant *max_time* value of 60. To see the actual results of these performance tests, please see Appendix G.

On the other hand, tests that varied the value of *max_time* were performed using 256 Place Objects and a constant *iterations* value of 10 (representing a "light" computational load for each callAll() being made). If you're interested in viewing the raw data collected from these sets of performance tests, please see Appendix H.

4.2.1.1.1 Test 1: callAll (null return value)

Figure 35 shows that the performance of an Agents callAll() function with varying degrees of computational load (*iterations*) produce a performance graph that matches with our expectations of parallel/distributed performance gains. To put this another way, as the number of hosts increase, the performance increases.

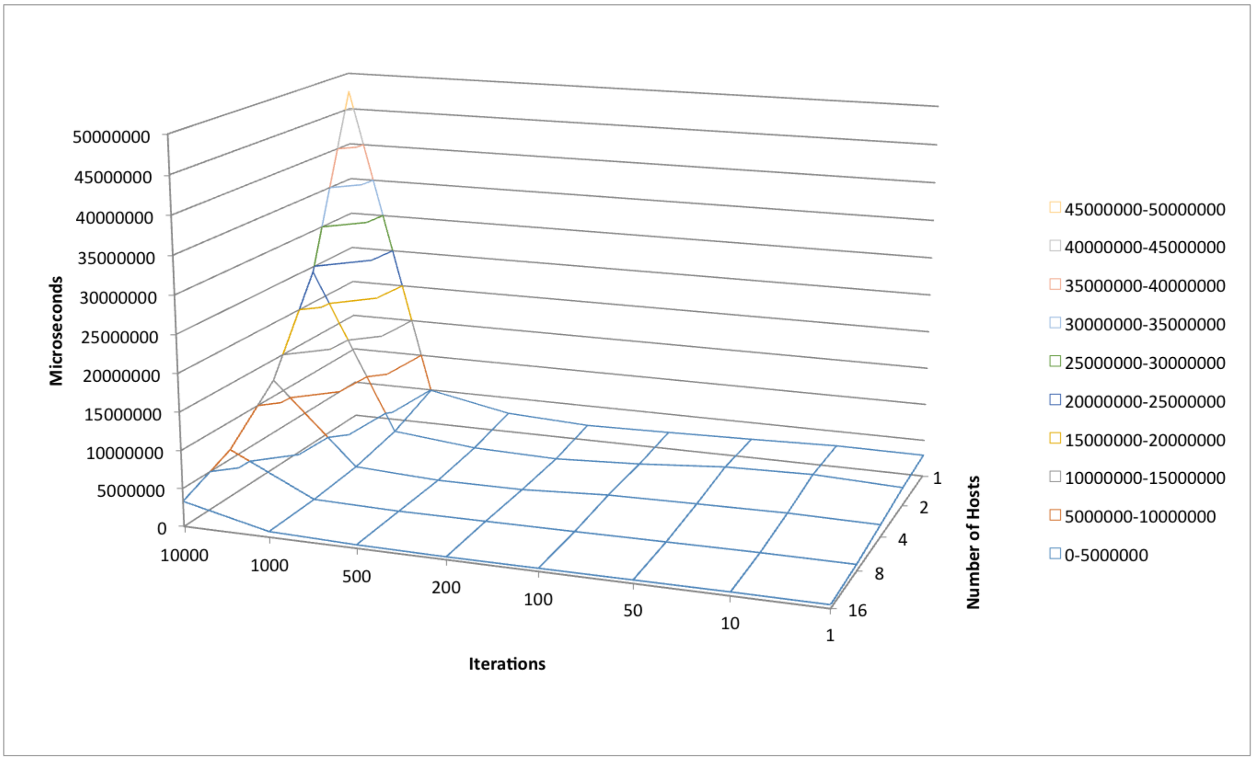

Figure 35: Agents: callAll (null return value) Performance Chart - Iterations

Looking at Figure 36, we see the same general trend as time of the simulation is increased. This indicates efficient use of resources as they become available to the simulation (good parallelization). There are a couple of anomalies present at 2 and 16 hosts, which correspond to using a poorly-performing node in our tests (uw1-320-09).

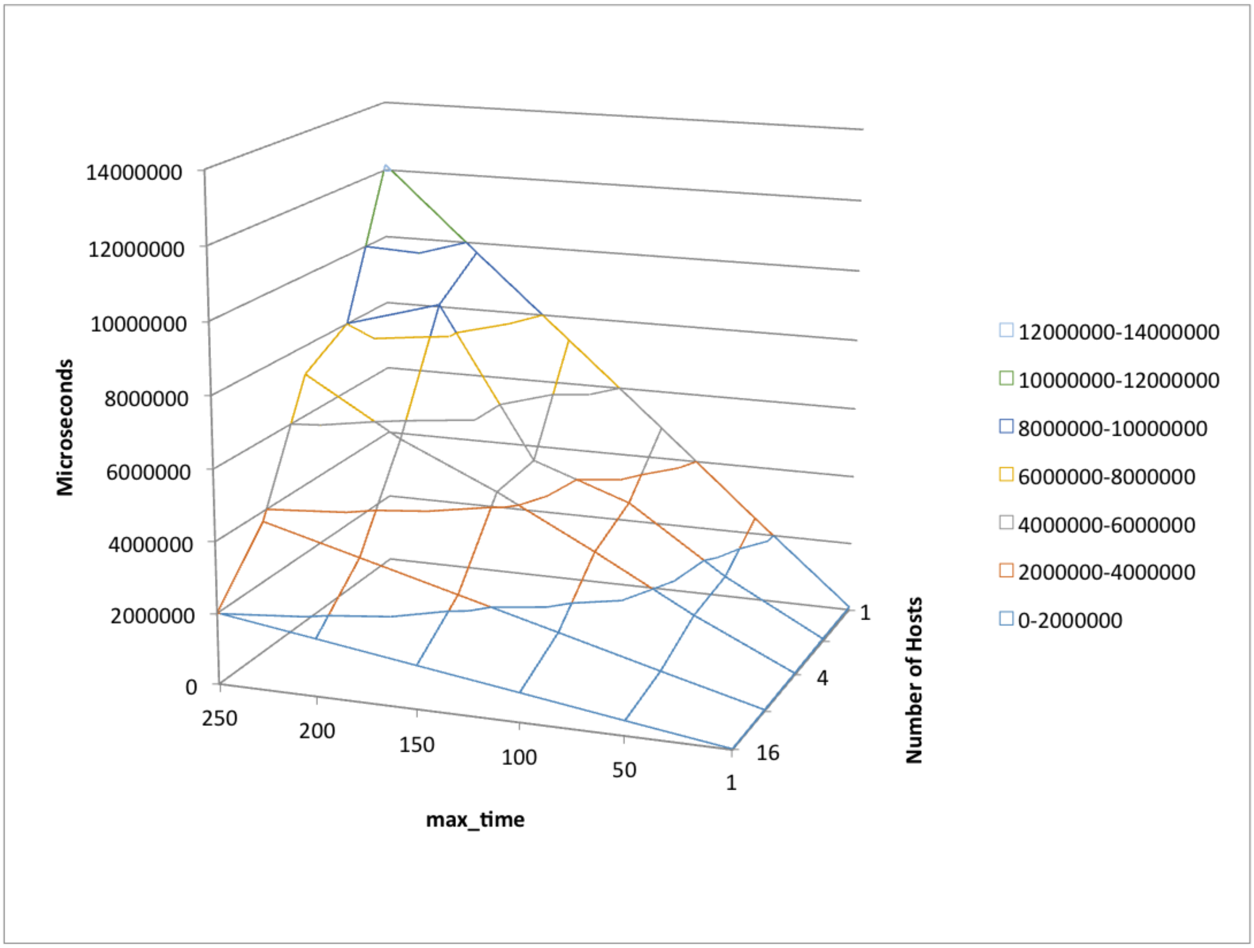

Figure 36: Agents: callAll (null return value) Performance Chart - Max Time

4.2.1.1.2 Test 2: Random Migration

Figure 37 shows a fairly constant performance, regardless of the varying degrees of computational load (*iterations*) being used. This is due to this test merely moving Agents from one Place to another - no computation is actually performed. So, since the number of Places and number of Agents are constant in this scenario, the difference in performance really comes down to the number of hosts involved in the test. As the number of hosts decreases, you can see the effect on migration calls, as Agents either move to Places on the same host or between hosts (cross-host migration allows computation of new Place location to benefit from parallel/distributed task breakdown). This difference will become more apparent in future tests that take away the "random" aspect of this migration.

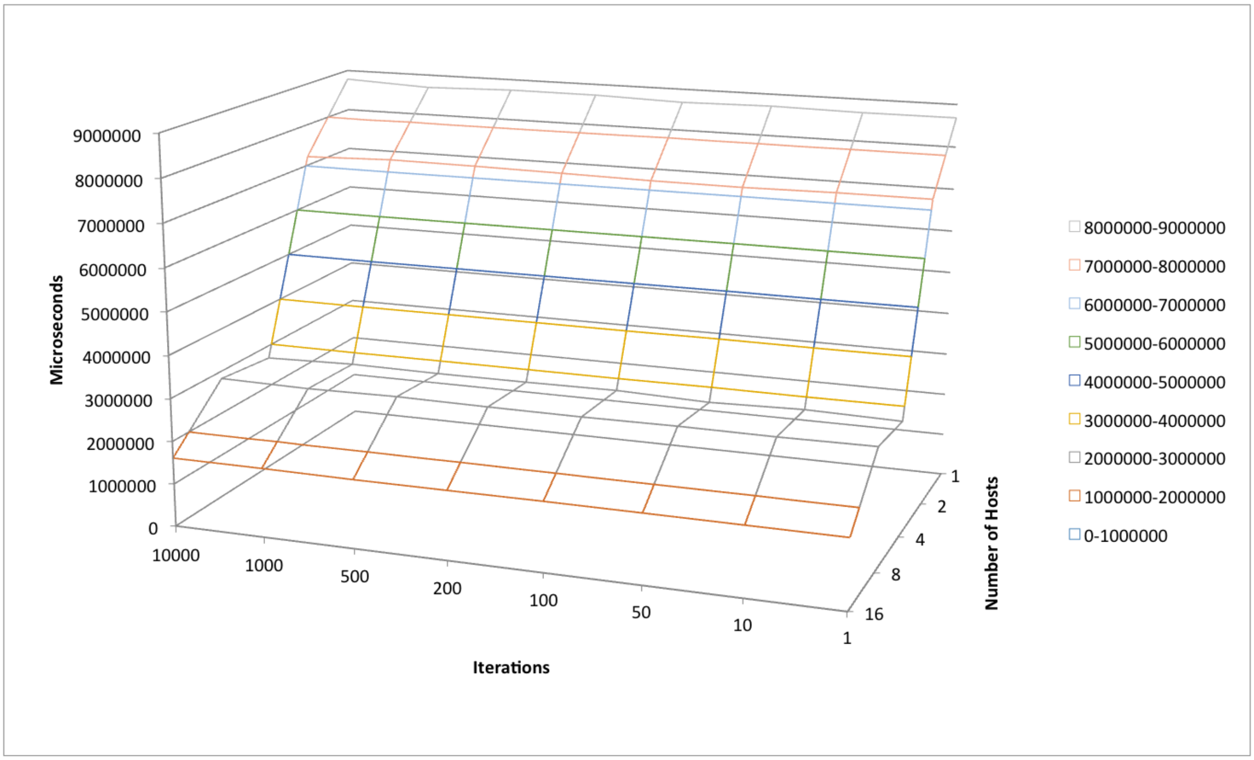

Figure 37: Agents: Random Migration Performance Chart - Iterations

If we vary the time of the simulation, instead of the computaitonal load, we will see (as in Figure 38) that the execution time increases accordingly. Once again, there are spikes at 2 and 16 hosts, due to the same poor-performing machine (uw1-320-09). However, if we ignore these lines, the general trend is a slight improvement of migration performance as the number of hosts are increased. This is a by-product of using a constant number of Places/Agents in our simulation. As the number of hosts increased, the actual number of Agents per host goes down - allowing each host to process migration requests faster.

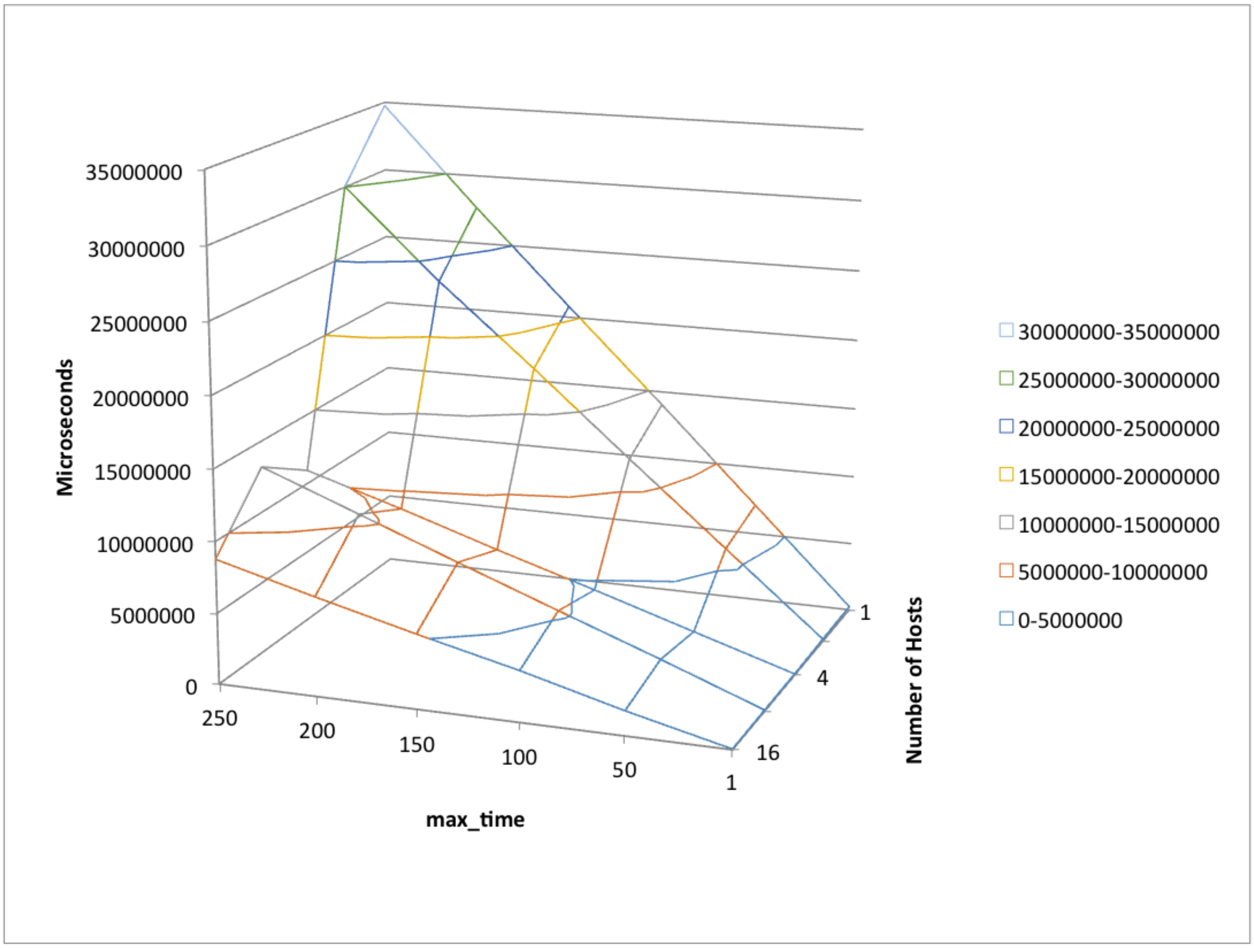

Figure 38: Agents: Random Migration Performance Chart - Max Time

4.2.1.1.3 Test 3: Full Migration

In Figure 39, we are presented with a view into performance that is completely unaffected by the number of iterations performed (computational load at each node). There is a noticable spike at two hosts, due to poor performance from uw1-320-09, but overall, the time taken for Agents to migrate to a new place drops according to the number of hosts involved in the simulation. This is a result of each node being able to distribute the work involved to reassign location for each Agent and the fact that the constant 256 Agents used in the simulation is spread more thinly across participating machines. It also makes sense that the value of *iterations* plays no part in the execution time, since no computation takes place during a migration.

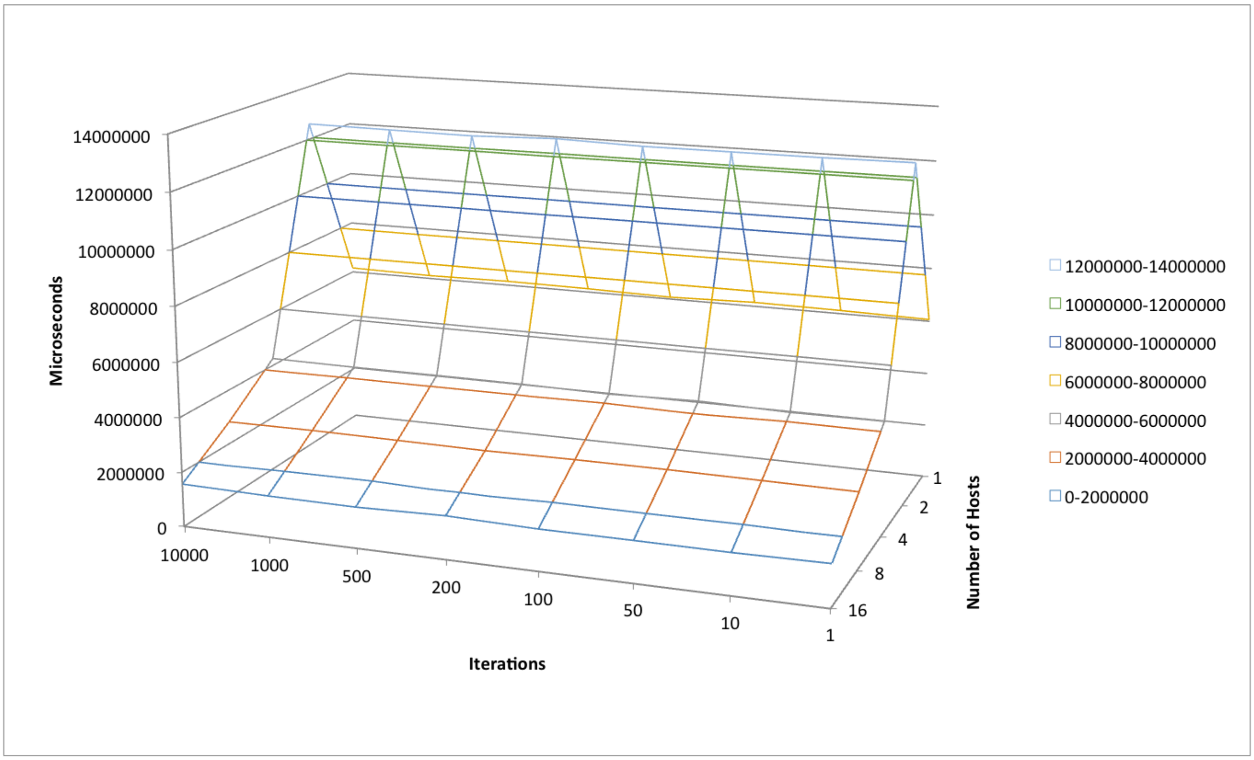

Figure 39: Agents: Full Migration Performance Chart - Iterations

In Figure 40 we are presented with a quite erratic view into performance as the simulation time is altered. Once again, if we are able to look past the poor performance at 2 and 16 hosts, we can observe an overall trend toward better performance with additional hosts. The effect of this test represents a “worst case” migration situation - as additional logic has been added to ensure that each Agent is not reassigned to its current location.

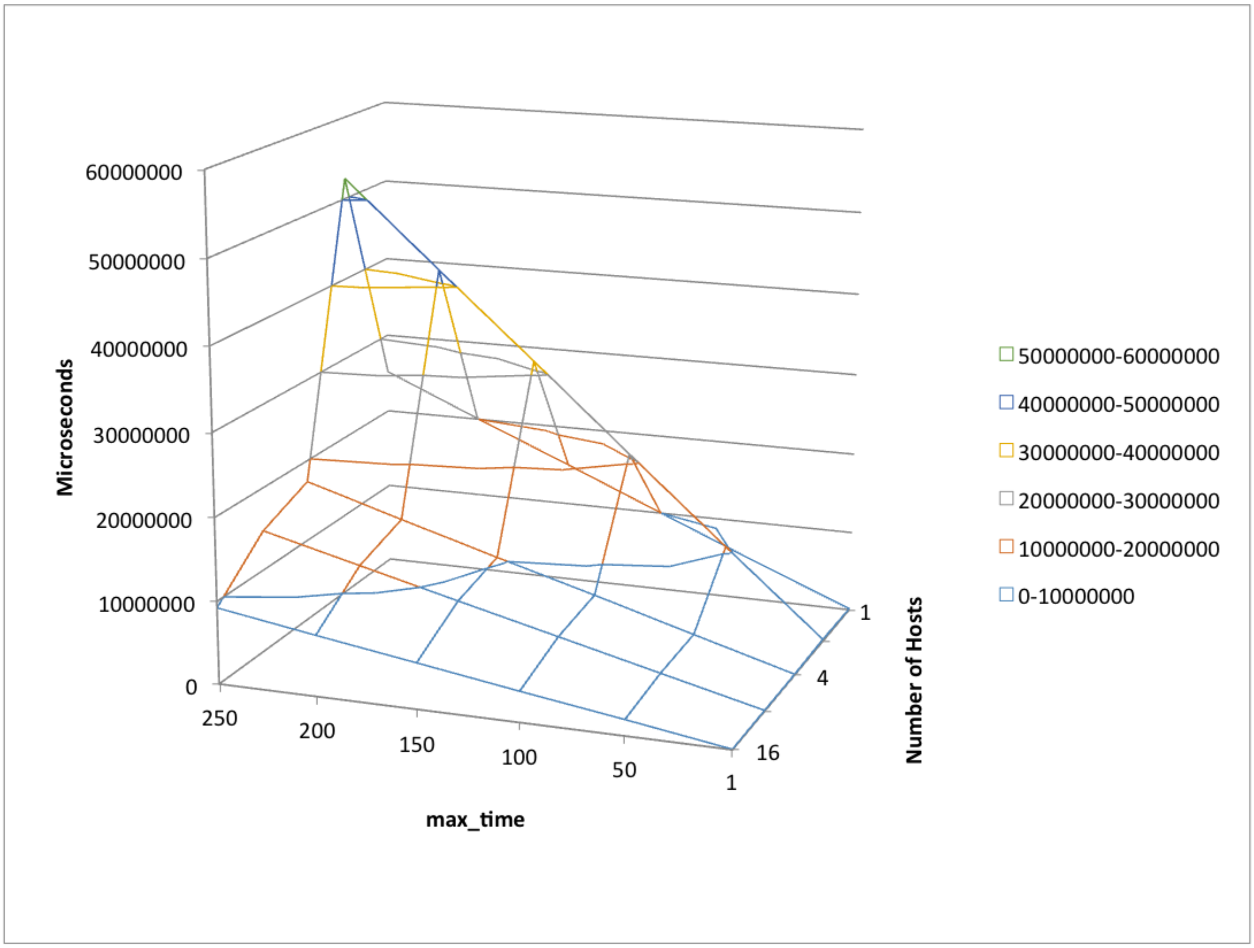

Figure 40: Agents: Full Migration Performance Chart - Max Time

4.2.1.1.4 Test 4: callAll (with return value)

Since this test involves computation taking place on each node, we see the familiar effect of distributing/parallelizing the work load in Figure 41. As the number of resources available to distribute work between increases, the execution time decreases.

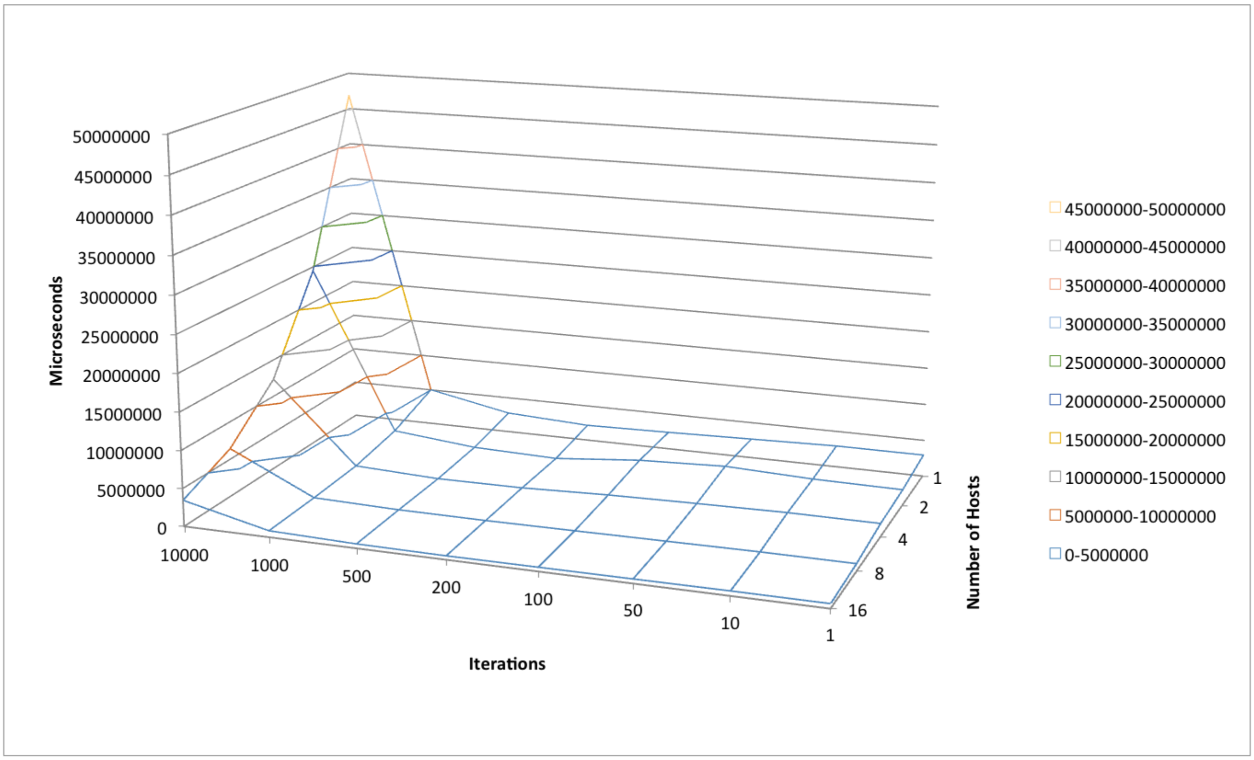

Figure 41: Agents: callAll (with return value) Performance Chart - Iterations

The familiar spikes at 2 and 16 hosts are also present in Figure 42 - which shows how MASS performs during a callAll operation, utilizing a return value. As expected, good parallelization continues to occur in this situation, showing that MASS is able to handle distributing and parallelizing work across machines and make efficient use of resources as they become available.

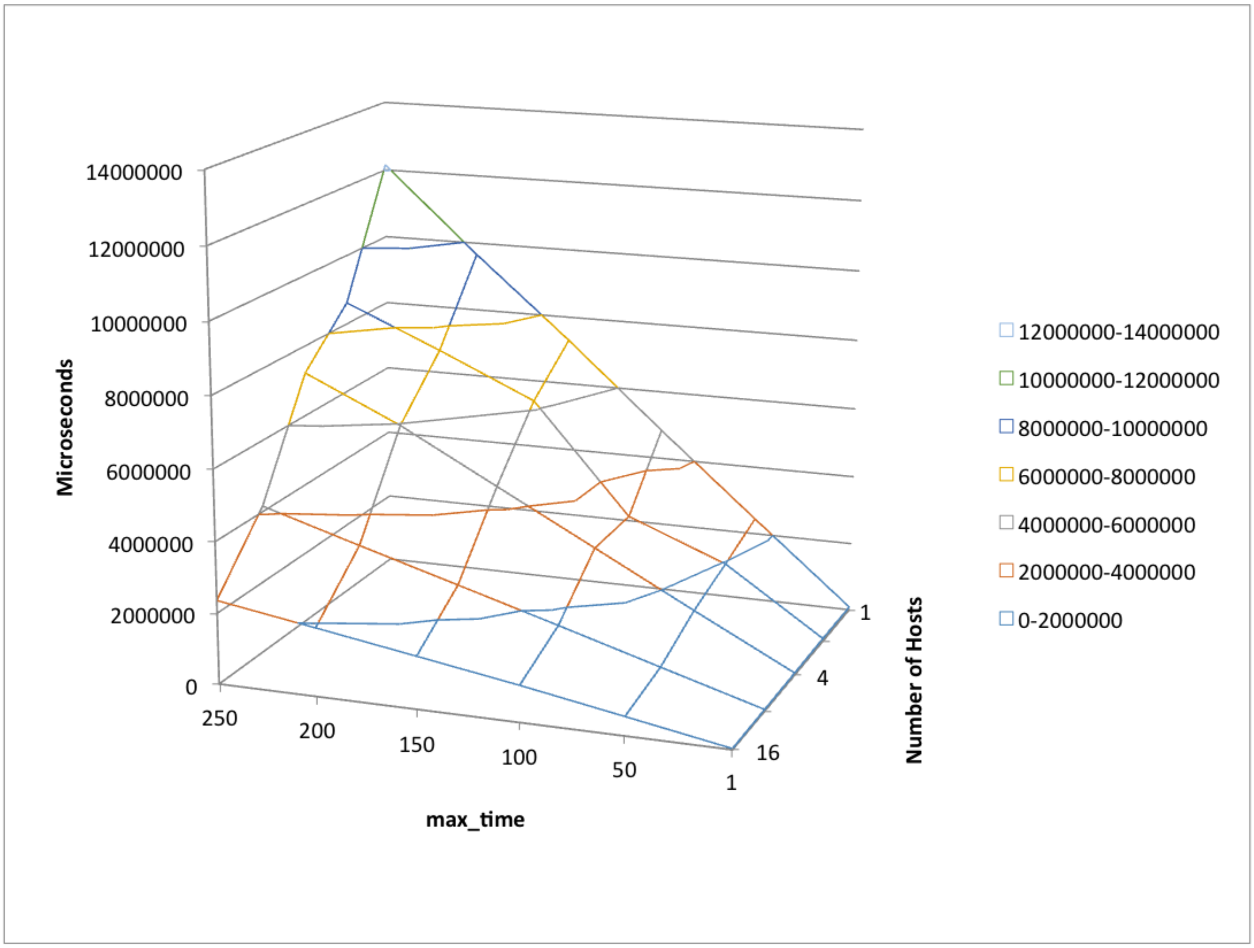

Figure 42: Agents: callAll (with return value) Performance Chart - Max Time

4.2.1.1.5 Test 5: Best Migrate (once)

As with "Test 3: Full Migration", Figure 39 shows a graph that is unaffected by the number of *iterations* performed (computational load). We also see the familiar spike at two nodes, that is most likely a result of our slow machine (uw1-320-09). You will notice that the scale of this test is about 25% smaller than the scale of the full migration test. This is due to the fact that the *max_time* attribute (normally set at 60 for these tests) is ignored and the migration only occurs once.

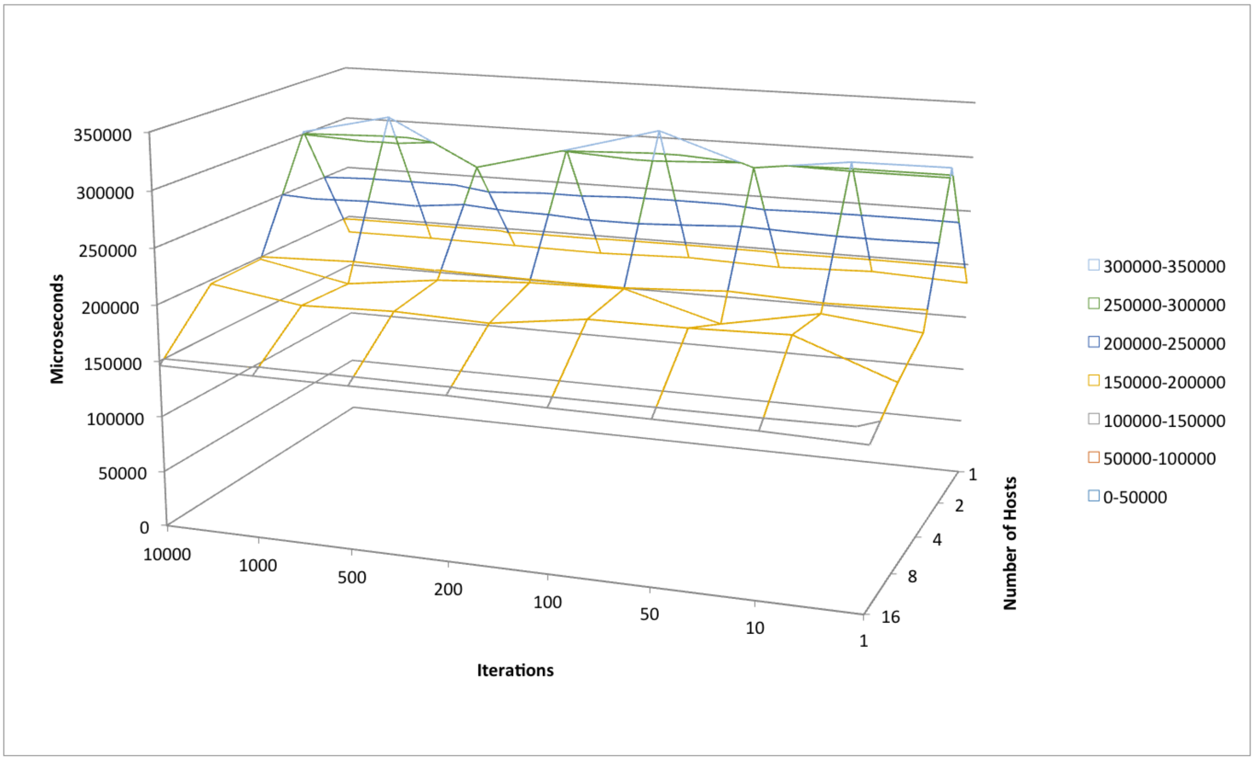

Figure 43: Agents: Best Migrate (once) Performance Chart - Iterations

In Figure 44 we see an uncharacteristic level response as resources are increased (with typical exceptions at 2 and 16 hosts; discussed previously). This is due to the *max_time* value being ignored for this test - so, each run is only performed once. We are also testing a "best case" scenario for the migration, which means that Agents do not actually move to a new location.

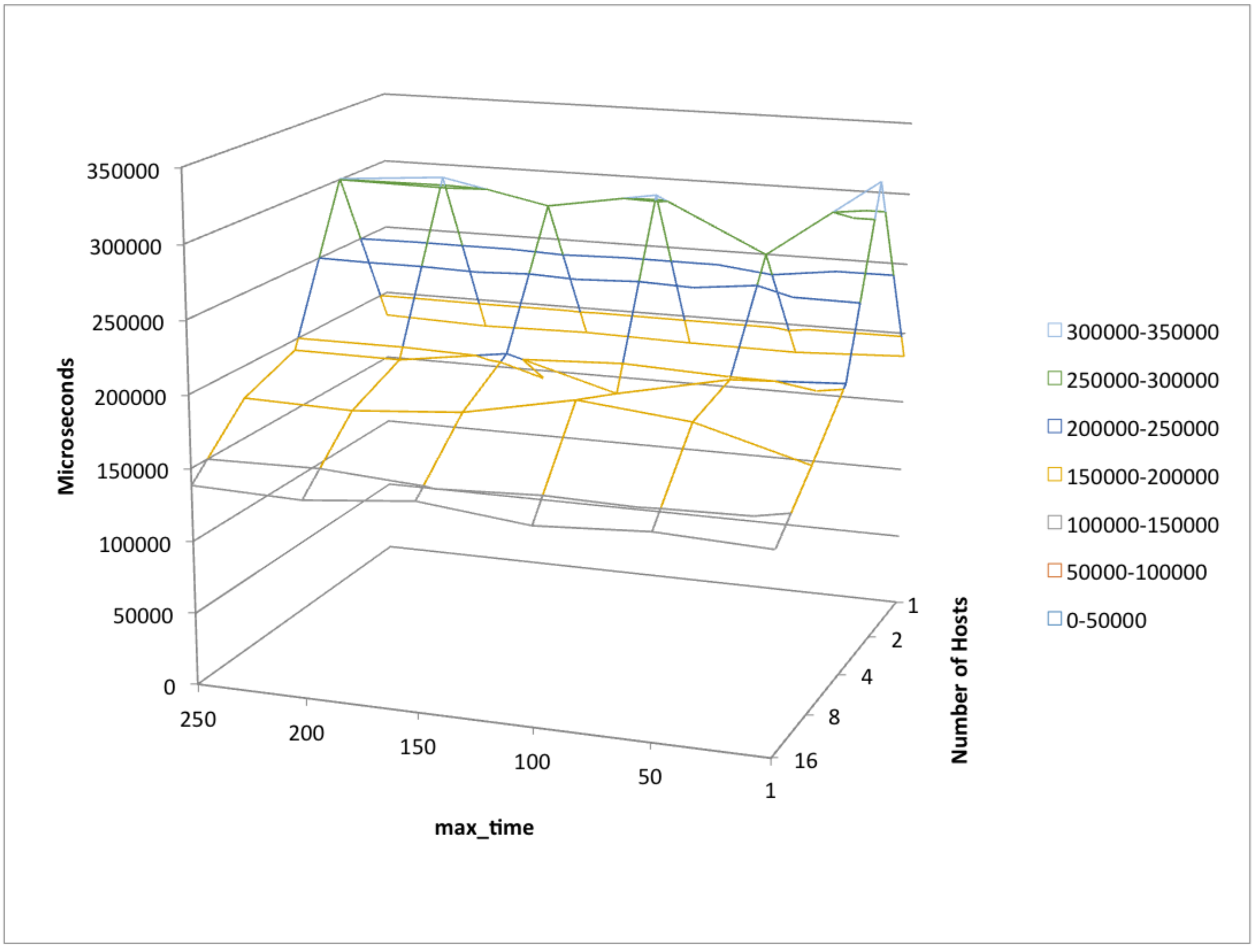

Figure 44: Agents: Best Migrate (once) Performance Chart - Max Time

4.2.1.1.6 Test 6: Random Migrate (once)

The data in Figure 45 shows us what a single run of the random migrate function looks like, in terms of execution time. Since no additional computational load is added (*iterations* value does not apply), we see a smooth decrease in effort as the number of hosts is increased. Once again, this is due to the number of actual Agents residing on each machine being spread out (spreading out the work required to computer and handle a migration). We also notice that the “bump” at two hosts is gone. Since this migration shoud technically take more time than the “best case” scenario (above), we can assume that the root cause is, indeed, an intermittently poor-performing node in our cluster (uw1-320-09) - which, appears to have decided to show up to work for this test.

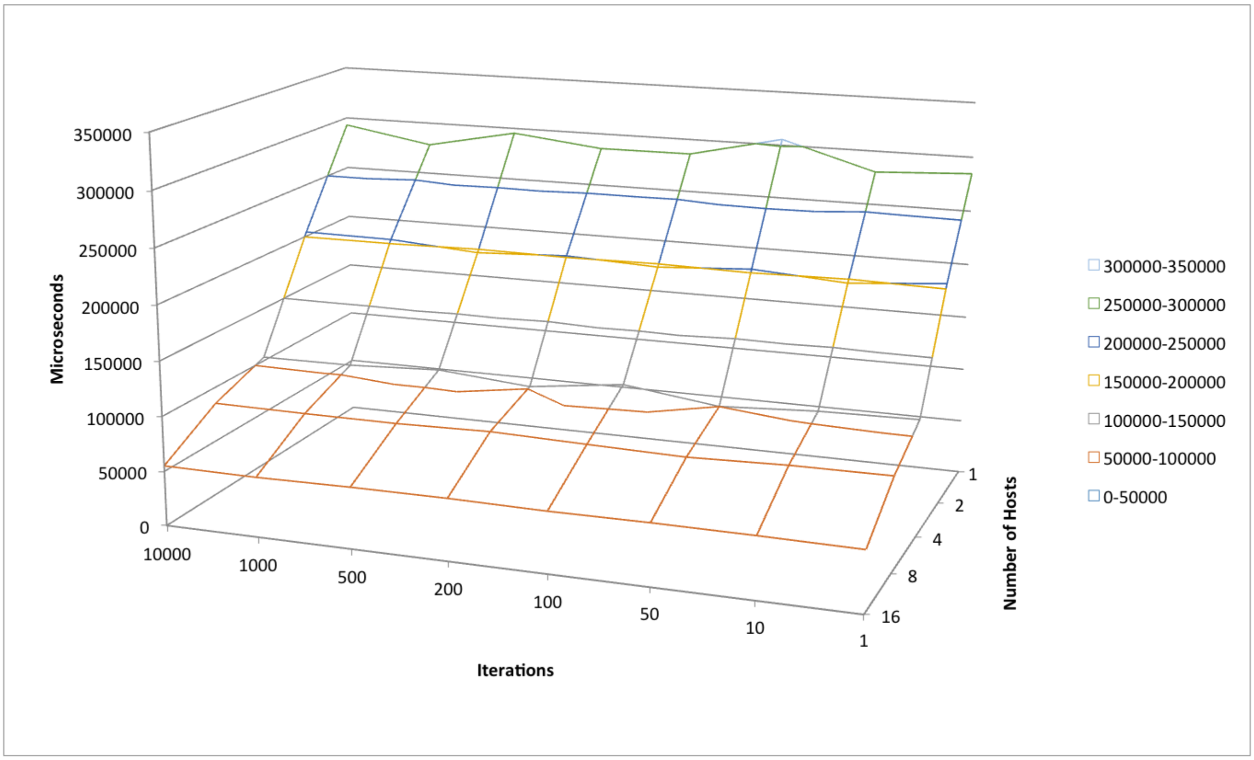

Figure 45: Agents: Random Migrate (once) Performance Chart - Iterations

In Figure 46 we can see a similar effect from varying the simulation size - this value is ignored! Once again, *max_time* is ignored and each test is only run once. This provides an individual look into how long a single random migration will take within MASS. We see familiar spikes at 2 and 16 hosts, but aside from these outliers, the overall trend is an growth in performance as hosts are also increased.

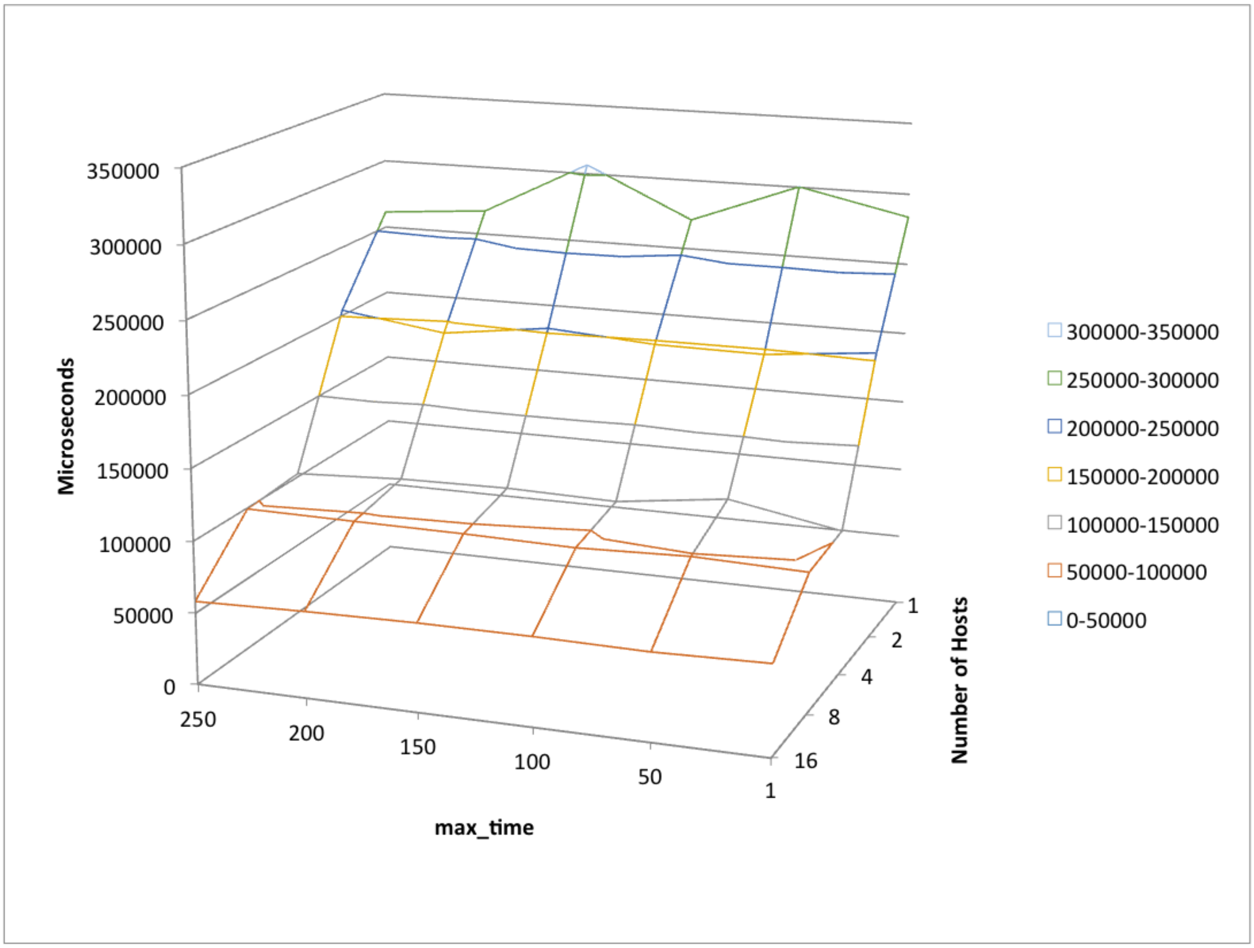

Figure 46: Agents: Random Migrate (once) Performance Chart - Max Time

4.2.1.1.7 Test 7: Worst Migrate (once)

The efforts of our faulty node (uw1-320-09) appear to have been short-lived, as we once again see a spike at two hosts in Figure 47. However, we also see a familiar trend toward better performance as the number of resources is increased (decreasing actual number of Agents per host, and subsequent calculations involved to migrate).

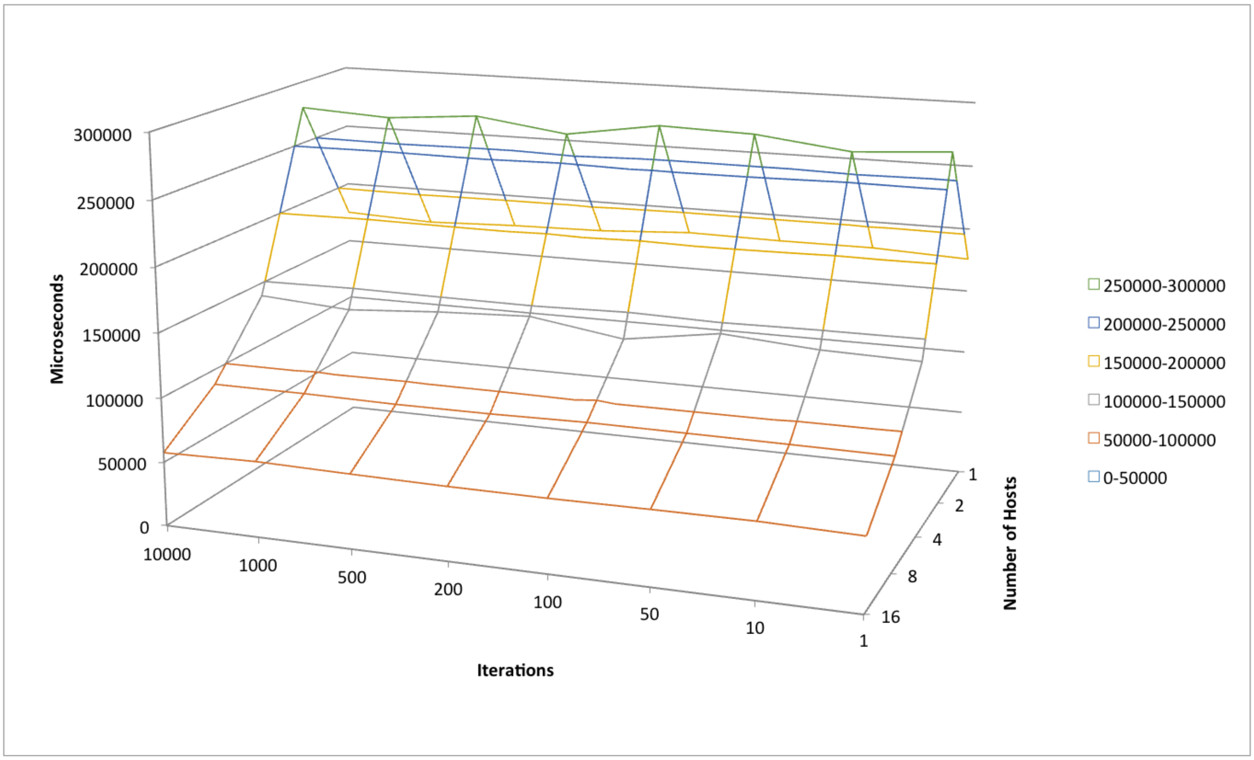

Figure 47: Agents: Worst Migrate (once) Performance Chart - Iterations

Figure 48 shows some expected spikes at 2 and 16 hosts - once again illustrating how one "bad apple" can ruin a bunch in a parallel simulation. However, ignoring these oddities, we once again see a pattern of good resource usage within MASS. The performance is relatively flat at each host as *max_time* is varied, which is due to this test just running once (ignores this value) - providing a baseline for a single execution of a worst migration scenario.

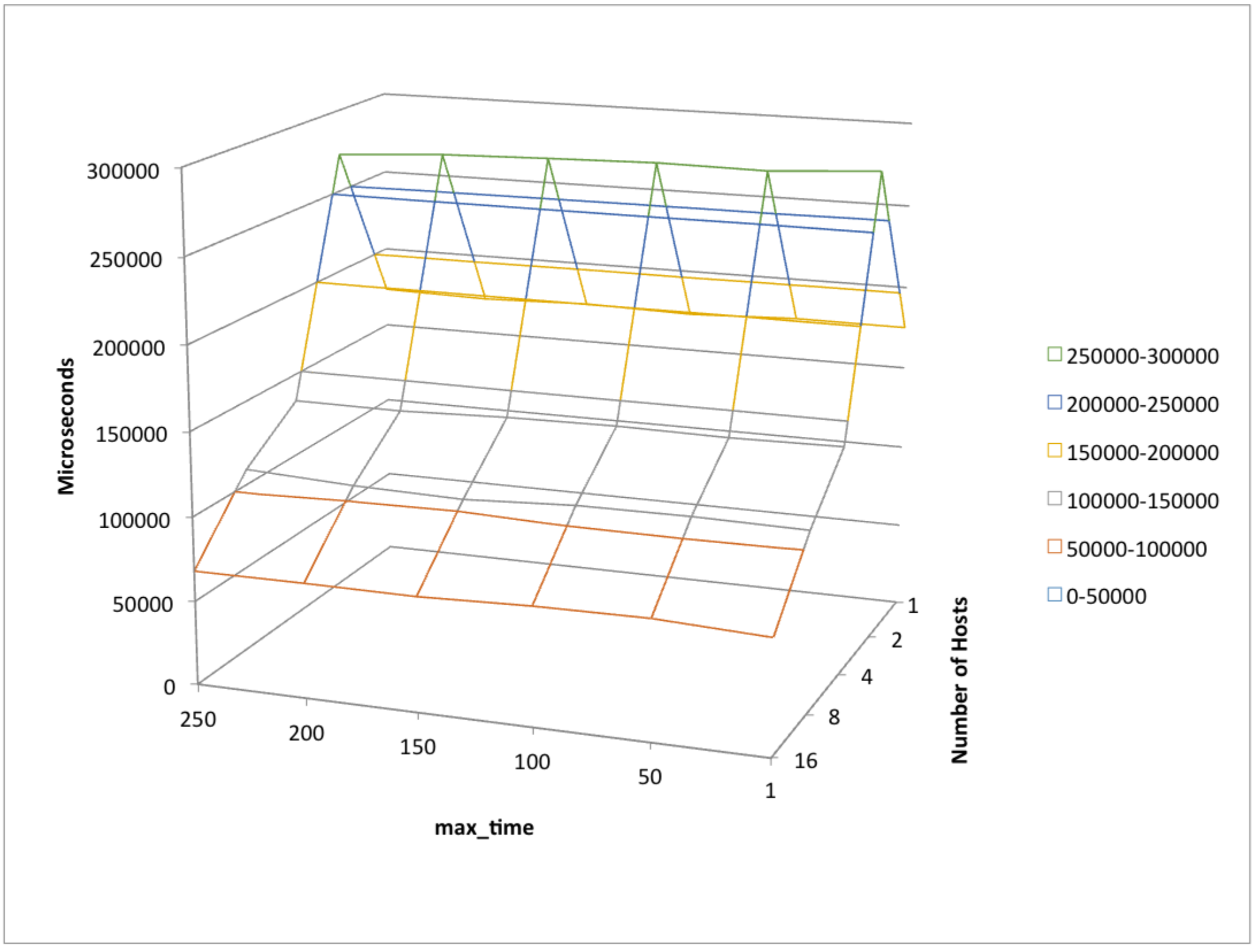

Figure 48: Agents: Worst Migrate (once) Performance Chart - Max Time

4.2.1.2 Places Performance

Section 3.2.1.1.1 details the particulars of the various tests that were run to get Places performance within MASS. In this section, we will present the results of these tests varying both the *iterations* and *max_time* values for simulations - providing a view into "computationally heavy" Place performance and "simulation time heavy" Place performance, respectively.

It should be noted that tests that varied the value of *iterations* were performed using 256 Place Objects and a constant *max_time* value of 60. If you would like to see the complete set of data collected from these sets of performance tests, please see Appendix I.

On the other hand, tests that varied the value of *max_time* were performed using 256 Place Objects and a constant *iterations* value of 10 (representing a "light" computational load for each callAll() being made). To see the complete set of data collected from these sets of performance tests, please visit Appendix J.

4.2.1.2.1 Test 1: callAll and exchangeAll

Figure 49 shows that the performance of a Places callAll() function followed by an exchangeAll() with varying degrees of computational load (*iterations*) produce a performance graph that matches with our expectations of parallel/distributed performance gains. To put this another way, as the number of hosts increase, the performance increases.

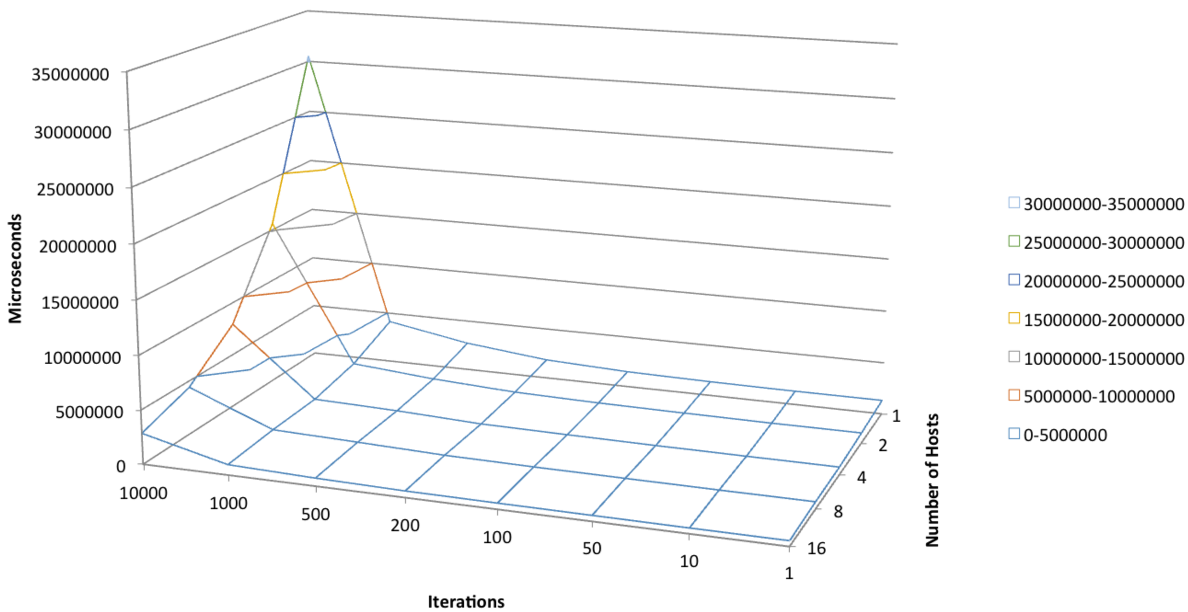

Figure 49: Places: callAll and exchangeAll Performance Chart - Iterations

If we consider Figure 50, we see a similar pattern but also a noticable anomaly at 16 hosts. This could be a relic of additional load from having to perform an exchangeAll() across more hosts or it could be the result of having a slow machine in the grid (or competition for resources, other applications running on a lab machine at the same time). One way to see if there is an underlying load from exchangeAll() that eventually surpasses the performance load of callAll() would be to consider the results of the individual callAll() test (see: Figure 54) and compare the difference in these graphs.

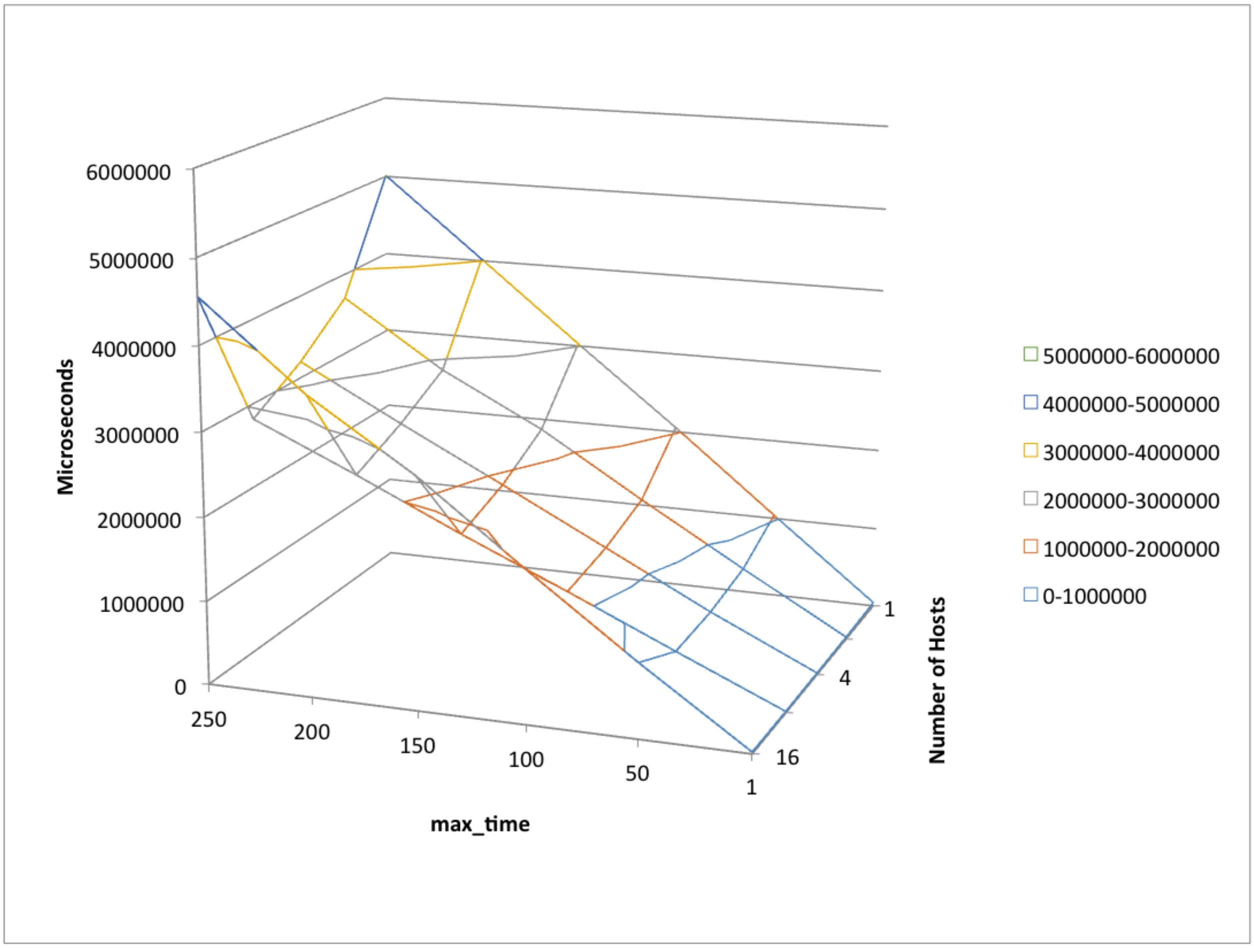

Figure 50: Places: callAll and exchangeAll Performance Chart - Max Time

4.2.1.2.2 Test 2: exchangeBoundary, callAll, and store output

We can see in Figure 51 that the performance when varying *iterations* once again matches our ideal projection for efficient distribution and parallelization of the work involved.

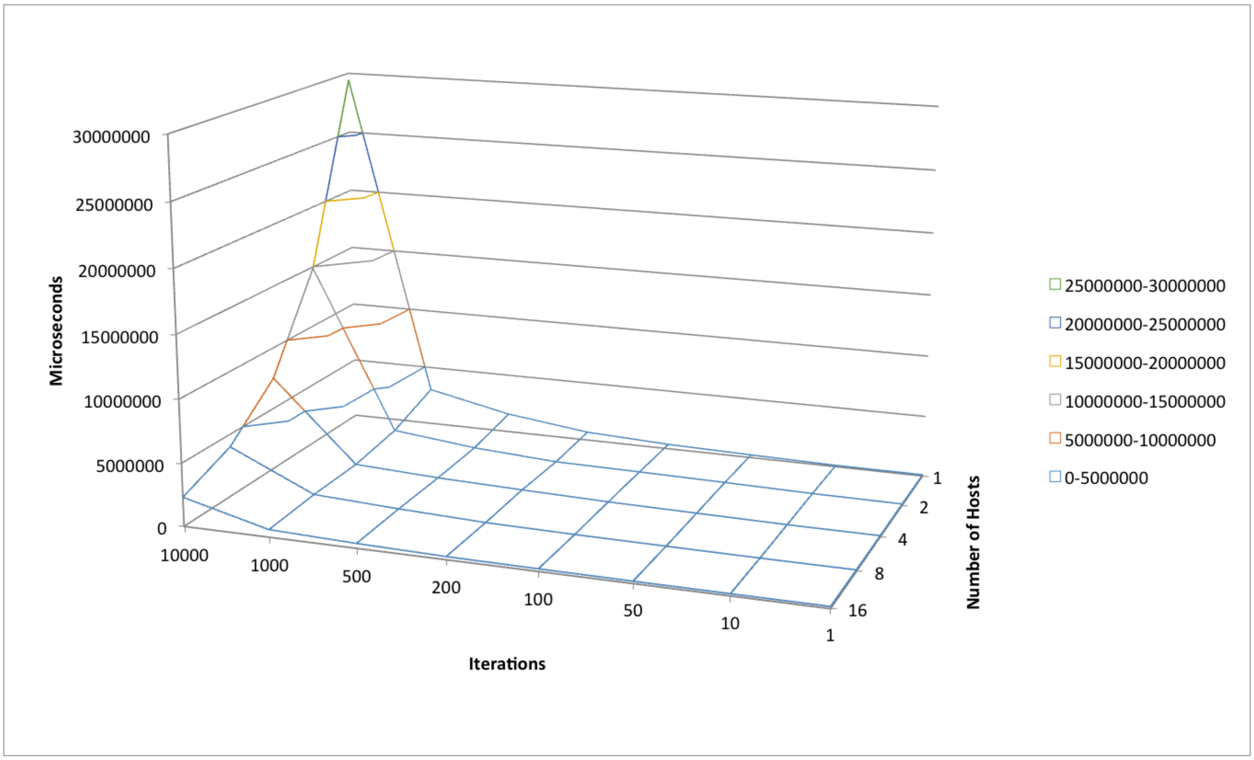

Figure 51: Places: exchangeBoundary, callAll, and store Performance Chart - Iterations

However, Figure 52 once again charts how we have an underlying negative performance profile when we are exchanging data between Places - a negative impact that ends up dominating performance when more hosts are involved. This is not suprising news, or is it an aspect that is unique to MASS in the realm of parallel/distributed computation. It is a common problem when you have a small amount of work and a lot of communication between nodes - as we are modeling here.

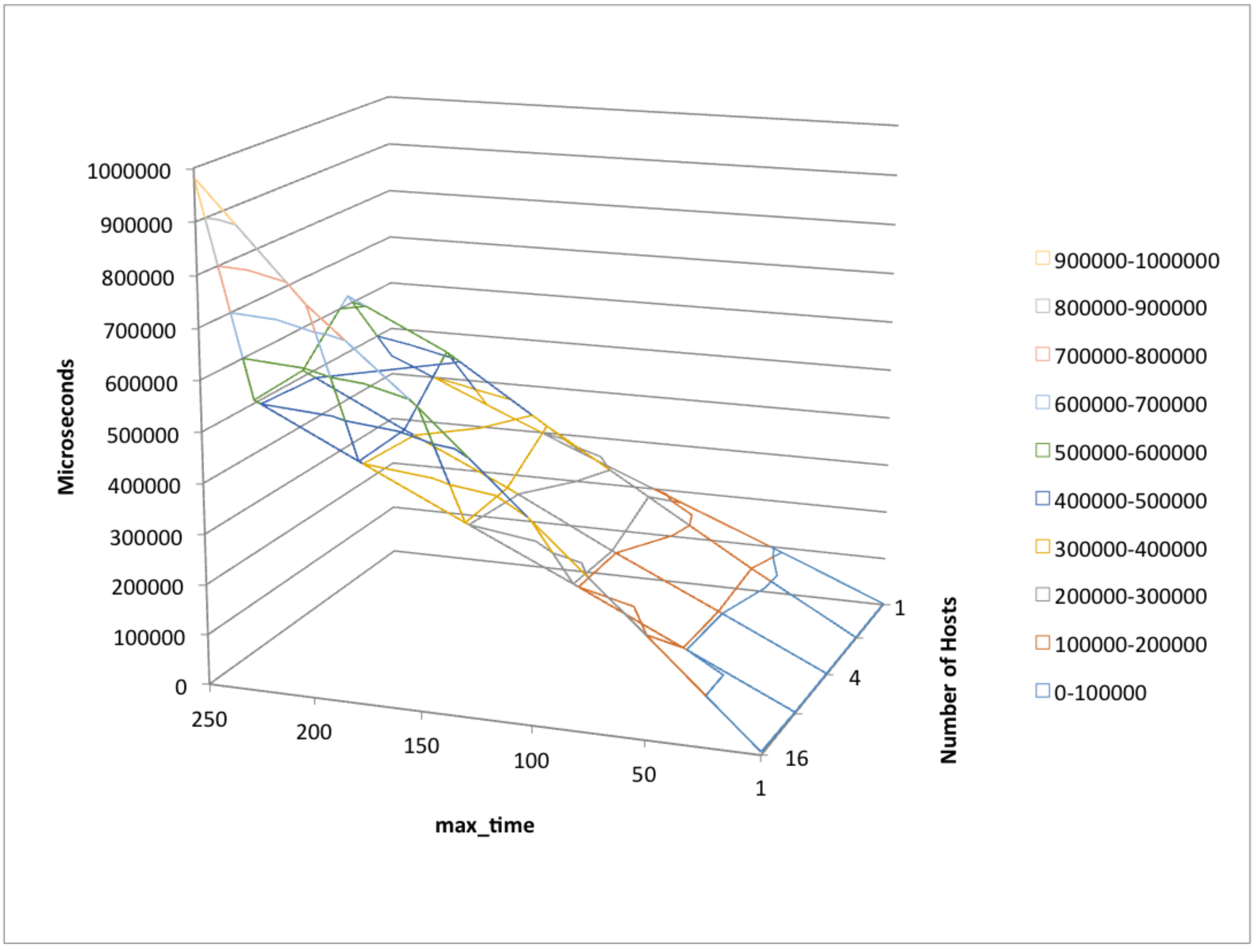

Figure 52: Places: exchangeBoundary, callAll, and store Performance Chart - Max Time

4.2.1.2.3 Test 3: callAll

Figure 53 is again displaying our very familiar results from parallelizing a heavy computational workload across the system. As more resources are allocated, the performance continues to improve.

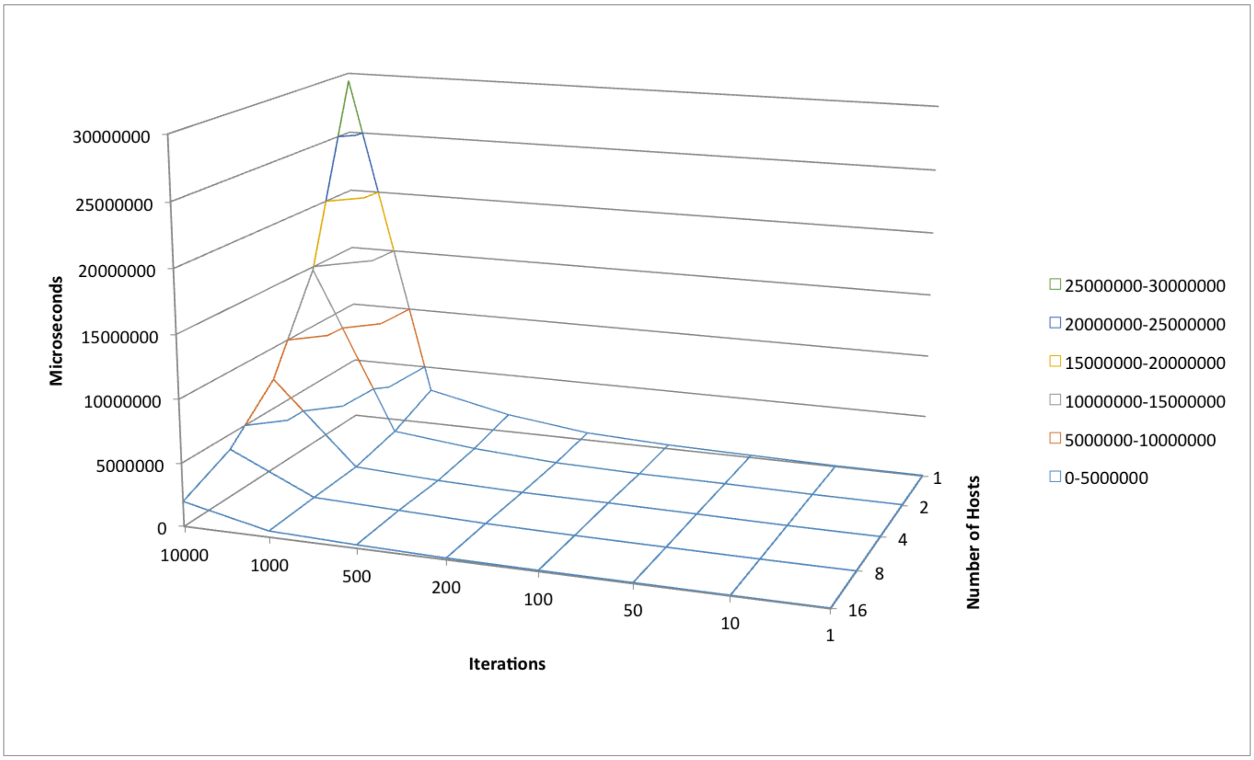

Figure 53: Places: callAll Performance Chart - Iterations

Figure 54 shows performance spikes at 2 and 16 hosts when performing a simple callAll() across hosts. However, the overall trend (ignoring these data points) shows a decline in execution time with more resources. It is slight, but it is there. This seems to indicate that an pre-existing problem we had experienced earlier with one of the lab machines underperforming (uw1-320-09) was not completely addressed. This could also account for the spike at 16 hosts noticed during our earlier test, seen in Figure 50 (above).

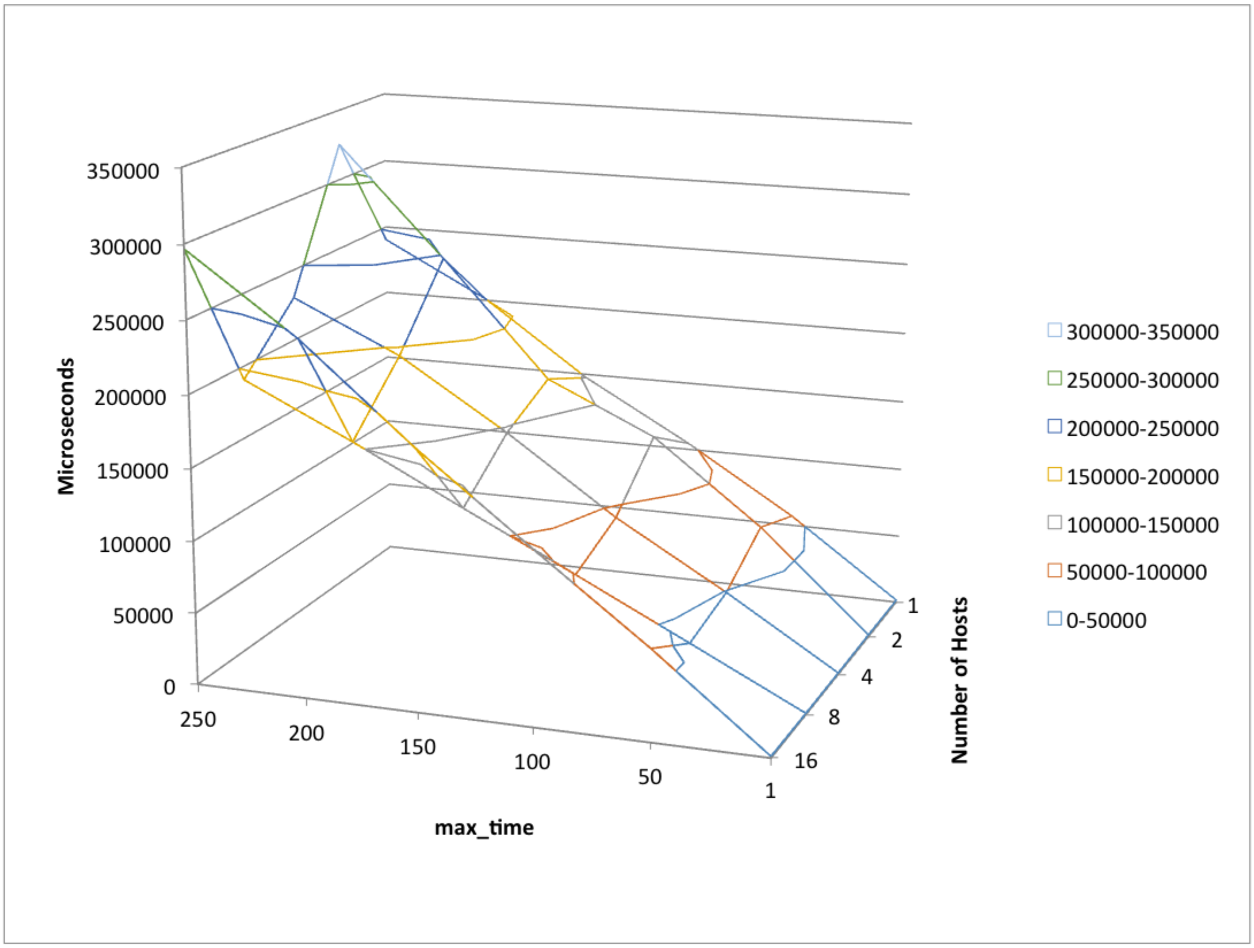

Figure 54: Places: callAll Performance Chart - Max Time

4.2.1.2.4 Test 4: callAll with periodic return value

Figure 55 wraps up our evaluation of how well Places within MASS are able to handle large computational effort - benefiting through increased performance and parallelization also increases.

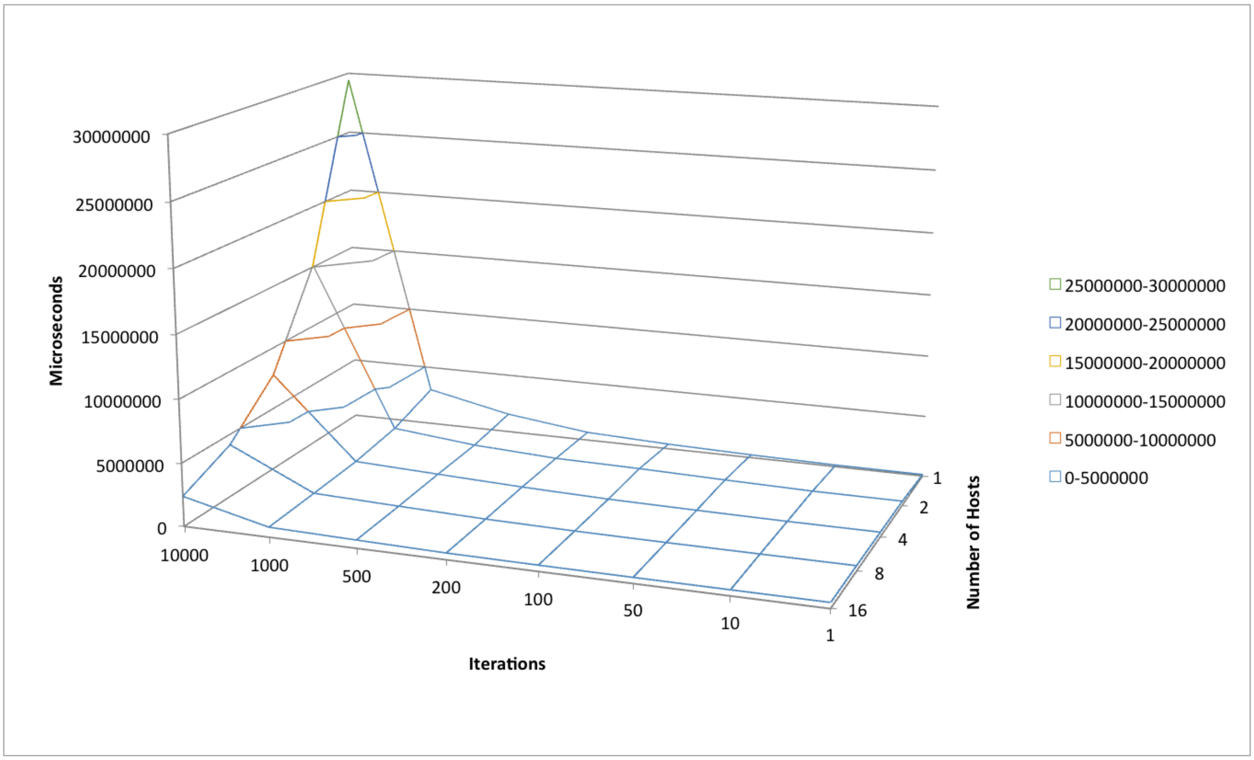

Figure 55: Places: callAll with periodic return Performance Chart - Iterations

Figure 56 shows a familiar spike at 2 and 16 hosts that could be related to a slow lab machine present in these configurations. However, unlike other test results, we can actually see an overall growth trend in intermediary results that seem to indicate that there is something else going on that is influencing performance and reducing the benefit of parallelization. The culprit? In this case, it is the "periodic return value" that is being printed on every interval of *max_time*. This basically means that as the hosts grow, the communication needed to obtain/print values is also growing slightly. However, the major cost to this test run appears to simply be dominated by the number of times values are returned (*max_time* value).

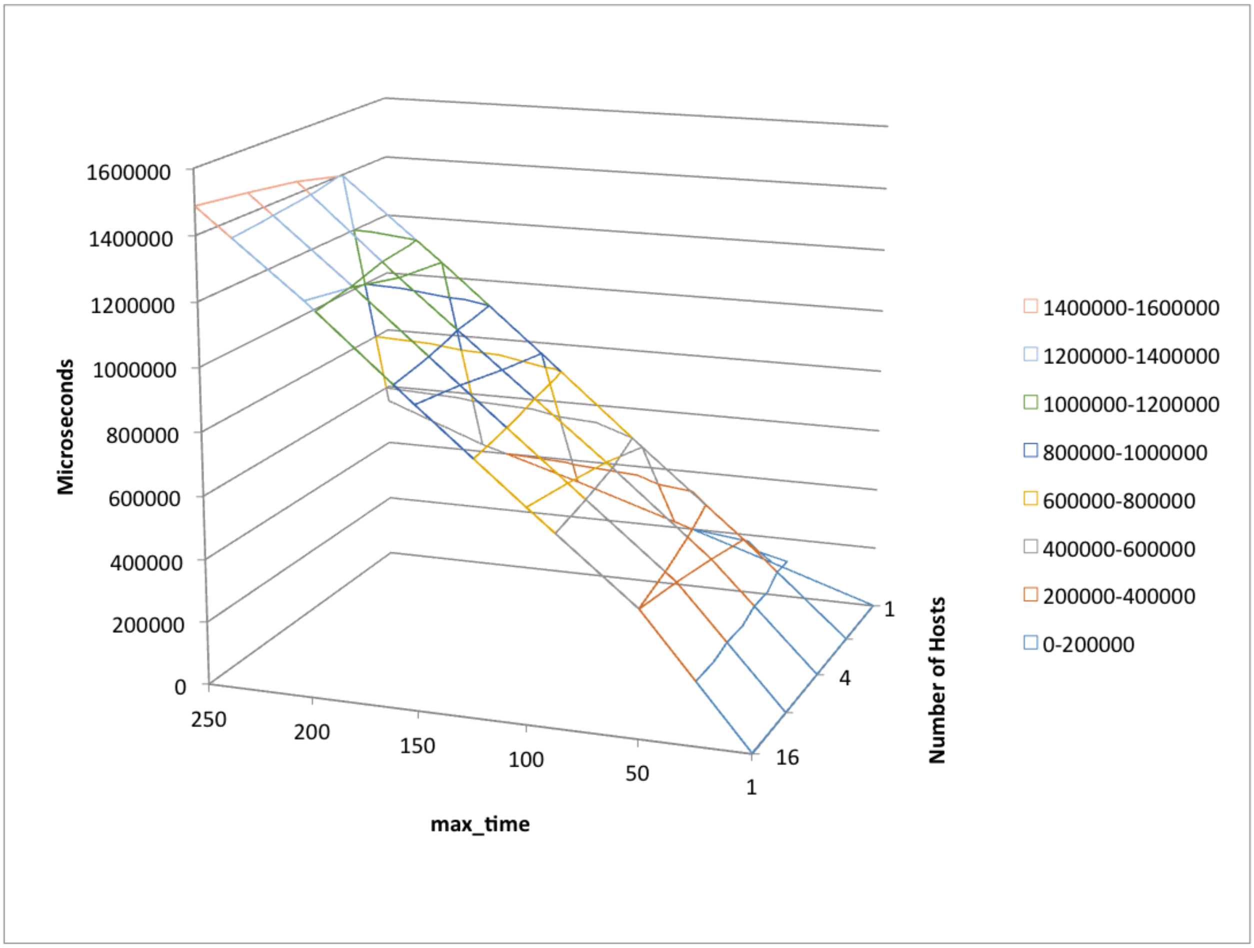

Figure 56: Places: callAll with periodic return Performance Chart - Max Time

4.2.1.3 General Performance Summary

In this section, we aggregate data collected/presented from previous tests and combine it to provide an "overall" look into the performance of basic functionality within MASS. We will begin by examining aggregated Agent data, before looking at Places. Then, we will wrap up with a side-by-side comparison showing Agents and Places averages against an average of averages - representing an overall picture of general performance within MASS.

4.2.1.3.1 Agent Summary

In Figure 57, we are presented with the actual average times that the collection of Agent test types took to complete, using different host configurations. However, to get a better idea of how this data actual looks and what sort of performance trends we can expect from Agents within MASS, we must look to Figure 58.

Agents Performance (µs)

Number of Hosts	Iterations	Max Time	Iterations & Max Time
	Overall Average	Overall Average	Combined Average
1	6123395.8	4638616.546	5381006.173
2	7283284.981	4330652.895	5806968.938
4	3137263.883	1839810.613	2488537.248
8	2630107.226	1294248.782	1962178.004
16	1645702.102	742365.3964	1194033.7492

Figure 57: Agent Performance Summary Table

Figure 58 provides a side-by-side line chart that shows how varying *iterations* affected performance, how varying *max_time* affected performance, and also how the average of these two variables ends up painting a picture for effective parallelization of Agents within MASS.

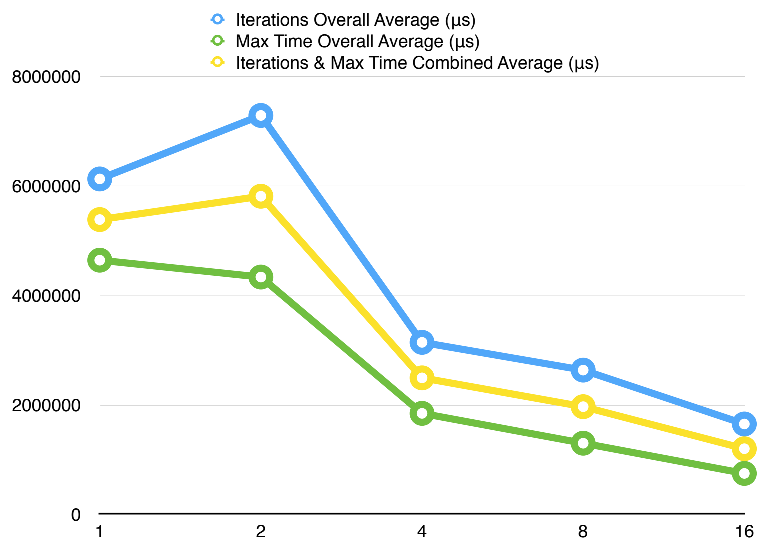

Figure 58: Agent Performance Summary Chart

As you can see, the effect of running these tests multiple times has a greater impact on overall execution time than varying the computational load. This is due to a predominance of "migration-oriented" tests within the Agents test plan. So, what you're really seeing is that *iterations* has no effect on the performance of a migration, whereas, repeating this migration a number of times (*max_time*) ends up having a substantial effect on overall performance.

It should be noted that there are tests that contain callAll functions that are impacted by the value of *iterations*. However, the main takeaway from this data is a cautionary tale in migration management. To increase the general performance characteristics of an application developed in MASS, an "unravelling" approach to tasks should be attempted, when possible (accomplish as much as you feasibly can between migration calls).

4.2.1.3.2 Place Summary

Once again, we present the raw data in Figure 59, that contains the actual average execution times from the collection of Place test types, using different host configurations. We also present a better view into this data within Figure 60.

Places Performance (µs)

Number of Hosts	Iterations	Max Time	Iterations & Max Time
	Overall Average	Overall Average	Combined Average
1	4717311.969	787521.9375	2752416.95325
2	2558371.747	757714.2208	1658042.9839
4	1454711.578	676777.5625	1065744.57025
8	952676.325	633251.4417	792963.88335
16	649772.8594	922031.0167	785901.93805

Figure 59: Place Performance Summary Table

In Figure 60 we see a clear difference between the runtime exhibited between varying *iterations* and *max_time* within Place tests. The nearly steady performance of the *max_time* tests can be attributed to the constant computational load performed during each time slice. Whereas, the dramatic improvement in time that we see from *iterations* points directly to the computational load placed on each Place during some of the more “extreme” scenarios (i.e. - 10000 iterations).

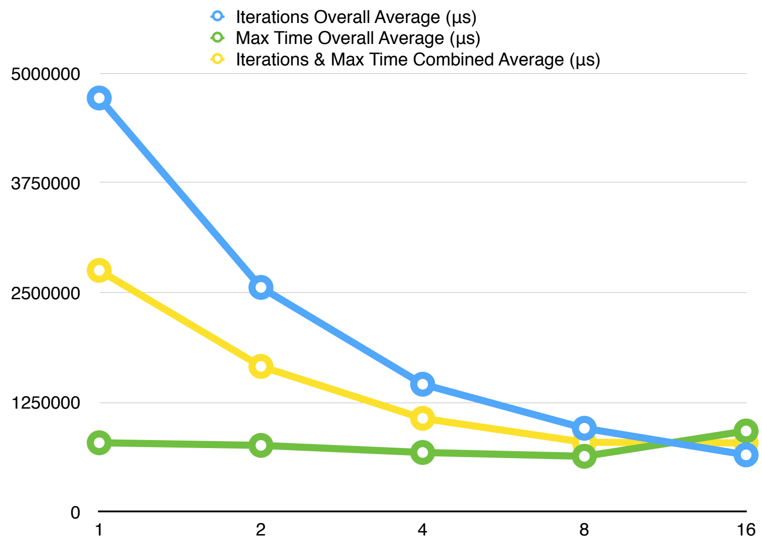

Figure 60: Place Performance Summary Chart

There are a couple of things worth noting here:

1. There is jump at 16 hosts for the *max_time* variable
   This is likely due to a combination of a poorly-performing machine (uw1-320-09), but also points to testing scenarios that include exchangeAll and exchangeBoundary calls
2. The benefits of parallelization across additional hosts are quite apparent
   There are a number of callAll scenarios tested within the Place benchmarks. So, this is not necessarily surprising, but it is pleasant to see represented in our data

All in all, the key takeaway when working with Places is that exchange calls will be expensive and will suffer from increased parallel resources (coordination/synchronization), however the benefit to performing complex computational operations at each Place is dramatic. Still, it is worth noting that as Places scale across additional hosts, there exists a point that the benefits to computational complexity are outweighed by the drawbacks of increased synchronicity costs. In the course of these tests, it appears as if that "magic number" is right around 12 hosts.

4.2.1.3.3 Overall Summary

So far, we have presented Agent and Place data fairly independently from one another. So, it is difficult to draw conclusions about how these two abstractions perform, compared to one another. Figure 61 shows the raw data of not only our previous Agent and Place aggregations, but includes a new "overall average" column that serves to track overall parallelization/performance of MASS (regardless of abstraction used in paradigm).

Agents/Places Performance (µs)

Number of Hosts	Agents			Places			Total Combined Average
	Iterations	Max Time	Iterations & Max Time	Iterations	Max Time	Iterations & Max Time	
	Overall Average	Overall Average	Combined Average	Overall Average	Overall Average	Combined Average	
1	6,123,395.80	4,638,616.55	5,381,006.17	4,717,311.97	787,521.94	2,752,416.95	4,066,711.56
2	7,283,284.98	4,330,652.90	5,806,968.94	2,558,371.75	757,714.22	1,658,042.98	3,732,505.96
4	3,137,263.88	1,839,810.61	2,488,537.25	1,454,711.58	676,777.56	1,065,744.57	1,777,140.91
8	2,630,107.23	1,294,248.78	1,962,178.00	952,676.33	633,251.44	792,963.88	1,377,570.94
16	1,645,702.10	742,365.40	1,194,033.75	649,772.86	922,031.02	785,901.94	989,967.84

Figure 61: Combined General Performance Summary Table

To aid in reviewing this content, we also provide a visual representation in Figure 62. According to this chart, the performance of Agents within MASS are far more costly to overall performance than the performance of Places. In fact, we see a really nice trend in our Places line - showing marked improvement as resources become available, with a slight uptick at the end (as a result of exchange-type tests).

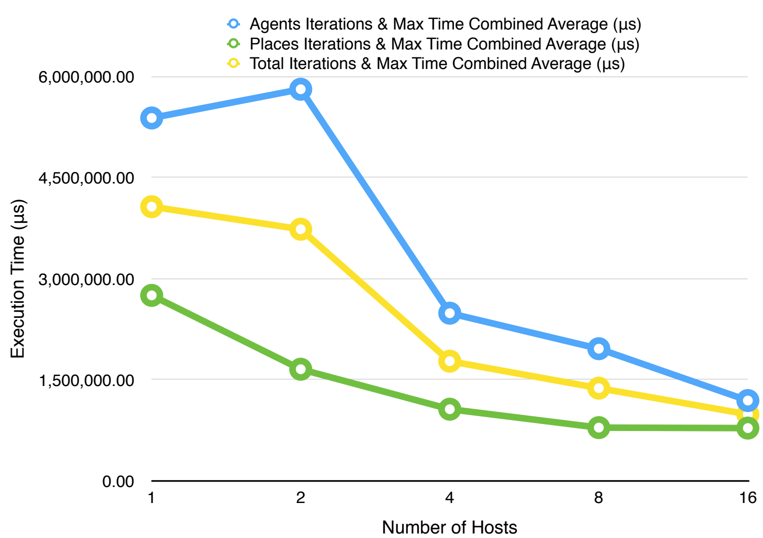

Figure 62: Combined General Performance Summary Chart

Looking at this same data, we are presented with the overall impact of Agents within MASS. We can see the effect of moving tests out to two hosts here - likely a result of sharing 256 Places between hosts and having to deal with a large number of competing resources on each machine as migrations occur. We can also see how this is severely decreased and continuously improved by applying more resources to the runtime environment.

The key takeaway from this look into the general performance of MASS is to stick with Places - if that's all you need. If you really need to model complex scenarios that require Agents on top of Places, then they're available to you and this paradigm scales well (as opposed to a potential scaling problem when a large number of nodes attempt Place exchange-type calls). However, be wary of situations where you end up placing a large number of Agents on a single machine, as this competition for resources could lead to resource contention, if you're not careful.

4.2.2 Practical MASS Performance

In this section, we will present the results of our performance testing using practical applications. These are applications that mimic (or, in the case of FluTE, "make") real world use of each framework. Whereas our general performance testing tended to isolate calls and vary resources available, these tests will generally use a combination of different calls and functionality available through each platform in the course of their execution.

4.2.2.1 Wave2D

4.2.2.1.1 Using Hybrid OpenMP/MPI

When testing Wave2D performance, Abdulhadi Ali Alghamdi [1] varied the test environment to see how the simulation would run with different resources provided. In Figure 63 we can see that as the number of threads available increased, the performance responded in kind. However, when increasing the number of processes (machines/hosts/nodes), it appeared to have a less-tangible effect on the overall performance.

	Test 1	Test 2	Test 3
Processes	1	1	4
Threads	1	4	4
Execution Time (μs)	7511	4202	3660

Figure 63: Wave2D Performance using Hybrid OpenMP/MPI

If we look at the graph in Figure 64, we really get a sense of how little gain was achieved by ramping up the number of processes in the simulation.

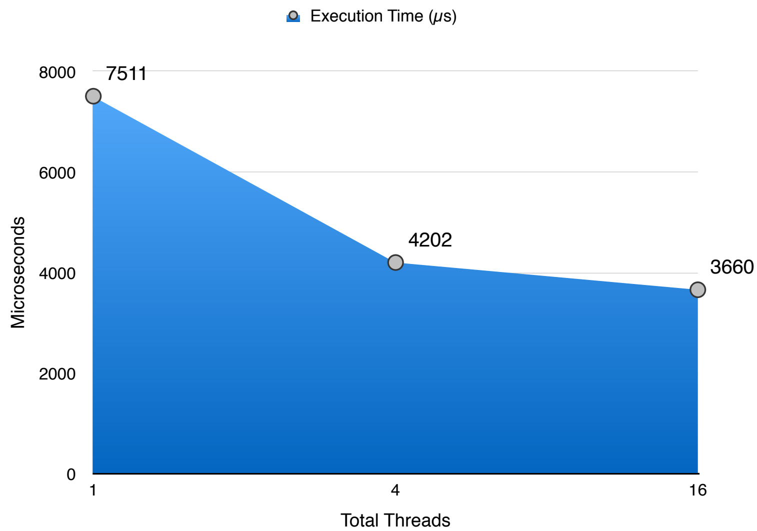

Figure 64: Wave2D Performance using Hybrid OpenMP/MPI

4.2.2.1.2 Using MASS

Using a similar approach for gathering MASS performance data, Abdulhadi Ali Alghamdi [1] varied the test environment in an identical fashion to the hybrid OpenMP/MPI performance tests. Figure 65 shows a familiar pattern of improvement with thread allocation, but also seems to suffer from a smaller effect size (in terms of execution time) when increasing the number of processes.

	Test 1	Test 2	Test 3
Processes	1	1	4
Threads	1	4	4
Execution Time (μs)	10590	5898	5053

Figure 65: Wave2D Performance using MASS

Looking at the graph in Figure 66, we see a familiar pattern in the performance across the three test scenarios. The largest gain is from increasing threads allocated, while increasing processes appears to have a minimal (positive) effect on the overall execution time.

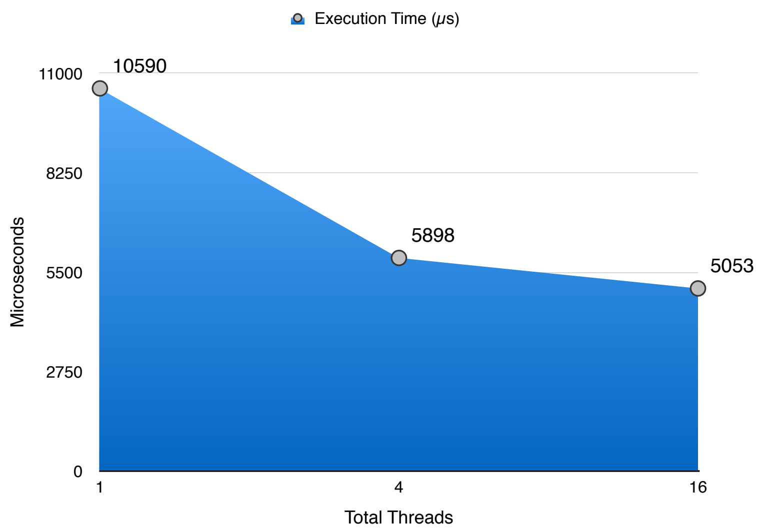

Figure 66: Wave2D Performance using MASS

4.2.2.1.3 Comparison Results

In order to get a better view of how these two frameworks stacked up side-by-side in a Wave2D application, we combined the data from the two tests into a single table. We also added another row that tracked the performance difference between the baseline application (Hybrid OpenMP/MPI) and MASS, in terms of a percentage difference. Figure 67 shows that the performance of MASS trailed across all three test scenarios.

	Test 1	Test 2	Test 3
OpenMP/MPI: Execution Time (μs)	7511	4202	3660
MASS: Execution Time (μs)	10590	5898	5053
MASS Difference (μs)	-40.99%	-40.36%	-38.06%

Figure 67: Wave2D Performance Comparison

If we put this same data into a chart, then we get a different perspective into this performance difference. As shown in Figure 68, the actual performance difference in this simulation decreased as more resources were provided to each framework.

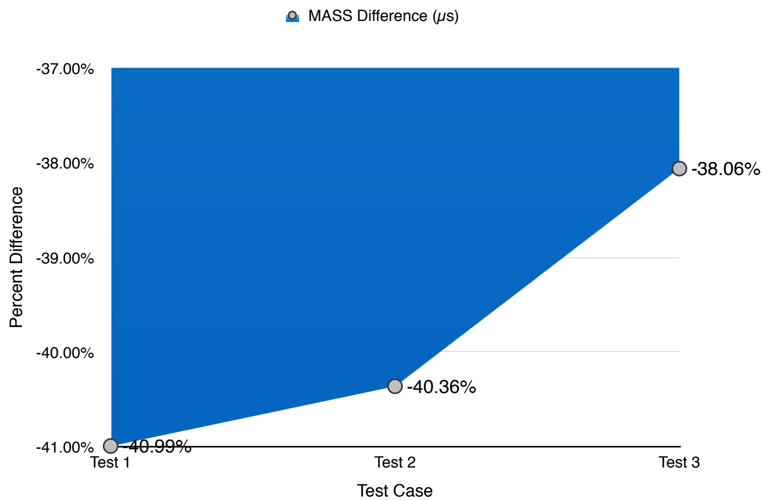

Figure 68: Wave2D Performance Comparison

4.2.2.2 Sugarscape

4.2.2.2.1 Using Hybrid OpenMP/MPI

Abdulhadi Ali Alghamdi [1] tested the performance of his Sugarscape implementation in similar fashion to the Wave2D tests - varying the resources available to the application framework and measuring the effect this had on the overall execution time. In Figure 69 we will once again see that as the number of threads available increased, the execution time dropped. We can also see a less-impactful drop as the number of processes jumped to four.

	Test 1	Test 2	Test 3
Processes	1	1	4
Threads	1	4	4
Execution Time (μs)	8922	5801	4914

Figure 69: Sugarscape Performance using Hybrid OpenMP/MPI

Observing the graph in Figure 70, we are once again given a visual representation of the small impact that increasing the number of processes had on overall performance (execution time).

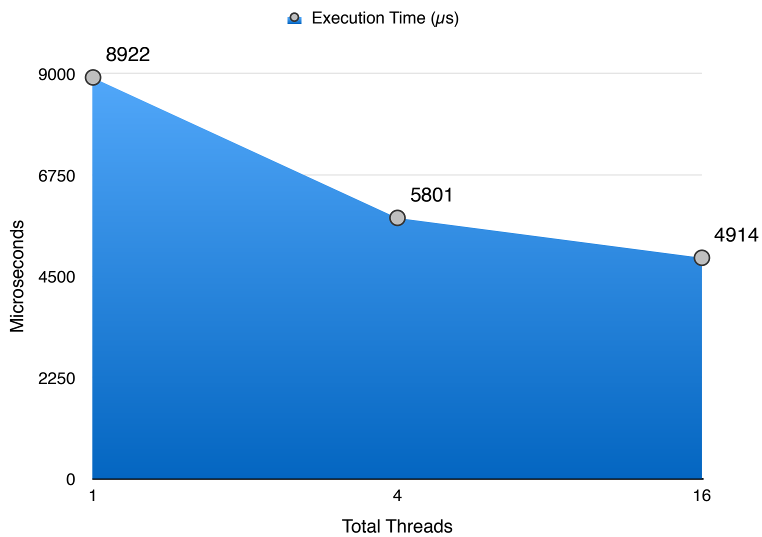

Figure 70: Sugarscape Performance using Hybrid OpenMP/MPI

4.2.2.2.2 Using MASS

Using an identical test schema, we can see in Figure 71 that MASS had a similar performance profile when threads/processes were increased. We can also see that the overall execution time was significantly higher with MASS.

	Test 1	Test 2	Test 3
Processes	1	1	4
Threads	1	4	4
Execution Time (μs)	12132	7819	6661

Figure 71: Sugarscape Performance using MASS

Figure 72 provides a visual representation of the trend that occurs as resources are increased within the MASS implementation of Sugarscape (threads have greater impact than processes).

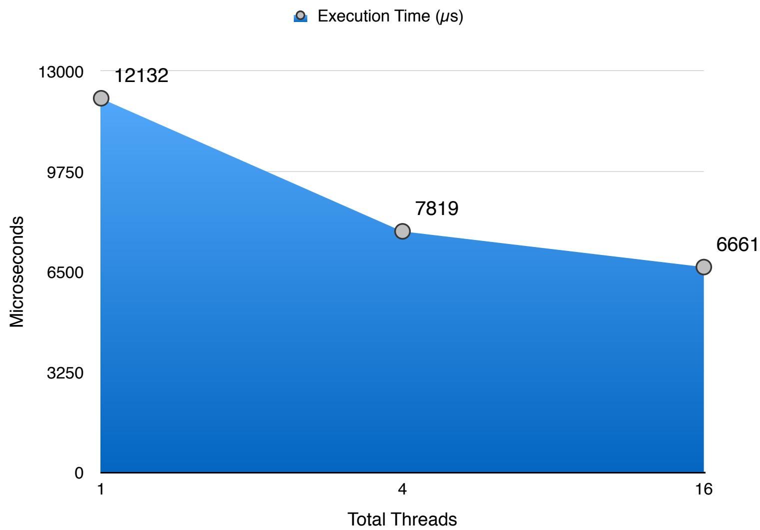

Figure 72: Sugarscape Performance using MASS

4.2.2.2.3 Comparison Results

Once again, we will attempt to provide a better view of how these two frameworks match up with one another, for Sugarscape, by combining the results from the two tests into a single table. We have also (once again) added another row that tracks the percent difference in execution time (performance) between the baseline application (Hybrid OpenMP/MPI) and MASS. Figure 73 shows that MASS under-performed across all three test scenarios.

	Test 1	Test 2	Test 3
OpenMP/MPI: Execution Time (µs)	8922	5801	4914
MASS: Execution Time (µs)	12132	7819	6661
MASS Difference (µs)	-35.98%	-34.79%	-35.55%

Figure 73: Sugarscape Performance Comparison

As with Wave2D, if we put this same data into a chart, we see that the difference between performance is generally pretty consistent. As shown in Figure 74, the percent difference fluctuates as more resources were provided to each application, but the overall deviation remains between 34 - 36%.

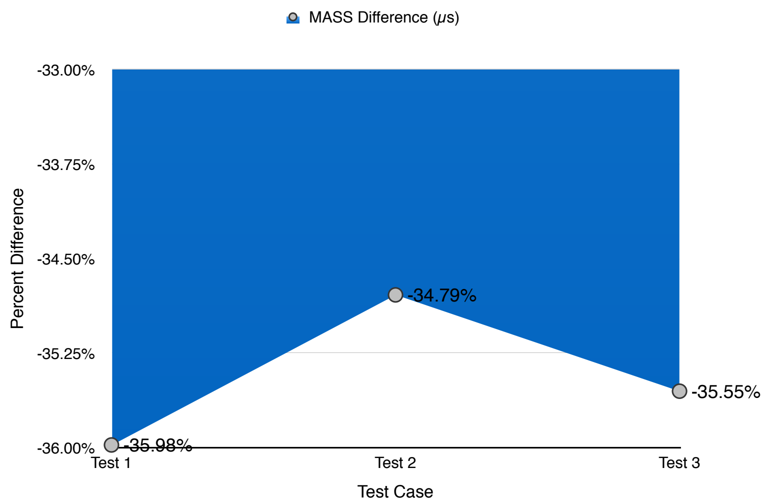

Figure 74: Sugarscape Performance Comparison

4.2.2.3 FluTE

Osmond Gunarso [17] tested the performance of his implementation of FluTE using a common data file (config.la-1.6). For more information on the details of this data file, please see Appendix K.

Of particular note is the "datafile" that was used for performance testing/comparison. This data file had the label "label la-1.6," which you will see referenced in results (below). The file itself is based on the "Los Angeles" file and describes a population with the following characteristics:

1. Tracts: 2049
   A tract represents a census tract, which "is an area roughly equivalent to a neighborhood established by the Bureau of Census for analyzing populations. They generally encompass a population between 2,500 to 8,000 people." [31]
2. Communities: 5547
   Communities are smaller groups located within census tracts. You can think of these as collections of co-workers, family members, friends, or neighbors.
3. Individuals: 11095039
   These are the actual number of people accounted for in our simulation.

As you can see, this is setting up a very massive and complicated scenario for our simulation.

To translate this data into MASS terms, Osmond [17] modeled each community as a place, each individual as an agent, and left tracts to become offsets into the data.

4.2.2.3.1 Using Hybrid OpenMP/MPI

In Figure 75 we can see that this simulation takes a lot of computing resources and time. More importantly, we also see that there is a definite effect on performance as more parallel/distributed resources become available. Unlike the previous examples, this effect only captures increasing the number of processes (hosts) available for the distributed execution of the program. However, we do see a continued, near-linear, decrease in execution time as processes are added.

	Test 1	Test 2	Test 3
Configuration File	config.la-16	config.la-16	config.la-16
Processes	1	2	4
Execution Time (s)	2338.04	1085.1	525.81

Figure 75: FluTE Performance using Hybrid OpenMP/MPI

We can see this near-linear behavior in Figure 76.

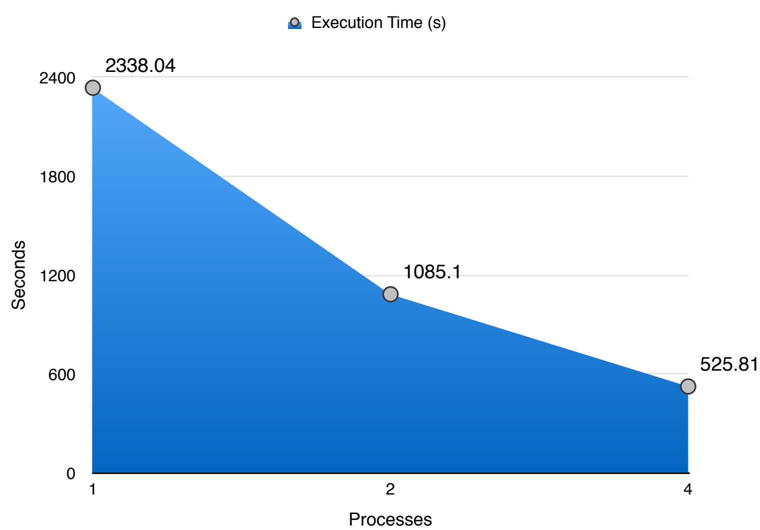

Figure 76: FluTE Performance using Hybrid OpenMP/MPI

4.2.2.3.2 Using MASS

Using an identical test schema, we can see in Figure 77 that MASS performed nearly as well as hybrid OpenMP/MPI using one process. However, it also appears to be nearly 50% slower when operating across multiple processes (hosts).

	Test 1	Test 2	Test 3
Configuration File	config.la-16	config.la-16	config.la-16
Processes	1	2	4
Execution Time (s)	2344.64	1852.63	905.16

Figure 77: FluTE Performance using MASS

Figure 78 provides a visual cue into this difference with a slower drop when utilizing two processes (hosts), and a more dramatic drop when transitioning to use four processes.

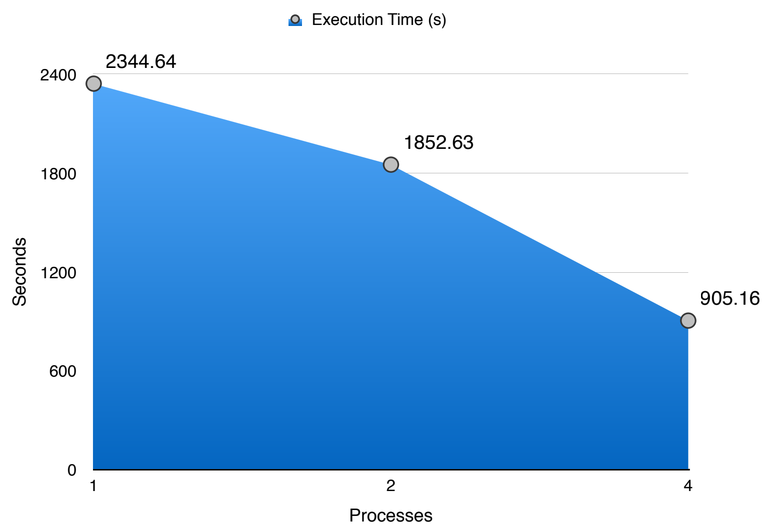

Figure 78: FluTE Performance using MASS

4.2.2.3.3 Comparison Results

Reviewing the performance data side-by-side, as in Figure 79, we see that the performance of Hybrid OpenMP/MPI and MASS applications of FluTE were nearly identical, given one process. However, when each application was provided with an additional process (host), the improvement for hybrid OpenMP/MPI was significantly greater than MASS. When four processes were assigned to the work, MASS had a better gain (in terms of execution time), but failed to keep pace with the improvement offered by hybrid OpenMP/MPI (in terms of percent difference).

	Test 1	Test 2	Test 3
OpenMP/MPI: Execution Time (s)	2338.04	1085.10	525.81
MASS: Execution Time (s)	2344.64	1852.63	905.16
MASS Difference (s)	-0.28%	-70.73%	-72.15%

Figure 79: FluTE Performance Comparison

Figure 80 illustrates the dramatic drop in competitiveness betweeen the two implementations. However, it also shows that the trend (difference in terms of percentage) appears to begin to level out as more resources are provided. Due to the massive size of this simulation and what we know about MASS's performance with high loads of Places/Agents per machine, it is a small wonder that the profile here is trailing the hybrid OpenMP/MPI approach.

Note: Osmond's parallelization heavily uses the master to maintain the shared data. Since I'm disclosing my thesis to the committee now - to provide ample time to review prior to my defense on Wednesday - I wanted to bring this to your attention: the data will be updated with new performance by the final defense.

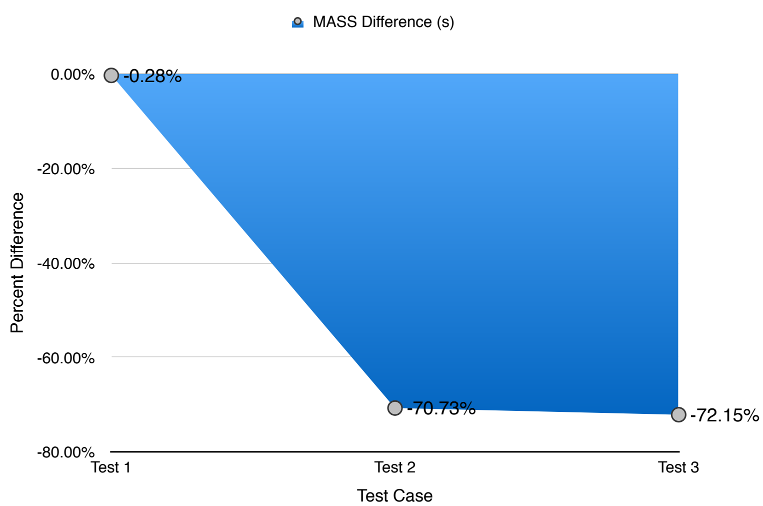

Figure 80: FluTE Performance Comparison

4.2.2.4 Combined Summary

In this section, we take a look at how hybrid OpenMP/MPI and MASS applications performed against one another considering the results of all of the practical applications tested. Ideally, we could take a sum of the execution times and get an overall average that encompassed a variety of configurations and subject domains to get a good overall picture of each framework. However, due to the nature of FluTE and its extensive runtime performance, all results would be confounded by these data points. So, instead we took a look at the average performance of each practical application and calculated the percent difference between frameworks.

Figure 81 shows us the results of these calculations and provides an overall average of the averages. What this data point is showing us is the overall average percent difference between each framework's performance. While it may seem odd to see different values here, they have been presented in a manner that is conducive toward discussion (i.e. - they do not assume one framework is the ultimate baseline in each computation).

		Test 1	Test 2	Test 3	Average	Average % Difference
OpenMP/MPI: Execution Time (μs)	FluTE	2,338,040,000	1,085,100,000	525,810,000	1,316,316,667	22.61%
	Sugarscape	8,922	5,801	4,914	6,546	26.21%
	Wave2D	7,511	4,202	3,660	5,124	28.63%
	Average Average % Difference	25.82%				
MASS: Execution Time (μs)	FluTE	2,344,640,000	1,852,630,000	905,160,000	1,700,810,000	-29.21%
	Sugarscape	12,132	7,819	6,661	8,871	-35.52%
	Wave2D	10,590	5,898	5,053	7,180	-40.12%
	Average Average % Difference	-34.95%				

Figure 81: Practical Application Performance Summary

Using the results of this summary comparison, we are able to make statements like, "Hybrid OpenMP/MPI applications typically perform 25.82% better than corresponding applications based on MASS." Conversely, we can also say that, "MASS applications typically perform 34.95% worse than corresponding applications based on a hybrid OpenMP/MPI framework." Though the numbers are different, you have to remember that this is due to how the comparison is being made.

Take for example a simpler case: What percent lower than 100 is 70? Most folks can answer this easily enough - it is 30% lower. The calculation to prove this is easy enough to perform, as well: *100 - (100 * .30) = 70*. However, it is another thing entirely to ask: What percent higher than 70 is 100? In this case, you have to consider what fraction of 70 makes up the difference between 70 and 100. Since 30% of 70 is 21, we can easily see that the reverse logic here and percentages are not consistent when switching comparators in our function (just in case you're wondering, the answer is ~42.86%).

4.3 Correlations

Since we had the survey data collected, we also wanted to see if there were any interesting correlations between data points in our responses. We used the Pearson product-moment correlation coefficient [8] measurement across all data points in our survey results, which have been broken down by framework and can be seen below.

Since there were a large number of variables to cross-correlate with one another, we have truncated the entire list below entries with a .20 *R value*. We have also highlighted (darker background color) all of the correlations that actually represent significant relationships.

4.3.1 OpenMP/MPI Correlations

Figure 82 shows generally expected results across the board. Still, it is interesting to see how different assessments of aspects of OpenMP/MPI play into how many lines of code it took people to complete their applications.

Sample size	47
Critical value (2%)	2.41212

Variable vs. Variable	**R**
OMPI: Difference between sequential and parallel programs vs. OMPI: Total LOC	-2.58992E+196
OMPI: Debugging difficulty vs. OMPI: Total LOC	1.49369E+196
OMPI: Application Suitability vs. OMPI: Total LOC	-1.00916E+196
OMPI: Learning curve vs. OMPI: Total LOC	3.27244E+195
OMPI: Parallelization-specific LOC vs. OMPI: Total LOC	0.62407
OMPI: Debugging difficulty vs. OMPI: Learning curve	0.50934
OMPI: Design the program vs. OMPI: Learn the library	0.4217
MASS vs OMPI: Places/Agents.callAll vs. OMPI: Design the program	0.35649
OMPI: Debug the program vs. OMPI: Write the program	0.29497
MASS vs OMPI: Places/Agents.callAll vs. OMPI: Application Suitability	-0.2946
OMPI: Application Suitability vs. OMPI: Learn the library	-0.28996
OMPI: Write the program vs. OMPI: Design the program	0.28006
MASS vs OMPI: Agents.manageAll vs. MASS vs OMPI: Places.exchangeAll	0.27629
OMPI: Total LOC vs. OMPI: Write the program	0.24669
OMPI: Parallelization-specific LOC vs. OMPI: Design the program	-0.24454
OMPI: Learning curve vs. OMPI: Learn the library	-0.24009
MASS vs OMPI: Places/Agents.callAll vs. OMPI: Learning curve	0.2385
OMPI: Debugging difficulty vs. OMPI: Debug the program	0.23662
OMPI: Debug the program vs. OMPI: Design the program	0.23591
OMPI: Difference between sequential and parallel programs vs. OMPI: Parallelization-specific LOC	0.23308
OMPI: Debugging difficulty vs. OMPI: Difference between sequential and parallel programs	0.2152
OMPI: Application Suitability vs. OMPI: Learning curve	0.21416
MASS vs OMPI: Agents.manageAll vs. OMPI: Learn the library	-0.20584
MASS vs OMPI: Agents.manageAll vs. OMPI: Difference between sequential and parallel programs	0.20486
MASS vs OMPI: Agents.manageAll vs. OMPI: Total LOC	-0.20435
OMPI: Difference between sequential and parallel programs vs. OMPI: Design the program	0.20314

Figure 82: OpenMP/MPI Variable Correlation

4.3.2 MASS Correlations

Figure 83 shows the same sort of relationships that one would expect to see - effort and lines of code, time taken and lines code, etc. Of particular note here would be the repeated correlations between the amount of time it takes to debug the MASS library.

Sample size	46
Critical value (2%)	2.41413
Variable vs. Variable	**R**
MASS: Parallelization-specific LOC vs. MASS: Total LOC	0.8142
MASS: Design the program vs. MASS: Learn the library	0.69755
MASS: Write the program vs. MASS: Design the program	0.68739
MASS: Debug the program vs. MASS: Learn the library	0.5206
MASS: Debug the program vs. MASS: Design the program	0.49857
MASS: Debug the program vs. MASS: Write the program	0.42874
MASS: Write the program vs. MASS: Learn the library	0.394
MASS: Debugging difficulty vs. MASS: Learning curve	0.33771
MASS: Application Suitability vs. MASS: Learning curve	0.3254
MASS vs OMPI: Places.exchangeAll vs. MASS: Learning curve	0.31451
MASS vs OMPI: Places/Agents.callAll vs. MASS: Debug the program	0.30151
MASS vs OMPI: Agents.manageAll vs. MASS vs OMPI: Places.exchangeAll	0.27629
MASS vs OMPI: Agents.manageAll vs. MASS: Learning curve	0.26657
MASS: Difference between sequential and parallel programs vs. MASS: Write the program	-0.25536
MASS: Learning curve vs. MASS: Debug the program	-0.23801
MASS: Learning curve vs. MASS: Write the program	-0.23149
MASS: Difference between sequential and parallel programs vs. MASS: Learning curve	0.2259
MASS: Difference between sequential and parallel programs vs. MASS: Parallelization-specific LOC	-0.21904
MASS vs OMPI: Places/Agents.callAll vs. MASS: Application Suitability	-0.20815
MASS: Application Suitability vs. MASS: Design the program	0.20695
MASS: Debugging difficulty vs. MASS: Learn the library	0.20553

Figure 83: MASS Variable Correlation

A more detailed analysis of these relationships is out of the scope of this research paper. These results are merely presented to further inform future research or efforts to increase programmability in the MASS framework.

5 Discussion

5.1 Summary

At this point, we have provided an overview of both MASS and hybrid OpenMP/MPI application frameworks, come up with a hypothesis regarding the ease-of-use and performance of these systems, designed experiments to test out our hypothesis, and presented the resuls of these experiments. In this discussion, we will highlight the findings of our research, discuss limitations to the studies performed, and finally, review our progress toward meeting the original goals of this investigation.

5.1.1 Ease of Use (Programmability)

During the course of our research, we found that according to the programmability characteristics in "Parallel programmability and the chapel language," (Chamberlain, et al; 2007) [3] MASS:

1. Had More of a Global View of Computation
2. Had Less Support for General Parallelism
3. Had Equal Separation of Algorithm and Implementation
4. Had Equal Support for Broad-Market Language Features
5. Had Less Data Abstractions
6. Was Less Performant
7. Had Less Execution Model Transparency
8. Had Equal Portability
9. Had Equal Interoperability with Existing Codes
10. Had Less Bells and Whistles

Which, set an initial expectation that MASS would continue to underperform against applications based on hybrid OpenMP/MPI. However, when we removed the inherent bias toward general parallel frameworks (over paradigm-oriented frameworks) in Figure 4, we ended up with a much more interesting comparison - one that pointed toward the main difference being related to additional features ("Bells & Whistles").

When we actually surveyed students that had used both frameworks to develop the same application, we also found a very close assessment of programmability.

According to survey results, we found that programmers using MASS:

1. Took 1 hour, 2 minutes, and 24 seconds (1.04 hours) longer to learn the libraries
2. Took 43 minutes and 48 seconds (0.73 hours) longer to design their applications
3. Took 39 minutes (0.65 hours) less to write their applications
4. Took 19 minutes and 12 seconds (.32 hours) longer to debug their applications
5. Had to write approximately 56 (55.88) more lines of code in their application
6. Had to write approximately 13 (13.15) less lines of parallell/distributed-specific lines of code in their application
7. Rated the Learning Curve around 23.36% (0.73 points) more difficult
8. Rated the Application Suitability around 2.52% (0.09 points) more difficult
9. Rated the Difference Between Sequential and Parallel Programs around 7.45% (0.22 points) more difficult
10. Rated the Debugging Difficulty around 8.88% (0.22 points) easier

5.1.2 Performance

Looking at the performance results between the same application developed using MASS and hybrid OpenMP/MPI, we found that:

1. FluTE
   The MASS implementation of FluTE ran 29.21% slower than the corresponding application based on hybrid OpenMP/MPI
2. Sugarscape
   The MASS implementation of Sugarscape ran 35.52% slower than the corresponding application based on hybrid OpenMP/MPI
3. Wave2D
   The MASS implementation of Wave2D ran 40.21% slower than the corresponding application based on hybrid OpenMP/MPI

5.1.3 Potential Confounding Issues

In nearly all statements of truth, there is a "grain of salt" to be considered, too. While we are generally pleased with the validity of the test design and results gathered in this research, it is prudent to also consider factors that may have positively or negatively influenced these results:

1. Order Topics Were Presented in Class
   Students were presented with OpenMP/MPI first and then had to recontextualize their point of view for parallel/distributed programming to adapt to a completely different model (MASS). This point is hard to avoid, since it is beneficial for students to learn the basics of parallelization strategies (data/task decomposition, striping, efficient cache use, etc), but at the same time, it is worth considering the added difficulty in learning how to do something you've become familiar with in a different manner. Our brains learn patterns for accomplishing tasks or thinking about problems, and as these patterns are used and reinforced, adapting to different approaches introduces its own difficulty
2. Class Time Spent Learning Each Framework
   Due to the nature of teaching these concepts (moving from small pieces and building up to larger/integrated frameworks), there is an inherent bias introduced in learning each framework, since these concepts are readily-transferrable to the "hands on" approach required when using MPI and OpenMP. In fact, looking at the course syllabus [15], we can see that a combined 4 weeks of lectures, 2 laboratory sessions, and programming assignments were provided that dealt with concepts beneficial to hybrid OpenMP/MPI development. On the other hand, we see 1 lecture, 1 laboratory session, and a single assignment that dealt directly with MASS
3. Competing Concepts Learned During MASS
   During the second half of the course (when topics related to MASS were presented), students were also responsible for researching and presenting literature reviews on other frameworks in the realm of parallel/distributed computing. These reviews had students independently learning about job management, file management, and fault tolerance approaches used in conjunction with complex systems that supported these ideas. On the other hand, during the first half of the quarter, the only expectation on student learning were the concepts presented in class (i.e. - students could entirely focus on OpenMP and MPI when they were presented)
4. Combined Survey Application
   Students were not asked to review hybrid OpenMP/MPI applications immediately after completing their corresponding assignment. Instead, the survey was provided after completing their applications using MASS. In terms of time, the second programming assignment (using hybrid OpenMP/MPI) was due on February 12, 2015. However, the survey they were asked to submit was due on March 18, 2015. This means that students were being asked to remember and assess the time and difficulty of a task that they performed over a month ago. This could

result in more "forgiving" assessments of the process difficulty or time taken during developing a hybrid OpenMP/MPI application. This could especially be true considering the potentially recent difficulty encountered by students while adapting previous applications to a new framework (MASS).

5. Overwhelming Use of Heat2D Application
   The results of the survey were based on evaluations that students provided after programming an application using both frameworks. We wanted to allow students to choose their own application to use, in order to reduce the possibility of confounding our data from students being assigned a domain that they had little interest in completing (or would find particularly difficult). Unfortunately, the result of this was that 33 out of the total 48 applications chosen by students were Heat2D. So, the average of results are dominated by this simulation. Since the remaining 15 applications were spread between a variety of other options, we did not have sufficient data to show (conclusively) that significant differences exist between frameworks per application type/area (e.g. - spatial simulations, big data analysis, or agent-based models)
6. Interest/Ability of Students in Second Class
   During the first course that the survey was administered, students had already completed a core programming class (required for their program) and had opted to enroll in CSS 534 "Parallel Programming in the Grid and Cloud" due to their interest in the subject matter. On the other hand, students in the second course that we surveyed had not taken a previous programming course and may have had little interest in parallel programming specifically, opting to enroll to merely fulfill graduation requirements. Furthermore, these students would not have had the benefit of a previous graduate-level programming course to aid in their general programming knowledge/capability. Since we have data on each course, we examined this area in great detail within Section 4.1.2.6

5.1.4 Generalizability of Results

The sampling method used was non-random and took advantage of convenience to obtain data. It would be incorrect to assume that we can generalize these same findings out to a wider population.

Statements regarding the findings of this study could be used to generally describe the trends of computer science students with entry-level experience in parallel/distributed programming, but drawing out the conclusion(s) contained herein to a wider group is unwarranted, given the test design chosen and implied limitations therein.

The main point of this research was to do an initial study into how these two frameworks compared with one another.

5.2 Academic Merit

At the beginning of this paper, we presented a hypothesis that pertained to the programmability of MASS. Our specific case went on to compare metrics around programmability against OpenMP/MPI, but that is not necessarily new knowledge, either. It is more like mixing some new hip-hop lyrics over a classic soul sample - it is a combination of things that already exist. While interesting to view things in this light, it is not introducing new knowledge that had not previously existed.

In our overview, we listed six goals that we wanted to achieve in this paper. This section will review these goals and provide additional insight into how we did on achieving them.

1. Provide Further Support for Programmability Claims
   Our research has added to the corpus on knowledge on programmability in MASS. We have discussed the paradigm-oriented approach to application development and the reduced burden to development that this approach presents to programmers
2. Provide First Programmability Assessment of C++ Implementation
   This research has also provided a stake in the ground for programmability using the C++ implementation of MASS. This paper is the first to breach this topic. Previously, all research and programmability claims for MASS had been isolated to the Java implementation

3. Track User Assessment of MASS
   Our paper has provided survey results that have tracked programmer assessment of MASS in terms of time, effort, and ease-of-use (programmability). This represents the first publication to present user-centered, quantifiable results related to MASS
4. Provide Insight into Effort and Time Using MASS
   This paper has provided very detailed looks into time and effort required during individual tasks of the development workflow for both frameworks (hybrid OpenMP/MPI and MASS), in addition to roll up summaries of these findings. This is the first paper to actually take a look into these factors for MASS
5. First Benchmarked Baseline MASS Performance Data
   This study is the first to gather and present baseline performance data for MASS. We have provided in-depth looks at the results of individual performance of dicrete MASS functionality, in addition to offering a synopsis of the overall performance characteristics of this framework
6. First Analysis of FluTE Performance in MASS
   We have presented performance data on the MASS implementation of FluTE - data that had previously only existed for sequential and hybrid OpenMP/MPI implementations of the simulation. This is the first performance analysis of a real-world, complicated simulation with interesting emergent properties in MASS. As such, it offers a glimpse into the ability for MASS to scale to handle realistic use-case scenarios

There are four additional outcomes from this research that were not specifically enumerated during our overview. These represent additional, important findings from this paper that are outcomes from work into proving/disproving our hypothesis.

The first outcome was that we have found MASS to be quite competitive with OpenMP/MPI in the fields of agent-based models, spatial simulations, and big data analysis. While the performance aspects give OpenMP/MPI a clear advantage, the programmability - across the board - is quite competitive. In fact, despite additional tooling to ease debugging, hybrid OpenMP/MPI applications still trail MASS in programmability for these categories.

Secondly, we have also found that a relative newcomer to the scene (MASS) could prove to be quite competitive with what could (arguably) be considered the dominant solution in this problem space - hybrid OpenMP/MPI. The programmability aspects of MASS are quite competitive with a system that has had the advantage of industry/organization-wide support, with a nearly two decade advantage. This is significant and represents a true opportunity for those working on developing MASS. After all, it is still coming into its own - there is active development on new features, functionality, and documentation that will all end up having a measurable effect on the overall ease-of-use (programmability) of this framework.

The third point is that our findings have set a baseline for future research into the programmability and performance of MASS. This is significant because we can use this data to track:

1. The effect of changes to framework
2. Performance changes when integrating new features (asynchronous automatic agent migration, built-in debugger)
3. Programmability changes when updating existing code, including:
   (a) Updating documentation
   (b) Adding persistent FAQ section
   (c) Bug/issue tracking and resolution
   (d) Implementation of additional methods for Places and Agents

Finally, we can extend the survey used in this study to include additional data points to actually develop an idea of preference (lacking in current study) and use this research as a basis for future studies. This is a particularly interesting subject. After all, you can build up a lot of research around the time it takes to do something, the effort involved in the process, and the easiness of discrete tasks within the activity - however, when it is all said and done, people could still prefer the seemingly harder task. We believe the assumption that time, effort, and ease-of-use necessarily lead to preferability is inherently flawed and fails to track intangible aspects like the true usefulness and attractiveness of a particular approach.

6 Conclusion

In this section, we will discuss the outcome of this work, answering the question: "Do programmers in big data analysis and ABM find MASS easier to use than hybrid OpenMP/MPI, despite its slower performance?"

If you will remember, the alternative hypothesis was defined as:

$$H_A = \mu \textit{ MASS Ease-of-Use} > \mu \textit{ Hybrid OpenMP/MPI Ease-of-Use}$$

The implied alternative hypothesis around performance was stated as:

$$H_A = \mu \textit{ MASS Performance} < \mu \textit{ Hybrid OpenMP/MPI Performance}$$

6.1 Ease of Use (Programmability)

Summarizing, in terms of time, effort (LOC), and programmability, we can say that:

1. Time
   Overall, programmers can expect to spend 1 hour, 26 minutes, and 24 seconds (1.44 hours) longer developing their applications, than they would by using a hybrid OpenMP/MPI approach
2. Effort (LOC)
   Programmers using MASS will have to write 56 more lines of code in their applications, but they will also be writing 8.17% less parallel/distributed-specific lines of code in those same applications
3. Programmability
   Programmers will generally find that MASS is 6.76% more difficult use, in terms of (learning, designing, writing, and debugging their applications).

Based on these findings, we are unable to reject the null hypothesis (accept the alternative hypothesis). In fact, we find that across the board (while results are close), hybrid OpenMP/MPI is slightly easier to use than MASS. So, the evidence supports/reinforces the null hypothesis:

$$H_0 = \mu \textit{ MASS Ease-of-Use} \leq \mu \textit{ Hybrid OpenMP/MPI Ease-of-Use}$$

While we have already managed to fail to accept our alternative hypothesis concerning ease-of-use, we still have the orthogonal issue of performance to consider.

6.2 Performance

The performance results presented in this paper allow us to make the following, general, statement about the performance of MASS: MASS applications typically perform 34.95% slower than corresponding applications based on a hybrid OpenMP/MPI framework.

Given these results, we are able to accept the alternative hypothesis for performance:

$$H_A = \mu \textit{ MASS Performance} < \mu \textit{ Hybrid OpenMP/MPI Performance}$$

While at the same time (due to the implication of accepting H_A) being able to reject the null hypothesis for performance:

$$H_0 = \mu \textit{ MASS Performance} \geq \mu \textit{ Hybrid OpenMP/MPI Performance}$$

6.3 Future Work

During the course of this paper, several outstanding issues or unanswered questions were brought up. This section details these, listing suggestions for possible future research into MASS.

1. Garbage Collection
   MASS does not currently make use of smart pointers - a potential area for future improvement.
2. Generic Programming
   MASS currently relies on inheritance (extending parent/base Place/Agent classes) to provide users a method to customize MASS for their own applications. Within these classes, it is possible to use C++ templates, but for greater flexibility, it would be a nice improvement to translate this paradigm into an actual template interface.
3. MASS Support
   Detailed information and illustrations of the underlying functionality in MASS are either very hard to find or non-existent - making it quite difficult to tune applications built using MASS. We would suggest creating an open message board or forum that users across classes can benefit from - asking questions and helping to find answers to common problems (at the same time, helping developers working on MASS identify/address pain points for users). We'd further suggest adding more examples for students to reference and provide a source for "living" documentation (meaning that it changes/develops along with MASS).
4. MASS Portability
   Currently, MASS has only been run on grids composed of machines that are running a Linux kernel. It would be interesting to see how portable it is across other architectures - indentifying and fixing potential bugs to increase its portability/usefulness.
5. Further Surveying of Class Matching Spring 2014 Composition
   There were a number of differences between the Spring 2014 and Winter 2015 populations that could account for significant differences in the programmability, effort, and time required using each framework. Unfortunately, the sampling size for the Spring 2014 course was rather small, so many differences were found to be statistically insignificant. However, the "trend" in some of these results were interesting and with more data, could point toward significant differences based on population characteristics.
6. Add Survey Question to Gauge Student's Ability/Interest
   Following with the previous idea, future surveys should take into account each student's programming ability (language backgrounds, number of months used, last time used, experience with C++ libraries) or interest in parallel/distributed computing. Adding additional questions to the survey to track this data could help reduce confounding variables in future test results and may help provide interesting correlations or additional conclusions. Being able to separate/classify groups according to their actual characteristics, like novice versus expert users or high GPA (grade 3.5+) versus low GPA (grade 3.5-), rather than simply "when they enrolled in the course" (Spring 2014 versus Winter 2015) would allow much more useful groupings and comparisons to be made regarding these frameworks.
7. Add Survey Question to Gauge Student's Preference
   One of the main pitfalls to the current survey is that it does not ask which framework students prefer. I like to think of this as the VHS vs Betamax problem. For those of you unfamiliar with this reference, the core problem is that we have presented a lot of data that shows that it is easier to learn, design, and write programs using hybrid OpenMP/MPI. We have also shown that it takes less overall lines of code and time to develop these applications. However, even though something takes less effort, less time, and is generally easier (at first), does not mean that people will prefer using it in the future. It could be that students have a hard time adapting to learning MASS at first (especially since they've previously spent weeks adapting a sequential algorithm and developing the same application in a different parallel framework), but if they were asked to develop a whole new application, they may really prefer to approach that MASS provides. I feel like this missing question would really help shed light on not just initial programmability, but also lasting preference - which, is an important thing to consider.

8. Split the Survey and Administer Immediately
   Currently, students are given a single survey and asked to remember details about what they were working on almost 5 weeks before. This can lead to estimation problems and comparative error (influence based on perceived experience recently using MASS). Instead, we should split the survey into two surveys and provide them to students immediately after each corresponding assignment - collecting OpenMP/MPI data separately from MASS data, but more importantly, collecting it while the estimations are still fresh in student's minds.
9. Assign Varied Applications
   We found that the dominate choice of students typically corresponded with the "path of least resistance" - meaning that, given the choice, students will choose the Heat2D application 68.75% of the time. This ends up skewing the data in favor of the time, effort, and programmability of this particular application, instead of providing a more complete, overall view into applications in general (or across domains)
10. Randomize Sampling in Future Experiments
   Accompanying this idea would be actually extended the scope of potential people surveyed beyond the classroom - taking into account the responses from programmers that are actively involved in parallel/distributed application development. This would allow a more useful study, in terms of being able to generalize results out to a wider audience
11. Investigate Ways to Detect/Manage Slow Nodes
   One of the big, recurring themes in our general performance results was anomalies found when using a slow node on our grid. While it is probably fiscally infeasible to use a hosted solution (AWS, Azure, etc), it would be worthwhile to spend some time looking into: machine state monitoring and redundancy solutions for lab machines (load balancing, mirroring, etc).

7 Appendix

A Actual Survey

MASS Programmability Analysis

Q1. State your time (in hours) needed to complete your HW2 and HW4 respectively.

Programming Stages	Hours you have spent for hybrid MPI/OpenMP	Hours you have spent for hybrid MASS
To learn the library		
To design the program		
To write the program		
To debug the program		

Q2. State the code size (in lines) of your HW2 and HW4 respectively.

	Hybrid MPI/OpenMP version	MASS version
Total lines (excluding comments)		
Parallelization-specific code		

Q3. State the programmability of HW2 and HW4: 1 quite hard, 2: hard, 3: fair, 4: good, 5: excellent

	Hybrid MPI/OpenMP version	MASS version
Learning curve		
The suitability to your application		
Degree of difference between sequential and parallel programs (1: big – 5: little difference)		
Debugging difficulty		

Q4. State the degree of easiness of the following MASS functions when you wrote your program, as compared to MPI/OpenMP funcitons: 1: quite hard, 2: hard, 3: fair, 4: easy, 5: quite easy, X: not used

MASS functions	Degree of easiness
Places/Agents.callAll	
Places.exchangeAll	
Agents.manageAll	

Q5. Estimate the degree of the following future functions' usefulness for your HW4 application as well as any applications you would like to code in the future: 1: not useful at all 2: probably not useful 3: maybe useful 4: useful, 5: quite useful

Future MASS functions	Degree of usefulness
Places.callSome	
Places.exchangeBoundary	
Agent.migrate (part 1): agent diffusion	
Agent.migrate (part 2): collision avoidance	
Parallel file I/Os	
Optimistic synchronization	

Q6. State the merits and demerits of hybrid MPI/OpenMP and MASs respectively.

Hybrid MPI/OpenMP merits:	Hybrid MPI/OpenMP demerits:
MASS merits:	MASS demerits:

Q7. In addition to your HW4 application, what applications else can you think take advantage of MASS?

Q8. Please report any bugs in MASS you found while you were developing your application.

No bugs found (if so, check the right box)	
Yes, some bugs found (if so, check the right box and list the bugs below)	

Bug #	Descriptions
1	
2	

Q9. May the professor's research group use your report for their future funding proposal submissions and paper publications, provided your name is recognized in acknowledgments/references or listed as one of our co-authors?

YES
YES under some conditions. Please write the conditions below: - -
NO

B Performance Test Program Command Line Arguments

1. username
   This is the name of the account to log into machines as (e.g. - UW Net ID).
2. password
   The password for this account (e.g. - UW Net ID password).
3. machinefile
   The path to a file, which lists remote machines (URLs) to use at runtime.
4. port
   The unique port to use for communication (e.g. - UW Student ID #).
5. nProc
   The number of processes to use at runtime.
6. nThr
   The number of threads each process should use at runtime.
7. test_type
   The type of test to run (see: Test Types; below).
8. size
   The size of the simulation space.
9. max_time
   The number of times to run the overall tests (not related to actual time - milliseconds, seconds, etc).
10. iterations
    The number of times individual Place Objects run through thier own computations (can be used to simulate applications with heavy or very light downstream computation).

C Places Performance Test Types

1. Numerical ID "1": Test Places callAll and exchangeAll
   This test accesses every place within the simulation and has this place perform a simple mathematical expression (in this case: **val *= 1.2;*). Depending on the value for *iterations*, this calculation is performed either one or many times. In addition, an exchangeAll() call is made after this operation, which simply returns the newly computed sum from the previous step across all place Objects in the simulation.
2. Numerical ID "2": Test Places exchangeBoundary, callAll, and store output
   This test accesses every place in the simulation and has that place exchange its current information (data type: *double*) with its neighbors (north/south/east/west or top/bottom/right/left - however you want to visualize it). It then makes another call to alter this value by performing a simple mathematical equation (in this case: **val *= 1.2;*), before making a final call to move its current value into the "outMessage" storage (area used to store values for future exchange calls).
3. Numerical ID "3": Test Places callAll
   This test accesses every place within the simulation and has this place perform a simple mathematical expression (in this case: **val *= 1.2;*). Depending on the value for *iterations*, this calculation is performed either one or many times.
4. Numerical ID "4": Test Places callAll with periodic return value
   Like the previous test, this test accesses every place within the simulation and has this place perform a simple mathematical expression (in this case: **val *= 1.2;*). Depending on the value for *iterations*, this calculation is performed either one or many times. The difference comes at every 10th time interval (based on *max_time* value), at which point each place is called and asked to return its current value.

D Agents Performance Test Types

1. Numerical ID “1”: Test Agents callAll (null return value)
   This test accesses every agent in the simulation and has the agent perform a simple mathematical expression (in this case: **val *= 1.2;*). Depending on the value for *iterations*, this calculation is performed either one or many times.
2. Numerical ID “2”: Test random migration
   This test accesses every agent in the simulation and has that agent migrate to another random Place in the simulation space. The location of this place is calculated by generating a random number, then dividing this number by the size of the simulation (to ensure that value remains in bounds). This calculation is performed to generate a new “x” and “y” coordinate pair, which is then used as this agent’s new location. Using this algorithm, it is entirely possible that the new location matches the current location - in this case, no movement is actually performed.
3. Numerical ID “3”: Test full migration
   This test is very similar to the random migration process, with one notable exception: logic has been added, when calculating the new coordinates, to ensure that the migration will actually occur (possibility of being assigned current place is removed). This represents a “worst case” scenario for migration performance.
4. Numerical ID “4”: Test Agents callAll (with return value)
   Like the callAll test above (Numerical ID “1”), this test accesses every agent in the simulation and has the agent perform a simple mathematical expression (in this case: **val *= 1.2;*). Depending on the value for *iterations*, this calculation is performed either one or many times. The difference is that this call actually returns the value calculated, which is then printed out by the calling test method.
5. Numerical ID “5”: Test Agent Migration: Best Migrate
   This test is very similar to the other migration tests that have been detailed, with one notable exception: it is only run once - the *max_size* attribute is ignored. It also targets the “best case” scenario - meaning that additional logic is in place to ensure that migrations result in Agents remaining in the same Place.
6. Numerical ID “6”: Test Agent Migration: Random Migrate
   This test is very similar to the other migration tests that have been detailed, with one notable exception: it is only run once - the *max_size* attribute is ignored.
7. Numerical ID “7”: Test Agent Migration: Worst Migrate
   This test is very similar to the other migration tests that have been detailed, with one notable exception: it is only run once - the *max_size* attribute is ignored. It also targets the “worst case” scenario - meaning that additional logic is in place to ensure that migrations actually occur (Agents can not be assigned a new location equal to their current location).

E Survey Results

MASS Survey Combined Results Summary (Spring 2014 & Winter 2015)

Question 1: State your time (in hours) needed to complete your HW2 and HW4 respectively

Student	OpenMP/MPI Hours				MASS Hours			
	To learn the library	To design the program	To write the program	To debug the program	To learn the library	To design the program	To write the program	To debug the program
Student 1	3	2	3	4	2	1	1	2
Student 2	0	0	20	20	10	2	8	12
Student 3	3	2	6	4	5	4	3	3
Student 4	8	2	2	4	4	1	4	4
Student 5	4	2	3	4	2	3	4	4
Student 6	4	4	20	20	4	8	30	20
Student 7	10	3	10	5	10	3	4	12
Student 8	4	3	5	14	8	2	4	12
Student 9	20	10	15	25	8	3	3	20
Student 10	3	6	6	4	8	3	2	3
Student 11	4	2	5	4	3	1	6	6
Student 12	10	2	5	1	6	4	5	0
Student 13	2	2	4	4	1	2	1.5	1.5
Student 14	4	3	6	6	2	1	2	3
Student 15	1.5	1.5	2	4	1.5	1.5	2	5
Student 16	8	6		2	8	6		12
Student 17	0.5	0.1	1	7	3	2	20	10
Student 18	8	4	25	10				
Student 19	2.5	5	2.5	6	5	8	3	6
Student 20	15	20	5	10	10	5	2	20
Student 21	1	1	4	4	5	2	2	6
Student 22	4	0.5	1	6	8	4	8	20
Student 23	5	5	7	9	7	7	10	9
Student 24	3.5	3	10	3	10.5	3.5	11.5	5
Student 25	10	20	5	6	10	10	5	5
Student 26	2.5	4.5	9	8	2.5	1.5	5	5
Student 27	2	2	1	4	2	3	2	4
Student 28	20	10	10	10	10	10	8	12
Student 29	8	10	15	17	14	16	20	18
Student 30	8	16	8	16	16	30	16	16
Student 31	3	10	50	10	15	20	50	15
Student 32	10	8	15	10	10	4	8	15
Student 33	10	8	8	6	6	4	4	2
Student 34	30	2	4	3	16	3	2	1
Student 35	4	8	5	5	6	6	4	9
Student 36	2	2	1	1	6	4	1	5
Student 37								
Student 38	7	3	5	7	4	4	4	5
Student 39	4	6	18	52	4	3	4	10
Student 40	3	2	6	5	6	4	5	10
Student 41	5	3	5	10	15	20	15	20
Student 42	6	2	10	3	10	4	15	2
Student 43	2	0.5	1	10	8	6	6	18
Student 44	4	4	4	4	6	6	6	6
Student 45	2	4	8	12	2	4	7	0
Student 46	2	1	4	2	6	1	2	10
Student 47	2	1	4	6	4	1.5	1	2
Student 48	7	12	15	8	14	15	15	16

MASS Survey Combined Results Summary (Spring 2014 & Winter 2015)

Student	Question 2: State the code size (in lines) of your HW2 and HW4 respectively			
	Homework 2 (Hybrid OpenMP/MPI)		Homework 4 (MASS)	
	Total lines (excluding comments and debug statements)	Parallelization-specific code	Total lines (excluding comments and debug statements)	Parallelization-specific code
Student 1	114	55	173	114
Student 2	192	14	260	3
Student 3	150	50	164	0
Student 4	243	27	239	9
Student 5	179	98	216	135
Student 6	6648	4880	4189	2389
Student 7	180	50	190	45
Student 8	163	18	411	23
Student 9	142	70	209	50
Student 10	160	56	180	35
Student 11	80	180	250	100
Student 12	483	65	155	100
Student 13	137	17	74	5
Student 14	250	55	150	10
Student 15	305	158	459	0
Student 16	118	58	192	132
Student 17	200	8	400	100
Student 18				
Student 19	190	43	300	30
Student 20	165	83	298	110
Student 21	102	32	71	6
Student 22	25	4	400	100
Student 23	279	46	262	21
Student 24	100	20	100	13
Student 25	200	20	150	10
Student 26	136	17	155	9
Student 27	150	27	190	10
Student 28	180	30	200	
Student 29	231	40	206	40
Student 30	124	60	220	100
Student 31	150	120	180	70
Student 32				
Student 33	246	70	300	10
Student 34	105	31	184	17
Student 35	152	24	186	9
Student 36	214	140	177	100
Student 37				
Student 38	175	41	196	10
Student 39	110	46	120	10
Student 40	256	124		
Student 41	230	40	250	50
Student 42	104	32	270	40
Student 43	20	15	400	350
Student 44	600	120	350	100
Student 45	230	70		
Student 46	236	119	312	10
Student 47	274	96	245	15
Student 48	375	153	346	132

MASS Survey Combined Results Summary (Spring 2014 & Winter 2015)

Student	Question 3: State the programmability of HW2 and HW4: 1 quite hard, 2: hard, 3: fair, 4: good, 5: excellent							
	Hybrid MPI/OpenMP version				MASS version			
	Learning curve	The suitability to your application	Degree of difference between sequential and parallel programs	Debugging difficulty	Learning curve	The suitability to your application	Degree of difference between sequential and parallel programs	Debugging difficulty
Student 1	2	4	2	2	4	5	3	4
Student 2								
Student 3	2	3	1	2	1	3	5	1
Student 4	3	5	2	2	2	3	3	2
Student 5	3	4	3	2	1	4	2	2
Student 6	3	4	4	3	2	5	1	1
Student 7	3	4	3	2	2	3	2	2
Student 8	3	3	1	2	1	3	3	1
Student 9	2	2	3	2	4	3		2
Student 10	4	4	2	3	2	4	4	4
Student 11			4				3	
Student 12	4	4	4	3	3	2	4	4
Student 13	2	3	5	1	4	3	1	4
Student 14	3	4	1	2	5	4	5	5
Student 15	4	4	4	1	3	3	2	1
Student 16	5	5	3	2	1	2	3	1
Student 17	5	4	5	3	1	2	4	1
Student 18								
Student 19	4	3	2	3	1	2	1	1.5
Student 20	3	5	3	4	1	5	4	2
Student 21	3	4	3	3	2	2	3	2
Student 22	5	3	4	4	1	2	1	5
Student 23	3	4	1	3	2	5	4	2
Student 24	4	3	4	3	3	3	4	3
Student 25	4	5	1	3	4	3	1	4
Student 26	3	4	1	2	4	5	3	2
Student 27	2	3	2	2	2	4	2	4
Student 28	2	4	3	3	3	4	2	3
Student 29	2	2	2	2	2	3	2	2
Student 30	3	4	4	3	1	3	2	2
Student 31	4	3	3	3	4	4	4	3
Student 32	1	5	4	1	1	5	1	1
Student 33	3	4	3	1	3	5	5	4
Student 34	3	3	4	3	2	4	2	5
Student 35	4	3	4	3	4	4	2	3
Student 36	1	5		1	4	5		4
Student 37	1	1	1	2	2	5	5	3
Student 38	3	4	2	2	3	5	2	5
Student 39	4	5	2	2	4	5	4	1
Student 40	3			3	1			1
Student 41								
Student 42	3	4	4	4	1	4	1	5
Student 43	3	3	5	2	2	2	1	4
Student 44	3	3	4	3	2	4	2	1
Student 45	3	3	4	1	3	1	1	1
Student 46	5	4	1	4	2	5	5	2
Student 47	3	4	3	2	2	4	3	4
Student 48	4	4	4	3	3	3	1	2

MASS Survey Combined Results Summary (Spring 2014 & Winter 2015)

Student	Question 4: State the degree of easiness of the following MASS functions when you wrote your program, as compared to MPI/OpenMP functions: 1: quite hard, 2: hard, 3: fair, 4: easy, 5: quite easy, (blank): not used			Question 5: Estimate the degree of the following future functions' usefulness for your HW4 application as well as any applications you would like to code in the future: 1: not useful at all 2: probably not useful 3: maybe useful 4: useful, 5: quite useful					
	Existing MASS Functions			Future MASS Functions					
	Places/Agents callAll	Places exchangeAll	Agents manageAll	Places callSome	Places exchangeBoundary	Agent.migrate (part 1): agent diffusion	Agent.migrate (part 2): collision avoidance	Parallel file I/Os	Optimistic synchronization
Student 1		4		4	5	4	4	2	2
Student 2									
Student 3	3	2		3	5	3	3	2	4
Student 4	2	2		4	5	1	1	1	3
Student 5		1		4	5	3	3	3	3
Student 6	3		5	5	5	5	1	1	3
Student 7		5		4	4	4	5	5	4
Student 8	1	1		4	3	3	4	5	4
Student 9	3	3		3	3	4	4	3	4
Student 10	5			5	5	1	1	3	2
Student 11	5	4		5	4	4	4	5	3
Student 12	3	3		3	5	4	4	5	3
Student 13	4	4		4	4	3	3	3	4
Student 14	5	5		5	4	5	5	5	5
Student 15	2.5	2		4	5	4	4	3	5
Student 16	3			5	5	3	3	5	4
Student 17	4		4	5	5	3	3	4	3
Student 18									
Student 19	4	4		4	5	4	3	4	3
Student 20	2	2		3	5	1	1	1	1
Student 21	5	3		5	5	1	1	1	1
Student 22	1	1	1	2	1	1	1	2	2
Student 23	4		4	5	4	4	4	5	4
Student 24	3	4		4	4	3	3	1	3
Student 25				2	5	3	3	5	5
Student 26	5	1		4	5	5	2	3	
Student 27	5				4			3	3
Student 28	4				4				
Student 29	3	3		4	4	4	4	4	4
Student 30	2			3	5	4	4	4	4
Student 31	5	4		4	5	4	5	5	5
Student 32	5	5		5	5	3	3	3	3
Student 33	3	4	4	3	5	3	3	5	5
Student 34	5			5	5	4	2	5	5
Student 35	3	3	3	3	4	4	4	5	5
Student 36	4	2	2	5	5	3	3	4	3
Student 37	5	2	5		5	5		1	5
Student 38	4	3		5	5	2	2	2	3
Student 39		5		3	5	4	5	5	5
Student 40	1	4		3	4	4	4	1	5
Student 41									
Student 42	4	2			5				5
Student 43	3	2	2	4	4	4	4	3	4
Student 44	4	2	5	4	4	3	3	3	3
Student 45	2			2	2	2	2	4	4
Student 46	5			5	5	4	5	5	5
Student 47	5	4	4	5	4	4	3	2	3
Student 48	3			3	4	3	3	4	2

F Detailed t-Test Results Between Surveyed Classes

F.1 Time to Learn the Library

In Figure 84 and Figure 85, we see that the difference in values between classes is not statistically significant enough to conclusively point to a marked difference in results.

Variable	Sample size	Mean	Variance
To learn the library	31	6.79032	43.3129
To learn the library	16	4.4375	8.69583
Summary			
Degrees Of Freedom	45	Hypothesized Mean Difference	0E+00
Test Statistics	1.35596	Pooled Variance	31.77388
Two-tailed distribution			
p-level	0.18188	t Critical Value (5%)	2.0141
One-tailed distribution			
p-level	0.09094	t Critical Value (5%)	1.67943

Figure 84: Two-Sample T-Test Result: OpenMP/MPI - To Learn the Library

Variable	Sample size	Mean	Variance
To learn the library	15	6.06667	12.53095
To learn the library	31	7.5	19.35
Summary			
Degrees Of Freedom	44	Hypothesized Mean Difference	0E+00
Test Statistics	1.09946	Pooled Variance	17.1803
Two-tailed distribution			
p-level	0.27755	t Critical Value (5%)	2.01537
One-tailed distribution			
p-level	0.13877	t Critical Value (5%)	1.68023

Figure 85: Two-Sample T-Test Result: MASS - To Learn the Library

F.2 Time to Design the Program

In Figure 86 and Figure 87, we also find that the difference between classes is not statistically significant.

Variable	Sample size	Mean	Variance
To design the program	16	3.4375	8.9625
To design the program	31	5.58387	28.39806
Summary			
Degrees Of Freedom	45	Hypothesized Mean Difference	0E+00
Test Statistics	1.4893	Pooled Variance	21.91954
Two-tailed distribution			
p-level	0.14339	t Critical Value (5%)	2.0141
One-tailed distribution			
p-level	0.07169	t Critical Value (5%)	1.67943

Figure 86: Two-Sample T-Test Result: OpenMP/MPI - To Design the Program

Variable	Sample size	Mean	Variance
To design the program	31	6.40323	43.87366
To design the program	15	3.9	12.75714
Summary			
Degrees Of Freedom	44	Hypothesized Mean Difference	0E+00
Test Statistics	1.36547	Pooled Variance	33.97295
Two-tailed distribution			
p-level	0.17905	t Critical Value (5%)	2.01537
One-tailed distribution			
p-level	0.08952	t Critical Value (5%)	1.68023

Figure 87: Two-Sample T-Test Result: MASS - To Design the Program

F.3 Time to Write the Program

In Figure 88 and Figure 89, the difference between classes is once again not statistically significant enough.

Variable	Sample size	Mean	Variance
To write the program	30	8.11667	86.64971
To write the program	16	8.4375	47.99583
Summary			
Degrees Of Freedom	44	Hypothesized Mean Difference	0E+00
Test Statistics	0.12091	Pooled Variance	73.47225
Two-tailed distribution			
p-level	0.90431	t Critical Value (5%)	2.01537
One-tailed distribution			
p-level	0.45216	t Critical Value (5%)	1.68023

Figure 88: Two-Sample T-Test Result: OpenMP/MPI - To Write the Program

Variable	Sample size	Mean	Variance
To write the program	15	5.73333	21.49524
To write the program	30	8.5	107.13793
Summary			
Degrees Of Freedom	43	Hypothesized Mean Difference	0E+00
Test Statistics	0.98276	Pooled Variance	79.25426
Two-tailed distribution			
p-level	0.33123	t Critical Value (5%)	2.01669
One-tailed distribution			
p-level	0.16561	t Critical Value (5%)	1.68107

Figure 89: Two-Sample T-Test Result: MASS - To Write the Program

F.4 Time to Debug the Program

In Figure 90 and Figure 91, we continue to find that the difference between classes is not statistically significant.

Variable	Sample size	Mean	Variance
To debug the program	31	9.25806	94.93118
To debug the program	16	6.75	19.4
Summary			
Degrees Of Freedom	45	Hypothesized Mean Difference	0E+00
Test Statistics	0.97554	Pooled Variance	69.75412
Two-tailed distribution			
p-level	0.33451	t Critical Value (5%)	2.0141
One-tailed distribution			
p-level	0.16725	t Critical Value (5%)	1.67943

Figure 90: Two-Sample T-Test Result: OpenMP/MPI - To Debug the Program

Variable	Sample size	Mean	Variance
To debug the program	15	6.53333	19.98095
To debug the program	31	9.79032	46.29624
Summary			
Degrees Of Freedom	44	Hypothesized Mean Difference	0E+00
Test Statistics	1.68156	Pooled Variance	37.92319
Two-tailed distribution			
p-level	0.09974	t Critical Value (5%)	2.01537
One-tailed distribution			
p-level	0.04987	t Critical Value (5%)	1.68023

Figure 91: Two-Sample T-Test Result: MASS - To Debug the Program

F.5 Effort: Total Lines of Code

In Figure 92 we find the closest statistical evidence to support a difference between the survey samples collected from each class. However, it is still shy of the cut-off and, along with Figure 93, we have to rule the difference between classes as not statistically significant.

Variable	Sample size	Mean	Variance
Total lines	12	231.58333	16,435.53788
Total lines	28	173.42857	3,830.84656
Summary			
Degrees Of Freedom	38	Hypothesized Mean Difference	0E+00
Test Statistics	1.94889	Pooled Variance	7,479.57299
Two-tailed distribution			
p-level	0.05872	t Critical Value (5%)	2.02439
One-tailed distribution			
p-level	0.02936	t Critical Value (5%)	1.68595

Figure 92: Two-Sample T-Test Result: OpenMP/MPI - Total Lines

Variable	Sample size	Mean	Variance
Total lines	28	255.03571	21,560.9246
Total lines	13	228.92308	9,123.57692
Summary			
Degrees Of Freedom	39	Hypothesized Mean Difference	0E+00
Test Statistics	0.58426	Pooled Variance	17,734.04839
Two-tailed distribution			
p-level	0.56241	t Critical Value (5%)	2.02269
One-tailed distribution			
p-level	0.28121	t Critical Value (5%)	1.68488

Figure 93: Two-Sample T-Test Result: MASS - Total Lines

F.6 Effort: Parallel-Specific Lines of Code

In Figure 94 we find the first case of a statistically significant difference between classes. This means that there was something about the Spring 2014 class that led to them writing more parallel-specific lines of code in their applications.

While we have identified a difference in this area for OpenMP/MPI applications, we still find in Figure 95, that MASS parallel-specific lines of code differences between classes were not statistically significant.

Variable	Sample size	Mean	Variance
Parallelization-specific code	12	29.5	224.45455
Parallelization-specific code	28	21.60714	33.50661
Summary			
Degrees Of Freedom	38	Hypothesized Mean Difference	0E+00
Test Statistics	2.42781	Pooled Variance	88.78102
Two-tailed distribution			
p-level	0.02004	t Critical Value (5%)	2.02439
One-tailed distribution			
p-level	0.01002	t Critical Value (5%)	1.68595

Figure 94: Two-Sample T-Test Result: OpenMP/MPI - Parallel-Specific Lines

Variable	Sample size	Mean	Variance
Parallelization-specific code	13	7.92308	4.24359
Parallelization-specific code	28	12.17857	64.59656
Summary			
Degrees Of Freedom	39	Hypothesized Mean Difference	0E+00
Test Statistics	1.86898	Pooled Variance	46.02642
Two-tailed distribution			
p-level	0.06915	t Critical Value (5%)	2.02269
One-tailed distribution			
p-level	0.03457	t Critical Value (5%)	1.68488

Figure 95: Two-Sample T-Test Result: MASS - Parallel-Specific Lines

F.7 Learning Curve

In Figure 96 and Figure 97, we return to the familiar pattern of not finding enough evidence to support a statistically significant difference in survey results.

Variable	Sample size	Mean	Variance
Learning curve	30	3.26667	1.09885
Learning curve	14	2.78571	0.7967
Summary			
Degrees Of Freedom	42	Hypothesized Mean Difference	0E+00
Test Statistics	1.48199	Pooled Variance	1.00533
Two-tailed distribution			
p-level	0.14581	t Critical Value (5%)	2.01808
One-tailed distribution			
p-level	0.0729	t Critical Value (5%)	1.68195

Figure 96: Two-Sample T-Test Result: OpenMP/MPI - Learning Curve

Variable	Sample size	Mean	Variance
Learning curve	14	2.5	1.5
Learning curve	30	2.33333	1.33333
Summary			
Degrees Of Freedom	42	Hypothesized Mean Difference	0E+00
Test Statistics	0.43756	Pooled Variance	1.38492
Two-tailed distribution			
p-level	0.66395	t Critical Value (5%)	2.01808
One-tailed distribution			
p-level	0.33197	t Critical Value (5%)	1.68195

Figure 97: Two-Sample T-Test Result: MASS - Learning Curve

F.8 Application Suitability

Figure 98 shows a very near significant difference between class survey results for OpenMP/MPI application suitability. However, this difference, along with the MASS difference found in Figure 99, were still not statistically significant enough.

Variable	Sample size	Mean	Variance
Application Suitability	29	3.51724	0.90148
Application Suitability	14	4.07143	0.37912
Summary			
Degrees Of Freedom	41	Hypothesized Mean Difference	0E+00
Test Statistics	1.98513	Pooled Variance	0.73585
Two-tailed distribution			
p-level	0.05385	t Critical Value (5%)	2.01954
One-tailed distribution			
p-level	0.02692	t Critical Value (5%)	1.68288

Figure 98: Two-Sample T-Test Result: OpenMP/MPI - Application Suitability

Variable	Sample size	Mean	Variance
Application Suitability	14	3.92857	0.99451
Application Suitability	29	3.44828	1.39901
Summary			
Degrees Of Freedom	41	Hypothesized Mean Difference	0E+00
Test Statistics	1.3092	Pooled Variance	1.27076
Two-tailed distribution			
p-level	0.19776	t Critical Value (5%)	2.01954
One-tailed distribution			
p-level	0.09888	t Critical Value (5%)	1.68288

Figure 99: Two-Sample T-Test Result: MASS - Application Suitability

F.9 Difference Between Parallel and Sequential Algorithms

In Figure 100 and Figure 101, we see that neither the OpenMP/MPI or MASS evaluation of parallel/sequential difference in algorithms produced statistically significant results between the classes surveyed.

Variable	Sample size	Mean	Variance
Difference between sequential and parallel programs	30	3.03333	1.48161
Difference between sequential and parallel programs	13	2.61538	1.75641
Summary			
Degrees Of Freedom	41	Hypothesized Mean Difference	0E+00
Test Statistics	1.00711	Pooled Variance	1.56204
Two-tailed distribution			
p-level	0.31979	t Critical Value (5%)	2.01954
One-tailed distribution			
p-level	0.1599	t Critical Value (5%)	1.68288

Figure 100: Two-Sample T-Test Result: OpenMP/MPI - Difference Between Parallel and Sequential Algorithms

Variable	Sample size	Mean	Variance
Difference between sequential and parallel programs	29	2.68966	1.86453
Difference between sequential and parallel programs	13	2.69231	1.89744
Summary			
Degrees Of Freedom	40	Hypothesized Mean Difference	0E+00
Test Statistics	0.0058	Pooled Variance	1.8744
Two-tailed distribution			
p-level	0.9954	t Critical Value (5%)	2.02108
One-tailed distribution			
p-level	0.4977	t Critical Value (5%)	1.68385

Figure 101: Two-Sample T-Test Result: MASS - Difference Between Parallel and Sequential Algorithms

F.10 Debugging Difficulty

In Figure 102 and Figure 103, we once again find no statistically significant differences in the results between surveyed classes (samples).

Variable	Sample size	Mean	Variance
Debugging difficulty	14	2.21429	0.7967
Debugging difficulty	30	2.53333	0.67126
Summary			
Degrees Of Freedom	42	Hypothesized Mean Difference	0E+00
Test Statistics	1.16976	Pooled Variance	0.71009
Two-tailed distribution			
p-level	0.24869	t Critical Value (5%)	2.01808
One-tailed distribution			
p-level	0.12435	t Critical Value (5%)	1.68195

Figure 102: Two-Sample T-Test Result: OpenMP/MPI - Debugging Difficulty

Variable	Sample size	Mean	Variance
Debugging difficulty	14	2.64286	2.55495
Debugging difficulty	30	2.65	1.70948
Summary			
Degrees Of Freedom	42	Hypothesized Mean Difference	0E+00
Test Statistics	0.01572	Pooled Variance	1.97117
Two-tailed distribution			
p-level	0.98753	t Critical Value (5%)	2.01808
One-tailed distribution			
p-level	0.49377	t Critical Value (5%)	1.68195

Figure 103: Two-Sample T-Test Result: MASS - Debugging Difficulty

F.11 Comparison: callAll Functionality

Figure 104 shows that we are unable to find evidence to support a statistically significant difference between the Spring 2014 and Winter 2015 survey results.

Variable	Sample size	Mean	Variance
Places/Agents.callAll	26	3.34615	1.75538
Places/Agents.callAll	14	3.96429	1.01786
Summary			
Degrees Of Freedom	38	Hypothesized Mean Difference	0E+00
Test Statistics	1.52094	Pooled Variance	1.50307
Two-tailed distribution			
p-level	0.13655	t Critical Value (5%)	2.02439
One-tailed distribution			
p-level	0.06828	t Critical Value (5%)	1.68595

Figure 104: Two-Sample T-Test Result: callAll Comparison

F.12 Comparison: exchangeAll Functionality

If we look at Figure 105 we are once again unable to prove a statistically significant difference between survey results for each course (sample).

Variable	Sample size	Mean	Variance
Places.exchangeAll	12	2.75	1.65909
Places.exchangeAll	20	3.15	1.60789
Summary			
Degrees Of Freedom	30	Hypothesized Mean Difference	0E+00
Test Statistics	0.8589	Pooled Variance	1.62667
Two-tailed distribution			
p-level	0.39721	t Critical Value (5%)	2.04227
One-tailed distribution			
p-level	0.1986	t Critical Value (5%)	1.69726

Figure 105: Two-Sample T-Test Result: exchangeAll Comparison

F.13 Comparison: manageAll Functionality

Finally, as shown in Figure 106 there is not enough evidence to point toward a statistically significant difference between the results of the two courses surveyed.

Variable	Sample size	Mean	Variance
Agents.manageAll	7	3.42857	2.28571
Agents.manageAll	4	3.75	1.58333
Summary			
Degrees Of Freedom	9	Hypothesized Mean Difference	0E+00
Test Statistics	0.35803	Pooled Variance	2.05159
Two-tailed distribution			
p-level	0.72857	t Critical Value (5%)	2.26216
One-tailed distribution			
p-level	0.36429	t Critical Value (5%)	1.83311

Figure 106: Two-Sample T-Test Result: manageAll Comparison

G Agents Baseline Results: Iterations

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	1	1	2900725	2901987	2970228	2874216	2871525	2995132	2899463	2917829	2944596	2873876	2914957.7	40266.96387	2995132	2871525
1	1	10	2994209	2966727	2981022	2972089	2931982	2936510	2947495	2943667	2972438	2965308	2961144.7	19321.70355	2994209	2931982
1	1	50	2659620	2650310	2618332	2567854	2631377	2619292	2653540	2680797	2616346	2625423	2632289.1	29485.78149	2680797	2567854
1	1	100	2412494	2405597	2346840	2356473	2346475	2379318	2268116	2379320	2370285	2372648	2363756.6	38101.19037	2412494	2268116
1	1	200	2103678	2098554	2150626	2143972	2153731	2198162	2167271	2170470	2131625	2174184	2149227.3	29658.87731	2198162	2098554
1	1	500	2704717	2698386	2673458	2688001	2724597	2840387	2682769	2698517	2677430	2741551	2712981.3	46943.24944	2840387	2673458
1	1	1000	4942017	4936779	4938613	4887317	4833901	4936899	4908764	4964416	4957441	5052348	4935849.5	53276.92111	5052348	4833901
1	1	10000	47241212	47591576	47403624	47276318	47834057	47458769	47386415	47327209	47806002	47301797	47462697.9	202420.0846	47834057	47241212
2	1	1	3323551	3347857	3371835	1442833	1473007	1452224	1511787	3413120	3371755	1463279	2417124.8	948881.5803	3413120	1442833
2	1	10	3409533	3481048	3479974	3471352	1480391	3433987	1489840	3486530	3416439	1502925	2865201.9	899970.7117	3486530	1480391
2	1	50	3098283	3119913	3069906	3111482	3118603	3106632	3127591	1375978	1368418	1378581	2587538.7	794376.1006	3127591	1368418
2	1	100	1215080	1204774	1224373	2760848	2850940	2789568	1213713	1302639	1211607	1203312	1697685.4	722729.6146	2850940	1203312
2	1	200	1110546	1126180	1075707	1161770	1126385	1143118	1064079	1118415	1126330	1170370	1122290	31723.52231	1170370	1064079
2	1	500	1403885	1421526	1451084	1387996	1411651	1389194	1424383	1397375	1374709	1426199	1408800.2	21535.21954	1451084	1374709
2	1	1000	2546064	2543598	2568777	2559750	2532875	2539379	2503593	2505412	2502867	2526365	2532868	22129.08779	2568777	2502867
2	1	10000	24167478	24152352	24164331	24104047	25908938	24043458	24185151	24053011	23991623	24161976	24293236.5	542083.4218	25908938	23991623
4	1	1	1715845	810035	1751054	1701580	1705950	1731749	1695568	1716330	1701513	1709696	1623932	271746.2752	1751054	810035
4	1	10	1762715	1735887	1746718	1741344	1744787	1749578	1771207	1795504	1744230	1749899	1754186.9	16888.61142	1795504	1735887
4	1	50	1568195	1580032	1609286	1587291	1616186	1577336	1575350	1591500	1597109	1615677	1591796.2	16415.37811	1616186	1568195
4	1	100	1390577	1407191	1416685	1409367	1428043	1428444	1401431	1418191	1429764	1429306	1415899.9	12851.97916	1429764	1390577
4	1	200	900037	657088	709840	581691	830861	1242057	825957	1018340	582458	626020	797434.9	202103.826	1242057	581691
4	1	500	731496	800233	719935	757659	740707	747427	722579	735817	729841	793996	747969	26786.46936	800233	719935
4	1	1000	1330584	1354397	1284103	1297138	1314531	1311615	1298540	1320665	1299746	1300352	1311167.1	19192.96622	1354397	1284103
4	1	10000	12251685	12311917	12155980	12307606	12219558	12183893	12125847	12129559	12228054	12249944	12216404.3	63566.13715	12311917	12125847
8	1	1	888144	890093	889833	892955	877909	878242	884605	880936	892251	888006	886297.4	5290.263079	892955	877909
8	1	10	902847	900618	908874	908522	906348	898870	897463	896952	910100	892339	902293.3	5703.236293	910100	892339
8	1	50	818194	829900	825259	826589	820050	810190	823672	825490	827784	828149	823527.7	5610.31288	829900	810190
8	1	100	733025	735678	748287	735745	740260	726846	719192	729523	747031	725995	734158.2	8799.171129	748287	719192
8	1	200	666051	668047	673709	675977	672862	666590	661829	667163	682414	664423	669906.5	5904.62426	682414	661829
8	1	500	624034	719499	691490	416946	621017	655140	530906	745615	431544	722322	615851.3	112704.9833	745615	416946
8	1	1000	1029835	708817	704332	683935	1039078	782620	739517	1032802	685331	891692	829795.9	145518.2745	1039078	683935
8	1	10000	6324698	6227898	6285506	6223875	6225769	6312067	6280342	6216166	6361475	6278409	6273620.5	47085.85599	6361475	6216166
16	1	1	481116	468795	478169	483170	475612	469802	476889	472309	483500	479020	476838.2	4955.028755	483500	468795
16	1	10	505509	489243	501691	490070	481576	490058	492941	482428	498390	487690	491959.6	7430.115708	505509	481576
16	1	50	445722	443958	451048	450285	438501	447643	452174	444444	453871	448134	447578	4349.370023	453871	438501
16	1	100	412160	402022	408073	402666	414388	407636	410400	407966	410164	401187	407666.2	4221.19421	414388	401187
16	1	200	374864	378153	376131	380667	372965	375508	366098	376344	379093	369893	374971.6	4142.839804	380667	366098
16	1	500	463232	462554	463537	452362	465384	449640	466919	458768	452099	462829	459732.4	5858.656317	466919	449640
16	1	1000	752031	776219	755813	706280	755065	764353	742997	748613	744573	719155	746509.9	19438.30568	776219	706280
16	1	10000	3235733	3380735	3275734	3338204	3369230	3336037	3290360	3364179	3333295	3353894	3327740.1	43952.35213	3380735	3235733

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	2	1	8433017	8791785	8634507	8429894	8876866	8780247	8525729	8741953	8630479	9029080	8687355.7	183937.659	9029080	8429894
1	2	10	8596820	9057924	8379991	8545326	8725285	8813825	8223867	8932061	8517516	9026698	8681931.3	263818.7116	9057924	8223867
1	2	50	8680795	8778002	8496849	8552175	8564668	8997775	8512212	9064930	8642543	8844858	8713480.7	191694.1565	9064930	8496849
1	2	100	8564687	8910475	8463632	8547868	8664465	8873822	8402226	8910365	8579276	8893901	8681071.7	188311.4439	8910475	8402226
1	2	200	8579489	8831919	8337143	8846822	8932224	8726885	8786370	8861088	8647449	8813711	8736310	166046.6723	8932224	8337143
1	2	500	8879485	8576978	8490876	8543500	9020009	9174490	8483109	8765885	8617841	8852972	8740514.5	225441.7267	9174490	8483109
1	2	1000	8518595	8710248	8480444	8694867	8932341	9002772	8621522	8654702	8624431	8643475	8688339.7	155773.5643	9002772	8480444
1	2	10000	8761771	8653969	8704930	8352424	8835379	8950705	9073842	8800017	8832637	8852153	8781782.7	182090.2554	9073842	8352424
2	2	1	7060630	7324203	7115989	7098510	7451796	7251682	7201808	7343563	7196683	7201052	7224591.6	115264.4569	7451796	7060630
2	2	10	6975717	7251471	7115831	7064328	7347739	7448430	7206231	7312460	7203755	7296746	7222270.8	133823.689	7448430	6975717
2	2	50	7125665	7161557	7121593	7170336	7186672	7239250	7106871	7378441	7152425	7297213	7194002.3	82229.91558	7378441	7106871
2	2	100	7119742	7201173	7102520	7136803	7135545	7208027	7116419	7389033	7244243	7378860	7203236.5	100359.6858	7389033	7102520
2	2	200	7144077	7366980	7105132	7213378	7343229	7248211	7154560	7244388	7200223	7317799	7233797.7	83376.46827	7366980	7105132
2	2	500	7134523	7662682	7248505	7071444	7239110	7261346	7166663	7323765	7084163	7407137	7259933.8	167142.6483	7662682	7071444
2	2	1000	7113068	7903901	7134821	7103393	7273337	7306996	7143415	7248652	7232742	7282765	7274309	221746.2858	7903901	7103393
2	2	10000	7100010	7322218	7163273	7185833	7223461	7278028	7153724	7263799	7034936	7390267	7211554.9	100721.3394	7390267	7034936
4	2	1	2634695	2667947	2713650	2719519	2804829	2726141	2733181	2654736	2674612	2663001	2699231.1	47633.28222	2804829	2634695
4	2	10	2704564	2813012	2855398	2783536	2778044	2724824	2661557	2669791	2689546	2759835	2744010.7	61103.9974	2855398	2661557
4	2	50	2638677	2813120	2649262	2612967	2678297	2679821	2654023	2854252	2661061	2659861	2690134.1	74595.93418	2854252	2612967
4	2	100	2711566	2963644	2693576	2742473	2735535	2702943	2702667	2709920	2796997	2768879	2752820	76887.76366	2963644	2693576
4	2	200	2717324	2677190	2647261	2658253	2790216	2837664	2750645	2768933	2667006	2828054	2734254.6	67456.84844	2837664	2647261
4	2	500	2818039	2748555	2711917	2695634	2737602	2770213	2678345	2754092	2634400	2744744	2729354.1	48866.43359	2818039	2634400
4	2	1000	2765423	2707614	2692440	2963174	2832205	2692842	2676295	2704036	2596053	2866489	2749657.1	102775.8509	2963174	2596053
4	2	10000	2818576	2687568	2690216	2679045	2670684	2641777	2630245	2730154	2679133	2747111	2697450.9	52310.54321	2818576	2630245
8	2	1	2800708	2976551	2885737	2828058	2796583	2801680	2825624	2788143	2775762	2828413	2830725.9	56634.34242	2976551	2775762
8	2	10	2778531	2844838	2814755	2814391	2866960	2803564	2763724	2792579	2802316	2818974	2810063.2	28489.00632	2866960	2763724
8	2	50	2792356	2834894	2864584	2830222	2795737	2830126	2813842	2770217	2824148	2834679	2819080.5	25514.63414	2864584	2770217
8	2	100	2821920	2813285	2794091	2792990	2825888	2804649	2800138	2745378	2809413	2809372	2801712.4	21362.42845	2825888	2745378
8	2	200	2786188	2808166	2808620	2798664	2771868	2809549	2813560	2857947	2818223	2805564	2807834.9	21229.8014	2857947	2771868
8	2	500	2853776	2773824	2782127	2837940	2785679	2775648	2968679	2815221	2819227	2849257	2826137.8	55481.8956	2968679	2773824
8	2	1000	2810031	2862273	2796716	2817655	2780251	2752217	2774808	2801804	2792569	2784810	2797313.4	27974.87838	2862273	2752217
8	2	10000	2808295	2783632	3049699	2828543	2798093	2788964	2773850	2782780	2810594	2820735	2824518.5	76900.84355	3049699	2773850
16	2	1	1612650	1585010	1619647	1610337	1580723	1605901	1600014	1617653	1631442	1607289	1607066.6	14621.88238	1631442	1580723
16	2	10	1627676	1597615	1613086	1598497	1615269	1615534	1573850	1623456	1625613	1603190	1609378.6	15621.2229	1627676	1573850
16	2	50	1611741	1631823	1596384	1596787	1603109	1623462	1609684	1626285	1620360	1608114	1612774.9	11692.52555	1631823	1596384
16	2	100	1653489	1585135	1618578	1587649	1601962	1590830	1598808	1618027	1608767	1675136	1613838.1	27931.00649	1675136	1585135
16	2	200	1622805	1600876	1616479	1583263	1588460	1606901	1591715	1598799	1637095	1614057	1606045	15911.0561	1637095	1583263
16	2	500	1606613	1625013	1607581	1592892	1590480	1576089	1603311	1598801	1614557	1586521	1600185.8	13584.50691	1625013	1576089
16	2	1000	1632217	1599498	1625412	1620187	1583441	1604999	1604636	1580786	1609359	1658708	1611924.3	22121.62612	1658708	1580786
16	2	10000	1616323	1601928	1630165	1588512	1603208	1611305	1594839	1617807	1629708	1615275	1610907	13115.42123	1630165	1588512

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	3	1	6108057	6171474	6091129	6122403	6138016	5988716	5962892	6118297	5957192	6005712	6066388.8	75302.05171	6171474	5957192
1	3	10	6078242	6181352	6099582	5895835	6156915	6182068	6101426	6326774	5997502	6199576	6121927.2	112156.6911	6326774	5895835
1	3	50	6155960	6145261	5962766	5953439	6258150	6343294	6207424	6244749	6018357	6235642	6152504.2	126367.0765	6343294	5953439
1	3	100	6119614	5964213	5982410	6016214	6200150	6141692	5977534	6070698	6125031	6140863	6073841.9	79214.24604	6200150	5964213
1	3	200	6181909	6087154	6041202	6027185	6224000	6040434	6066664	6351313	6043076	6239142	6130207.9	105972.9305	6351313	6027185
1	3	500	5956581	6318789	6020856	6181565	6281777	6241677	6087966	6283599	5896405	6280330	6154954.5	145634.0552	6318789	5896405
1	3	1000	6062926	6013302	6211527	6190711	6098506	6206615	6047568	6057899	5920238	6331331	6114062.3	113709.8529	6331331	5920238
1	3	10000	6198872	6143471	5994194	6162977	6293984	6171689	6247521	6240324	6059907	6077875	6159081.4	88453.17055	6293984	5994194
2	3	1	12561204	12665088	12482770	12620390	12576222	12567977	12673231	12607691	12512277	12542885	12580973.5	58523.81562	12673231	12482770
2	3	10	12492611	12572804	12603056	12573002	12609094	12573398	12485456	12569812	12435602	12583604	12549843.9	54746.18612	12609094	12435602
2	3	50	12504335	12526585	12534417	12457166	12593671	12615658	12541625	12476846	12488837	12716260	12545540	73701.21954	12716260	12457166
2	3	100	12603393	12528084	12640412	12376019	12580519	12515429	12467820	12561115	12642995	12506224	12542201	77932.4482	12642995	12376019
2	3	200	12660655	12698962	12542771	12799919	12717021	12566810	12587179	12542133	12507379	12569945	12619277.4	89882.97998	12799919	12507379
2	3	500	12401984	12672508	12495814	12517563	12569907	12550536	12526218	12472063	12456948	12563173	12522671.4	70217.69163	12672508	12401984
2	3	1000	12649142	12598632	12467651	12430150	12705114	12650307	12572661	12489260	12414740	12516748	12549440.5	95426.48288	12705114	12414740
2	3	10000	12627481	12566928	12610199	12607112	12487454	12527861	12606194	12646819	12611725	12524209	12581598.2	49598.46106	12646819	12487454
4	3	1	4510953	4130325	4267843	4074984	4122866	4269987	4171352	4168855	4047365	4343315	4210784.5	133075.2335	4510953	4047365
4	3	10	4014436	4124192	4218615	4249311	4094316	4254431	4224823	4285573	4156190	4276492	4189837.9	84715.6211	4285573	4014436
4	3	50	4402497	4174404	4187572	4218608	4431875	4064016	4306624	4332982	4331805	4222434	4267281.7	107739.9775	4431875	4064016
4	3	100	4266134	4119232	4266733	4229642	4112935	4352767	4041698	4322940	4144297	4086818	4194319.6	101445.1943	4352767	4041698
4	3	200	4195117	4271632	4291230	4232966	4320122	4345912	4120201	4238034	4218509	4143239	4237696.2	68886.20226	4345912	4120201
4	3	500	4100567	4275774	4276708	4168854	4158311	4185750	4279764	4371472	4363956	4286644	4246780	85260.69289	4371472	4100567
4	3	1000	4367160	4307870	4239488	4232800	4298654	4225581	4090938	4217232	4145477	4297200	4242240	77052.2077	4367160	4090938
4	3	10000	4159730	4093308	4256851	4281651	4446170	4251448	4213730	4285434	4496227	4173100	4265764.9	117839.5729	4496227	4093308
8	3	1	2861912	2839870	2667945	2821439	2879976	2794883	2663555	2852727	2779516	2839152	2800097.5	72864.91852	2879976	2663555
8	3	10	2823970	2853750	2876188	2819921	2880934	2667270	2854475	2835922	2715942	2867258	2819563	67714.7238	2880934	2667270
8	3	50	2838566	2868273	2770878	2845582	2833667	2859232	2794141	2851225	2839191	2875313	2837606.8	30724.32603	2875313	2770878
8	3	100	2806737	2853984	2833120	2838338	2708624	2844605	2827444	2820522	2868897	2885565	2828783.6	45632.34668	2885565	2708624
8	3	200	2617194	2840278	2854898	2767236	2810218	2797184	2765591	2814276	2804879	2870413	2794216.7	67286.75314	2870413	2617194
8	3	500	2905270	2834428	2838534	2656486	2865789	2782878	2766655	2839810	2843907	2858342	2819209.9	65808.57771	2905270	2656486
8	3	1000	2853546	2858744	2874798	2819281	2814111	2826763	2819094	2859528	2831267	2824025	2838115.7	20334.28527	2874798	2814111
8	3	10000	2881129	2827482	2865733	2551900	2898049	2818796	2826858	2836065	2808544	2847510	2816206.6	92143.47826	2898049	2551900
16	3	1	1568112	1510304	1488896	1517235	1482049	1472985	1532311	1779298	1514918	1470106	1533621.4	86649.52148	1779298	1470106
16	3	10	1496236	1471686	1507082	1494335	1471318	1521891	1465664	1540317	1502852	1489575	1496095.6	22290.72176	1540317	1465664
16	3	50	1514532	1560562	1555608	1458973	1477573	1525746	1511338	1481202	1488310	1473893	1504773.7	33003.7734	1560562	1458973
16	3	100	1504018	1500935	1569459	1520255	1510643	1476812	1489382	1476005	1502563	1466001	1501607.3	27799.21192	1569459	1466001
16	3	200	1489157	1789428	1522001	1493577	1470046	1722238	1534782	1481401	1510752	1713000	1572638.2	113567.4436	1789428	1470046
16	3	500	1526252	1508098	1516537	1474321	1494488	1449576	1479277	1511477	1510131	1463729	1493388.6	24089.14942	1526252	1449576
16	3	1000	1497003	1477862	1491723	1504365	1488341	1714961	1479282	1473801	1492785	1511108	1513123.1	68190.69459	1714961	1473801
16	3	10000	1726835	1471969	1518976	1472520	1489211	1475033	1808581	1707147	1493672	1531234	1569517.8	120380.7797	1808581	1471969

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	4	1	2953999	2952027	2913160	2951333	2918955	2945407	2873661	2957525	3016574	2931716	2941435.7	35044.47064	3016574	2873661
1	4	10	2952131	2916854	2938337	2912658	2966762	2963242	2941676	2926076	2938343	2934969	2939104.8	17048.09007	2966762	2912658
1	4	50	2634061	2668852	2669933	2730635	2711707	2645564	2637902	2642410	2640346	2627581	2660899.1	33087.89274	2730635	2627581
1	4	100	2348347	2333686	2404313	2389311	2379563	2400828	2348942	2378486	2393003	2382041	2375852	22862.17063	2404313	2333686
1	4	200	2175294	2133597	2138867	2179567	2203233	2170014	2151526	2126229	2200422	2149833	2162858.2	25686.12568	2203233	2126229
1	4	500	2705992	2699193	2733605	2700337	2950657	2761853	2692322	2756525	2750351	2765313	2751614.8	71578.52531	2950657	2692322
1	4	1000	4990580	4925654	4892948	4982915	5794326	4875733	4894180	4871655	4907370	5044174	5017953.5	264337.4477	5794326	4871655
1	4	10000	46430460	47115957	46367130	47128034	48392597	46734747	46834432	46871949	46235961	46915770	46902703.7	574840.7558	48392597	46235961
2	4	1	3485741	1520228	1515613	1613282	1528673	1526056	3483961	1538672	1562272	3484198	2125869.6	889918.1334	3485741	1515613
2	4	10	2723068	1564471	1556074	3499147	1566164	3472670	3514826	1507543	1543006	1501347	2244831.6	889341.3507	3514826	1501347
2	4	50	3138364	3132305	3175248	3137768	1369860	3170370	1912354	3154046	3160926	1410267	2676150.8	740526.3389	3175248	1369860
2	4	100	1264318	1255308	2827304	2902487	2851536	2848350	2860910	2793981	1310423	1299290	2221390.7	767298.1516	2902487	1255308
2	4	200	1162486	1258671	1157987	1218166	1158876	1188134	1167617	1161519	1169029	1166643	1180912.8	31200.63108	1258671	1157987
2	4	500	1509398	1426033	1502026	1500564	1501893	1484768	1483433	1495901	1480969	1456164	1484114.9	24189.1229	1509398	1426033
2	4	1000	2647874	2607632	2600193	2644012	2589586	2599004	2598600	2603228	2621439	2564907	2607647.5	23484.86938	2647874	2564907
2	4	10000	27308444	24254047	24213542	24031760	24220647	24056743	24282888	24029226	24136975	24216938	24475121	948586.3012	27308444	24029226
4	4	1	1772848	1805603	1782864	1780111	1793364	1773701	1766818	1792234	1794311	1798129	1785998.3	11959.34212	1805603	1766818
4	4	10	1808952	1812418	1806180	1815983	1804520	1806594	1809083	1822880	1815972	1812698	1811528	5341.314351	1822880	1804520
4	4	50	1679777	1577849	1678685	1641493	1630517	1658950	1625414	1634983	1672122	1637566	1643735.6	29204.22306	1679777	1577849
4	4	100	1521624	815088	1475560	1493170	1478599	1502362	1504330	1502601	1484938	1481177	1425944.9	204069.7661	1521624	815088
4	4	200	1346671	988861	835813	796050	737279	924145	1333893	951302	1326378	1171061	1041145.3	222788.0778	1346671	737279
4	4	500	929454	950357	913233	975831	1018124	1048566	985447	913605	875538	1052563	966271.8	57266.55537	1052563	875538
4	4	1000	1447203	1445898	1417631	1430769	1418516	1425924	1454235	1438489	1428347	1425013	1433202.5	11980.02516	1454235	1417631
4	4	10000	12403371	12443853	12354592	12294268	12355013	12327496	12355117	12356954	12209313	12399156	12349913.3	61318.15609	12443853	12209313
8	4	1	983749	982490	976812	986577	974089	980387	982641	969135	991594	991852	981932.6	6865.344	991852	969135
8	4	10	1005066	1003169	990430	998184	995277	986574	988285	1003205	1010915	995454	997655.9	7520.122478	1010915	986574
8	4	50	922201	947347	910362	919448	921146	909134	914346	909842	924181	911064	918907.1	10889.55033	947347	909134
8	4	100	839919	840053	837037	834902	841129	828535	822783	849135	836866	840863	837122.2	6892.908817	849135	822783
8	4	200	774747	771952	770785	771685	763709	752336	770971	778515	760099	772734	768753.3	7381.079556	778515	752336
8	4	500	822786	798408	825997	813686	819241	758607	796529	849209	807119	772926	806450.8	25039.18998	849209	758607
8	4	1000	1050430	1059031	1085020	1060572	1063162	1058878	1068106	1053076	1031344	1046157	1057577.6	13408.47625	1085020	1031344
8	4	10000	6424702	6424520	6401631	6377851	6441544	6445345	6417516	6403394	6397822	6422135	6415646	19625.97454	6445345	6377851
16	4	1	611870	605633	608478	606931	618762	607398	602673	609776	604665	611346	608753.2	4316.362191	618762	602673
16	4	10	622583	615501	617166	614270	613810	618252	615430	610621	622139	622413	617218.5	3886.257306	622583	610621
16	4	50	572511	570423	564079	568082	575980	570096	566469	571901	569870	582369	571178	4860.088621	582369	564079
16	4	100	541846	538919	557674	527653	536599	527645	531422	534781	539626	541126	537729.1	8272.035523	557674	527645
16	4	200	502656	497477	502513	502338	506649	506611	504444	503670	497488	505221	502906.7	3088.911978	506649	497477
16	4	500	575584	578777	563326	572574	566936	576758	561467	579390	584937	584399	574414.8	7816.443984	584937	561467
16	4	1000	853068	850279	855300	853252	840448	846417	840466	862754	838267	864020	850427.1	8591.832045	864020	838267
16	4	10000	3463589	3451731	3452881	3445520	3461860	3418994	3442166	3462217	3447173	3598294	3464442.5	46305.43211	3598294	3418994

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	5	1	175122	179349	177843	182892	195885	179074	183716	186310	177840	187557	182558.8	5833.950031	195885	175122
1	5	10	189369	179487	187022	185852	196373	184199	184194	181200	185477	189528	186270.1	4516.313285	196373	179487
1	5	50	173342	184919	175911	176201	203772	182563	184464	181815	188464	179741	183119.2	8179.172401	203772	173342
1	5	100	183119	183735	185253	175309	201941	188672	186145	184856	182699	183542	185527.1	6368.459177	201941	175309
1	5	200	181177	185138	184185	176480	197566	179460	178587	178435	186903	181784	182971.5	5775.176971	197566	176480
1	5	500	177878	180434	180757	177679	195200	190575	184423	185317	183635	180968	183686.6	5287.678379	195200	177679
1	5	1000	186026	184600	175638	184780	196299	187750	185799	181466	181975	179440	184377.3	5229.266202	196299	175638
1	5	10000	187404	182793	180066	182213	198204	178937	182328	179893	186723	182689	184125	5364.15518	198204	178937
2	5	1	392412	264868	263030	391941	288558	249136	346730	266903	283844	329930	307735.2	51136.89643	392412	249136
2	5	10	262895	348842	252540	338836	343471	383394	302103	263163	320902	254155	307030.1	44466.97348	383394	252540
2	5	50	264590	348563	256767	290459	375823	264156	246148	283010	387901	243841	296125.8	51502.71695	387901	243841
2	5	100	273633	384697	385356	292708	400089	339021	349070	261945	299791	257562	324387.2	51437.13354	400089	257562
2	5	200	303926	357634	278620	361399	261470	294208	381571	257208	266814	248369	301121.9	46175.32746	381571	248369
2	5	500	340327	267998	251220	257049	252545	262560	252416	341270	296534	269576	279149.5	33295.72307	341270	251220
2	5	1000	267261	396535	352730	296722	331553	291831	383185	388321	248523	257173	321383.4	53736.62593	396535	248523
2	5	10000	391373	272931	364366	262094	322569	278399	273394	343734	256396	256038	302129.4	46953.946	391373	256038
4	5	1	190365	190564	186498	146925	191509	195376	180068	185104	183513	180528	183045	12923.02035	195376	146925
4	5	10	167712	224781	182388	215605	234689	151813	210494	178532	176541	178006	192056.1	25902.95247	234689	151813
4	5	50	181114	190931	136134	175212	171114	195660	168596	143793	199184	189440	175117.8	20129.21758	199184	136134
4	5	100	232232	147707	219599	228476	234667	139281	210914	194430	172583	213459	199334.8	33069.03817	234667	139281
4	5	200	192709	196217	228234	147556	165502	188895	231088	164807	227631	232912	197555.1	29866.94197	232912	147556
4	5	500	176924	187899	161427	233906	197286	185604	213798	167775	228830	172098	192554.7	24114.81032	233906	161427
4	5	1000	184342	189284	233894	159480	167241	175774	173440	172758	174260	188220	181869.3	19422.62534	233894	159480
4	5	10000	190026	231976	168398	166703	191219	155022	215032	235595	196204	230257	198043.2	27774.1217	235595	155022
8	5	1	202631	129326	113544	191013	211208	176706	168759	223925	121464	116070	165464.6	40073.66668	223925	113544
8	5	10	229862	208185	182099	225550	202368	195365	194157	122466	202784	211321	197415.7	28414.54497	229862	122466
8	5	50	205593	196443	185776	184587	197252	179567	226183	226911	120022	230154	195248.8	30657.18877	230154	120022
8	5	100	234195	190164	168914	191063	170822	116529	201034	227598	226006	224036	195036.1	34498.36934	234195	116529
8	5	200	224854	121148	229338	120931	197673	184482	227520	207630	192333	130822	183673.1	41458.98783	229338	120931
8	5	500	226808	197212	215150	198988	129919	234067	121422	189462	121619	230118	186476.5	43053.77056	234067	121422
8	5	1000	207878	178813	235610	190459	205334	208239	121728	193036	114877	185196	184117	36171.50432	235610	114877
8	5	10000	204318	185890	230713	157636	186275	182944	232590	193151	199389	192383	196528.9	21170.59378	232590	157636
16	5	1	138755	136170	150628	149298	145523	145018	104760	150210	143780	145504	140964.6	12855.69738	150628	104760
16	5	10	138924	141827	146887	150416	144895	140430	145118	138089	144150	138909	142964.5	3807.11805	150416	138089
16	5	50	147271	144612	140763	145583	144796	144545	141070	143049	143621	139918	143522.8	2213.755894	147271	139918
16	5	100	143722	147797	143296	140753	149240	138902	144565	144617	141225	145996	144011.3	3022.641231	149240	138902
16	5	200	147410	148097	146036	141386	147413	146711	143780	149809	145112	140226	145598	2865.769565	149809	140226
16	5	500	144882	143400	151734	147839	145615	143391	147035	137487	145106	144696	145118.5	3456.838592	151734	137487
16	5	1000	141851	141963	149455	144766	149702	146916	148345	146860	141780	140443	145208.1	3319.208172	149702	140443
16	5	10000	149890	141408	151503	144939	144457	158378	141002	142858	141320	144439	146019.4	5314.231388	158378	141002

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	6	1	266527	263751	253932	268955	297189	336392	277126	258171	368856	256920	284781.9	36615.26777	368856	253932
1	6	10	338854	249154	298436	261253	303515	277542	260597	261134	279244	275217	280494.6	25335.75897	338854	249154
1	6	50	342229	267281	242946	390670	296528	270387	319139	263205	392292	270519	305519.6	51011.24376	392292	242946
1	6	100	386169	281333	249287	255412	307307	261678	305461	279586	272134	263372	286173.9	38090.00832	386169	249287
1	6	200	238531	286459	244723	428524	318380	275114	257661	284562	253003	270492	285744.9	52508.26927	428524	238531
1	6	500	252121	351387	281189	341247	287709	276277	255561	279047	239951	389147	295363.6	46318.4644	389147	239951
1	6	1000	263784	258146	257501	263053	312370	362241	237594	287049	273630	269461	278482.9	33648.1047	362241	237594
1	6	10000	389731	262937	270759	240321	293652	278731	382157	280260	255034	275564	292914.6	48570.64493	389731	240321
2	6	1	208670	207163	201357	183666	219813	180355	173278	249926	208193	206587	203900.8	20829.35822	249926	173278
2	6	10	204268	191887	206282	193924	213138	185056	202266	188242	196912	188536	197051.1	8681.77586	213138	185056
2	6	50	216151	206080	214346	199356	205274	188057	197846	206504	198698	195331	202764.3	8176.288462	216151	188057
2	6	100	203276	186440	201661	210621	191401	192098	189106	177461	219616	204045	197572.5	11897.32078	219616	177461
2	6	200	214036	210386	195891	212132	196253	201959	228633	185425	181124	186138	201197.7	14329.05499	228633	181124
2	6	500	205319	171270	211409	202610	203838	180285	200494	207349	200538	189843	197295.5	12160.00098	211409	171270
2	6	1000	214223	204554	201222	173383	212373	259066	176813	188846	217413	185154	203304.7	23772.52332	259066	173383
2	6	10000	194483	199916	171862	206874	207380	193608	215529	187660	205922	256777	204001.1	21111.13586	256777	171862
4	6	1	133396	114483	103268	99746	95550	102706	96714	103428	105577	108200	106306.8	10421.39159	133396	95550
4	6	10	99189	97552	102965	101945	129047	109333	102891	103835	103622	96758	104713.7	8800.9241	129047	96758
4	6	50	89034	102999	107681	94384	100476	103913	95157	103636	99211	104089	100058	5378.721651	107681	89034
4	6	100	99160	106146	103762	112905	106464	101621	128504	120475	138367	96122	111352.6	13013.59085	138367	96122
4	6	200	94179	94852	106315	99473	103032	113244	96986	98529	100648	100506	100776.4	5394.770305	113244	94179
4	6	500	100386	102493	99769	103711	97549	115855	130587	133028	97072	98672	107912.2	12993.45368	133028	97072
4	6	1000	105523	107291	101028	96998	107202	104161	106794	92392	106434	112479	104030.2	5492.27184	112479	92392
4	6	10000	96754	97654	97843	117306	101016	108834	104573	100079	106889	103216	103416.4	6010.52399	117306	96754
8	6	1	90472	89308	82364	84603	90497	83742	86979	76849	84718	86921	85645.3	3966.966248	90497	76849
8	6	10	84239	82797	82828	88289	90087	91699	81337	83099	85492	80119	84998.6	3644.436587	91699	80119
8	6	50	83791	91244	78627	83307	79440	77288	83532	82036	83503	80886	82365.4	3674.429104	91244	77288
8	6	100	81252	85670	83498	83317	78194	87478	87033	87253	81565	86434	84169.4	2970.062834	87478	78194
8	6	200	83925	87094	87065	90151	87751	88755	81067	86504	87540	82594	86244.6	2692.325434	90151	81067
8	6	500	84353	81509	82818	85960	86186	85122	82560	88635	79224	87282	84364.9	2709.48148	88635	79224
8	6	1000	85915	75272	84231	83399	91123	83925	80592	89224	88256	82175	84411.2	4355.002544	91123	75272
8	6	10000	87102	74439	85222	85754	88535	84626	84173	94097	86240	80479	85066.7	4843.334761	94097	74439
16	6	1	55437	54923	53550	53992	53997	55656	53685	56580	54395	53503	54571.8	987.4671438	56580	53503
16	6	10	58001	57052	54192	55242	54904	57704	57241	53355	51620	55147	55445.8	1960.502578	58001	51620
16	6	50	56090	53083	52884	54071	58333	57852	54329	55366	58015	57075	55709.8	1959.28981	58333	52884
16	6	100	55432	54507	54968	53666	56198	55466	56095	55327	56033	53532	55122.4	907.7919585	56198	53532
16	6	200	54165	60928	53656	54633	55869	56175	56818	54810	55662	57294	56001	1973.248945	60928	53656
16	6	500	55575	56525	52136	55913	56254	56784	53491	56470	56950	56880	55697.8	1528.172752	56950	52136
16	6	1000	54896	52218	53811	53255	54435	53631	53267	56915	56055	56455	54493.8	1474.471146	56915	52218
16	6	10000	57180	52087	53749	52440	56014	54879	55965	54369	57670	55760	55011.3	1772.964075	57670	52087

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	7	1	173878	175105	171672	175941	196640	174587	167980	176084	164785	181033	175770.5	8166.720336	196640	164785
1	7	10	181937	180451	169008	176888	198577	169081	181188	180991	172277	179550	178994.8	8068.022382	198577	169008
1	7	50	169544	183529	181957	177564	193398	174688	188026	166248	186304	168214	178947.2	8719.91963	193398	166248
1	7	100	178189	171278	187240	166549	202605	172057	168817	187642	181317	183892	179958.6	10426.41811	202605	166549
1	7	200	182306	181348	181050	168941	174503	180649	167056	177622	178657	167073	175920.5	5804.02211	182306	167056
1	7	500	182073	179361	173606	175297	178895	167344	182672	169973	170109	166875	174620.5	5634.625742	182672	166875
1	7	1000	168188	170696	171093	169581	179464	174515	170573	166632	171269	178350	172036.1	3957.752808	179464	166632
1	7	10000	170179	180924	178929	172078	180195	176837	174820	181889	167493	167520	175086.4	5219.14798	181889	167493
2	7	1	278837	292957	272476	254707	277088	287311	286568	266060	254730	287522	275825.6	12984.11631	292957	254707
2	7	10	249078	275033	261667	280611	261782	281124	293508	261322	278801	268030	271095.6	12384.02656	293508	249078
2	7	50	283262	287471	266959	277495	285687	293415	259660	264640	290647	289825	279906.1	11465.38114	293415	259660
2	7	100	283059	258774	285862	287379	285325	279898	295522	296283	282967	265058	282012.7	11290.28203	296283	258774
2	7	200	258172	266091	293365	274681	266672	276531	267788	273456	261352	269929	270803.7	9283.947889	293365	258172
2	7	500	274820	253160	288890	287957	273890	291930	292539	285879	292523	265975	280756.3	12683.91252	292539	253160
2	7	1000	279620	290907	284915	288925	277433	259133	292499	277339	250762	248645	275017.8	15549.98553	292499	248645
2	7	10000	273957	281538	286111	285169	261172	282822	256472	286955	283831	291645	278967.2	10986.16439	291645	256472
4	7	1	138094	135336	138560	130440	131083	133771	152489	135128	126427	132106	135343.4	6682.664861	152489	126427
4	7	10	133572	151410	124251	119876	157970	129782	145830	144075	126475	136991	137023.2	11856.65608	157970	119876
4	7	50	129014	137154	161505	146265	136152	149490	127990	143471	155912	139459	142641.2	10370.08851	161505	127990
4	7	100	121378	124352	130862	127770	138240	121089	129630	138377	136720	146246	131466.4	7855.93348	146246	121089
4	7	200	148508	155998	126438	149026	156935	145299	146878	137724	137528	122178	142651.2	11020.54957	156935	122178
4	7	500	158368	154397	125034	141545	125582	135153	133811	143244	147242	131094	139547	10867.08753	158368	125034
4	7	1000	131526	130208	153466	130291	135757	126021	134313	128787	135206	141407	134698.2	7474.288311	153466	126021
4	7	10000	144706	146983	139541	126854	144493	158847	123244	130194	140490	141988	139734	9998.695095	158847	123244
8	7	1	88903	86414	88832	88790	89662	87277	86184	89869	84313	90163	88040.7	1816.01465	90163	84313
8	7	10	85922	91341	91727	93532	86420	83444	86160	85003	89335	91638	88452.2	3294.958021	93532	83444
8	7	50	88949	87521	94208	89476	88192	88733	87605	88538	92494	81713	88742.9	3115.620402	94208	81713
8	7	100	92928	85385	87332	87022	94197	84938	88793	90667	89249	89555	89006.6	2861.679549	94197	84938
8	7	200	86784	88075	86963	88958	87493	87004	86591	83976	88769	92759	87737.2	2131.974193	92759	83976
8	7	500	84183	85104	88233	86457	91962	89904	88645	82986	88723	88442	87463.9	2609.873271	91962	82986
8	7	1000	85925	90807	89758	86518	89494	85760	88200	88978	89360	88124	88292.4	1634.866368	90807	85760
8	7	10000	89144	86812	88958	86941	88573	90812	89108	89445	86320	86751	88286.4	1409.710552	90812	86320
16	7	1	57946	53872	55804	56069	58029	57203	58696	54990	55828	56020	56445.7	1423.184531	58696	53872
16	7	10	57045	62578	54613	54341	57801	60650	57876	56675	59745	58038	57936.2	2420.103832	62578	54341
16	7	50	58897	57288	57507	56927	55347	56494	60738	58038	57425	56544	57520.5	1402.206208	60738	55347
16	7	100	56185	58385	57860	56556	59902	54700	55961	55213	55336	58711	56880.9	1644.939175	59902	54700
16	7	200	57404	58888	57501	55990	59554	53245	55207	57525	56044	58505	56986.3	1796.924041	59554	53245
16	7	500	57719	60959	58408	56846	57914	57957	56801	57932	56417	57110	57806.3	1212.727673	60959	56417
16	7	1000	55407	57381	56727	57858	59226	68419	55727	60670	56166	62047	58962.8	3762.625355	68419	55407
16	7	10000	54886	58951	59261	54830	56200	58152	58765	55893	59391	58778	57510.7	1751.40561	59391	54830

H Agents Baseline Results: Max Time

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	1	1	83729	87869	84986	85264	87347	84485	85459	86696	85322	84232	85538.9	1287.113084	87869	83729
1	1	50	2491648	2469067	2461653	2455614	2449545	2480698	2427105	2442246	2479706	2483909	2464119.1	19573.12555	2491648	2427105
1	1	100	4945004	4891136	4918102	4938998	4908130	4890310	4939213	4936585	4880641	4892814	4914093.3	23299.7209	4945004	4880641
1	1	150	7296363	7394902	7362228	7253844	7393965	7210377	7182291	7392601	7354713	7342422	7318370.6	74654.35281	7394902	7182291
1	1	200	9666574	9665222	9673648	9602849	9745026	9805257	9677283	9799045	9862486	9696441	9719383.1	76517.46423	9862486	9602849
1	1	250	12086676	12088166	12152259	12255791	12147575	11991417	12095367	12156836	12218456	12428689	12162123.2	113181.367	12428689	11991417
2	1	1	57559	50744	105084	47777	98300	52512	103310	105279	54626	106734	78192.5	25740.59237	106734	47777
2	1	50	1289772	1298965	1219422	1246577	2867124	1254988	1269974	1262511	1299291	2894060	1590268.4	645610.8392	2894060	1219422
2	1	100	2539056	5882039	2450179	2460153	2422005	5757433	2504932	5657390	2495724	2503560	3467247.1	1505796.581	5882039	2422005
2	1	150	3753165	3716415	8503303	3797517	3691694	3700535	3680457	3658051	5046071	4866775	4441398.3	1440940.416	8503303	3658051
2	1	200	11299345	4936131	10970429	4929345	11422048	8009067	4835283	11339700	11601913	8454780	8779804.1	2801865.87	11601913	4835283
2	1	250	6574104	6057560	6183568	7300810	14302170	6048571	11319754	6188631	6217677	9864599	8005744.4	2721669.805	14302170	6048571
4	1	1	62456	56413	52517	54631	64840	32096	56989	57838	53493	58430	54970.3	8434.134598	64840	32096
4	1	50	632910	1473222	1449985	1463752	1463115	1439117	1476827	1478471	1464880	1450255	1379253.4	249069.4363	1478471	632910
4	1	100	2862823	2885928	2898025	2909939	2869183	2878810	2917483	2880623	2880322	2884464	2886760	16242.7733	2917483	2862823
4	1	150	4340188	4310903	4283687	4347538	4346950	4278844	4281922	4337038	4319290	4340871	4318723.1	26734.67886	4347538	4278844
4	1	200	5682898	5751378	5782658	5770229	5809366	3715544	5789644	5812562	5749210	5817824	5568131.3	618695.4018	5817824	3715544
4	1	250	7112197	7214140	7189925	7169062	7161481	7193969	7206303	7300328	7232856	7140132	7192039.3	49622.03171	7300328	7112197
8	1	1	36323	41029	39221	37004	36770	37326	37509	36994	36889	36909	37597.4	1359.928469	41029	36323
8	1	50	769121	764020	757155	758396	756379	762131	767369	763693	761699	754996	761495.9	4485.114747	769121	754996
8	1	100	1503374	1497920	1500932	1497294	1510114	1516461	1516354	1518591	1509302	1488669	1505901.1	9364.434862	1518591	1488669
8	1	150	2278155	2236646	2241385	2240009	2220184	2255686	2226561	2249080	2242293	2250143	2244014.2	15224.1568	2278155	2220184
8	1	200	3040361	3016500	2990847	2979835	2952943	3003687	2979594	2936759	2983681	2956786	2984099.3	29395.38406	3040361	2936759
8	1	250	3734526	3736498	3741734	3681358	3719273	3745932	3704964	3719258	3669607	3736694	3718984.4	24811.7276	3745932	3669607
16	1	1	22748	22770	22046	22938	22430	21935	21246	24367	24256	21798	22653.4	962.0664426	24367	21246
16	1	50	422698	414138	422382	415565	416569	414584	416496	412494	414983	418013	416792.2	3208.222492	422698	412494
16	1	100	822923	816696	831866	815560	823940	810207	818003	830388	800930	826129	819664.2	8963.274109	831866	800930
16	1	150	1211005	1222085	1207197	1223647	1229455	1216053	1207898	1216917	1215145	1233578	1218298	8380.804639	1233578	1207197
16	1	200	1618390	1623276	1646295	1617505	1614725	1624260	1628917	1605248	1591875	1608996	1617948.7	13871.22658	1646295	1591875
16	1	250	2009619	2042934	2028567	2017880	2008163	2000004	1999691	2004974	2006378	2021526	2013973.6	13093.98617	2042934	1999691

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	2	1	226340	332028	215240	248392	251932	224256	235797	220390	221830	238326	241453.1	32316.45293	332028	215240
1	2	50	6941208	7277523	7249263	7016571	7022094	7168741	7234250	7295789	7307210	7020209	7153285.8	131920.0682	7307210	6941208
1	2	100	14038239	13905766	14056470	13933516	13846721	14082367	13959914	14059772	14086029	14490683	14045947.7	167471.9346	14490683	13846721
1	2	150	21049279	20590125	20742059	20836516	20656595	20322752	20939171	20551227	21207727	21109291	20800474.2	265021.7565	21207727	20322752
1	2	200	27595296	27648211	27447079	27488941	27062808	27338943	27611636	27611216	28549246	28101090	27645446.6	390711.8863	28549246	27062808
1	2	250	34831661	34071540	34602968	34032333	34171406	34991946	34284655	34877008	35671108	36124984	34765960.9	660764.72	36124984	34032333
2	2	1	180718	167417	193135	151753	153383	190696	158659	152250	180209	182899	171111.9	15479.3514	193135	151753
2	2	50	5962096	5999438	6077118	6018260	5988557	5992166	6171627	5944011	6035885	6127374	6031653.2	69277.10303	6171627	5944011
2	2	100	11818199	11765831	11960936	11833995	11696608	11759062	12478442	11909419	11932010	11830750	11898525.2	208441.0523	12478442	11696608
2	2	150	17963276	18002340	17644057	18056763	17681872	17679047	17708772	17713169	17744557	18428547	17862240	237001.9177	18428547	17644057
2	2	200	23665584	23730305	23741841	23508182	23694669	23479418	23832577	23622286	23712244	23528907	23651601.3	109062.836	23832577	23479418
2	2	250	29788616	29615139	31637116	29499744	29676344	29789063	29696762	30839608	30131611	29553850	30022785.3	654257.4516	31637116	29499744
4	2	1	81884	90311	85923	80142	83020	84584	83463	81791	86184	117920	87522.2	10491.11578	117920	80142
4	2	50	2248104	2202926	2296238	2148155	2307824	2505300	2221626	2275035	2250721	2228245	2268417.4	90424.13764	2505300	2148155
4	2	100	4467453	4370759	4364577	4502706	4458980	4681267	4487180	4402118	4492805	4465583	4469342.8	84989.71553	4681267	4364577
4	2	150	6666894	6450275	6400763	6568047	6637004	7103942	6669603	6488247	6553077	6556779	6609463.1	185322.2128	7103942	6400763
4	2	200	8994465	8722035	8991007	8862433	8786702	9373481	8667335	8593000	8667040	8839746	8849724.4	216406.0937	9373481	8593000
4	2	250	10824296	10869616	10870072	10968421	10713762	11429665	10953659	11028166	11156147	10844037	10965784.1	192933.7925	11429665	10713762
8	2	1	75979	79034	81374	79422	78188	79964	75241	75087	81215	79251	78475.5	2195.22369	81374	75087
8	2	50	2733931	2719378	2736358	2702971	2685082	2697711	2675730	2911739	2712216	2706782	2728189.8	63840.70164	2911739	2675730
8	2	100	5427547	5305462	5286538	5273746	5346654	5184493	5316512	5330515	5229115	5247378	5294796	64353.54279	5427547	5184493
8	2	150	7844496	7816282	7857863	7873163	7879133	7882865	7943112	7906347	8099656	7829169	7893208.6	77221.39309	8099656	7816282
8	2	200	10583513	10445424	10467621	10562482	10520478	10554817	10685199	10534012	10623942	10488408	10546589.6	68974.76653	10685199	10445424
8	2	250	13199800	13233871	13326412	13165543	13122887	13227745	13277932	13017641	13185907	13105059	13186279.7	84669.30042	13326412	13017641
16	2	1	45897	46018	44119	44833	59157	44469	48860	59961	53312	42325	48895.1	6054.7968	59961	42325
16	2	50	1550377	1550024	1950114	1888280	1611232	1486329	1753553	1824387	1704015	1840432	1715874.3	152308.4537	1950114	1486329
16	2	100	3099510	3093401	3834990	3596618	3444143	3252729	3717881	3874080	4197400	3546348	3565710	338271.5048	4197400	3093401
16	2	150	4595189	4612988	5521319	5523082	5113514	4753417	5488533	5627867	5387383	5506475	5212976.7	389692.2305	5627867	4595189
16	2	200	6151142	6135763	7468431	7569110	6807586	6292304	7236285	7502764	7350461	7033523	6954736.9	544122.9207	7569110	6135763
16	2	250	7642209	7657722	9576014	9581963	8381523	7462618	9735619	9479878	8824831	9425216	8776759.3	868699.2481	9735619	7462618

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	3	1	131833	133931	141633	142315	131116	133169	141725	143148	137955	141809	137863.4	4602.564724	143148	131116
1	3	50	5036753	4918326	5007057	4854801	5072195	5114766	5010305	4985213	4994807	4951275	4994549.8	70686.79041	5114766	4854801
1	3	100	10187490	9810802	10099507	9835677	10033737	10182691	10081793	10114731	9959975	10080907	10038731	124646.1384	10187490	9810802
1	3	150	14414935	15250119	14918901	15101544	15230637	15519340	15215613	15159414	15048880	15205685	15106506.8	273462.363	15519340	14414935
1	3	200	20211886	20049308	20099091	19754763	19763249	20215027	20154841	20430364	20092667	20037267	20080846.3	193230.0398	20430364	19754763
1	3	250	24929257	24406910	26714839	24971956	24699247	25537121	24856845	25636007	25029962	25163481	25194562.5	612474.2664	26714839	24406910
2	3	1	257409	270158	255167	267755	265635	239372	259292	237980	264980	262766	258051.4	10632.62041	270158	237980
2	3	50	10621277	10594463	10656060	10618150	10578868	10600457	10624064	10436727	10477981	10600110	10580815.7	65458.16136	10656060	10436727
2	3	100	20950459	21044291	21164460	20993532	21091645	21119910	20855546	20805244	20986533	20982857	20999447.7	106560.4026	21164460	20805244
2	3	150	31402331	31339110	31379577	31677126	31517990	31856263	31174151	31774219	32002049	31410504	31553332	249307.13	32002049	31174151
2	3	200	41877214	42268603	42068061	41776238	41487814	41991096	42094385	41859336	42309118	41883886	41961575.1	229631.4535	42309118	41487814
2	3	250	52073644	52453827	52888752	52601495	52587369	52936714	52343977	52171503	52684727	52371117	52511312.5	269332.8458	52936714	52073644
4	3	1	127344	108791	110774	109169	108844	100819	124118	114079	107661	104049	111564.8	7883.833126	127344	100819
4	3	50	3609973	3468572	3546464	3510898	3674701	3527274	3545403	3554839	3362012	3578434	3537857	79090.81137	3674701	3362012
4	3	100	7000502	6978116	7178387	6931481	7007804	6997624	7064420	6880257	6928979	7157784	7012535.4	91607.43278	7178387	6880257
4	3	150	10333230	10343470	10560789	10663510	10216857	10252368	10211414	10520177	10213940	10413988	10372974.3	153557.2715	10663510	10211414
4	3	200	13683220	13839360	13884030	14053064	13939124	13544895	14255613	13768864	13802662	13616521	13838735.3	199485.164	14255613	13544895
4	3	250	17378956	17326316	17381579	17663094	17534686	16940685	17675712	17405203	17185942	17337695	17382986.8	206734.8506	17675712	16940685
8	3	1	88963	90331	87172	87605	90984	86266	90214	91271	89314	87888	89000.8	1625.974219	91271	86266
8	3	50	3045125	2992632	2980162	3014059	3051818	2955891	2985527	3041685	3023396	3064359	3015465.4	33995.93867	3064359	2955891
8	3	100	6053570	5945871	6017855	6004196	6010629	5817903	5970316	5945285	5709319	5965995	5944093.9	98833.25793	6053570	5709319
8	3	150	8813122	9051914	8948821	8919155	9004237	8615108	8982818	8864433	8539655	8926311	8866557.4	159023.7101	9051914	8539655
8	3	200	11860973	11952355	12303467	11767437	11937623	11407661	11882852	11803110	11430001	11876094	11822157.3	244743.4016	12303467	11407661
8	3	250	14951452	14951375	14727609	14874887	14997562	14264545	14833657	14860765	14301701	15125585	14788913.8	272093.191	15125585	14264545
16	3	1	49971	53144	57421	61252	84902	48534	63105	61452	44881	52703	57736.5	10722.53915	84902	44881
16	3	50	1764924	1730027	1886350	2125934	2097085	1668261	1938778	1781245	1973740	2208054	1917439.8	174302.6604	2208054	1668261
16	3	100	3465107	3455682	3831135	3715970	3659219	3322231	3774612	3742358	3794050	4113916	3687428	215091.2363	4113916	3322231
16	3	150	5208812	5120871	5608209	5714687	5507337	4932376	5720963	5930184	5870673	5728160	5534227.2	319406.1913	5930184	4932376
16	3	200	6842011	6869383	7489403	7916487	7259701	6821350	7608381	7633979	7589250	7496739	7352668.4	366485.6724	7916487	6821350
16	3	250	8964751	8675176	9498918	9620038	9083676	8229625	9431101	9775735	9555739	9495150	9232990.9	462534.6857	9775735	8229625

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	4	1	86304	85085	84240	82509	83972	84486	84633	85039	86434	85009	84771.1	1071.447474	86434	82509
1	4	50	2421281	2486953	2382114	2470148	2471737	2462339	2487446	2458621	2370585	2447600	2445882.4	39305.19641	2487446	2370585
1	4	100	4961091	4854867	4858291	4794346	4663168	4921269	4837366	4778882	4893351	4858700	4842133.1	78784.50284	4961091	4663168
1	4	150	7363262	7339224	7480150	7335908	7179090	7228562	7293672	7343388	7294978	7221283	7307951.7	81558.86163	7480150	7179090
1	4	200	9827127	9636042	9447860	9697352	9756564	9529767	9797972	9529384	9680688	9706276	9660903.2	118257.0623	9827127	9447860
1	4	250	12233625	12069182	12079499	12078521	12274742	12172492	12079871	12236175	12073539	12195605	12149325.1	77418.69137	12274742	12069182
2	4	1	71291	106775	110052	116902	107486	59958	75581	79156	59189	63000	84939	21709.8825	116902	59189
2	4	50	2915397	2917639	1311131	1305168	1326986	1319702	2952190	1299775	2978785	1295520	1962229.3	799382.5716	2978785	1295520
2	4	100	2622725	2553123	2552961	5862385	2507894	2557393	2590517	2587285	2505246	4307194	3064672.3	1068780.572	5862385	2505246
2	4	150	3789107	8705599	3803326	8727473	8648460	3843587	8747035	3776220	3795434	8180369	6201661	2404913.808	8747035	3776220
2	4	200	8162531	6446362	8588547	11484729	5134870	5071523	11638130	5097298	7677852	5113263	7441510.5	2422694.203	11638130	5071523
2	4	250	10534881	9493911	9401056	6327349	6293071	6342685	7453032	6303996	9611424	14409450	8617085.5	2492464.758	14409450	6293071
4	4	1	58475	63387	62565	63161	63588	50476	63114	61249	60775	69767	61655.7	4623.714374	69767	50476
4	4	50	1518993	1501447	1492592	1525547	1513335	1523569	1511653	1506410	1529962	1529674	1515318.2	11884.23844	1529962	1492592
4	4	100	3036082	2987378	3001638	2982893	2980553	2994407	3018779	2996384	2993257	2989485	2998085.6	16282.30898	3036082	2980553
4	4	150	4495275	4467902	4490338	4430610	4510313	4462604	4495802	4426592	4511076	4462045	4475255.7	28831.52095	4511076	4426592
4	4	200	5983105	6044965	5935156	5944387	5933200	5973515	5951987	5923232	5978600	5950537	5961868.4	33662.25001	6044965	5923232
4	4	250	7473329	7475783	7451423	7494378	7362319	5183225	7429004	7522296	7421480	7450583	7226382	682317.2627	7522296	5183225
8	4	1	46846	48659	46401	45785	48182	46503	46955	46896	46716	45276	46821.9	948.8747494	48659	45276
8	4	50	879280	872750	869867	879756	874737	876633	860048	875499	866369	876939	873187.8	5867.780514	879756	860048
8	4	100	1710399	1691026	1703038	1708680	1696346	1713633	1715024	1713231	1686228	1704983	1704258.8	9520.286559	1715024	1686228
8	4	150	2510937	2517247	2511754	2523182	2521013	2546244	2526716	2507743	2527753	2516421	2520901	10572.37565	2546244	2507743
8	4	200	3359381	3343219	3317333	3343392	3321360	3362326	3336326	3297556	3302577	3329099	3331256.9	20801.60983	3362326	3297556
8	4	250	4164496	4192580	4176684	4133608	4174493	4207753	4121828	4151472	4118769	4146066	4158774.9	28206.26062	4207753	4118769
16	4	1	32713	31909	32006	33599	33141	30084	32797	32850	34236	31593	32492.8	1101.860227	34236	30084
16	4	50	544742	537837	527100	531052	532858	515492	531822	525789	529971	529825	530648.8	7237.30835	544742	515492
16	4	100	1026870	1024112	1014132	1031061	1026994	1010417	1022020	1023185	1030997	1020298	1023008.6	6374.678128	1031061	1010417
16	4	150	1490277	1498629	1473104	1467106	1476642	1450666	1445513	1467752	1496848	1481213	1474775	16999.74423	1498629	1445513
16	4	200	1923636	1930854	1915804	1934345	1921354	1896057	1916483	1938083	1915276	1926634	1921852.6	11405.63396	1938083	1896057
16	4	250	2380388	2401464	2342228	2356491	2353881	2351412	2355876	2356229	2353930	2359873	2361177.2	16212.70384	2401464	2342228

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	5	1	183881	187696	175511	178191	180293	184754	181173	190181	189632	183463	183477.5	4574.57063	190181	175511
1	5	50	187693	185997	175879	179343	174615	175848	185939	176354	179834	176495	179799.7	4682.673319	187693	174615
1	5	100	176081	185661	187930	174431	178317	182318	173925	178546	180034	185853	180309.6	4709.88047	187930	173925
1	5	150	185133	180718	177742	186260	177901	181593	178129	183394	180531	186215	181761.6	3177.869481	186260	177742
1	5	200	177711	177246	179488	176596	184112	185305	179646	182493	182451	178667	180371.5	2871.750311	185305	176596
1	5	250	184738	181228	182294	180234	183853	186068	174820	183152	183427	186683	182649.7	3223.434102	186683	174820
2	5	1	300888	346740	385568	302489	357521	259625	387619	279217	349700	261871	323123.8	45976.22384	387619	259625
2	5	50	260510	288783	318025	264342	254874	259178	260087	264717	249748	253615	267387.9	19656.41132	318025	249748
2	5	100	331771	287044	279582	398483	307820	254792	265719	369444	279927	263041	303762.3	45850.80822	398483	254792
2	5	150	362490	252710	363281	264218	249527	263671	358522	258076	286752	251410	291065.7	47131.97148	363281	249527
2	5	200	392016	279250	255038	299871	282874	263327	271332	338851	282409	399334	306430.2	49650.5591	399334	255038
2	5	250	400820	258476	261967	338084	259003	259478	299646	277715	351906	298713	300580.8	46149.60896	400820	258476
4	5	1	192888	202139	166627	228381	196675	149314	224547	191799	235962	236417	202474.9	27897.21136	236417	149314
4	5	50	173477	190310	207335	232171	198914	145173	203049	222682	177337	233533	198398.1	26513.20948	233533	145173
4	5	100	210576	189490	187529	165696	173161	140118	169295	236038	181687	165315	181890.5	25221.41499	236038	140118
4	5	150	222054	224569	232184	210726	234796	143878	201041	162447	228342	171254	203129.1	30893.87061	234796	143878
4	5	200	192690	174264	170361	226973	229520	144282	231397	162729	166236	223641	192209.3	31248.31249	231397	144282
4	5	250	171625	190224	194782	183221	229147	158839	212933	190750	230860	163506	192588.7	23960.28306	230860	158839
8	5	1	124242	228664	110370	183149	195561	118077	129947	187642	213823	191828	168330.3	41106.90467	228664	110370
8	5	50	199191	213624	202432	193878	233093	117099	130560	184017	195517	228072	189748.3	36099.74328	233093	117099
8	5	100	182559	229703	220380	189971	224048	129443	195784	199932	207216	195119	197415.5	26992.32517	229703	129443
8	5	150	186029	230953	115869	179371	178359	123778	220904	170163	227372	186855	181965.3	37325.22092	230953	115869
8	5	200	228917	134874	174329	180853	201400	111245	201694	181536	116569	228845	176026.2	40461.99512	228917	111245
8	5	250	230417	229005	172826	197454	231455	130093	117658	231797	118566	118712	177798.3	49486.49576	231797	117658
16	5	1	152272	140771	143065	143104	146458	108538	146045	105088	143641	136325	136530.7	15373.94069	152272	105088
16	5	50	142899	148860	139071	143084	141696	115169	137401	146312	140743	141785	139702	8753.166593	148860	115169
16	5	100	143535	145533	140484	141355	143006	110835	116782	144356	148333	119904	135412.3	13136.52278	148333	110835
16	5	150	169681	149650	139770	147962	150890	104630	148562	137072	142778	144832	143582.7	15465.27966	169681	104630
16	5	200	112729	135649	155682	155460	142059	115448	119530	137503	138983	149520	136256.3	14937.81528	155682	112729
16	5	250	144702	140113	140074	149558	136450	110654	145540	141876	133733	141541	138424.1	10189.69561	149558	110654

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	6	1	275899	272211	256943	242628	345882	351429	266064	319016	243704	262988	283676.4	38257.5839	351429	242628
1	6	50	257118	351961	353099	245261	263488	252962	260158	394863	274678	349693	300328.1	52532.15027	394863	245261
1	6	100	386029	260106	306959	262180	258674	233071	251410	253034	257158	243908	271252.9	42359.43174	386029	233071
1	6	150	264317	294333	296803	234163	322451	382659	360263	259009	256427	394651	306507.6	53602.70631	394651	234163
1	6	200	352805	257063	255792	252355	258408	252546	264155	246152	262473	272266	267401.5	29280.4089	352805	246152
1	6	250	245738	243875	263397	249351	255986	274877	258672	303860	258553	260646	261495.5	16526.97041	303860	243875
2	6	1	208341	200588	192572	222325	214073	179994	211496	192927	214460	208611	204538.7	12170.52377	222325	179994
2	6	50	200904	185586	195939	204902	205680	177407	201552	214085	190330	193001	196938.6	10204.78866	214085	177407
2	6	100	207589	177338	222668	189491	176318	184028	195111	213900	187751	221845	197603.9	16728.17914	222668	176318
2	6	150	205035	209209	219466	199280	203976	211484	185713	203709	188522	202636	202903	9537.217802	219466	185713
2	6	200	201865	190989	193688	181399	223962	187615	182707	206136	192248	170568	193117.7	14097.71081	223962	170568
2	6	250	190699	205600	214830	195593	207796	209418	204427	183966	199101	226596	203802.6	11605.77755	226596	183966
4	6	1	111003	105687	96028	106696	98324	105703	93774	104544	106630	83341	101173	7851.65757	111003	83341
4	6	50	106094	92535	111705	115648	139117	139979	110857	141152	97345	98671	115310.3	17547.97446	141152	92535
4	6	100	96067	91450	109893	100834	94344	106584	108975	103622	139247	105784	105680	12677.95157	139247	91450
4	6	150	102920	93477	98235	94663	104462	115673	137974	136838	94572	94481	107329.5	16340.12503	137974	93477
4	6	200	96046	97393	123823	134304	103546	99652	99178	89821	112898	105114	106177.5	13017.71713	134304	89821
4	6	250	102980	84307	108421	113008	95281	111014	104441	101351	101685	101834	102432.2	7793.866855	113008	84307
8	6	1	101652	99280	96449	97447	96822	96380	97523	96518	99637	89183	97089.1	3102.33333	101652	89183
8	6	50	99759	97177	103719	97585	100948	91275	100747	103348	100079	97664	99230.1	3406.137327	103719	91275
8	6	100	98829	94283	98227	98350	97806	96844	93838	92493	97850	98264	96678.4	2152.360434	98829	92493
8	6	150	98472	97707	98324	98025	101626	98216	98018	94745	104105	91250	98048.8	3271.29677	104105	91250
8	6	200	100808	98586	100488	98029	107282	94967	94123	97118	98172	96005	98557.8	3556.418305	107282	94123
8	6	250	99813	100894	97278	98148	101090	101235	95669	101224	97495	100082	99292.8	1895.333364	101235	95669
16	6	1	60753	59855	57606	60053	85315	55020	61293	65541	57648	62219	62530.3	8067.080749	85315	55020
16	6	50	56475	60310	57684	75065	52682	55518	61223	62310	64290	61283	60684	5836.820299	75065	52682
16	6	100	59161	58746	63472	63560	58926	57582	59395	72785	63735	58906	61626.8	4318.675139	72785	57582
16	6	150	59651	58211	60248	56981	61353	58437	60551	61629	76083	62462	61560.6	5099.986435	76083	56981
16	6	200	61481	59027	58612	60266	59762	56037	71231	57077	60459	59135	60308.7	3944.421911	71231	56037
16	6	250	58553	58062	57724	58551	60773	54074	57288	57402	62350	57583	58236	2078.374653	62350	54074

Agents Baseline Test Results (256 Places, 256 Agents, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	7	1	181610	168719	178365	167208	178735	176168	169793	166885	180052	183518	175105.3	6011.906687	183518	166885
1	7	50	172226	177811	180999	174513	185014	168507	166844	169627	176430	181869	175384	5808.650256	185014	166844
1	7	100	189224	179770	171082	177916	166736	158956	167325	167261	175207	175089	172856.6	8046.983089	189224	158956
1	7	150	166802	173607	179909	172677	189996	169111	171761	176353	172692	169199	174210.7	6350.376761	189996	166802
1	7	200	169895	169331	170750	189024	175665	170453	167503	164839	169542	175899	172290.1	6408.310315	189024	164839
1	7	250	167298	178222	171265	179510	171128	173919	168709	184850	170138	170185	173522.4	5307.944898	184850	167298
2	7	1	285149	287398	266723	292112	276324	265218	289667	289140	285975	292277	282998.3	9538.134996	292277	265218
2	7	50	255447	277961	269026	310858	274675	261937	288394	271907	289379	290678	279026.2	15371.2007	310858	255447
2	7	100	288828	269964	294967	287003	256017	285079	269090	266521	292117	288067	279765.3	12508.98631	294967	256017
2	7	150	259973	259384	271142	269881	300608	271354	279621	290403	293044	287928	278333.8	13490.754	300608	259384
2	7	200	279209	285137	265753	289213	265432	270073	279237	285151	262025	285272	276650.2	9441.614055	289213	262025
2	7	250	250976	281281	288724	275891	274846	268454	291162	276721	257960	261350	272736.5	12402.58557	291162	250976
4	7	1	154250	123822	139656	123695	132243	132589	143777	137098	150033	128820	136598.3	9891.818377	154250	123695
4	7	50	139509	135806	120000	132233	154256	114575	147777	153087	130333	128522	135609.8	12617.56426	154256	114575
4	7	100	124512	131793	130331	129723	149555	123214	137900	125454	155529	155423	136343.4	12002.97777	155529	123214
4	7	150	146842	129295	134141	149157	133886	132938	127640	124641	133079	142912	135453.1	7779.118388	149157	124641
4	7	200	115955	150481	138037	132457	139749	136904	129631	142376	125853	121633	133307.6	9803.955724	150481	115955
4	7	250	128414	131348	129156	156889	125087	127637	133224	112252	153499	138766	133627.2	12572.39092	156889	112252
8	7	1	100910	107586	104300	113069	108460	104453	105226	102419	114671	112452	107354.6	4490.176348	114671	100910
8	7	50	104955	108340	108290	110133	104975	112940	105931	108852	107353	111034	108280.3	2478.313703	112940	104955
8	7	100	105166	116018	108534	109223	108403	105844	103624	106666	105508	109399	107838.5	3282.482056	116018	103624
8	7	150	98009	98067	105147	106179	103384	99507	106844	110924	108693	108817	104557.1	4425.799034	110924	98009
8	7	200	105368	98697	100620	107309	112366	100859	106254	109913	104673	114351	106041	4880.638852	114351	98697
8	7	250	111217	111595	104427	105081	104069	115505	104879	113996	108980	112548	109229.7	4105.527129	115505	104069
16	7	1	68522	67939	63989	68340	67321	63231	84657	66355	66858	64543	68175.5	5768.821478	84657	63231
16	7	50	65821	66853	69610	71466	80457	65493	91590	64665	64116	70260	71033.1	8233.694365	91590	64116
16	7	100	66148	68850	65876	75310	71140	65945	71606	88007	65244	64992	70311.8	6728.039816	88007	64992
16	7	150	64904	64128	68956	74531	65065	65983	76587	62695	72349	65600	68079.8	4553.215629	76587	62695
16	7	200	68031	66424	67263	66550	67070	62650	67881	74485	69993	72624	68297.1	3180.533491	74485	62650
16	7	250	65198	68748	62374	77278	67062	62906	73972	64320	67435	70790	68008.3	4585.655549	77278	62374

I Places Baseline Results: Iterations

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	1	1	1251863	1341601	1297171	1282954	1291578	1221523	1635269	1354330	1199359	1331042	1320669	115218.5511	1635269	1199359
1	1	10	1283905	1215949	1280625	1690915	1229687	1428037	1342395	1230291	1225108	1246060	1317297.2	139296.096	1690915	1215949
1	1	50	1429693	1341308	1323571	1336843	1348973	1325617	1392257	1430222	1453402	1317051	1369893.7	48923.35819	1453402	1317051
1	1	100	1492678	1486125	1470677	1497651	1464013	1601255	1472693	1494560	1487491	1523147	1499029	37572.88751	1601255	1464013
1	1	200	1775434	1780687	1882329	1996444	1976983	1828562	1832835	1786399	1786421	1797472	1844356.6	77628.85073	1996444	1775434
1	1	500	2645416	2633441	2726285	2880429	2695066	2673043	2972950	2629282	2631703	2674104	2716171.9	111188.3899	2972950	2629282
1	1	1000	4157837	4112822	4118991	4158759	4108456	4160057	4123765	4124023	4122144	4102545	4128939.9	20651.10892	4160057	4102545
1	1	10000	30507485	30518838	30522620	30558411	30537812	30531069	30574195	30513888	30518940	30550949	30533420.7	20523.89286	30574195	30507485
2	1	1	1036620	1071539	969164	950486	912302	971725	879466	1067418	930579	981159	977045.8	61098.85455	1071539	879466
2	1	10	908396	1065282	1059774	1019467	855866	1027672	870709	927305	863141	1042909	964052.1	82337.75305	1065282	855866
2	1	50	905377	1137839	1010448	1046942	1121036	973176	1019162	947683	917814	979697	1005917.4	74535.82984	1137839	905377
2	1	100	939595	1181955	1033492	985549	1210290	1147030	1056496	962471	948217	1259478	1072457.3	112266.0094	1259478	939595
2	1	200	1333787	1347702	1313068	1142922	1205500	1225124	1145686	1043449	1308389	1259909	1232553.6	94136.44317	1347702	1043449
2	1	500	1536561	1931565	1933536	1656128	1429456	1649151	1451311	1493856	1659399	1847957	1658892	179877.9459	1933536	1429456
2	1	1000	2205277	2703695	2243005	2258103	2354938	2267921	2255912	2463173	2216009	2229477	2319751	147542.9686	2703695	2205277
2	1	10000	15970304	16392578	15433893	15479460	15408289	15555518	15462141	15521562	15378847	15480453	15608304.5	305585.7053	16392578	15378847
4	1	1	753059	648173	751652	697859	731043	732866	785817	688371	777749	651036	721762.5	46285.80513	785817	648173
4	1	10	696206	781253	668443	743059	761019	786843	684343	729792	736974	737191	732512.3	37475.12074	786843	668443
4	1	50	820904	718052	734336	695374	790438	816249	718433	763669	793917	704666	755603.8	44894.0311	820904	695374
4	1	100	820619	834829	728687	784571	752964	765465	856219	891329	810875	887054	813261.2	52731.85415	891329	728687
4	1	200	916693	845872	714037	970301	971297	844656	887456	970850	1030661	814700	896652.3	89606.12454	1030661	714037
4	1	500	1197516	1228856	1190400	1202865	1323237	1228119	1339508	1294447	1275945	1059681	1234057.4	76914.21277	1339508	1059681
4	1	1000	1658082	1516751	1326552	1558392	1521660	1261488	1157706	1776880	1342997	1318218	1443872.6	183341.48	1776880	1157706
4	1	10000	7896016	8075371	7798121	8060173	7833418	8019348	8037106	7867756	7780125	7909376	7927681	105884.8273	8075371	7780125
8	1	1	666234	628695	654272	663456	668986	656278	671274	645332	707860	682097	664448.4	20202.59603	707860	628695
8	1	10	655314	666519	667968	687522	659779	664252	674140	682054	686918	697724	674219	13098.59388	697724	655314
8	1	50	688055	694022	724673	693010	714394	688803	695697	714638	739329	705920	705854.1	16249.60064	739329	688055
8	1	100	753446	726306	745581	760599	759886	747425	752206	732191	731423	720110	742917.3	13668.97249	760599	720110
8	1	200	844509	815580	839303	827413	829614	785939	843602	843120	849002	830964	830904.6	17788.13556	849002	785939
8	1	500	1022498	1093787	1084006	1087165	1068849	1089610	1060024	1085738	1058629	1085235	1073554.1	20691.15113	1093787	1022498
8	1	1000	1234270	1267644	1338089	1256091	1293205	1338831	1286063	1292706	1231838	1389825	1292856.2	47698.87516	1389825	1231838
8	1	10000	4470845	4412800	4294733	4522646	4362018	4287909	4450755	4387530	4559983	4347141	4409636	86893.46429	4559983	4287909
16	1	1	511577	504893	488597	499868	497239	512513	499717	484936	486554	490436	497633	9462.847394	512513	484936
16	1	10	496385	503696	503554	490127	519505	512650	489197	511825	496978	507013	503093	9468.298728	519505	489197
16	1	50	520751	526134	538573	546273	531573	517816	518737	525676	526275	529018	528082.6	8445.998262	546273	517816
16	1	100	544155	561910	572708	545146	564000	561428	538553	554968	554461	553484	555081.3	9855.87846	572708	538553
16	1	200	585222	595474	605927	591489	583062	596786	599797	601757	613761	597929	597120.4	8703.605623	613761	583062
16	1	500	735440	733121	749594	724044	743337	719925	726949	723685	727487	743746	732732.8	9528.314434	749594	719925
16	1	1000	959788	950549	968020	958557	965024	958559	945352	960447	953121	950924	957034.1	6630.072043	968020	945352
16	1	10000	2876661	2856580	2838700	2808409	2908021	2914767	2887688	2824798	2882572	2889374	2868757	33647.18421	2914767	2808409

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	2	1	103863	108108	98657	107620	104334	97779	98209	106767	107003	99501	103184.1	4014.385767	108108	97779
1	2	10	120227	108039	131620	102046	104094	101377	101384	124428	105871	101430	110051.6	10588.83415	131620	101377
1	2	50	222628	223181	244225	237813	246030	222621	239483	226509	223495	223095	230908	9270.992719	246030	222621
1	2	100	381144	375644	380458	375643	373250	381620	374819	373920	372073	375579	376415	3245.5056	381620	372073
1	2	200	707355	662955	698878	704867	709169	674143	677595	669680	724731	690197	691957	19199.31962	724731	662955
1	2	500	1548900	1547801	1569108	1550793	1577626	1556869	1552625	1566156	1577474	1551864	1559921.6	11063.40733	1577626	1547801
1	2	1000	3020422	3046863	3028358	3019391	3022748	3032390	3022256	3017386	3020335	3023999	3025414.8	8294.068179	3046863	3017386
1	2	10000	29425615	29417319	29435782	29430159	29442054	29430098	29425594	29442562	29421752	29427725	29429866	7804.124294	29442562	29417319
2	2	1	105685	100027	120105	133282	112931	100652	111709	124533	124174	130021	116311.9	11294.64049	133282	100027
2	2	10	119625	141812	154902	154937	161684	133651	142695	146186	173559	157357	148640.8	14493.03455	173559	119625
2	2	50	195830	189246	278008	273274	295123	225279	264386	290837	291651	290565	259419.9	38686.39936	295123	189246
2	2	100	266727	284736	359005	352718	331375	283160	304556	464874	430955	405632	348373.8	64015.71307	464874	266727
2	2	200	427492	421124	444894	423480	434793	392002	397558	410138	451853	403294	420662.8	18926.56689	451853	392002
2	2	500	884666	896974	834852	842731	824282	838568	849634	824964	837867	859512	849405	23116.61195	896974	824282
2	2	1000	1678592	1686068	1578921	1581352	1567994	1565392	1601273	1587136	1588459	1557836	1599302.3	43210.88483	1686068	1557836
2	2	10000	14787413	15619499	14815962	14769695	14772602	14774759	14811910	14780973	14792318	14776720	14870185.1	250228.5644	15619499	14769695
4	2	1	102286	93566	111037	114528	114728	113130	116458	112430	111966	108441	109857	6610.695879	116458	93566
4	2	10	128845	127738	129691	121611	127069	123645	128977	126616	125102	116122	125541.6	3952.369219	129691	116122
4	2	50	206208	211407	198752	208114	206360	193847	206378	206936	204397	206531	204893	4751.026016	211407	193847
4	2	100	283844	320869	299900	297335	281197	280525	285697	289775	291847	285895	291688.4	11489.96988	320869	280525
4	2	200	451224	458672	477024	457407	427938	461139	459881	469400	476111	447762	458655.8	13726.72408	477024	427938
4	2	500	537757	856925	746250	643228	751323	618743	722506	589268	492314	578202	653651.6	107145.5557	856925	492314
4	2	1000	1069952	1051675	1037722	822338	1031387	821954	1070263	1052829	1055478	846033	985963.1	102847.9157	1070263	821954
4	2	10000	7559831	7519951	7540157	7448340	7551495	7468522	7468317	7495805	7440390	7481019	7497382.7	40949.07704	7559831	7440390
8	2	1	109727	112662	116084	112907	114776	116333	115320	116091	115738	114497	114413.5	1982.525019	116333	109727
8	2	10	117511	122580	122738	121809	123235	122390	122766	120185	121533	120591	121533.8	1632.694019	123235	117511
8	2	50	162502	158502	161934	160106	165066	160275	158871	158178	173912	157447	161679.3	4630.885165	173912	157447
8	2	100	202257	205313	208812	206952	209636	203110	205217	206196	218374	210976	207684.3	4416.192411	218374	202257
8	2	200	303013	304915	296995	294904	292210	296381	303427	296122	299084	296799	298385	3921.763838	304915	292210
8	2	500	557181	566113	560347	549197	553748	560224	571189	548472	553764	549272	556950.7	7213.614268	571189	548472
8	2	1000	878739	883298	900654	900546	895008	899278	900176	900084	904718	900110	896261.1	7997.937415	904718	878739
8	2	10000	3997497	4026541	3963667	3945498	3998462	3946632	3997380	4013446	4046461	4009181	3994476.5	31586.76547	4046461	3945498
16	2	1	160115	156782	157939	156064	158346	153978	157781	165833	158070	153448	157835.6	3284.264703	165833	153448
16	2	10	169010	161360	162910	160270	159634	159218	167942	161560	163624	158437	162396.5	3405.427528	169010	158437
16	2	50	177555	180622	184503	180318	154315	180066	157559	176279	181791	179168	175217.6	9894.934827	184503	154315
16	2	100	203727	201007	206743	202250	205711	204869	210707	178066	206508	197167	201675.5	8601.191572	210707	178066
16	2	200	246604	246521	259590	257887	244255	231678	246196	246175	245447	236523	246087.6	7905.071008	259590	231678
16	2	500	382911	373646	387375	382836	374689	384377	382693	377387	385314	370838	380206.6	5333.663191	387375	370838
16	2	1000	584799	598098	617595	600759	607912	596975	582012	597740	611143	602741	599977.4	10377.61363	617595	582012
16	2	10000	2324556	2332681	2294087	2324586	2283713	2337046	2327701	2305227	2311964	2335242	2317680.3	17326.35292	2337046	2283713

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	3	1	55085	54850	61568	58581	53497	53487	61478	56329	53719	58032	56662.6	2957.404443	61568	53487
1	3	10	71474	74558	73365	75746	59610	58033	78035	74810	73479	75313	71442.3	6524.380523	78035	58033
1	3	50	197587	175392	197703	199785	191970	203236	202987	197827	203404	183879	195377	8735.261461	203404	175392
1	3	100	333724	323082	330937	322524	326760	322438	332696	328907	332331	322741	327614	4425.343331	333724	322438
1	3	200	649752	629463	654450	620101	617045	622383	625854	625230	619203	617541	628102.2	12600.79738	654450	617045
1	3	500	1500753	1506529	1498462	1505855	1504053	1499672	1498787	1499970	1503853	1501305	1501923.9	2781.774666	1506529	1498462
1	3	1000	2968366	2965939	2969338	2968014	2966571	2971889	2970553	2985119	2973705	2978977	2971847.1	5726.746187	2985119	2965939
1	3	10000	29390250	29413462	29369216	29378407	29373560	29379949	29372954	29366066	29369155	29384568	29379758.7	13292.01758	29413462	29366066
2	3	1	65731	54341	56125	67312	57171	55588	67835	42331	34044	55791	55626.9	10171.47464	67835	34044
2	3	10	103412	71382	58209	105494	82509	78365	86748	77611	68302	95287	82731.9	14502.74909	105494	58209
2	3	50	120314	122561	254552	107830	252655	127967	224724	143146	124924	117263	159593.6	56371.87939	254552	107830
2	3	100	197106	203422	197564	200457	182498	184345	220828	183473	193824	182726	194624.3	11541.78129	220828	182498
2	3	200	330209	366839	340864	351100	351817	343357	331731	336871	361131	325726	343964.5	12893.11317	366839	325726
2	3	500	788615	829446	776038	778547	787820	772819	785138	784651	808276	766121	787747.1	17577.44871	829446	766121
2	3	1000	1515861	1609894	1514608	1513974	1504619	1525922	1505168	1505765	1511579	1510575	1521796.5	29975.30206	1609894	1504619
2	3	10000	14773835	15028424	14742780	14763925	14727824	14712402	14723805	14717414	14721925	14716580	14762891.4	90681.14198	15028424	14712402
4	3	1	46029	43943	50381	46733	47728	48956	43759	48390	38871	41312	45610.2	3430.760639	50381	38871
4	3	10	55836	61108	64835	61639	62438	57696	59006	64018	59804	54584	60096.4	3198.331884	64835	54584
4	3	50	139474	140172	144836	148260	136745	133337	139244	140535	134699	131906	138920.8	4793.076482	148260	131906
4	3	100	234528	223427	229532	140839	120394	226407	233005	232661	226531	210835	207815.9	39387.93489	234528	120394
4	3	200	190157	374508	186666	194366	395096	230832	202794	276627	196937	229859	247784.2	73293.70599	395096	186666
4	3	500	437144	464752	411778	425612	427090	423530	426868	424326	417985	432801	429188.6	13594.07553	464752	411778
4	3	1000	775709	780898	784098	772732	770286	782470	790913	779027	787817	771271	779522.1	6666.007447	790913	770286
4	3	10000	7415853	7470582	7407169	7409516	7389533	7384118	7406283	7373787	7380601	7384176	7402161.8	26539.52714	7470582	7373787
8	3	1	38348	38487	41699	45649	39543	39292	40480	40612	43177	40144	40743.1	2138.298878	45649	38348
8	3	10	47535	47209	52729	44685	47025	47158	44852	42344	46651	48302	46849	2577.157271	52729	42344
8	3	50	86708	85178	88167	88649	85824	88656	84190	88962	84280	86595	86720.9	1736.097661	88962	84190
8	3	100	132507	133731	128337	140651	130490	132270	130509	131719	133956	134217	132838.7	3128.460805	140651	128337
8	3	200	221466	220263	221182	222837	223225	224797	222562	222415	221334	231213	223129.4	2950.447464	231213	220263
8	3	500	446456	412952	317077	465287	464608	451059	460168	459679	456996	450789	438507.1	42927.39915	465287	317077
8	3	1000	689892	698273	485011	448286	677541	764837	512423	814282	743839	774649	660903.3	124437.9056	814282	448286
8	3	10000	3778478	3979690	3754261	3743102	3753009	3806828	3777580	3787625	3742980	3819508	3794306.1	66557.75033	3979690	3742980
16	3	1	39228	40735	38476	37067	37131	37562	38957	44496	35364	41349	39036.5	2481.272224	44496	35364
16	3	10	48671	48069	42121	42353	47171	45550	43354	40895	51189	43907	45328	3181.019711	51189	40895
16	3	50	64448	86460	66218	62414	72798	63021	64902	63312	62218	64498	67028.9	7095.199736	86460	62218
16	3	100	82644	85498	87460	94281	88805	83842	87057	87269	88111	82836	86780.3	3253.497135	94281	82644
16	3	200	134113	137013	134115	133864	134564	132573	132770	135974	130838	131384	133720.8	1815.181357	137013	130838
16	3	500	273377	277993	268186	273345	266820	272663	284297	268594	271435	268454	272516.4	5044.635412	284297	266820
16	3	1000	490259	502539	497550	484014	491771	486525	484170	495444	493973	490343	491658.8	5633.405751	502539	484014
16	3	10000	2025061	2040630	2043532	1942157	1942944	1989703	1981098	1951825	1943181	1976086	1983621.7	38310.60025	2043532	1942157

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Iterations	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	4	1	126393	107890	130175	119130	128989	126184	127825	125751	122569	123816	123872.2	6134.989532	130175	107890
1	4	10	148528	121164	138175	128609	129776	136353	138813	147961	154304	131118	137480.1	9814.376398	154304	121164
1	4	50	257767	224147	232015	237875	244644	229812	229940	234773	248392	221699	236106.4	10690.64082	257767	221699
1	4	100	394315	418448	396220	409815	376435	417663	370058	386908	403416	417008	399028.6	16462.04761	418448	370058
1	4	200	685656	672953	683419	685131	661553	673973	662969	696210	659465	698503	677983.2	13313.43411	698503	659465
1	4	500	1541540	1542829	1554770	1547789	1545096	1546709	1560930	1566558	1543911	1539550	1548968.2	8454.470602	1566558	1539550
1	4	1000	3026844	3009494	3018674	3011129	3010428	3008511	3013972	3026023	3007950	3008480	3014150.5	6853.096413	3026844	3007950
1	4	10000	29414859	29428983	29430485	29427039	29426185	29418858	29437914	29420167	29424571	29432638	29426169.9	6534.837526	29437914	29414859
2	4	1	323513	325555	331028	347944	360537	337103	352817	342910	338701	355866	341597.4	12061.87452	360537	323513
2	4	10	347447	348454	341049	357228	366257	338531	333689	356507	348080	360877	349811.9	9834.285957	366257	333689
2	4	50	460640	470376	466315	455418	441475	485078	465673	505058	462146	505381	471756	19711.51451	505381	441475
2	4	100	612344	616492	680998	623596	572261	624776	624300	628612	672819	621338	627753.6	28964.70324	680998	572261
2	4	200	699671	680434	708953	728573	746929	727045	717925	732068	700427	730974	717299.9	18787.33631	746929	680434
2	4	500	1069011	1060732	1089389	1091610	1105364	1093691	1068954	1090406	1073565	1099647	1084236.9	14226.50664	1105364	1060732
2	4	1000	1760121	1769111	1790716	1794802	1792038	1781118	1770240	1773706	1776696	1780443	1778899.1	10619.34594	1794802	1760121
2	4	10000	15249124	15093157	15231844	15098230	15138235	15068578	15124732	15132430	15109330	15117236	15136289.6	55676.00849	15249124	15068578
4	4	1	384167	385256	390866	384709	387759	391095	398711	397683	389000	380567	388981.3	5531.80956	398711	380567
4	4	10	385499	395133	387393	383452	397080	397150	382626	386636	388830	385650	388944.9	5219.001388	397150	382626
4	4	50	425022	456719	442704	453100	438291	403470	468877	441850	430511	440974	440151.8	17170.35712	468877	403470
4	4	100	515027	515296	528370	514317	519965	517986	512022	509872	491399	519213	514346.7	9047.279061	528370	491399
4	4	200	679288	686075	693288	697402	684499	691247	706985	707718	697507	696705	694071.4	8749.491245	707718	679288
4	4	500	951768	950846	983101	976597	959304	938238	965049	944890	944856	939930	955457.9	14540.94434	983101	938238
4	4	1000	1222338	1220190	1222959	1217183	1237465	1222149	1250443	1231042	1240523	1238089	1230238.1	10395.95348	1250443	1217183
4	4	10000	7794475	7787360	7762266	7785188	7761989	7756585	7795246	7775536	7755606	7770170	7774442.1	14535.90415	7795246	7755606
8	4	1	429096	424814	443057	428704	451846	417177	452101	441920	436628	431563	435690.6	10929.95147	452101	417177
8	4	10	431814	412387	454903	452443	460140	418750	428088	436424	441957	438621	437552.7	14710.94953	460140	412387
8	4	50	436153	423934	441006	451243	451526	459001	429457	449282	458808	456940	445735	11846.23195	459001	423934
8	4	100	493555	469206	496176	488003	488099	499700	468376	481267	463399	486232	483401.3	11895.83339	499700	463399
8	4	200	552414	554759	570783	583744	595622	581398	586079	585144	573037	581782	576476.2	13168.02793	595622	552414
8	4	500	780723	757282	787032	787548	765982	776391	772062	784775	762600	765617	774001.2	10335.22099	787548	757282
8	4	1000	996554	1019072	995829	995365	993768	1000272	994188	996622	982317	982885	995687.2	9582.316409	1019072	982317
8	4	10000	4162181	4165000	4177535	4169843	4187474	4176393	4154701	4156278	4175594	4148268	4167326.7	11591.41396	4187474	4148268
16	4	1	442671	447905	459602	456629	467448	452734	447791	446469	449381	453520	452415	6913.405268	467448	442671
16	4	10	441599	448625	464597	466805	439235	449745	453952	448391	457888	455609	452644.6	8519.669174	466805	439235
16	4	50	455436	457221	468905	454854	470081	463108	464225	459684	453637	456788	460393.9	5593.059314	470081	453637
16	4	100	471527	481857	471809	499434	464412	491895	495498	477892	480307	470479	480511	11123.19168	499434	464412
16	4	200	485764	513699	527042	544210	534484	526696	513781	517904	511953	519379	519491.2	14858.54329	544210	485764
16	4	500	613713	649569	654640	664704	650980	645751	637107	647089	657895	644914	646636.2	13127.96265	664704	613713
16	4	1000	781845	787259	795181	811133	791938	788772	784719	785002	779698	800187	790573.4	9012.295193	811133	779698
16	4	10000	2396393	2401529	2394676	2408252	2403412	2385043	2358516	2383489	2355455	2410870	2389763.5	18426.15625	2410870	2355455

J Places Baseline Results: Max Time

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Max Time	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	1	1	32070	33212	50486	51904	38215	32248	32416	32733	55155	56768	41520.7	10109.12407	56768	32070
1	1	50	1018715	1034104	1043011	1020715	1008647	1007563	1038547	1013007	1215955	1010170	1041043.4	59559.88441	1215955	1007563
1	1	100	2024908	2013618	2020382	2010280	2073415	2042387	2047024	2076336	2156781	2073887	2053901.8	41877.62466	2156781	2010280
1	1	150	2986386	3006215	3003097	3001875	2996766	3043293	3056521	3002152	3145262	3033277	3027484.4	44694.02236	3145262	2986386
1	1	200	3962644	4200560	4000245	3999341	4006951	3975836	4007125	4022087	4036161	4005924	4021687.4	62793.60926	4200560	3962644
1	1	250	4991837	5013961	5042553	5024573	5015647	4992842	4993254	5091065	4977070	4974178	5011698	33368.65458	5091065	4974178
2	1	1	22992	34929	26198	21032	26927	25099	22282	26551	31876	23201	26108.7	4141.705061	34929	21032
2	1	50	766448	686634	765669	748652	795867	683136	710974	745758	755350	776154	743464.2	35964.16015	795867	683136
2	1	100	1351469	1608002	1474270	1619415	1436581	1616053	1400530	1456446	1533737	1337685	1483418.8	101323.4167	1619415	1337685
2	1	150	2270045	2411000	2426394	2084260	2146852	2170546	2319238	2159084	2412915	2120913	2252124.7	125729.4985	2426394	2084260
2	1	200	2760747	2820097	3126672	2776349	2807168	2963925	2989532	2961247	2943230	2755586	2890455.3	117783.8002	3126672	2755586
2	1	250	4269006	3764924	3680121	3571074	3475455	3568704	3547681	3573865	3724154	3824349	3699933.3	216290.6146	4269006	3475455
4	1	1	21628	20148	25321	21280	20529	26183	19460	20780	19377	18552	21325.8	2386.219512	26183	18552
4	1	50	653297	642135	611035	609700	507535	554283	659322	650529	673434	618743	618001.3	49038.01983	673434	507535
4	1	100	1239359	1315308	1302566	1333345	1174023	1262574	1145477	1224013	1328776	1327881	1265332.2	64450.9702	1333345	1145477
4	1	150	1981309	1891482	1863369	2025316	1834360	1904909	1985839	1775044	1829610	2005341	1909657.9	81340.89477	2025316	1775044
4	1	200	2569826	2588021	2567170	2421877	2480862	2482128	2517692	2589482	2613128	2604840	2543502.6	60916.90511	2613128	2421877
4	1	250	3229953	3199167	3206738	3175454	3178932	3080387	3063013	3279825	3346493	3198193	3195815.5	79197.69311	3346493	3063013
8	1	1	16776	18424	17359	17225	16303	17881	17412	18248	17262	15357	17224.7	866.5949515	18424	15357
8	1	50	562499	577307	581592	566733	569130	576126	578181	565871	574765	574411	572661.5	5901.639641	581592	562499
8	1	100	1148558	1140797	1128310	1140012	1131194	1153809	1169415	1112472	1134883	1117550	1137700	16059.52718	1169415	1112472
8	1	150	1691203	1686984	1698492	1696242	1657590	1734012	1685637	1708306	1635395	1665968	1685982.9	26193.7051	1734012	1635395
8	1	200	2230259	2246549	2269993	2242806	2284422	2246556	2290308	2282785	2258177	2294902	2264675.7	21630.32063	2294902	2230259
8	1	250	2807513	2800415	2847801	2851213	2784282	2845481	2832739	2786148	2837396	2779958	2817294.6	27115.92145	2851213	2779958
16	1	1	14253	17669	14847	27911	18544	25383	29331	14784	39651	13764	21613.7	8197.432294	39651	13764
16	1	50	730776	870957	963762	860292	955017	912736	904661	859182	952226	841854	885146.3	66258.33858	963762	730776
16	1	100	1612854	1767155	1745708	1640601	1960989	1734700	1742439	1919506	2003794	1860889	1798863.5	125483.5173	2003794	1612854
16	1	150	2563332	2708529	2556804	2536031	2772747	2786366	2976373	2669663	3036474	2846145	2745246.4	164406.8553	3036474	2536031
16	1	200	3218170	3036357	3447223	3484269	3600222	3708121	3925672	3628601	3773751	3660038	3548242.4	250159.5501	3925672	3036357
16	1	250	4160542	4150150	4184219	4218050	4799649	4564770	4790644	4640685	4603687	5484360	4559675.6	394866.4217	5484360	4150150

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Max Time	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	2	1	2225	2510	2174	2187	2248	2212	2156	2456	2381	2456	2300.5	128.3684151	2510	2156
1	2	50	97933	84339	95717	88384	86883	84162	85274	98769	93241	86749	90145.1	5423.705992	98769	84162
1	2	100	209992	168925	169382	173228	167756	168586	170367	184597	175852	167431	175611.6	12480.00299	209992	167431
1	2	150	253201	258524	252067	253709	256776	309077	258675	253853	260002	261609	261749.3	16066.70429	309077	252067
1	2	200	335227	398076	342882	338495	340654	335048	336613	400590	335014	337112	349971.1	24804.66999	400590	335014
1	2	250	420775	431801	504108	419211	444563	423873	427067	472268	429133	426041	439884	26040.43132	504108	419211
2	2	1	1950	2420	2389	1838	2404	2113	2250	2562	2379	2343	2264.8	217.0008295	2562	1838
2	2	50	122678	111135	134531	128649	105886	131362	128822	120881	113252	131687	122888.3	9366.523689	134531	105886
2	2	100	236265	257253	244950	241540	249154	234897	253462	271078	265112	260514	251422.5	11584.88612	271078	234897
2	2	150	401893	378008	381504	357488	362739	377152	379276	387534	391648	382586	379982.8	12239.37323	401893	357488
2	2	200	525552	523827	513898	491374	521074	516649	490791	510886	578819	503302	517617.2	23521.23907	578819	490791
2	2	250	582056	622818	619894	636342	658887	581147	594772	592401	634525	665447	618828.9	28978.20268	665447	581147
4	2	1	2599	2426	2238	2034	2178	2359	2476	1810	2290	2178	2258.8	216.1776122	2599	1810
4	2	50	107210	104949	106470	109343	103272	103044	101000	99446	103804	97088	103562.6	3506.238132	109343	97088
4	2	100	213552	205302	191575	193179	199139	211063	201449	205472	208573	206206	203551	6858.301568	213552	191575
4	2	150	308110	311129	306605	299165	330175	320590	312213	321569	294634	304967	310915.7	10221.29065	330175	294634
4	2	200	401308	395027	413159	409844	426483	408406	392971	408408	420769	418701	409507.6	10323.52084	426483	392971
4	2	250	540162	517763	509225	526688	499691	506671	519119	502265	482056	526970	513061	15762.59914	540162	482056
8	2	1	2417	2442	2503	2442	2603	2500	2735	2186	2437	2175	2444	160.0905993	2735	2175
8	2	50	103951	99014	102569	107958	100127	102875	102576	104871	102132	104546	103061.9	2379.40154	107958	99014
8	2	100	212264	216721	201258	197843	198821	200908	203805	208636	205698	202255	204820.9	5767.557862	216721	197843
8	2	150	305241	292181	303431	304546	300036	305887	305209	305911	307347	303818	303360.7	4163.974953	307347	292181
8	2	200	396750	402330	406365	397609	407820	400962	405772	407207	406544	409240	404059.9	4158.483364	409240	396750
8	2	250	497100	502056	511145	509958	507633	492306	500618	501621	509334	534068	506583.9	10820.14142	534068	492306
16	2	1	3956	6989	4240	6383	8789	10104	3937	5301	3861	7832	6139.2	2133.167401	10104	3861
16	2	50	193563	194985	191504	182955	203291	205130	201067	205170	232462	211816	202194.3	12804.54599	232462	182955
16	2	100	358474	372092	397310	363934	393601	394629	408600	408355	415039	424387	393642.1	21074.8353	424387	358474
16	2	150	559598	548933	556140	559807	558715	632228	600612	599122	604185	643600	586294	32501.54988	643600	548933
16	2	200	729834	736297	747025	729167	746853	788540	832092	740738	751568	808313	761042.7	33971.86035	832092	729167
16	2	250	900581	956211	928780	941234	1025653	1008243	1009756	976944	1016123	1062182	982570.7	47716.74196	1062182	900581

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Max Time	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	3	1	1436	1436	1460	1907	1407	1345	1323	1428	1248	1449	1443.9	167.111011	1907	1248
1	3	50	54935	52103	51622	64768	62261	62163	60994	64328	48722	47998	56989.4	6253.447181	64768	47998
1	3	100	92943	97968	113002	105469	126405	114831	94246	96103	96572	107672	104521.1	10414.30394	126405	92943
1	3	150	140571	145057	142687	139256	154082	155860	143459	142248	147231	164009	147446	7581.712115	164009	139256
1	3	200	211889	186865	188829	241316	232791	193587	185811	189719	184680	195329	201081.6	19529.22075	241316	184680
1	3	250	242472	245661	233864	243334	253403	232650	244978	242699	231431	233379	240387.1	6844.111987	253403	231431
2	3	1	1032	1774	1483	1577	1336	1526	1403	1467	1185	1290	1407.3	198.9422278	1774	1032
2	3	50	63577	86352	87563	68615	52564	85469	70176	67252	67976	70385	71992.9	10650.98522	87563	52564
2	3	100	154060	126534	130454	137518	92538	171118	132739	134131	144996	80223	130431.1	25430.40133	171118	80223
2	3	150	199786	213785	180122	129823	158882	198653	118404	110621	149143	198982	165820.1	35712.53359	213785	110621
2	3	200	272371	259340	270483	248412	211278	195456	255043	251233	305756	209097	247846.9	32010.30175	305756	195456
2	3	250	429837	325368	314688	308724	405134	272900	321995	313199	234655	327831	325433.1	53679.32276	429837	234655
4	3	1	971	989	1085	983	930	1034	815	1051	947	1018	982.3	71.66456586	1085	815
4	3	50	50640	49861	42918	52846	53050	46506	47786	51378	48595	48787	49236.7	2908.843449	53050	42918
4	3	100	97595	92673	96355	89451	96376	89408	96921	84223	108898	90444	94234.4	6370.434023	108898	84223
4	3	150	150658	140478	139129	157420	147921	148674	150394	152231	142252	141837	147099.4	5639.865888	157420	139129
4	3	200	187917	204205	191332	204797	184109	201956	197227	190465	198650	190140	195079.8	6882.879758	204797	184109
4	3	250	225423	230215	231790	241393	242748	208551	216208	231199	255234	220697	230345.8	13001.27864	255234	208551
8	3	1	1114	776	884	800	790	833	951	948	874	929	889.9	96.65345312	1114	776
8	3	50	40187	34503	39658	42395	40950	40859	37375	40283	39787	40072	39606.9	2078.97424	42395	34503
8	3	100	79452	71593	75289	73295	84842	76172	78076	83012	80716	78414	78086.1	3942.066551	84842	71593
8	3	150	123906	108490	117327	112421	114356	118517	115169	114269	118053	116213	115872.1	3876.81115	123906	108490
8	3	200	158642	153114	157455	140720	154431	151152	160074	156578	156879	154421	154346.6	5193.549619	160074	140720
8	3	250	191049	173693	191827	187988	201426	193250	192093	188184	197649	188957	190611.6	6921.37059	201426	173693
16	3	1	1088	1082	1160	1099	1294	1254	1196	1065	1120	1152	1151	72.77087329	1294	1065
16	3	50	54243	57936	63514	44510	56969	89708	71277	61048	68849	61220	62927.4	11418.44645	89708	44510
16	3	100	117822	103937	108394	109601	118047	129578	143569	125913	113829	133750	120444	11902.8908	143569	103937
16	3	150	170065	172461	169191	163895	183097	194887	173088	177278	180979	206573	179151.4	12320.82387	206573	163895
16	3	200	233530	227710	227552	215442	239107	246756	259025	279083	260688	246735	243562.8	17981.2227	279083	215442
16	3	250	268075	272861	283417	279767	286141	315528	321520	318841	302464	320363	296897.7	20039.62545	321520	268075

Places Baseline Test Results (256 Places, Time in Microseconds)

Hosts	Test Type	Max Time	Time (Run 1)	Time (Run 2)	Time (Run 3)	Time (Run 4)	Time (Run 5)	Time (Run 6)	Time (Run 7)	Time (Run 8)	Time (Run 9)	Time (Run 10)	Time (Average)	Time (St. Dev)	Time (Max)	Time (Min)
1	4	1	1254	1483	1410	1421	1455	1412	1458	1260	1417	1508	1407.8	81.37788397	1508	1254
1	4	50	117311	119542	103569	112338	103071	99708	103743	121276	114278	115217	111005.3	7395.749537	121276	99708
1	4	100	210392	205047	205932	246017	202328	209984	242226	215746	237692	220088	219545.2	15576.76602	246017	202328
1	4	150	333688	310695	304249	299720	324547	336535	368113	323517	341246	316813	325912.3	19148.43292	368113	299720
1	4	200	431908	439928	391466	409653	449698	408769	418186	421693	450992	426464	424875.7	17997.22611	450992	391466
1	4	250	579615	503722	543436	594898	547006	529322	506607	541956	597040	545536	548913.8	31117.05434	597040	503722
2	4	1	994	1655	1503	1431	883	1198	945	1046	1038	1366	1205.9	252.4513617	1655	883
2	4	50	303231	308850	287671	282878	316923	305841	286432	297587	281647	298867	296992.7	11366.72844	316923	281647
2	4	100	592853	553996	555570	556752	566064	595136	586432	583884	562419	572850	572595.6	15067.95065	595136	553996
2	4	150	870462	857672	821419	856858	873805	858640	893242	865381	817226	834303	854900.8	22736.72048	893242	817226
2	4	200	1154455	1105127	1130228	1126267	1182039	1104786	1128267	1113600	1097599	1176063	1131843.1	28256.18804	1182039	1097599
2	4	250	1375232	1434777	1367905	1376154	1400890	1419435	1380169	1468186	1375213	1363662	1396162.3	32604.127	1468186	1363662
4	4	1	993	1136	1005	859	920	985	1064	941	899	959	976.1	76.69608856	1136	859
4	4	50	332551	344378	333965	343886	331115	331089	343786	330444	336679	342789	337068.2	5683.571356	344378	330444
4	4	100	612935	628712	589633	625802	624174	602807	612467	611780	622711	632510	616353.1	12449.16825	632510	589633
4	4	150	908647	896422	890037	873203	864238	865665	881705	879614	891246	883811	883458.8	13098.18851	908647	864238
4	4	200	1159985	1125429	1169151	1152167	1147575	1158953	1138980	1160223	1178214	1140637	1153131.4	14739.46005	1178214	1125429
4	4	250	1420686	1446000	1421993	1430394	1443470	1431105	1422028	1466492	1456538	1443329	1438203.5	14811.6583	1466492	1420686
8	4	1	889	881	779	758	885	880	848	853	874	871	851.8	43.71452848	889	758
8	4	50	379301	379041	374123	367359	369033	378593	376116	372713	354249	364339	371486.7	7548.516464	379301	354249
8	4	100	644101	672382	644411	622758	643207	645756	648227	648093	645124	639694	645375.3	11384.91407	672382	622758
8	4	150	902402	952922	919073	914150	915828	922033	919805	925571	912386	906705	919087.5	13067.69189	952922	902402
8	4	200	1186160	1216315	1193712	1169592	1204281	1198420	1199330	1195790	1203493	1202711	1196980.4	11786.76089	1216315	1169592
8	4	250	1469161	1506878	1482382	1422627	1459535	1440388	1456052	1478853	1464007	1469807	1464969	21909.72074	1506878	1422627
16	4	1	1159	1160	7783	1242	1146	3567	1183	1138	1030	1096	2050.4	2043.491776	7783	1030
16	4	50	414953	400701	402067	393106	399509	396697	390290	403713	385900	407080	399401.6	8002.87616	414953	385900
16	4	100	670216	662295	674298	674523	662590	679510	659489	665428	669454	679948	669775.1	6883.542249	679948	659489
16	4	150	956462	950088	963933	928605	927229	948155	966977	937370	910511	969994	945932.4	18592.96925	969994	910511
16	4	200	1221554	1201440	1231961	1224821	1243211	1246227	1229313	1248260	1204411	1238346	1228954.4	15501.42426	1248260	1201440
16	4	250	1477340	1518562	1462244	1476666	1480147	1512176	1490913	1472049	1491021	1497135	1487825.3	16854.79325	1518562	1462244

K FluTE Data File Details

1. label la-1.6
   The “label” is simply a convenience name that the output will use when referring to results of this FluTE simulation. In this case, the value (label) used was “la-1.6”
2. R0 1.6
   This is the reproductive number assigned to outbreak virus (influenza) being modeled. This can be thought of as a numerical measure of how transmissable a given disease is - for comparison, the default value is set to 0.1 (so, this represents a fairly nasty flu - and, thus, a bit more computational cycles to deal with its transmission)
3. seed 1
   This is simply the value used to seed the random number generator
4. datafile la
   This is the prefix to use for the data file names. In this case, the “la” is meant to model “Los Angeles”
5. logfile 0
   This value controls how often results are printed to a log file. In this case, the zero value indicates that no log file was generated
6. individualfile 0
   This is a binary value that controls whether or not an individuals file is created as part of the simulation. A zero value here means that no individuals file was created during the application’s execution
7. prestrategy none
   String values can be assigned here to mimic strategies for dealing with outbreaks before the exist. In this case, no strategies were employed, simulating absolutely zero preparation on the part of the population under test
8. reactivestrategy none
   String values can be assigned here to indicate vaccination strategies that can be used during the simulation (post-outbreak). A value of “none” here indicates that the population has no vaccination strategy in place
9. vaccinationfraction 0.7
   This value is used to control the percentage of folks that receive a vaccination. However, since there is no vaccination strategy being modeled, this value is irrelevant for the simulation that was run
10. responsethreshhold 0.0
    This value indicates the percentage of the population that needs to be affected by an outbreak before responsive strategies (vaccinations, etc) are employed. Since the value used in this simulation is 0%, it means that responsive strategies will begin with the first incidence of infection
11. responsedelay 9
    This value tracks how many days to wait before responsive strategies are actually employed. So, while the *responsethreshhold* indicates that a responsive strategy should begin on the first infection, this value will delay actual implementation of countermeasures for 9 days (from first infection)
12. ascertainmentdelay 1
    This value represents the number of days that it takes medical personnel to correctly diagnose an incidence of the influenza being simulated
13. ascertainmentfraction 0.6
    This value indicates the percentage of people displaying symptoms of the disease, whom will be diagnosed by medical personnel. For this simulation, the percentage was set to 60%
14. seedinfected 10
    This number represents the initial number of infected people across the entire population. In this case, 10 individuals were infected at the start of the simulation
15. seedinfecteddaily 0
    This number indicates whether or not new infected individuals should be introduced to the population each day. The zero value here indicates that they should only be introduced when the simulation begins

16. AVEs 0.3
    This is how effective the antiviral is at preventing people from contracting infection. The value for our performance test set this to 30%
17. AVEp 0.6
    This is how effective the antiviral is at preventing illness from people that have contracted the infection. The value for our performance test sets this to 60%
18. AVEi 0.62
    This is how effective the antiviral is at preventing those infected from further infecting others. The value for our performance test sets this at 62%

List of Figures

References

[1] Abdulhadi Ali Alghamdi. Performance analysis of hybrid omp/mpi and mass. From PDF attached to email sent to thesis author, May 2015.

[2] Dieter an Mey. compunity: The community of openmp users, researchers, tool developers and providers. `http://www.compunity.org/training/faq/index.php`, Nov 2012.

[3] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[4] Narayani Chandrasekaran. Implementation of mass c++ library over cluster of multi-core computing nodes. CSS596 Capstone Final Report, Computing and Software Systems, University of Washington Bothell, appearing at `http://depts.washington.edu/dslab/MASS/reports/NarayaniChandrasekaran_au12.docx`, December 2012.

[5] DL Chao, ME Halloran, VJ Obenchain, and IM Longini Jr. Flute, a publicly available stochastic influenza epidemic simulation model. *PLoS Comput Biol*, 6(1), 2010.

[6] Timothy Chuang. Design and qualitative/quantitative analysis of multi-agent spatial simulation library. Master's thesis, University of Washington, 2012.

[7] Timothy Chuang and Munehiro Fukuda. A parallel multi-agent spatial simulation environment for cluster systems. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 143–150, Dec 2013.

[8] Wikipedia contributors. Pearson product-moment correlation coefficient. `http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient`, April 2014.

[9] Wikipedia contributors. Sugarscape. `http://en.wikipedia.org/wiki/Sugarscape`, April 2014.

[10] Wikipedia contributors. Message passing interface. `http://en.wikipedia.org/wiki/Message_Passing_Interface`, May 2015.

[11] Wikipedia contributors. Openmp. `http://en.wikipedia.org/wiki/OpenMP`, April 2015.

[12] J. Emau, T. Chuang, and M. Fukuda. A multi-process library for multi-agent and spatial simulation. In *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on*, pages 369–375, Aug 2011.

[13] The Eclipse Foundation. Eclipse ptp. `https://eclipse.org/ptp/`, Jan 2015.

[14] Munehiro Fukuda. Css 543 - program 2: Multithreaded schroedingers wave simulation. `http://courses.washington.edu/css543/prog/prog2.pdf`, Dec 2013.

[15] Munehiro Fukuda. Css 534 - parallel programming in grid and cloud. `http://courses.washington.edu/css534/`, Dec 2015.

[16] Munehiro Fukuda and Distributed Systems Lab Members. Sensor cloud integration: An agent-based workbench for on-the-fly senor-data analysis. `http://depts.washington.edu/dslab/SensorCloud/`, April 2012.

[17] Osmond Gunarso. Flute-mass. From PPT presentation provided to Distributed Systems Lab, May 2015.

[18] Jay Hennan. Baseline. MASS Performance Testing Application, Dec 2014.

[19] Hung Ho. Asynchronous and automatic migration of agents in mass. CSS596 Capstone Final Report, Computing and Software Systems, University of Washington Bothell, to appear in http://depts.washington.edu/dslab/MASS, June 2015.

[20] Cameron Hughes and Tracey Hughes. *Parallel and distributed programming using C++*. Addison-Wesley, Boston, 2004.

[21] IBM. Dynamic performance monitoring for openmp. http://www.research.ibm.com/actc/projects/dynaperf2.shtml, Jan 2004.

[22] Somu Jayabalan. Field-based job dispatch and migration. Master's thesis, University of Washington, 2012.

[23] Prabhanjan Kambadur, Douglas Gregor, and Andrew Lumsdaine. Openmp extensions for generic libraries. In *OpenMP in a New Era of Parallelism*, volume 5004, pages 123–133, West Lafayette, IN, USA, May 2008. Springer-Verlag Berlin Heidelberg. In 4th International Workshop, IWOMP 2008, Proceedings.

[24] Jennifer Kowalsky. Mass c++ updates. CSS497 Capstone Final Report, Computing and Software Systems, University of Washington Bothell, to appear in http://depts.washington.edu/dslab/MASS, June 2015.

[25] Elad Mazurek and Munehiro Fukuda. A parallelization of orchard temperature predicting programs. In *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 179–184, Aug 2011.

[26] Microsoft. Using the hpc cluster debuggers for soa and mpi applications. https://www.microsoft.com/en-us/download/details.aspx?id=23213, Jan 2015.

[27] Bhargav A. Mistry. Dynamic load balancing in mass. Master's thesis, University of Washington, 2013.

[28] Chris Rouse. Agents implementation for c++ mass library. CSS497 Capstone Final Report, Computing and Software Systems, University of Washington Bothell, appearing at http://depts.washington.edu/dslab/MASS/reports/ChrisRouse_wi14.pdf, Winter 2014.

[29] J. S. Squire and S. M. Palais. Programming and design considerations of a highly parallel computer. In *Proceedings of the May 21-23*, pages 395–400, Detroit, Michigan (USA), May 1963. Spring Joint Computer Conference.

[30] Cherie Lee Wasous and Munehiro Fukuda. Distributed agent management in a parallel simulation and analysis environment. Master's thesis, University of Washington, 2014.

[31] Kathleen Weessies. Finding census tract data: About census tracts. http://libguides.lib.msu.edu/tracts, May 2015.

[32] Michael Wong, Eduard Ayguad, Justin Gottschlich, Victor Luchangco, Bronis R. de Supinski, Barna Bihari, and other members of the WG21 SG5 Transactional Memory Sub-Group. Towards transactional memory for openmp. In *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766, pages 130–145, Salvador, Brazil, September 2014. Springer International Publishing. In 10th International Workshop on OpenMP, IWOMP 2014, Proceedings.