Louisiana State University
LSU Digital Commons

LSU Doctoral Dissertations | Graduate School

2017

Decentralized Scheduling for Many-Task Applications in the Hybrid Cloud

Brian Lyle Peterson
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer Sciences Commons

Recommended Citation
Peterson, Brian Lyle, "Decentralized Scheduling for Many-Task Applications in the Hybrid Cloud" (2017). *LSU Doctoral Dissertations*. 4223.
https://digitalcommons.lsu.edu/gradschool_dissertations/4223

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contactgradetd@lsu.edu.

DECENTRALIZED SCHEDULING FOR MANY-TASK APPLICATIONS IN THE HYBRID CLOUD

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Electrical Engineering and Computer Science

by
Brian Peterson
B.Software Engineering, Auburn University, 2005
May 2017

Table of Contents

Abstract iii

Chapter 1: Introduction 1
1.1 Problem Description . . . 1
1.2 Problem Statement . . . 1
1.3 Simulation Introduction . . . 2
1.4 Lessons Learned from Simulations . . . 2
1.5 Cloud Vector Scheduling Approach . . . 4

Chapter 2: Related Work 6
2.1 Improving Scheduling Algorithms . . . 6
2.2 Literature Review . . . 7
2.3 Scheduling Algorithms . . . 8

Chapter 3: Framework 12
3.1 Simulation Methodology . . . 12
3.2 Cloud ATC Experimental Methodology . . . 14
3.3 Vector-Based Scheduling . . . 16

Chapter 4: Software Architecture 19
4.1 Platforms and Resource Management . . . 19
4.2 Intra-Platform Communication . . . 25
4.3 Vector-Based Work Redistribution . . . 37
4.4 Experiment Parameterization . . . 39
4.5 Experiment Timeline . . . 42

Chapter 5: Experimental Results 47
5.1 Simulation Results . . . 47
5.2 Cloud ATC Results . . . 52
5.3 Vector Scheduling Experimental Definition . . . 54
5.4 Vector Scheduling Experimental Description . . . 56
5.5 Initial Cloud Vector Scheduling Results . . . 58
5.6 Cloud Vector Scheduling Experimental Results . . . 60

Chapter 6: Summary 67

Chapter 7: Future and In Progress Work 70

Chapter 8: Vita 76

Abstract

While Cloud Computing has transformed how we solve many computing tasks, some scientific and many-task applications are not efficiently executed on cloud resources. Decentralized scheduling, as studied in grid computing, can provide a scalable system to organize cloud resources and schedule a variety of work. By measuring simulations of two algorithms, the fully decentralized Organic Grid, and the partially decentralized Air Traffic Controller from IBM, we establish that decentralization is a workable approach, and that there are bottlenecks that can impact partially centralized algorithms. Through measurements in the cloud, we verify that our simulation approach is sound, and assess the variable performance of cloud resources. We propose a scheduler that measures the capabilities of the resources available to execute a task and distributes work dynamically at run time. Our scheduling algorithm is evaluated experimentally, and we show that performance-aware scheduling in a cloud environment can provide improvements in execution time. This provides a framework by which a variety of parameters can be weighed to make job-specific and context-aware scheduling decisions. Our measurements examine the usefulness of benchmarking as a metric used to measure a node's performance, and drive scheduling. Benchmarking provides an advantage over simple queue-based scheduling on distributed systems whose members vary in actual performance, but the NAS benchmark we use does not always correlate perfectly with actual performance. The utilized hardware is examined, as are enforced performance variations, and we observe changes in performance that result in running on a system in which different workers receive different CPU allocations. As we see that performance metrics are useful near the end of the execution of a large job, we create a new metric from historical data of partially completed work, and use that to drive execution time down further. Interdependent task graph work is introduced and described as a next step in improving cloud scheduling. Realistic task graph problems are defined and a scheduling approach is introduced. This dissertation lays the groundwork to expand the types of problems that can be solved efficiently in the cloud environment.

Chapter 1
Introduction

1.1 Problem Description

Many scientific fields, such as quantum chemistry, genomics, phylogenetics, astrophysics, geophysics, computational neuroscience, or bioinformatics, require massive computational power and resources that might exceed those available on a single local supercomputer. Historically, there have been two drastically different approaches for harnessing the combined resources of a distributed collection of machines: traditional computational supercomputer grids and large-scale desktop-based master-worker grids. Over the last decade, a commercial alternative has become available with cloud computing. For many smaller scientific applications, using cloud computing resources might be cheaper than maintaining a local supercomputer. However, cloud computing has not yet been successful for high-performance computing applications.

Virtual machines may never be competitive for running large high-performance computing applications with fine-grained parallelism, such as large dense matrix multiplications. For applications that can be broken into a set of smaller tasks, however, it may be possible to match the performance requirements of a task with the performance characteristics of a subset of the cloud nodes. For grids and supercomputers, this *many-task computing* approach [32] has proven very effective. E.g., Rajbhandari et al. [33] structured the computation of a quantum chemistry tensor contraction equation as a task graph with dependencies and were able to scale the computation to over 250,000 cores. They dynamically schedule tasks on groups of processors and use work-stealing for load balancing.

A possible solution for running high-performance or many-task computing applications in the cloud is to identify the performance characteristics of the cloud nodes and the network connections between them and to map computational tasks onto subsets of the nodes with appropriate performance characteristics. Maintaining this information centrally for a large number of machines, however, is prohibitively expensive. IBM's Air Traffic Control (ATC) algorithm [4] attempts to solve this problem by arranging cloud nodes in groups and by letting the leader of each group (the air traffic controller) direct the compute tasks (aircraft) from a central job queue to their destination worker nodes that will then execute the task. While this approach distributes the load of maintaining performance information for worker nodes, the central job queue is still a potential bottleneck. Also, the air traffic controllers may not have enough information about the computational requirements and communication patterns of the individual tasks.

1.2 Problem Statement

The thesis of this dissertation is that a fully decentralized approach in which the applications decide on which nodes to run will improve the performance of some applications in the cloud. Our approach relies on the nodes to advertise their performance characteristics. If performance information is available from the hypervisor or operating system, we will use

it. Otherwise, the nodes periodically measure their own performance as well as any other relevant metrics. This performance information is then distributed in aggregate form to neighboring nodes along an overlay network. Computational tasks can be inserted into the cloud at any point. The application queries the resource availability through an API and migrates the tasks along the overlay network to where appropriate resources are available. By letting the application decide which nodes to run on, the decision can be based on information about the communication and computation characteristics of the application that may not be available if the decision is made by leader nodes.

Similar to real-world air traffic control, applications can enter the cloud at any point instead of through a centralized job queue. The pilot (application) communicates with air traffic control, e.g., to obtain information about the weather (resource availability) at the destination, but the final decision rests with the pilot in command. As in our approach, aircraft (tasks) travel along air routes (an overlay network) to their destination.

Our approach to high-performance and many-task computing in the cloud is based on our prior work on the Organic Grid (OG), a decentralized desktop grid prototype [11, 12]. Evidence supporting our design is provided by simulations of the Organic Grid and by both simulations and cloud measurements of Barsness et al.'s Air Traffic Control algorithm. A comparison of our ATC simulation and measurements performed on machines requisitioned from the CloudLab system [15] validates our simulation approach. All sets of measurements support pursuing a decentralized approach to scheduling work in the cloud.

1.3 Simulation Introduction

In the next sections, we first describe the Organic Grid and ATC approaches in more detail and then present our simulation approach. We present the results of our simulations that demonstrate that our proposed scheduling approach would result in decreased communication overhead. Cloud experiments are performed and compared to the simulation result. We interpret both the cloud and simulation measurements, and what the OG and ATC comparison tells us about decentralized scheduling. Comparing the simulated and cloud ATC runs allows us both to justify the simulation methodology and present interesting measurements of cloud performance.

After presenting simulations, we present a framework for decentralized scheduling on the cloud. This framework utilizes multiple metrics combined via a vector to drive where work moves on a fully decentralized system. We measure different metrics to drive scheduling, and present the benefits of a variety of approaches. We determine what measurements drive a decentralized scheduler to better scheduling decisions. We present several approaches that did not produce an advantage, and the disadvantages actually encountered. Finally, we discuss task graphs and many-task computing, and how such problems can be approached on the cloud.

1.4 Lessons Learned from Simulations

Utilizing simulations we notice the possibilities of the ATC model of partial centralization along with its drawbacks. We also see the potential benefits of the Organic Grid's performance-based restructuring. Restructuring of the overlay network, as in the Organic Grid, however, has disadvantages for the type of applications we are targeting. Any move-

ment, either of work by scheduling, or of nodes via restructuring, complicates job-to-job communication, as the location of the job and the path of communication itself is subject to change. Additionally, more complicated locking procedures would be required to be sure that jobs are not torn away mid communication by a restructured node, and that a job with a message to send knows where to look to send its result.

The first change we make in our second approach is to prefer work pushing rather than pulling. A node with excess work analyzes itself and its immediate neighbors, and as a result of that analysis it targets the correct neighboring nodes to send work to. We hypothesize that this will be preferable to a work-stealing approach for several reasons. First, and most obviously, rescheduling is done when there is something to reschedule. A work-stealing approach has the fundamental downside of unfulfillable requests in the event that there are fewer jobs than nodes. While proving this was not a fundamental goal of our experiments, we present results in the experimental section that point towards this downside. While we did not implement pure work stealing, we did experiment with allowing nodes to request work from neighbors. These request messages had some benefits in initial phases, but when already overburdened resources were given the additional responsibility of responding to these types of messages, a significant amount of time was wasted that pushed back the completion of the entire set of work. Additionally, implementing work stealing required a slightly more complicated procedure programmatically, as a single entity could no longer be responsible both for initiating rescheduling and starting a new job on a node. A work-pushing approach makes coding the locks around the work queue much more straightforward, as both dequeues, dequeue to start work, and dequeue to move work, can be done by the same entity.

For these reasons, we have built an intermediary system based on the ideas of the Organic Grid, but without some of its key components, especially network restructuring. We will enhance and more completely describe the node quality measurement, and discuss different mechanisms for measuring node performance and the benefits received for having done so. This system can be extended to work with other metrics for scheduling. E.g., for adding measurements of network speed and bandwidth, we can reuse the existing framework for information distribution and use.

We posit that a good job scheduler in the Cloud will have the following capabilities. It should be scalable, able to handle a variety of hardware platforms, as well as multi-site systems. It should match available hardware to job requirements. It should maintain up-to-date information about the status of computing resources. It should be extensible to allow jobs to tailor their requirements to any resource data is available. Our goal is to build a system that could be extended to work with a variety of measurable traits, but we will need to provide at least two specific traits to test the viability of our model. The two traits that seem to be the most important would be the queue length, or the number of jobs that a particular resource has queued to perform, and a measurement of the resource's performance. As a research tool, we need to provide verbose and accurate logging from which we can analyze performance and determine the usefulness of different metrics for scheduling.

Harnessing a variety of computing resources is the fundamental challenge of the hybrid cloud. A hybrid cloud consists of either a set of cloud resources with varying characteristics, or a system that contains a combination of cloud computing and other computing resources.

For example, a company providing a web service might utilize local resources for most of their traffic, but during high traffic times supplement their computing power with resources temporarily allocated from a cloud provider. A research center might have a variety of computing resources available to be allocated from a common portal. Such hybrid clouds provide an interesting scheduling challenge. It may also be the case that even resources allocated from a homogeneous cloud system may differ in performance characteristics. This will be more likely if computing hardware is shared between several processes.

1.5 Cloud Vector Scheduling Approach

We use a vector-based model to describe how we will utilize node information. By describing a vector of coefficients, we can weigh the value of any node information that can be described numerically. By plotting the nodes on an N-dimensional space, the coefficient vector and the dot product formula can be used to find the node with characteristics most similar to the vector. Therefore, it is possible to think of the vector as the direction in which work should flow. Trivially, any scheduler should move work from a high queue length node to a low queue length node. However, if we also take into account a measurement of node performance, it might be beneficial to weigh that value as well when determining the correct location for scheduling. A central component of our experiments will be determining an optimal vector to minimize the time for computing jobs on a decentralized network of nodes.

We are targeting programs that can take advantage of specialized hardware, but do not need to be precisely written for one exact system, programs that lie between extremely parallel programs that can take advantage of any hardware via Map-Reduce and programs written to precisely use a specific piece of hardware, such as a typical supercomputer program. Many-task computing problems seem to be ideal for this environment. While other programs should still be able to be scheduled on our system, we hypothesize that these middle-ground programs would receive the most advantage from a hardware-conscious automatic scheduler.

In a decentralized network, any choice to move work has to be done based on a limited amount of information available at a certain point in the network. We organize our experimental network as a graph of worker nodes. When a set of jobs exist at a node that node will have to make a decision based on its view of the network about whether to leave those jobs to be completed locally or to move them to a adjacent nodes.

Our most immediate issue is how much information an individual node should possess. Our current experiments use nodes that view themselves and a set of immediate neighbor nodes. We have considered the possibility of more distant information but we run into an immediate problem with doing so. When receiving information about more distant resources from neighbor nodes, there does not seem to be an obvious or consistent way to show that those resources are ideally gained through those nodes. In fact, it is possible that this information might have traversed a loop that includes the node which itself is asking for information. Because of scaling, we do not want to maintain specialized information sets for each querying node.

Additionally, we need to measure the usefulness of the data we are gathering. If we measure performance, i.e., the speed at which independent tasks are completed, is our measurement going to correlate to actually completing jobs more quickly? While we had several benchmarks available to us in the form of the NAS benchmark set, which has been used in prior

work to test the effectiveness of cloud computing platforms [19], how relevant are these benchmarks to actual computing tasks? While we were not able to completely answer this question before we started work, with hindsight we can measure this fairly accurately. In Figure 1.1 we can see an imperfect correlation between the job completion times and the measured performance times from the data of specifically the Queue and Performance mixed test that we discuss more in our experimental section. This test utilized the performance data to dictate where jobs were scheduled.

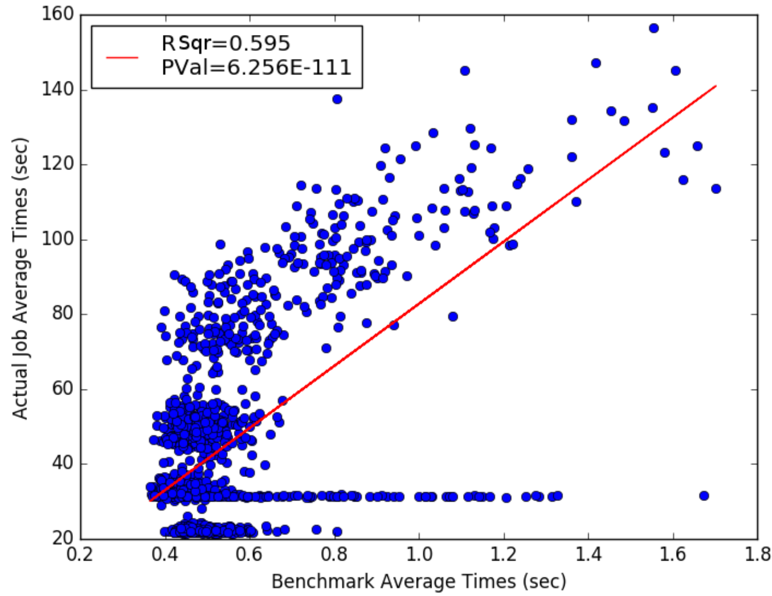

FIGURE 1.1. Average Benchmark Performance vs Average Measured Actual Performance of Performance and Queue test populations

Chapter 2
Related Work

2.1 Improving Scheduling Algorithms

Prior work has shown the possibility of using a mix of decentralized strategies for work scheduling. The Organic Grid used a hierarchical strategy to move high performing nodes closer to a source of work. The ATC uses controller nodes to organize groups of workers, and controllers compete to get work for their worker nodes. Simulations of each of these algorithms have identified the comparative benefits and drawbacks of each approach.

The Organic Grid's organization helps move work to high performing nodes more quickly, however it builds an overlay network centered around a single work source. In a very large scale cloud system, we may not always have a single source for work, so on a larger scale we have to consider how to build a network that takes performance into account when scheduling, but does not rely on a single point source of all work. We want to take the idea of measuring performance and using that information to drive scheduling. We experimented with simulating the Organic Grid, and one trend we noticed was that when work was added in to a new location, the overlay network that was built was destroyed and recreated around the new insertion point. Because in the original Organic Grid, the agents that reorganized the network were associated with individual jobs, the knowledge gained in the previous session was lost and unavailable to the next job to use the system. The Organic Grid is still a useful strategy for scheduling a very large amount of work from a consistent insertion point, but there is room for improvement.

The ATC algorithm contains a degree of mid-level centralization, nodes are categorized into worker and controllers. Controllers organize a group of workers, and take on groups of tasks for workers. Our simulations show that this can improve performance. Centralization allows the benefit of full utilization. A controller knows how many workers it has, and is able to pull a job that will most completely utilize the nodes. A fully decentralized solution will not know exactly how to make an ideally sized job at each scheduling step. However the downside to this solution in our simulations was the communications burden that was placed on the controller nodes.

These experiments suggest many more possibilities than could be tested. The Organic Grid scheduling algorithm is appropriate for a reliable work source, and it might be useful for organizing a set of nodes under a controller, such as the one used in the ATC model. Using the Organic Grid's measurement tools is valuable, however different jobs may have different requirements than a single performance metric. Characteristics such as access speed to a data source, proximity to other nodes, or communication speed may also need to be considered. As jobs might have different requirements, it would be desirable to allow jobs themselves to choose how to prioritize variables that can be used for scheduling.

We want to benefit from the node information used by the Organic Grid, but reorganization may be detrimental both because of multiple job sources and because the initial organization may represent a beneficial hardware organization. We also want to develop an algorithm that

can use different types of node information to make a job-specific scheduling decision. When our only concern is node performance, the correct decision is always to move towards the highest performing node. When we have multiple sources of node information, we need a cost model in order to prioritize different node variables.

It has been shown that for certain problems, especially those that require complex intercommunication between multiple jobs, typical cloud resources, such as Amazon's EC2, do not provide adequate performance. [38] [19] Methods such as MapReduce [16] provide the basis of cloud computing, but require two things: some level of centralization, to map the tasks onto a known set of processors, and independence, so that tasks do not have to be coordinated between the working resources until the reduce step. These are both reasonable restrictions for a wide range of problems, and have allowed the cloud to become a radically successful platform displacing older grid-computing-based systems.

However, what if the problem requires some intercommunication, thereby being unsuited to a simple MapReduce approach, but also is not composed of tasks of perfectly predictable scope and size? We will attempt to schedule many-task problems on cloud hardware in a variety of conditions and test what approaches may be leveraged to improve throughput and ensure resources remain utilized. We will test a mix of interdependent and independent tasks, and analyze how many independent tasks are necessary to keep resources utilized while waiting for dependencies, and what tradeoffs can be made to complete a set of interdependent tasks as quickly as possible.

2.2 Literature Review

Research on traditional grid scheduling has focused on algorithms for determining an optimal computation schedule based on the assumption that sufficiently detailed and up-to-date knowledge of the systems state is available to a single entity (the metascheduler) [22, 7, 1, 37]. While this approach results in a very efficient utilization of the resources, it does not scale to large numbers of machines, since maintaining a global view of the system becomes prohibitively expensive. Variations in resource availability and potentially unreliable networks might even make it impossible.

A number of large-scale desktop grid systems have been based on variants of the master/workers model [6, 35, 39, 20, 14, 29, 30, 24, 27, 9, 26, 25]. The fact that SETI@home had scaled to over 5 million nodes and that some of these systems have resulted in commercial enterprises shows the level of technical maturity reached by the technology. However, the obtainable computing power is constrained by the performance of the master, especially for data-intensive applications. Since networks cannot be assumed to be reliable, large desktop grids are designed for independent task applications with relatively long-running individual tasks.

Cloud computing has been very successful for several types of applications, especially for applications that do not require frequent communication between different cloud nodes, such as MapReduce [16] or graph-parallel [40] algorithms. However, for applications with fine-grained parallelism, such as the NAS MPI benchmarks or atmospheric monitoring programs, it shows less than satisfactory performance [38, 19]. The main reasons are that the performance of cloud nodes is often not predictable enough, especially with virtualization, and that the communication latency is typically worse than that of a cluster [38]. While it is

possible to rent dedicated clusters from cloud providers, they are significantly more expensive per compute hour than virtual machines (VMs). With virtualization, however, the user may not know whether a pair of VMs run on the same physical machine, are in the same rack, or are on different ends of the warehouse.

Additionally, there have been many advances in supercomputer and high-performance cluster computing for problems that require advanced communication [17, 8]. Automatic handling of small-scale and compiler-level parallelization to optimize task-graph problems is a proven approach. If the scope of the problem is known before it is started, and individual task size can be predicted, many specific alterations can be done to improve performance and synchronize communication steps. Problems that alternate between computation and communication steps can be scheduled on known and homogeneous hardware precisely to optimize throughput.

Other cloud computing projects have expanded the realm of what computation is possible in the cloud. Spark [41], a system from Berkeley's Amp Lab, is designed to improve performance on iterative algorithms, such as machine learning algorithms, which repeatedly access a single data set. The primary mechanism is a "resilient distributed dataset" that allows multiple independent processes to access the same set of data more efficiently, at least for read operations, than would otherwise be possible.

Google has dedicated research time to improving distributed application performance [10]. They have moved monitoring of application performance into applications themselves. This was used to improve the understanding of application performance across locations and when scaled up, but not obviously for actual scheduling decisions. Omega, another system described in [10], decentralized some of the scheduling systems, allowing peer schedulers to operate and make optimistic decisions, and avoid a "monolithic centralized master".

2.3 Scheduling Algorithms

In prior work, we have built a prototype of a decentralized desktop grid system, the Organic Grid [11, 12]. In the Organic Grid, Java mobile agents are utilized to wrap and transport tasks to the nodes on which they are executed. Nodes start with connections to a set of neighbor nodes, as shown in Figure 2.1. The tasks collectively form an application. An application can be added to any node, and will be completed by the network of nodes connected to it. Nodes without work will utilize work stealing to get jobs from neighbors who possess extra work. Some applications are too large for a single agent to complete. In this case an agent may clone itself, and move that clone to a location requesting work. The new clone will receive a portion of the application tasks. Cloned agents will complete their tasks and any results will be reported back to the original work source. If the work assigned to a clone is still too large, the cloning process may be repeated, creating an overlay network. This network is defined by the connections between cloned agents as the application tasks are spread through the network of nodes. This overlay network will form a tree in which the root is the location at which the application was originally added to the network.

The Organic Grid is designed to move high-performing nodes closer to the root node of the overlay network. Because work radiates outwards from the root node, this means that high-performers will be the immediate choices for any work being moved out of the root. Parent nodes measure the time taken by child nodes when completing individual tasks. This

FAST
MEDIUM
SLOW
KNOWS ABOUT

FIGURE 2.1. Organic Grid Initial Network

information is used to identify high performing child nodes. High performing child nodes are candidates to be promoted to a parent's parent, moving them up the overlay network and closer to the root of the tree, as seen in Figure 2.2. This process could result in too many high performing nodes communicating directly with the root. To avoid this, the number of possible direct children of any one node is limited. This limit may be altered on a per-application basis. When tasks are moved to a child node, an algorithm similar to the TCP sliding window protocol is used. This allow us to overlap communication and computation. It has been demonstrated that this approach will result in improved performance for two applications. The BLAST sequence alignment application [11], and a Cannon-style distributed matrix multiplication [12]. The overlay network restructuring algorithm was identical for both application. However to achieve optimal performance some parameters were altered in the different runs. The maximum number of children a node could have, and the details of the sliding-window protocol to stream tasks, were different for the two different applications.

In the Organic Grid, the agents themselves measure the performance of their child nodes without any support from the underlying agent platform. In a cloud computing environment, this would be undesirable, since the performance information would be lost when an application terminates. We, therefore, propose that the nodes measure the performance independently of any task and provide that information to applications through an API.

We also build upon the intelligently self-organizing network described in IBM's Air Traffic Controller patent [4]. In this algorithm, groups of computational nodes are defined at the start, and provided with a leader (air traffic controller). These leaders query a central job queue, and communicate with other leaders to acquire work for their groups. A leader will

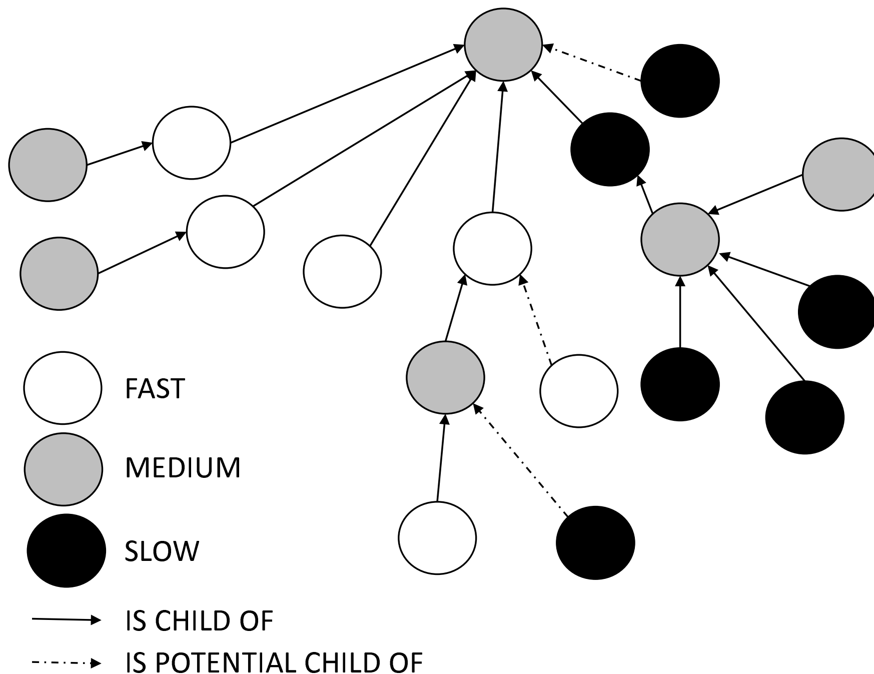

FIGURE 2.2. Organic Grid after Network Restructuring

choose the best task for its group, and then either request more workers from other leaders, or spawn a smaller group to size itself correctly to the task. While these groups may resize, this middleman centralization means that tasks are always only two hops away from the node that will complete them. This, however, does create a high communication burden both on the single job queue and the leader nodes that are organizing workers.

In Figure 2.3, we show the organization of an ATC network, with each leader controlling a subset of worker nodes. Two jobs are currently in the queue, the smaller Job 1, and the larger Job 2. Figure 2.4 displays how the network changes as the jobs are distributed. Group leader A will split off a smaller network, A', to size itself appropriately for the smaller Job 1. Group leader B will merge with C to create a larger group to tackle the larger Job 2. Notice that the new merged or split groups maintain the intra-leader relationships that existed before the changes, so that the graph of group leaders cannot become disconnected. Additionally, a new relationship between the split groups lead by A and A' is created. If, in the future, A's group needed to grow to take on a larger job, it would attempt first to re-merge with A' before querying other nearby leaders, as it is assumed that the initial group layout of the network reflects the physical organization of the system. Arranging groups in a manner that reflects hardware organization will result in groups of jobs being assigned to groups of nodes with superior interoperability.

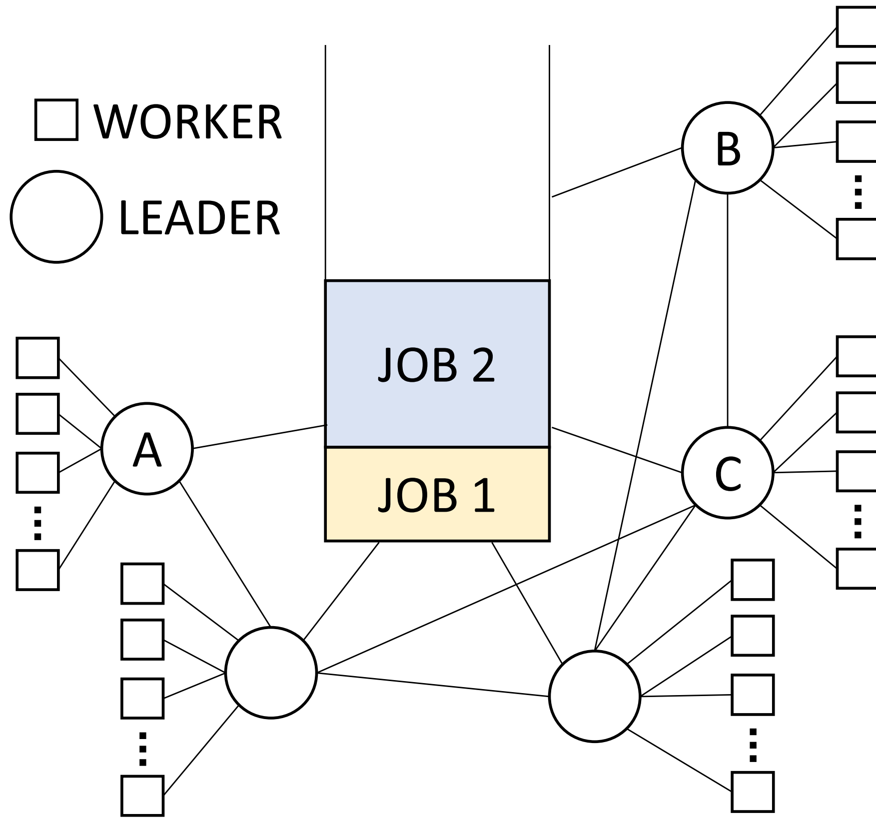

FIGURE 2.3. Air Traffic Controller Initial Network

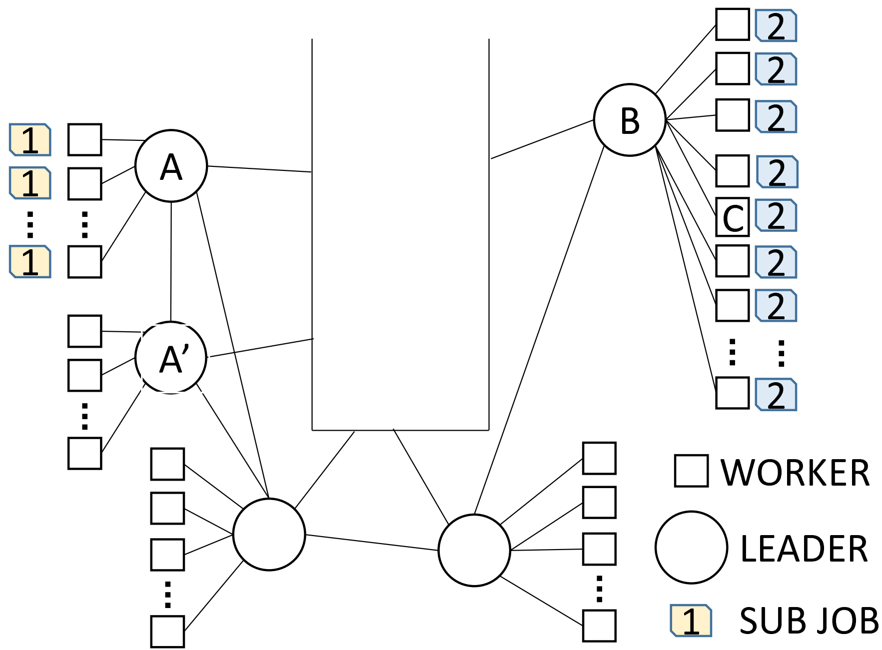

FIGURE 2.4. Air Traffic Controller after Work Distribution

Chapter 3
Framework

3.1 Simulation Methodology

We have developed a simulation application to help us understand the strengths and weaknesses of different methods used to distribute work. This tool provides the measurements necessary to justify our approach to distributed computing and guide us towards how it will be implemented. It gives us a way to measure and compare the behavior of existing algorithms, and understanding what factors play the most significant role in how different approaches distribute work. We have measured simulations of both the Organic Grid (OG) and Air Traffic Control (ATC) algorithms in several different configurations to try to understand where each approach is superior and how we can develop a new system that may combine the strengths and mitigate the weaknesses of each.

A primary concern when attempting to create a simulation is showing that it is an accurate representation of the system that it simulates. We do not attempt to measure the computational time taken, we simply measure the messages sent and the burdens on individual computing nodes. By observing and categorizing the messages sent by each algorithm, we are able to compare the ways each algorithm places burden on the network. Our primary assumption is that any implementation of these algorithms on a real network would require the same types of messages in order to distribute and complete similar work. The computational time taken by the simulation, on the other hand, might be an inferior measurement because it might not match with real-world implementations of the tested algorithms. For this reason, we do not present time measurements in this preliminary investigation, we are more interested in examining how each algorithm's distribution method can be optimized than proving which is fastest.

Additionally, how can we be certain that what we have done will be relevant in an implementation on the cloud? The simulation attempts to accurately mimic different distributed algorithms on a common simulated set of nodes, and produces measurements to distinguish how those nodes will be burdened and how they will be utilized by each algorithm. On a cloud implementation, the differences in node performance may come from other factors, such as multiple processes competing for processor time, or delays due to communication overhead, but the simulation will still show how each distributed algorithm responds to these stressors even if the source of the stress is a simplified simulation parameter. And more importantly, the simulation can still guide us to the best experiments to run on a real implementation, and help us identify the variations on each algorithm that can result in the largest changes in performance or balance.

With the Organic Grid, we have the benefit of past prototypes against which we can compare our simulation. The Organic Grid organizes its overlay network and moves high-performing nodes closer to the root, or source of work. In the original Organic Grid paper [12] this was done by starting with a random graph of "fast" and "slow" nodes, constructing an overlay tree for disseminating tasks and collecting results, and, based on the performance

of the nodes, migrating the fast nodes to near the top, leaving the slow nodes near the bottom. Figures 2.1 and 2.2 show the start configuration for a sample run and the result of restructuring the overlay network, respectively. Our simulation produces similar results, although instead of designating only two categories of nodes, we assign each node a performance modifier as a floating point number between 0.5 and 1.0. In Figure 3.1, we can see the resulting overlay network from a simulated run of the Organic Grid algorithm. The nodes begin the experiment with randomly selected neighbors, and a randomly selected node is chosen to be the source of work. Note that the root, or source of work, cannot be switched out, and, therefore, remains a relatively slow node. However, its immediate children are very high performing nodes. The ordering is not completely based on performance, as not enough time has elapsed to force every high performing node higher in the network and there is a limit to the number of immediate children any node can have. Comparing this to Figure 2.2 demonstrates that the simulation can accurately reproduce the behavior of the Organic Grid algorithm.

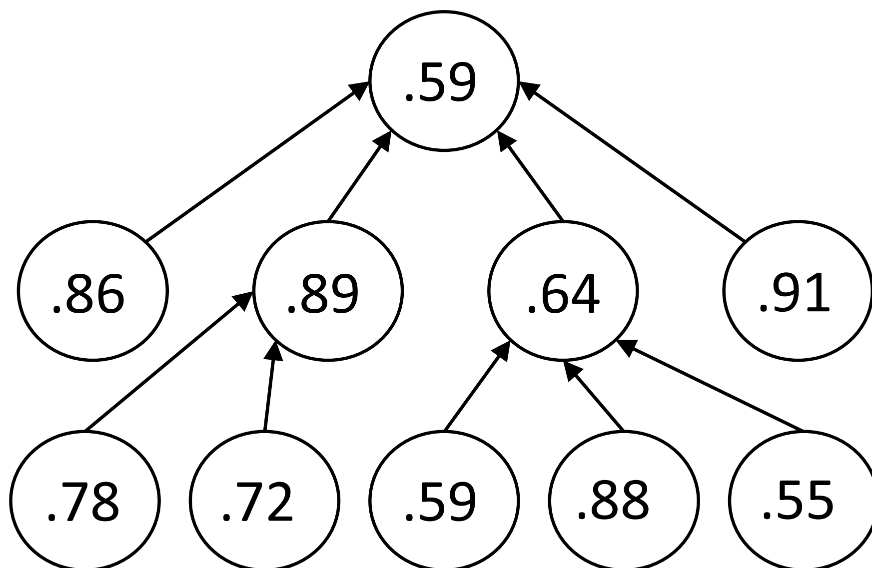

FIGURE 3.1. Node Tree From OGSimulation

Defending the simulations of the Air Traffic Controller algorithm is more difficult, as we are referring to a patent and not a prior implementation. There are a range of possible implementations discussed in IBM's patent [4]. We simulate the basic system as described in Section II, monitoring communication as it takes place through leader nodes, while the leader nodes utilize a decentralized network to reorganize themselves and their worker nodes appropriately based on the sizes of jobs added to the job queue. To accurately compare the ATC and OG algorithms, these nodes have the same performance modifiers, however, because the ATC does not utilize this information when reorganizing the network, it has no visible effect on the algorithm's actions, only the speed at which work is completed. We monitor messages from the job queue, among the leader nodes as they organize themselves, and from leader to worker nodes as jobs are assigned.

Messages in our simulated network are organized into information, job, and result types. We are most interested in job messages, which move work from node to node through the network, and the result messages, which move the results of completed work back to the point at which that work was inserted. A message that moves computation or results from one location to another is in all likelihood larger than a simple status or heartbeat message, and unlike a request for work, will occur more often as the network becomes more heavily burdened. It is, therefore, not just a good measurement of how well the algorithm distributes work, but by examining which nodes participate in which messages, it is also a good measurement for estimating which nodes are disproportionally burdened by the communication requirements of a distributed algorithm.

Grouping messages by type also allows us to distinguish signal from noise. For example, as both algorithms we examine are work-stealing approaches, there will be an increasing amount of unfulfilled requests for work as a system becomes less burdened. These requests are much less interesting than work transfers, and their overhead will increase when the system does not have other meaningful work to do. Finally, by examining the participants in specific messages, we can analyze which nodes in the system are playing a disproportionally large role in managing information. While we remove statistical outliers in most of our measurements, it is useful here to examine individual participants that are consistent outliers. We examine the top three most overburdened nodes under each configuration, organized by the type of message, to determine if any interesting patterns arise.

In order to compare algorithms in different environments, many simulations were performed with several different values for relevant variables. The ones we compare here are based on the number of nodes in the network, (Node Count or NC in the figures), the number of jobs added to the system (Job Count or JC), and the amount of time each piece of a job takes to complete (Job Time or JT), which we measure in milliseconds. These three variables are shared by the two algorithms we simulate and make the best points of comparison, both for these algorithms and for any future algorithms we may wish to simulate and measure. While we do not scale the number of nodes in our simulation as far as we will eventually scale our implementation, the varying size measurements still serve to show measurable differences between the OG and ATC methodologies.

3.2 Cloud ATC Experimental Methodology

For validating the simulation results, we have run experiments on the CloudLab platform [15]. CloudLab is designed to allow for repeatable experiments performed in the cloud. By defining a profile that describes the machines used and using randomly generated but preserved machine configurations, we can create a variety of experiments and maintain an experimental log that will allow these experiments to be reliably replicated [34]. We used a machine template running Docker 1.6.2 on Ubuntu 12.04 LTS. These machines were provisioned on the APT Utah Cluster, which possesses two classes of nodes. The information in Table 3.1 on the available hardware is from the CloudLab Utah hardware description [3]:

TABLE 3.1. CloudLab Utah Hardware

Name	CPU	RAM
r320	1x Xeon E5-2450 processor	16 GB
c6220	2x Xeon E5-2650v2 processors	64 GB

Each node is connected to three network interfaces, a public Internet facing control network, a "flexible fabric" network, and a "commodity fabric" network [3].

The cloud experiment utilized nodes that were requisitioned from CloudLab in a common manner. While the simulation artificially created performance distinctions between different nodes, that was not the case in the cloud simulation. Distinctions in performance are due to differences in the hardware, in the load on that hardware, on the communication latency, and on differences in workload on the participating nodes. Even when all nodes are pulled from the same system, there may be some measurable variation in performance. Measurements of node performance are taken and compared to judge whether these similar nodes will perform measurably differently.

Our experimental platform is a Python application that uses the Python-twisted libraries for networking, packaged in a Docker container. Each running Docker instance maintains a server program that listens for incoming messages and a thread of execution to generate messages to send to other instances. This program is implemented primarily in a base class called `SimMgr`. This class is extended by other classes to specialize into an `ATCSimMgr` and `ExpSimMgr`. `ATCSimMgr` has the specialized logic required to take on the roles of nodes in an ATC system, including the Queue, Controller, and Worker nodes. It also contains extensive local logging, for both debugging and experimental measurement purposes. We will refer to Docker instances running `ATCSimMgr` nodes as *experiment nodes*. `ExpSimMgr` allows a Docker instance to manage and report on an entire experiment as an *experimental monitor*. These nodes contain information about every experimental node in an experiment, they dictate roles to experimental nodes, and once a given experiment is completed they contact the nodes to gather and collect their individual log information. Logs are written each time useful messages are sent or received. These collected logs are the basis for the cloud-based measurements presented below.

Experimental nodes listen for the following messages:

- Send Experimental Logs
- Terminate
- Add Batch Job
- Batch Job Done
- Add Sub Job
- Sub Job Done
- Set Type

  - Controller, Queue, or Worker
  - Also informs controllers of queue location

- Add Sub Nodes (assigns workers to controller)

The experimental monitor listens for the following messages:

- Terminate
- Receive Logs
- Receive experimental node
- Batch Job Done

We simulated work in the form of 200 batch jobs, each of which contains 18 parallel portions. Each of the parallel portions range in time from 100 to 200 milliseconds. These 200 batch jobs are added to the queue node and given on a first-come first-served basis to controllers that request work. A controller then allocates work for each of its worker nodes. Once a worker completes a piece of work, it reports that to the controller, and once the controller completes a batch job, it reports to the queue. Once all jobs in the queue have completed, the experiment is over and the logs are collected.

In the cloud application, we bundled messages when communicating from a controller to the queue, which results in a reduced number of messages for the queue. This can be seen as an improvement over the original simulation methodology, however it does not necessarily translate into a reduced communication load overall, since the same information must be transmitted. Depending on the algorithm implemented, and how much information returned to the point of origin could be reduced in size, the burden on the queue could grow or shrink. However the burden on the controllers, which is more interesting, is primarily based on the number of workers that it manages. Although the overall burden will be determined by the nature of the work, the disproportionate burden will be determined by the structure of the network.

3.3 Vector-Based Scheduling

What we are calling *vector-based scheduling* is the idea that there is a direction in which work should move through a network of nodes. We theorize that if the nodes can be placed in an n-dimensional space based on n measurable attributes, the best direction for work to move can be defined as a vector in that space. Therefore, we measure attributes about nodes at a given time, such as their current workload, or their performance on a given benchmark. After we describe each node's position, we can compare those positions to an ideal vector using the dot product operation in order to determine which nodes most fit the direction defined by that vector. By collecting the dot product values of a group of nodes, we can create a weighted job distribution from any individual node to any set of neighbors. Using this methodology, we can make distributed scheduling decisions based on recent information at any point in the network of nodes.

As an example, suppose we measure only node performance and queue length. Node A enters a redistribution cycle, as a result of having a job surplus. Its neighbors are nodes B, C, and D. A has the information about itself and its neighbors as seen in Table 3.2.

TABLE 3.2. Node Information

Node	Queue Length	Benchmark (sec)
A	7	0.8
B	1	0.1
C	0	0.2
D	1	0.6

According to the table, A has a high number of jobs, and a it took 0.8 seconds to complete the benchmark run. As a general rule, we expect that for the work we are doing, moving jobs towards unburdened nodes will be a good scheduling strategy. Unless the cost of movement is greater than the total cost of computation time, this will be the case. We also hope to show that moving jobs towards higher performing nodes will be beneficial, and to what degree this is the case. B in this scenario is the highest performing node, and C is the node with the fewest jobs in its queue.

We can either view these characteristics as facts about the nodes, or we can view them comparatively by transforming them to a known range. Transformed, we can distinguish between comparatively good characteristics. For evenly burdening the network, which is important if no one node has global information, this is valuable, however it can have the negative result of magnifying small differences in performance. Note that this example occurs for one scheduling choice at one point in time, therefore we will transform the set 7 1 0 1 into the range [-1, 1]. If another scheduling decision is made at a later time, the data that exists at that point will be transformed independently of this decision point. The transformed data is shown in Table 3.3

TABLE 3.3. Node Info on [-1,1]

Node	Queue Length	Benchmark
A	1	1
B	-0.714	-1
C	-1	-0.714
D	-0.714	0.429

Now we can describe the direction we want work to move in terms of a vector in this space. We experimentally test several vectors. For this example, we use (-0.9, -0.3). If we consider the first coordinate to represent the weight given to Queue Length, and the second coordinate as the weight given to Benchmark Performance, we have a space as shown in Figure 3.2.

An additional factor to consider when determining where to move jobs is the cost of movement. The communication burden of moving a job from one location to another is always present, and may vary from job to job depending on the memory requirements of

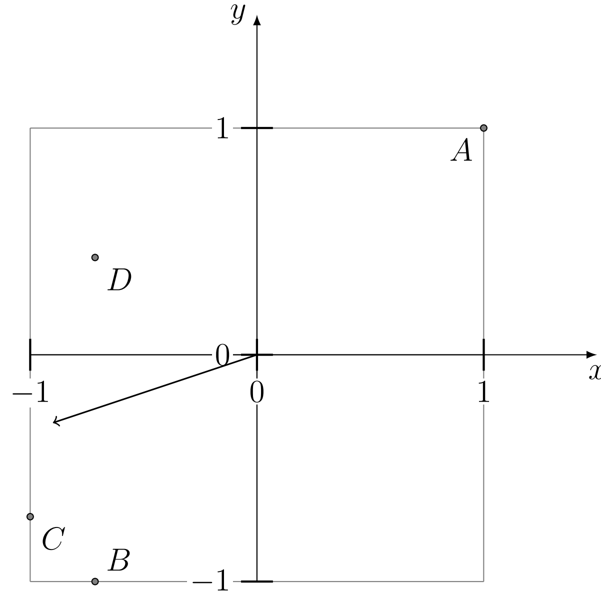

FIGURE 3.2. Node Locations

a job. Additionally, in a decentralized network of independent decision makers, there is a danger of a thrashing behavior, in which jobs are repeatedly shifted around between the same set of nodes. We have implemented two policies to mitigate this problem, although it may be counterproductive to completely eliminate it.

The first and simplest rule is to provided a minimum queue length below which a node will not attempt to move jobs. For our initial experiments that queue length is 1. The second possibility, which is more in line with our experimental methodology, is to provide a stasis vector as an experimental parameter. This vector offsets the metrics of the work source node to make it a more likely candidate for work redistribution. This stasis vector is not large enough to override a significant difference in measured node quality, but should be large enough to keep insignificant changes in measured node quality from prompting wasteful rescheduling. Proving whether a stasis vector is useful, and what characteristics it should have will be a focus of the experimental stage.

Chapter 4
Software Architecture

4.1 Platforms and Resource Management

To test the vector scheduling algorithm, we have implemented a decentralized scheduling program for the cloud. The primary software artifact is the platform. Platforms are stationary on a piece of hardware, and they present that hardware's resources to the decentralized network of platforms through neighbor relationships. They also control a queue of work for that piece of hardware, starting jobs and reporting completed ones. Platforms consist primarily of a TCP/IP server that listens for incoming messages and responds to them. The specific class hierarchy of the current Platform utilizes the `BenchNodePM` class, which inherits from the `NodePlatformManager` class, which inherits from the `PlatformManager` class. Certain basic functionality that defines the platform is spread through these three classes.

Platforms are initiated via the script StartNBenchPM.bash, which will start a parameter defined number of platforms based on input arguments. This directly calls the BenchNodePM.py file as an argument, therefore the entry point for execution of one individual Platform is at the end of the BenchNodePM.py file. This file pulls the IP and Port that the platform will listen on, as well as the IP and Port of an experimental manager that it will report to. An instance of `BenchNodePM` is created, and its member function `StartAll` is called. `StartAll` is defined in `NodePlatformManager`, which also calls the `StartAll` implementation in the parent class `PlatformManager`. `StartAll` is designed to start the fundamental threads used by the platform manager.

The three threads started by each platform are `filewaiter`, `serverThread`, and `managerThread`. `serverThread` will listen for incoming messages at the parameter defined platform IP and Port. It will be utilized to respond to incoming messages. `managerThread` will manage the local work queue, dequeuing available work and checking and responding to completed work. `filewaiter` is designed for the limited purpose of handling larger incoming messages that we do not want to block on. When the conceptual object that is a job is being transferred from platform to platform, it may transfer one or more files that may involve a few connections. `filewaiter` manages partially transferred jobs, keeping them in an incoming queue, monitoring for the transfer to be complete, and then finally dropping them into the actual queue once transfer has completed successfully.

The platform carries a few member variables that fundamentally define its state. The following is not an exhaustive list of member variables, but instead a grouping of the most meaningful data structures that will be acted on by the Platform to complete work and interact with neighbors:

1. `workqueue`, the list of local jobs.

2. A list of neighbors, and the IP/Port information to message them

3. `ExecContainer`, the thin manager of a currently executing job.
4. Measurement data about itself that it can present to neighbors.
5. Times when measurements were taken, in case they should be retaken.
6. A data structure containing the information neighbors have provided it.
7. An instance of `MsgMonitor`, which is in charge of making sure that outgoing messages send successfully.

The breakdown of the platform classes is visible in Figure 4.1, as well as separate figures for each class, Figure 4.2 (`PlatformManager`), Figure 4.3 (`NodePlatformManager`), Figure 4.4 (`ExpPlatformManager`), Figure 4.5 (`BenchNodePM`), and Figure 4.6 (`TaskGraphPM`). The `PlatformManager` class is the parent class for all platforms, both `ExpPlatformManager`, which runs experiments and does not perform work and the `NodePlatformManager` then `BenchNodePM`, then `TaskGraphPM` classes, which are experimental platforms that gradually build more sophisticated experimental capabilities. The `msgmon` variable, which is utilized to send messages between platforms, is defined at the `PlatformManager` level, as it is used by both the experimental platform and node platform hierarchy. The `workqueue`, which is relevant only to nodes that actually perform work, is defined at the level of the node platform. Benchmarking and vector scheduling work is divided, but much of it is added at the level of `BenchNodePM`. The ability to execute task graphs is largely defined at the level of `TaskGraphPM`. This design is largely the result of the gradual implementation and testing of different and enhanced features over time, with the desire to impact as little as possible previously implemented functionality, so that comparison between different implementations would be as fair as possible.

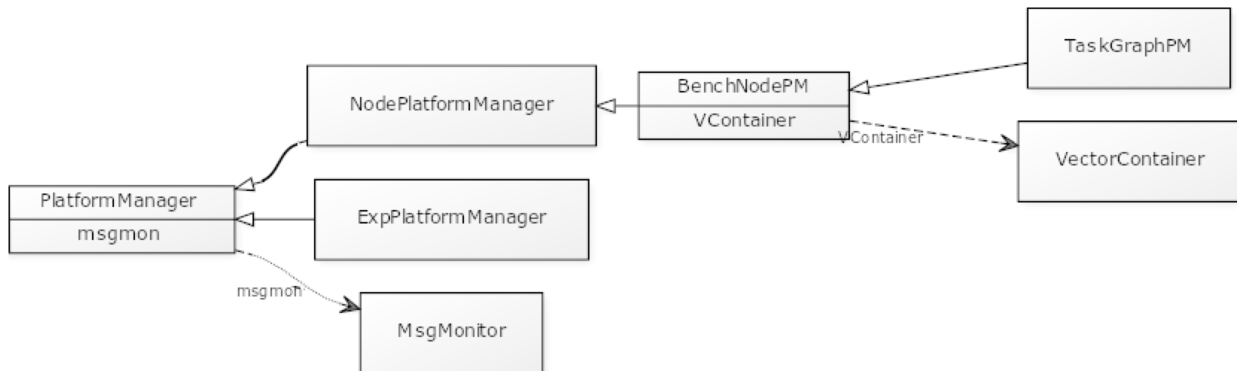

FIGURE 4.1. UML Diagram of Platform Manager Hierarchy

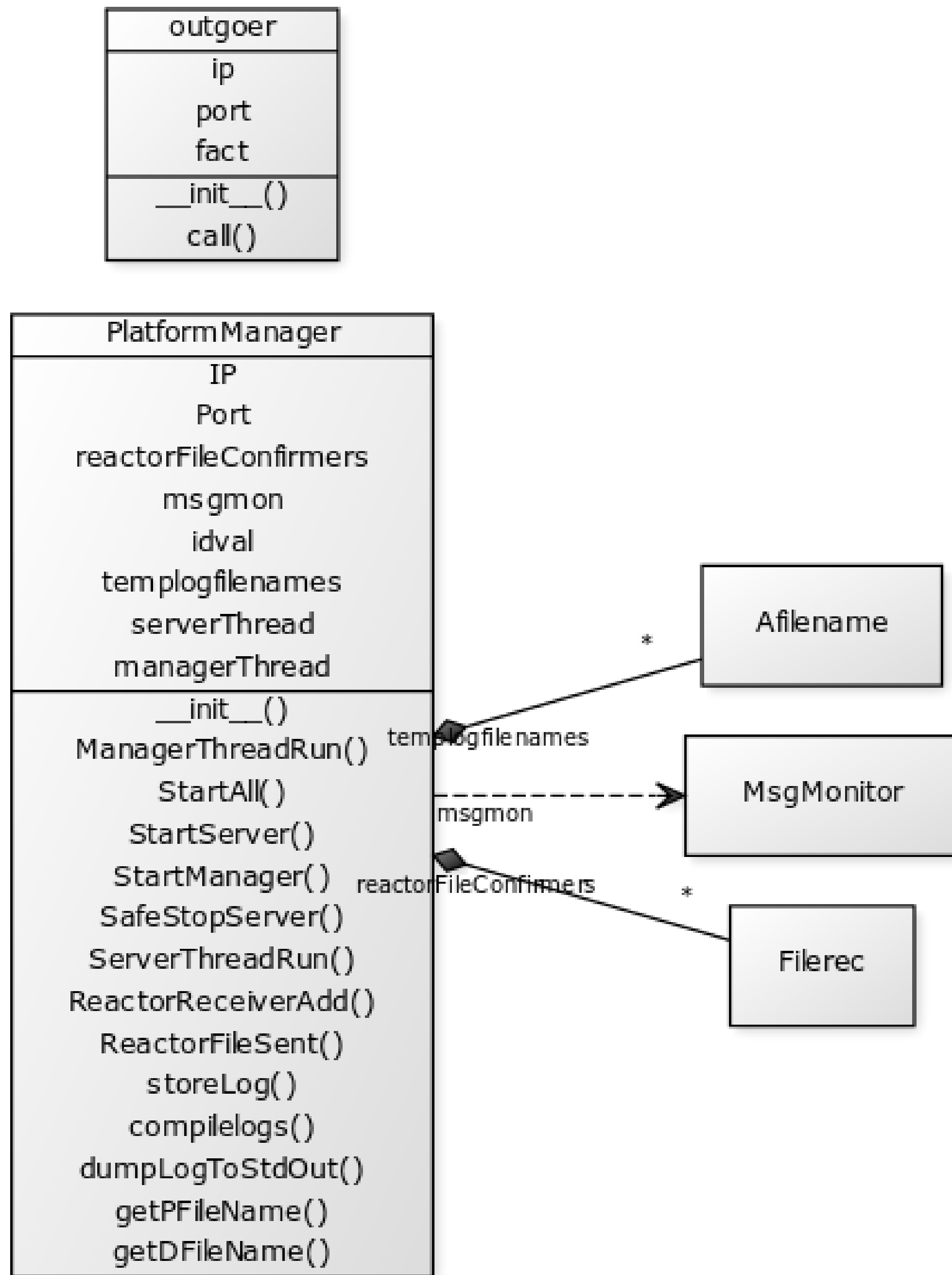

FIGURE 4.2. UML Diagram of Platform Manager

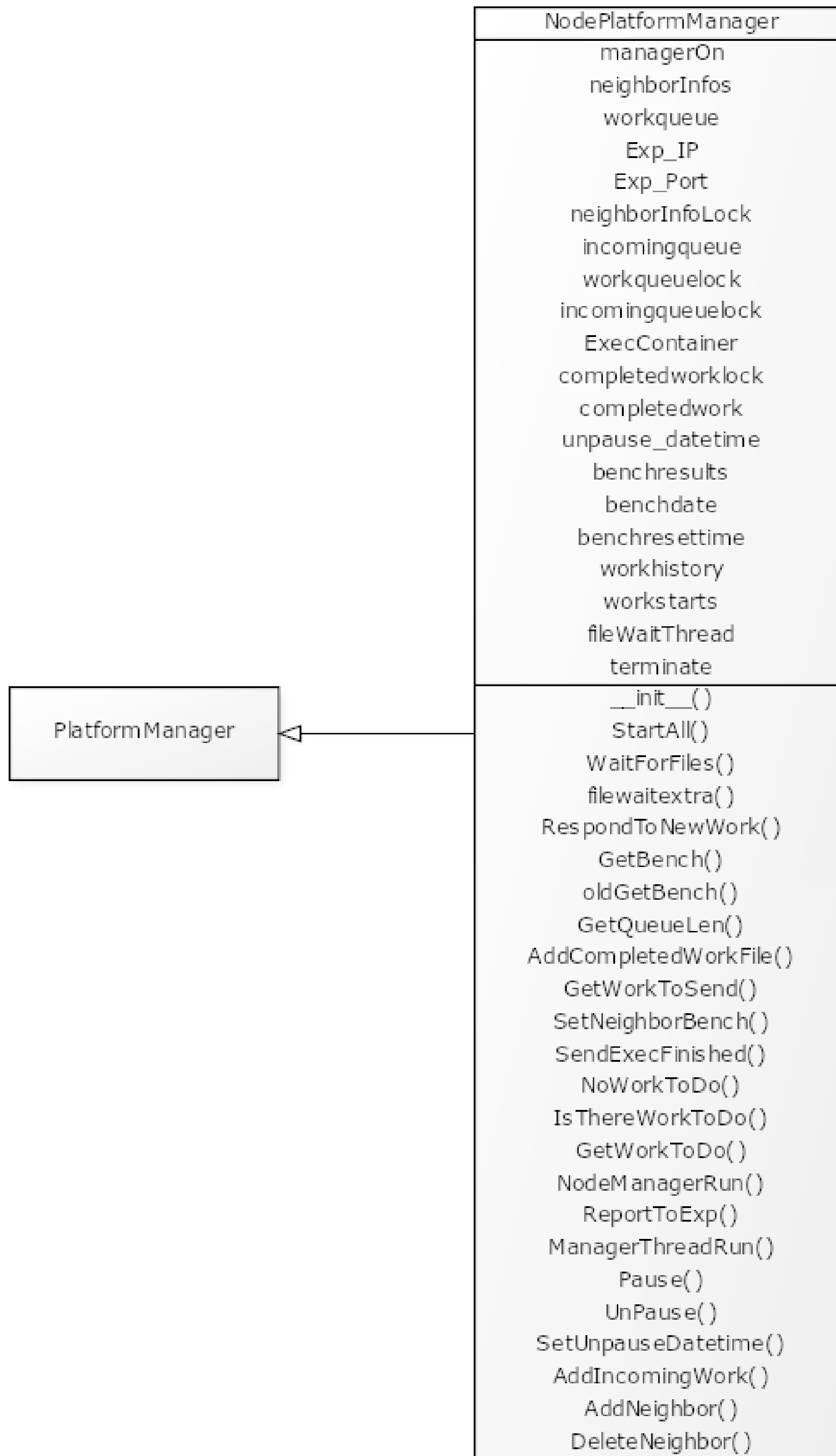

FIGURE 4.3. UML Diagram of Node Platform Manager

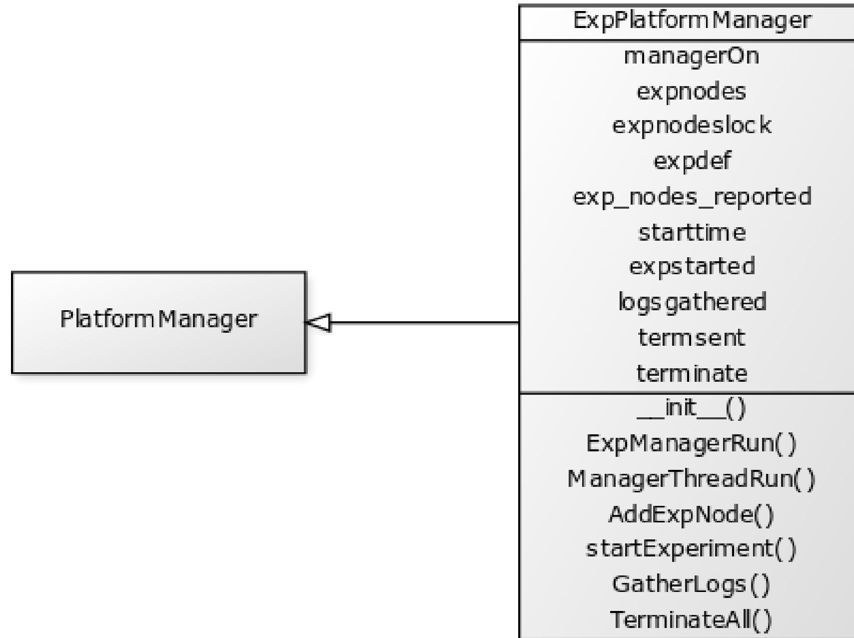

FIGURE 4.4. UML Diagram of Experiment Platform Manager

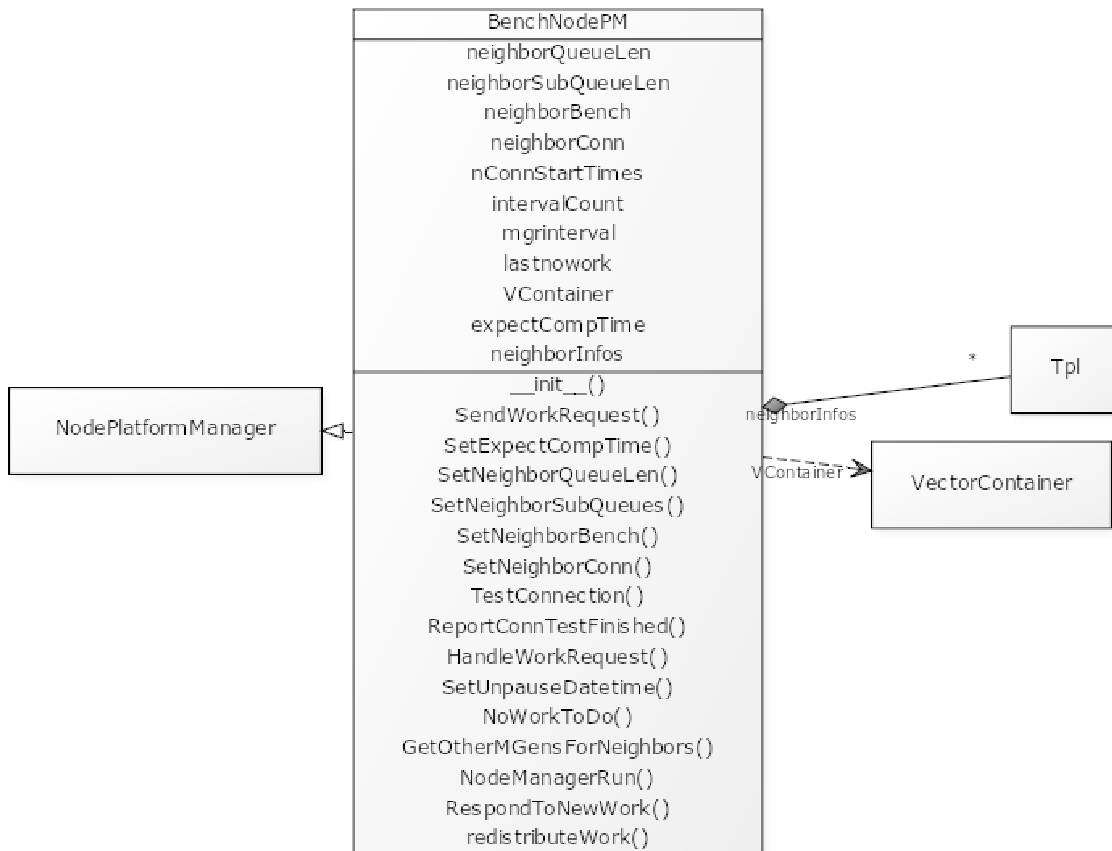

FIGURE 4.5. UML Diagram of Benchmark Based Node Platform Manager

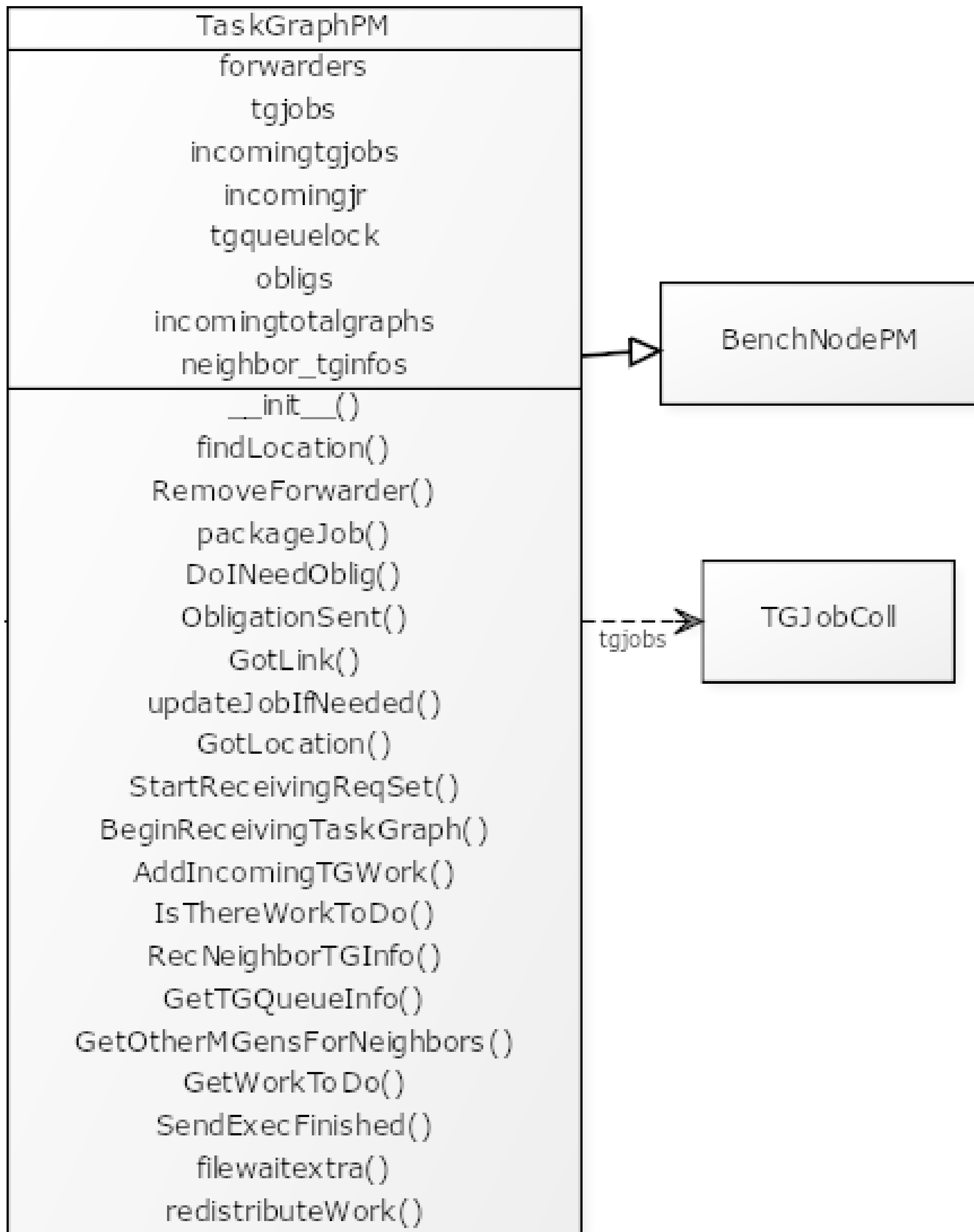

FIGURE 4.6. UML Diagram of Task Graph Platform Manager

`NodePlatformManager`'s primary functions can be seen as pseudocode in Algorithm 1. These functions control how the platform is started and stopped, and how jobs are actually processed in the node. Once a job is started it is started as a separate process and monitored, not run within the same thread as the platform. The platform therefore consists of

at least three threads of execution. The monitor thread in Algorithm 1, the thread(s) of execution of any work that is being performed, and the thread that maintains ongoing TCP communications with other platforms.

Algorithm 1 NodePlatformManager's primary monitor thread functions. These functions control the lifetime of a platform and the execution and reporting of any jobs the platform is responsible for.

```
1: function NodeManagerRun(self)
2:     if self.ExecContainer = None then
3:         self.ExecContainer ← self.GetWorkToDo()
4:         self.ExecContainer.Start()
5:     else if self.ExecContainer.IsFinished() then
6:         self.SendExecFinished()
7:         self.ExecContainer ← None
1: function ManagerThreadRun(self)
2:     self.ReportToExp()
3:     while True do
4:         time.sleep(MANAGERCHECKTIME)
5:         if self.terminate then
6:             self.SafeStopServer()
7:         if self.managerOn then
8:             self.NodeManagerRun()
9:         else
10:            if time.now() > self.unpausetime then
11:                self.managerOn ← True
```

`BenchNodePM`, which inherits from `NodePlatformManager`, contains additional functionality that controls how information is shared between neighbors, and how work is moved from platform to platform. In Algorithm 2, we can see the additional functionality that surrounds work redistribution. A platform can only redistribute the work that is contained in its own queue, it relies on information from its neighbors to understand their queue lengths, and works under the knowledge that they will also be making independent decisions about where to send the work in their own queues. The vector method allows us to choose nodes with less burden and send them a proportional amount of work. The direction that work should flow is described by a vector, and the dot product of that vector and the statistics of any given node let us build a ratio that will determine the amount of work to send any node. In Algorithm 3 we can see the functionality that performs this task. `FlowVector` is one of several classes that were designed to test different ways to handle different experimental methods as this project evolved.

4.2 Intra-Platform Communication

The classes that define information to be sent are called message generators and inherit from the `MesssageGenerator` class, shown in Figure 4.7. Message generators are created,

Algorithm 2 BenchNodePM. Much of this class's functionality is inherited from NodePlatformManager. It adds the ability to collect statistics about neighbor nodes, and to use those statistics along with the experimentally parameterized vector to redistribute the work in its queue.

```
1:  function NodeManagerRun(self)
2:      NodePlatformManager.NodeManagerRun(self)
3:      if self.sendInfoInterval() then
4:          outmgens ← self.GetOtherMGensForNeighbors()
5:          outmgens.send()
6:          self.redistributeWork()

1:  function redistributeWork(self)
2:      nbench ← []
3:      nqls ← []
4:      nbench[0] ← self.GetBench()
5:      nqls[0] ← self.GetQueueLen()
6:      for nid in self.neighborQueueLen do
7:          nbench.append(self.neighborBench[nid])
8:          nqls.append(self.neighborQueueLen[nid])
9:      dictRedist ← FlowVector.getRedistDict(nbench, nqls)
10:     for nid in dictRedist do
11:         for i in range(dictRedist[nid]) do
12:             msg ← self.dequeueWorkForNeighbor()
13:             msg.send()
```

Algorithm 3 FlowVector. The functionality here is used only by BenchNodePM, but is not stored in that class because it is much more developmentally volatile as we test different vector functionality. The purpose here is to use the statistics about each node, including the local node at index zero, to determine how many jobs from the local nodes queue each should have. Nodes should have more jobs if their statistics match with the vector that defines the ideal parameters. This is accomplished via the dot product operation.

```
1:  function GETREDISTDICT(nbenches, nqls)
2:      flowvector ← getFlowVectorParameter()
3:      nbenches ← normalize(nbenches)
4:      nqls ← normalize(nqls)
5:      dptotal ← 0
6:      dpnodes ← new dictionary()
7:      for nid in nbenches do
8:          dpval ← dotproduct([nqls[nid], nbenches[nid]], flowvector)
9:          dpnodes[nid] ← dpval
10:         dptotal ← dptotal + dpval
11:     jobsToDist ← nqls[0]
12:     extra ← 0
13:     jobdict ← new dictionary()
14:     for nid in dp nodes do
15:         dpratio ← dpnodes[nid] / dptotal
16:         jval ← jobsToDist * dpratio
17:         jobdict[nid] ← floor(jval)
18:         extra ← extra + (jval - floor(jval))
19:     nids sort ← keys of jobdict sorted descending by values
20:     while extra > 1 do
21:         for nid in nids sort do
22:             jobdict[nid] ← jobdict[nid]+1
23:             extra ← extra - 1
24:             if extra < 1 then
25:                 break
26:     return jobdict
```

then passed to a platform's `msgmon` variable, an instance of the class `MsgMonitor`, shown in Figure 4.8. This class is provided with message generator objects, and is charged with ensuring that messages are completely received at their destinations. `MsgMonitor` calls the twisted library's `connectTCP` method, which actually opens a connection to a specific IP address and port, when it is provided with a `MessageGenerator` to send to a location. Once the connection is opened, the `MessageGenerator` is read, as a stream might be, until its message is sent. After the message is sent, a message generator might close the connection, or it might expect a message back from the destination. This is the value of the `OneShot()` function, it defaults to `True`, which closes the connection when a message is sent, but if overridden to return `False`, the connection will remain open expecting a response back.

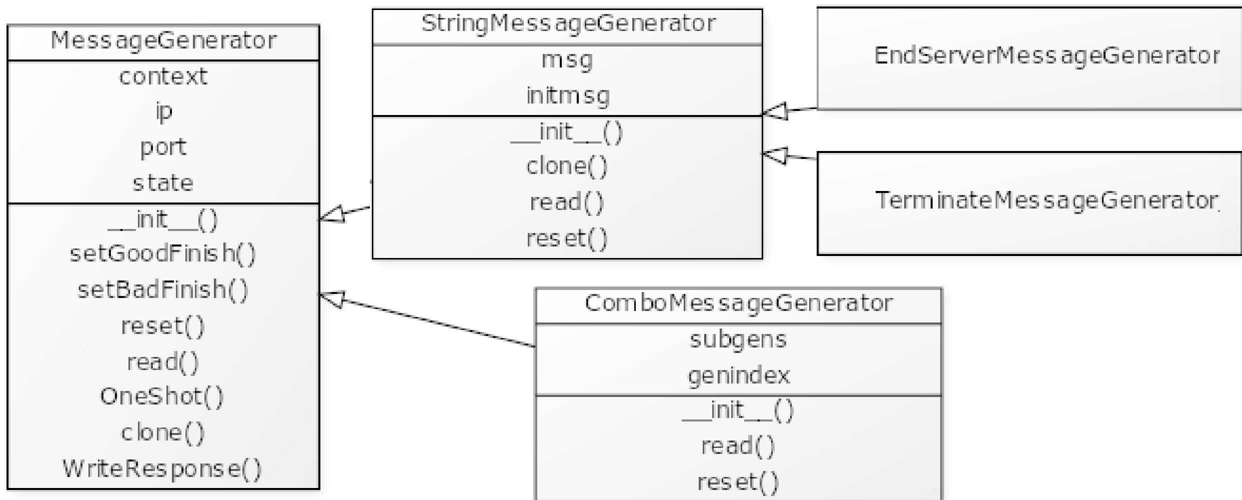

FIGURE 4.7. UML Diagram of Message Generators

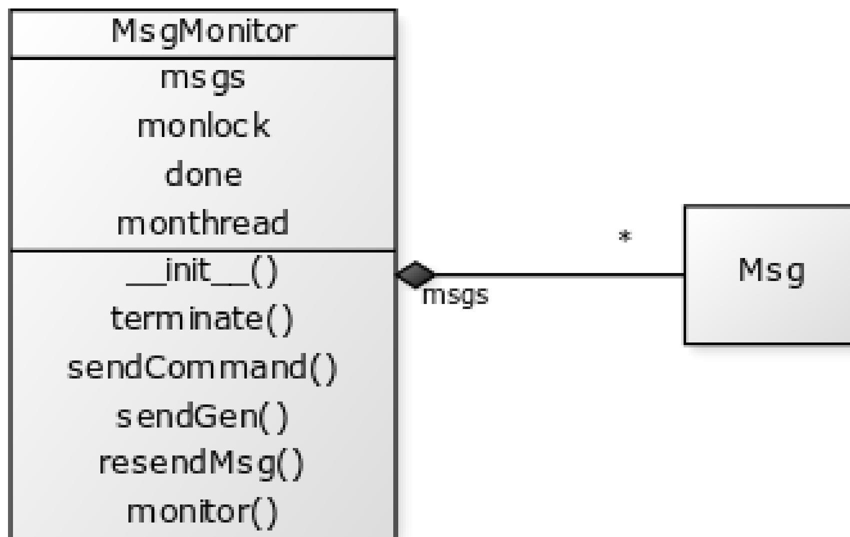

FIGURE 4.8. UML Diagram of Message Monitor

In Figure 4.9, we can see some of the most basic inheritors of the `MessageGenerator` class. Most message generators inherit from `StringMessageGenerator`. Not all message generators we use are visible in Figure 4.9, as there are too many to easily show. Classes that inherit from `StringMessageGenerator` send simple, small amounts of information, such that the entire message can be defined by a single string variable. More interesting message generators inherit from `ContainerMessageGenerator`, which sends a message that essentially contains a portion of work, and `FileMessageGenerator`, which sends a file. `ContainerMessageGenerator` must send potentially multiple files, and must be managed at the destination so that once an entire job container is sent, it can be marked as available to either be worked or transfered once again. We do not implement any kind of job container pipelining, a job must be completely moved from one location to another before it is moved again.

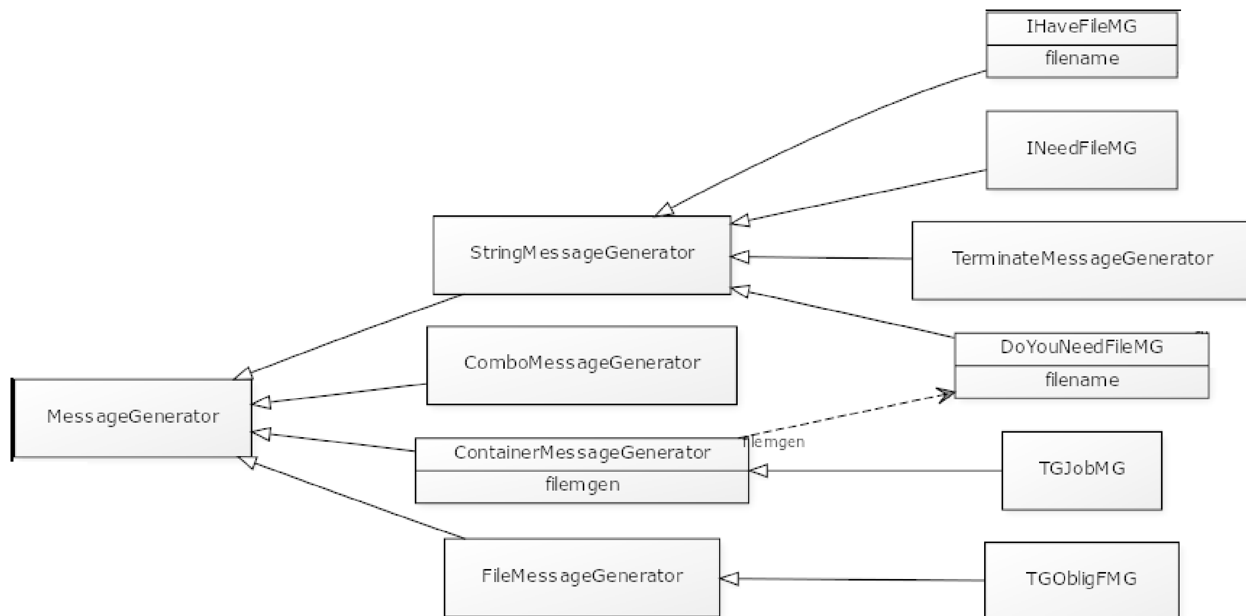

FIGURE 4.9. UML Diagram of Message Generator Hierarchy

`FileMessageGenerator`, as seen in Figure 4.10 is used to send a file, but it is typically not utilized unless a file transfer is first proven to be necessary. Most file messages are preceded by the `DoYouNeedFileMG`, in Figure 4.11, which sends the unique file identifier to a destination to see if the file is actually necessary. As many jobs we test utilize the same code, with different arguments, needlessly sending the same files around would waste time. If a destination responds that a file is needed, a `FileMessageGenerator` will at that point be created to actually send the file. `TGObligFMG` is an extension of this class with the added ability to relate a file to a specific task id, and to send extra information along with the file. `ComboMessageGenerator` is simply a combination of multiple message generators.

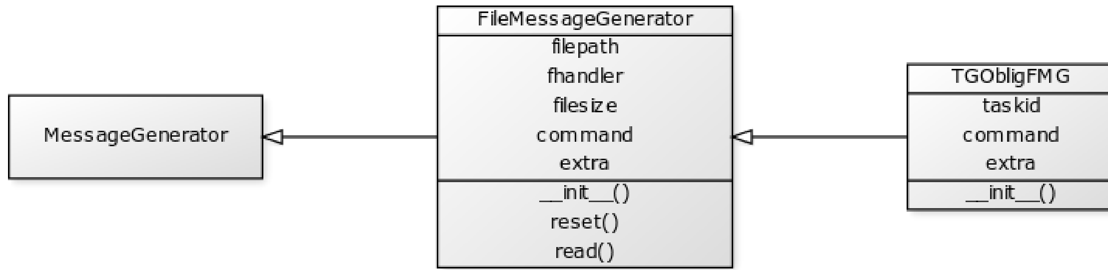

FIGURE 4.10. UML Diagram of File Message Generator

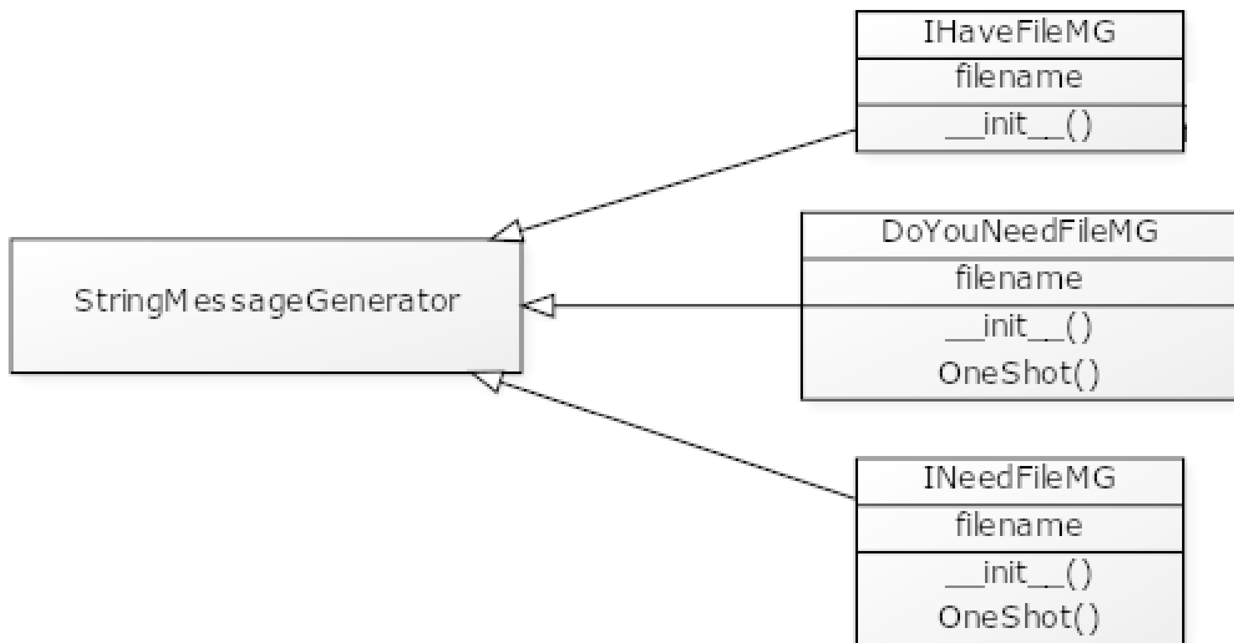

FIGURE 4.11. UML Diagram of Conditional File Transfer Classes

Each `MessageGenerator` at a source has a corresponding `MessageReader` at that message's destination. We can see the UML breakdown of some of the readers in Figure 4.12. Once a message is received by a platform, it is sent to the `ReaderFactory`, which examines its header and creates an instance of a message reader class specifically designed to handle that type of message, with the `buildMessageHandler` function. One thing to notice at this point in both Figure 4.13 and Figure 4.7 is the `context` variable that each possess. The `context` is the instance of the platform that the message is relevant to. A message is sent from one platform/context, to another platform/context. In order to read relevant information from a platform, and write relevant information to a platform, the message generators and message readers must have access to those platforms, and the `context` variable allows them to get it.

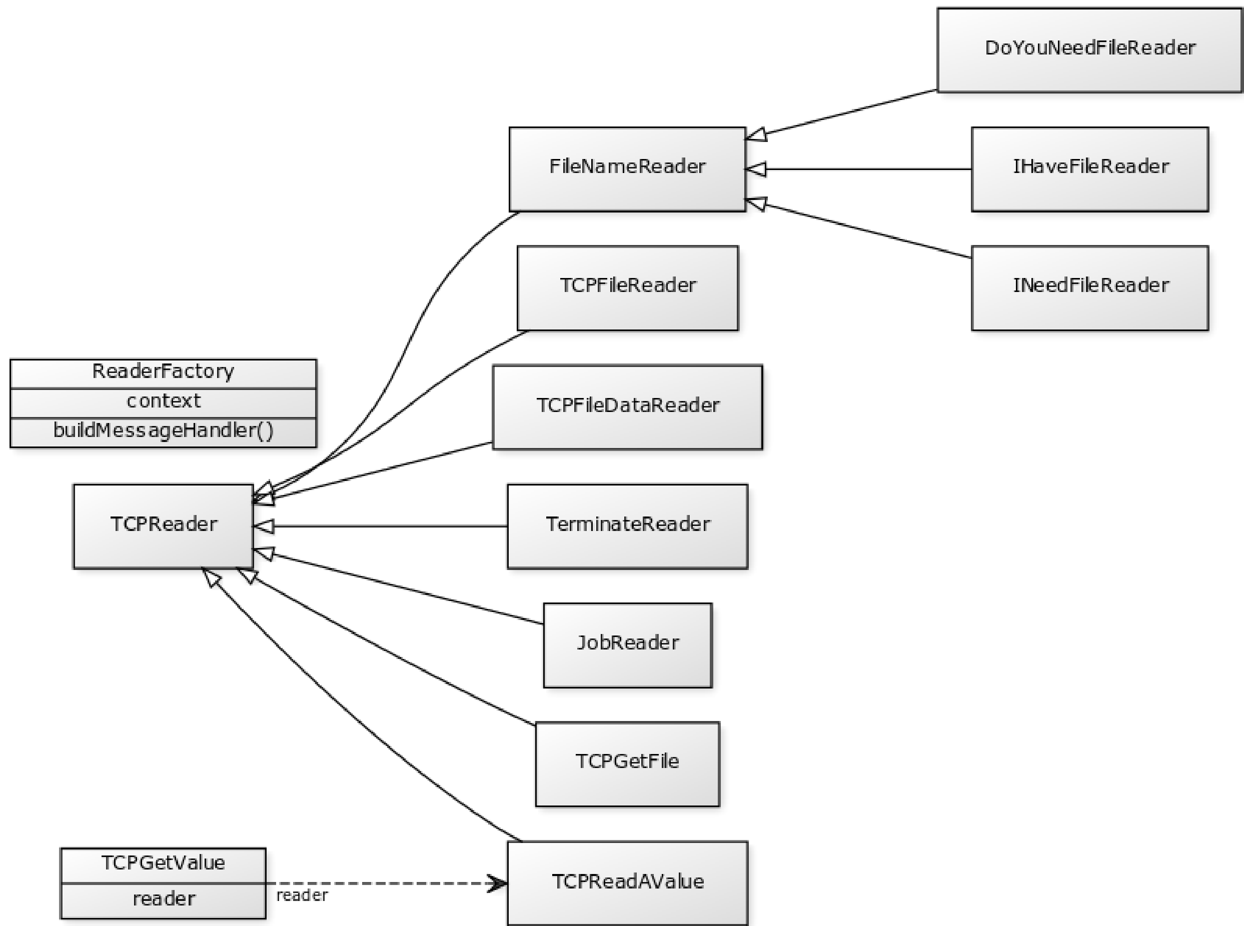

FIGURE 4.12. UML Diagram of Message Readers

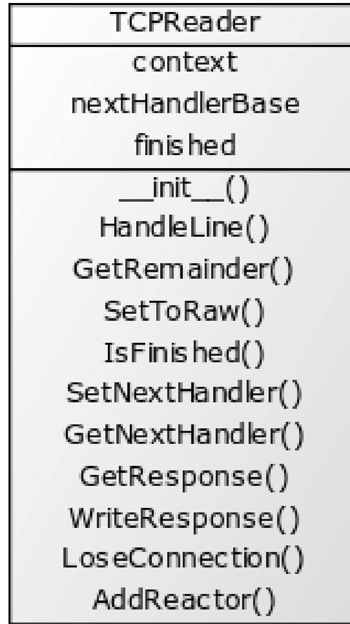

FIGURE 4.13. UML Diagram of TCPReader

Notice that readers have the ability to define a method called `GetResponse`, which essentially lets the reader create a message generator to send a response message, which will be managed by a corresponding reader at the original location of the message. So while each message is managed by a temporary class that is really only relevant to the individual message currently being delivered, a more complex exchange might involve the creation of several message generators and message readers as information is passed back and forth. Readers default to reading ASCII encoded text from the TCP connection, but with the method `SetToRaw`, they can switch to reading byte arrays of defined length. This means that any raw byte information to be sent must be prefixed by an ASCII encoded header defining the amount of byte information to be sent.

Work is managed by wrapper classes we can see in Figure 4.14. The two primary classes, `ContainerManager` and `TGManager`, wrap around specific pieces of work, allowing them to be transfered from platform to platform, and once completed, allow them to report results. `TGManager` has additional capabilities related to handling task graph dependencies. To move work from one node to another, the `Package` function is called to create the message generators necessary to transfer the work. `JobRequirement` and `LocationForwarder` are helper classes used by `TGManager` to define dependencies between jobs and the locations to send those dependencies.

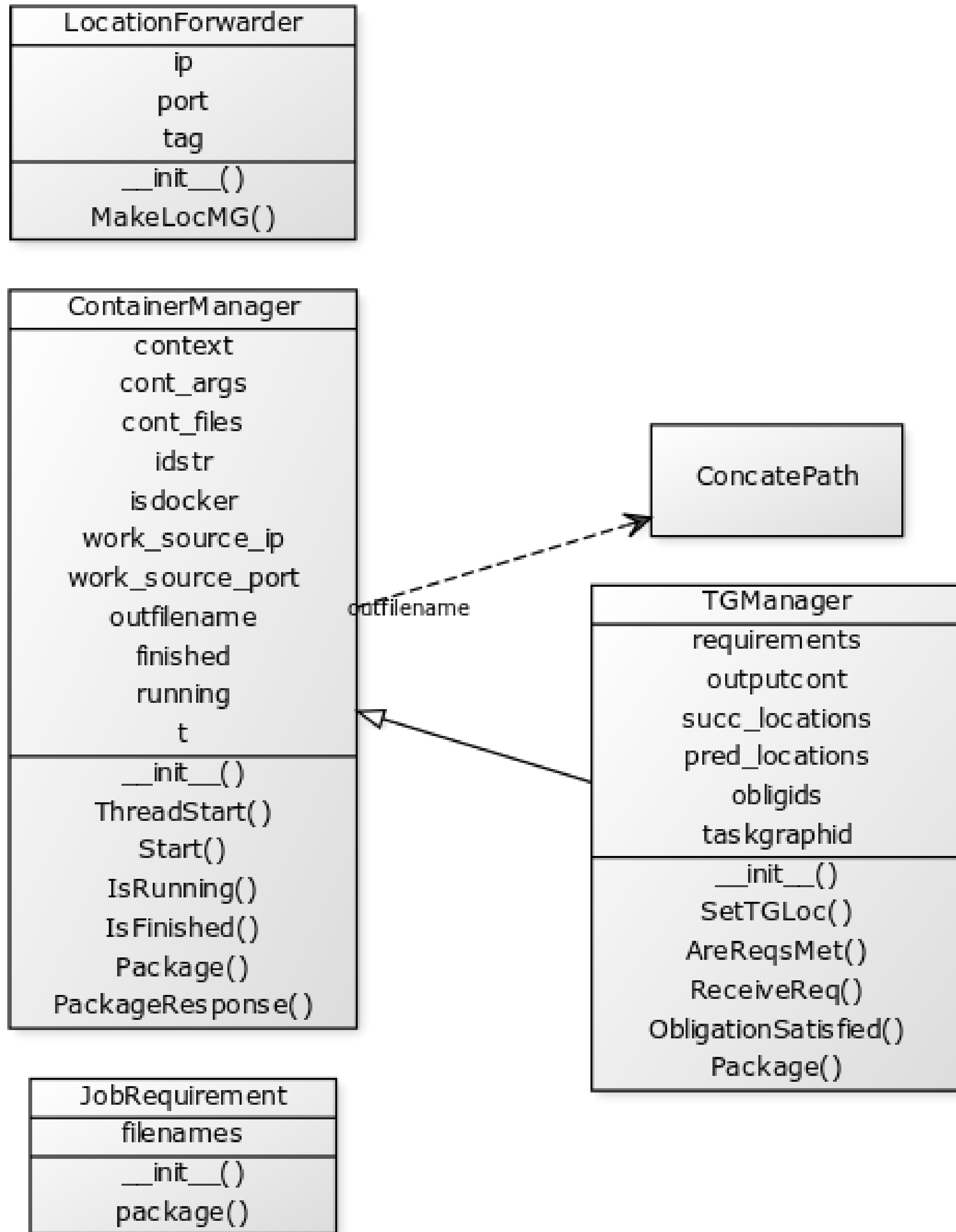

FIGURE 4.14. UML Diagram of Container Managers

With a decentralized system containing the ability to move multiple jobs, scheduling a task graph and making sure dependencies end up in the correct locations is a major challenge. The `LocationForwarder` allows interdependent jobs that move from one node to another to leave what amounts to a forwarding address, so messages for that job can be forwarded to a new location. Forwarders should be removed as jobs are updated with new locations, and maintain an index that will increment every time a job is moved, so that the most recent information about a job's location can be identified.

serverThread will create a listener based on an instance of MessageDispatcherFactory, which will actually utilize the python twisted library we are basing our TCP/IP server on. There are several layers of abstraction around which our messages are actually built designed to make the creation of new types of messages as easy as possible, but this also makes it somewhat hard to understand exactly what happens when information is sent through TCP/IP. When a platform listening for incoming messages receives an incoming connection, MessageDispatcherFactory creates an instance of MessageDispatcher, which inherits from twisted's LineReciever class. This instance will handle information on this connection until that individual connection is terminated. A connection may consist of one or more back and forth messages. Each message will start with a portion of string information separated by commas. While we can and do transfer byte data, we always first send a message that explains what byte data to expect and what to do with it, and when we can complete that transfer and return to looking for ASCII encoded data.

The first full line of new information will be provided to the ReaderFactory's buildMessageHandler method. This method is the entry point to best understanding specialized message handling. While MessageDispatcher does a lot of the work of siphoning information from the connection, the class instantiated by ReaderFactory contains the context specific handling of the information that is actually received. This abstraction also serves to keep the details of the twisted library implementation away from the code that actually handles the message. Any reader, (inheritor of the TCPReader class), is instantiated with a member variable called context. context is always the instance of the PlatformManager class that may need to be read or altered based on the content of the message. This is the link that allows an incoming message to alter the platform's neighbor information, or add a job to its queue.

While the TCPReader class defines a few methods, most are not overridden in most readers. Two which often are are HandleLine, and GetResponse. HandleLine is necessary to handle any information passed by the message other than the type of message which it is. There are some messages for which this is unnecessary, for example PauseReader pauses the platform (via context), and prepares a response, but there is no extra information in the line to handle. It is also possible for a reader to choose to lose a connection. GetResponse always creates either a null value, or an instance of a class inheriting from the MessageGenerator class.

While the subclasses of `TCPReader` handle received messages at a target platform, outgoing messages from a source platform are dealt with by subclasses of `MessageGenerator`. Many simple messages inherit from an intermediate class, `StringMessageGenerator`, for any message that can be completely represented by a single string. Therefore, message passing between a source platform and a destination platform involves:

1. Creating an instance of a subclass of `MessageGenerator` at the source platform specific to the message.
2. Enqueuing that instance into the `MessageMonitor` instance attached to that platform.
3. Using the information in the `MessageGenerator` to open a connection and send the message.
4. At the destination, once at least one complete line of the message has been read, a reader will be created by `ReaderFactory` and given the platform instance as context.
5. The reader will process the message, altering the state of the platform if necessary.
6. Potentially, a response `MessageGenerator` will be created, utilizing the same connection but otherwise returning to step 3.
7. Potentially, the connection will be closed.

Messages may therefore go back and forth almost as a tennis ball through a single connection, but all messages we use will eventually end in a completion state that terminates the connection. The complete set of messages we can generate is constrained by the classes we have implemented that inherit from `MessageGenerator`, some of which are deprecated at this point. Message generators exist for the following general purposes, each will have a corresponding reader that will process the message at its destination:

1. Ask for work
2. Sending Benchmark Information
3. Sending a neighbor introduction - (sent from experimental controller)
4. Sending queue length information
5. Pause and Unpause the platform - (sent from experimental controller)
6. Sending the result of a piece of work
7. Sending a piece of work.

The last two are some of the more complex messages, which involve multiple parts. Work is wrapped by the `ContainerManager` class in SubContManager.py. This thin wrapper lists a set of necessary files for a job, an origin for the job's completion to be reported to, a set of arguments to be used to start the job, an expected result to send once the job is completed,

and as of writing this, we are expanding this class to also contain details about requirements and obligations for the purposes of the task graph implementation. One of the member functions of `ContainerManager` is `Package`, which essentially serializes the instance into a `MessageGenerator`, allowing the piece of work to then be sent to another platform. This `ContainerMessageGenerator` sends the specific information about the job, and then for each necessary file, additionally creates and sends an instance of `DoYouNeedFileMG`. This is because, especially for most of our test work, each job utilizes the same basic compiled code, even if it runs differently, and constantly sending redundant copies of this code around would be extremely wasteful. `DoYouNeedFile` messages can either be responded to with instances of `IHaveFileMG`, or `INeedFileMG`, depending on what files are present at the destination. A response that requires the file to be sent will trigger the use of a more complicated `FileMessageGenerator`, which uses a combination of ASCII prefixes, as well as byte data transfers through the connection to send a file. A job can only be considered ready to be executed, or moved again, after all its files have been transferred or verified. Making sure this rule is held for all mid-transfer jobs is the purpose of the `filewaiter` thread.

This brings up a point that will have to be addressed when this implementation is provided as a usable library, but is not particularly interesting experimentally. What makes a file unique is not at the moment clear if the system is handling multiple pieces of work, potentially added by multiple users, ideally not requiring that those users avoid file name collisions. This is not difficult to solve but it is worth pointing out that it is not solved right now, as it seemed to be unnecessary functionality for a proof of concept.

While the message generators and readers control the transmission of specific messages, they are supported by a generalized message handling system. Each platform attaches an instance of `MessageDispatcherFactory`, as shown in Figure 4.15, to the port that it listens on. This factory is able to create a `MessageDispatcher` to manage an incoming connection. The connection will send data, which will be sent to the `ReaderFactory`, which will determine which reader will actually handle the incoming information. By cycling through a sequence of generators at a sending location, and corresponding readers at a destination, many simple class instances can be created to handle modularized pieces of a complex message. Several message generators and readers are created to send an entire piece of work from a source to a destination. When a platform wants to establish an outgoing connection, a generator is passed to a `MessageSenderFactory`, shown in Figure 4.16 that is able to establish a TCP connection and manage communication for any message that we might want to send. That `MessageSenderFactory` creates a `MessageSender`, which is a simple extension of a `MessageDispatcher` that is primed to start initial communication. After that initial unsolicited communication, `MessageDispatcher` will handle passing information to readers, and any additional information will be essentially a response. However, responses can provoke responses, and this cycle can continue until all relevant information is transfered. To clarify, the dispatch concept in `MessageDispatcher` involves dispatching incoming information to an appropriate reader.

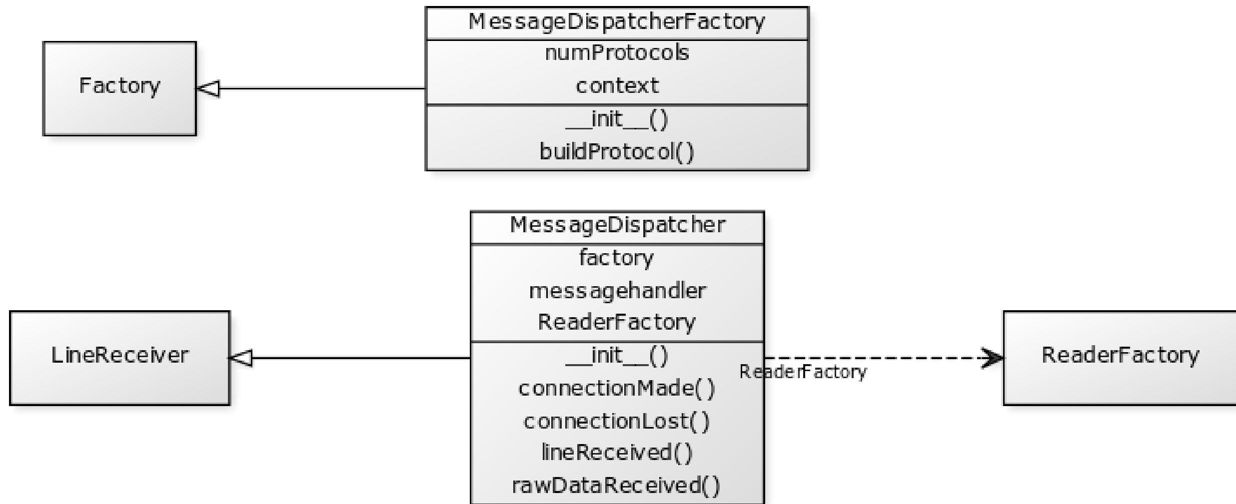

FIGURE 4.15. UML Diagram of Message Managements

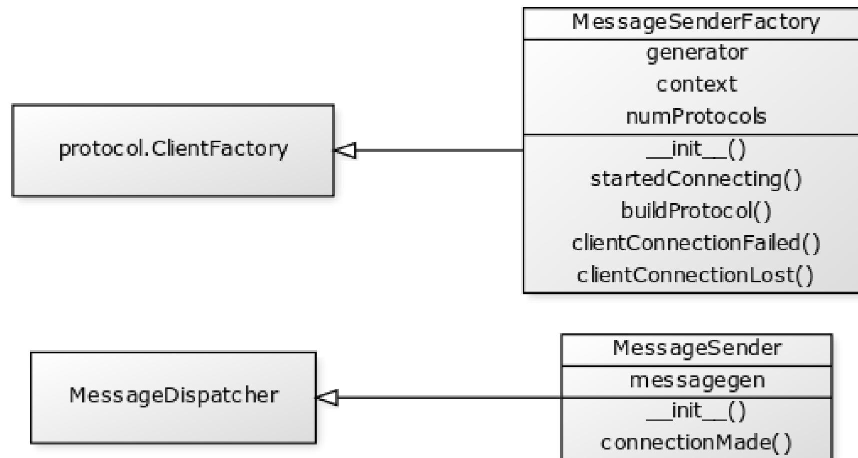

FIGURE 4.16. UML Diagram of Message Sender

While many messages can be completely transmitted in a simple fashion, some larger messages must be marked as in transit, and finalized when file information has been completely transferred. This is the purpose of the `reactorFileConfirmer` object in the `PlatformManager`, see Figure 4.2. This allows us to move a job, mark that job as in transit at a destination, and then mark a job as completely transferred when all its files are confirmed as delivered. At this point a job can be moved to an active queue for execution or further movement through the platform network.

4.3 Vector-Based Work Redistribution

`VectorContainer`, shown in Figure 4.17 stores a flow vector, to be used to direct work through the graph, and a stasis vector, which is meant to prejudice the scheduling algorithm towards keeping work where it is already located, and avoid rescheduling based on differences

too minor to be valuable. As our scheduling mechanism evolved over time with our experiments, the stasis vector remained but was unused, as we weren't able to create experimental support for its utilization. We also added the swap time ability. The primary idea behind the vector scheduler is that the direction in which work should flow in a decentralized network can be determined only by the knowledge available at that point in the network. Information, starting with node queue lengths and relative node performance, can be utilized to make intelligent scheduling decisions. Positioning a vector on each node will provide it with the ability to weigh the value of each piece of information that it possesses.

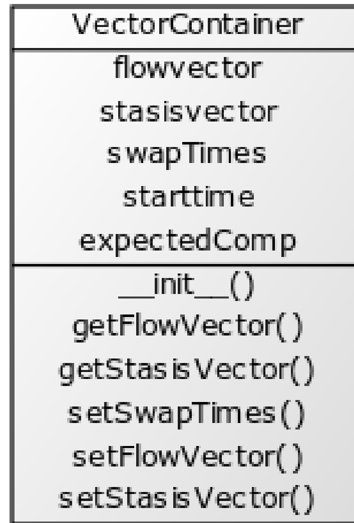

FIGURE 4.17. UML Diagram of Vector Container

We had originally suspected that the vectors would need to gradually change over the lifetime of an experiment, but our initial results with single vector experiments showed that work actually progressed in three fairly distinct stages. In stage one, the work is concentrated and most of the network is unutilized. Improving that stage involves more rescheduling actions. The second stage is when the available hardware can be fully utilized. Scheduling based on queue length is most important at this point, as every node should remain utilized. In the third phase, we will not have enough work remaining for all nodes to remain busy, so we should prioritize high performers. To enable the shift between phase two and phase three, we implemented swap times, which would shift the scheduling strategy from one flow and stasis vector combination, to another combination, after a certain amount of time had passed.

Vector containers, along with a node's information about itself and its neighbors, will be passed to a redistribution function that will gather the information sets, apply the vector coefficient to each piece of data, and determine what jobs available from a node will be distributed to its neighbors. If a node does not possess any work to distribute, this process will be skipped as it is a computational overhead. An additional benefit of vector weighing is that it allows us not just to move work in a direction, but provide a weight for each neighbor node that can be used to determine the proportion of the available work that it will receive.

4.4 Experiment Parameterization

Alongside the experimental platform, we have built a set of experiment definition classes to describe experiments parameters. These classes build the experiment, but do not play a role in the system once an experiment has actually started and the worker nodes are processing jobs. One such class, the `BenchDefinition` class, in Figure 4.18 defines a single bench type of experiment, which was the low level name for experiments utilizing the vector scheduling approach with the inclusion of benchmark measurements.

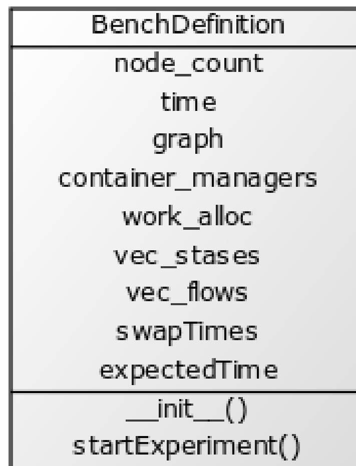

FIGURE 4.18. UML Diagram of Bench Experiment Definition

The `BenchDefinition` class contains the basic variables to define the experiment, how many nodes will participate, the graph that will be used to create the neighbor relationships between those nodes.

1. `node_count` - how many worker nodes in experiment
2. `graph` - the graph that defines node neighbor relationships
3. `container_manager` - the work that will be done
4. `work_alloc` - which nodes will start with work
5. `vec_stases` - the stasis vectors to be used
6. `vec_flows` - the flow vectors to be used
7. `swapTimes` - the time points to switch which vector is used
8. `expectedTime` - how long we expect the experiment to take

An instance of this class can be serialized, recorded to or created from a json file. This allows us to store an actual experiment that was performed to improve repeatability. Initially, we assumed each experiment would be defined by a json file, but this proved to be a tedious and unwieldy method. We instead created the `BenchDefFactory` class, which is able to

predictably and repeatably generate `BenchDefinition`s within certain parameters. By using the graphseed variable to seed a random number generator, we can utilize this class to create a set of random experiments, and by using the same seed we can generate the same set of graphs and make our experiments repeatable without saving a json file for every single experiment run. Instead, we define the parameters of the experiment in a json file, and have the `BenchDefFactory`, in Figure 4.19 use those parameters to generate as many instances of `BenchDefinition` as we want to run. Instead of a single graph, we have a defined random seed and the value of `prob_connected`, which controls how likely it will be that any given two nodes are connected by a neighbor relationship. `prob_work_alloc` controls the probability that a node will start with work. If set to zero, all the work will start on one single node. Other than those randomized variables, most of the other values, such as the vectors to be used, are passed directly to the generated `BenchDefinition`.

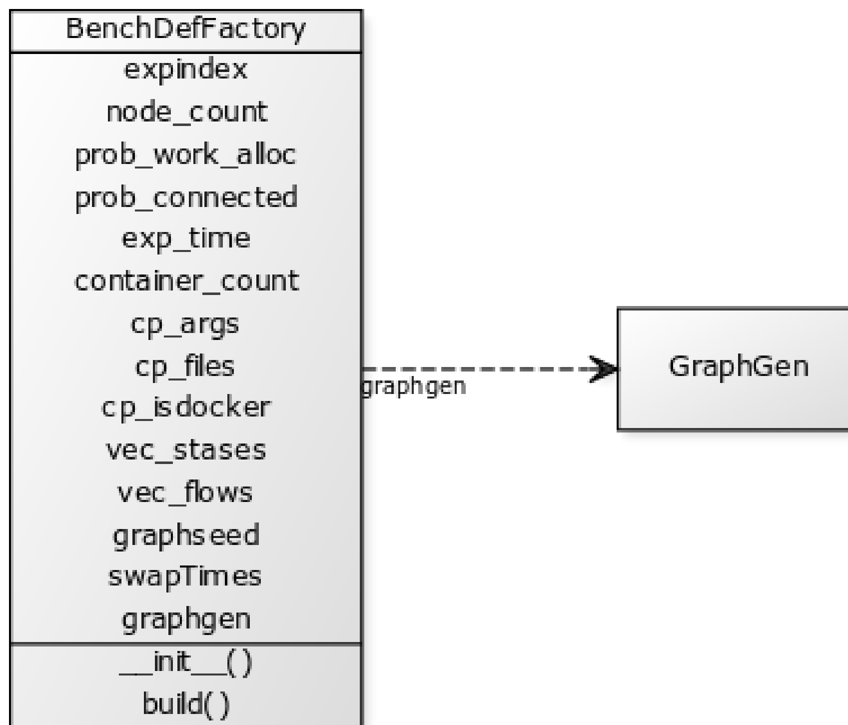

FIGURE 4.19. UML Diagram of Bench Experiment Factory

This process was expanded on in the task graph portion of our work to generate repeatable task graph experiments. This work was completed under some more stringent time constraints, so rather than build a more sophisticated inheritance mechanism, existing functionality was simply copy pasted into the `TaskGraphDefinition` (Figure 4.20) and `TaskGraphDefFactory` (Figure 4.21) classes. These are augmented with the ability to create task graph work for the nodes to perform. For a more detailed review of the reasoning behind how task graph jobs are created see the experimental section 5.3.

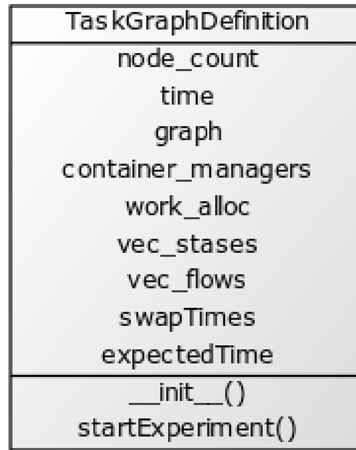

FIGURE 4.20. UML Diagram of Task Graph Experiment Definition

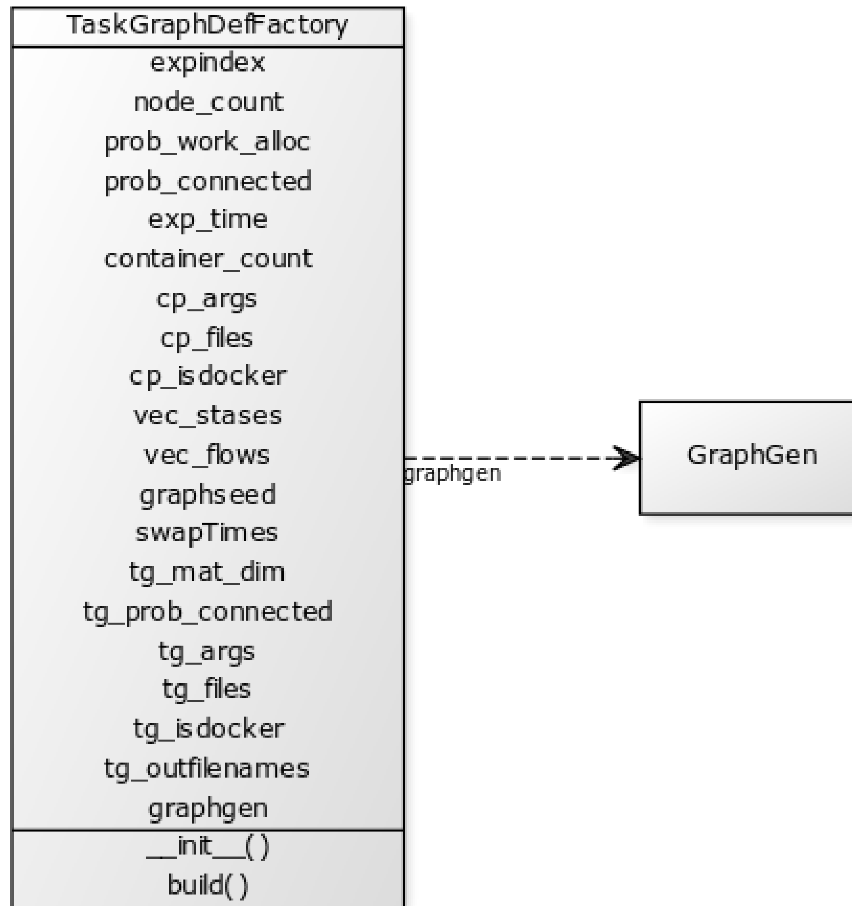

FIGURE 4.21. UML Diagram of Task Graph Experiment Factory

One last implementation detail is the distinction between worker/node platforms and the single experimental platform/manager that controls the experiment. The experimental

manager is contained in the `ExpPlatformManager` class, which also inherits from `Platform-Manager`, and therefore uses essentially the same system to communicate with the platforms that are part of the experiment. It is capable of communicating with the worker nodes, and does to set up the experiment, providing the graph of neighbors, as well as the initial work. After providing this startup information it sends out a time at which the nodes will unpause, starting the experiment. After this message, the experimental controller does not communicate with the workers during the experiment until after a certain timeout by which the experiment should have completed. At that point it sends termination messages to all participating workers, and collects their individual logs of the work done and messages sent. It is from these logs that the figures and information presented in the experimental section are created. The experimental manager also always has its own set of processor cores set aside for only it, so that it impacts the hardware of the experiment as little as possible. During the duration of the experiment, it does not perform any actions except to occasionally check for the timeout. Pseudocode of the `ExpPlatformManager` can be seen in Algorithm 4. When the `ExpPlatformManager` starts an experiment, it will refer to an experimental definition that will describe that experiment's parameters. It does this by taking the worker nodes that have reported to it and sending them to an instance of `BenchDefinition` or `TaskGraphDefinition`'s `startExperiment` method, which can be seen in pseudocode in Algorithm 5

4.5 Experiment Timeline

From this we can now construct the timeline of an entire single experiment. An experimental manager is started on a specific set of cores, and is given information about the experiment, as well as how many platforms will be participating. It starts listening on a defined port. Platforms are started on the rest of the hardware, usually with deliberately unbalanced resource allocation, and each platform is provided the IP address and port of the experimental manager. Each platform identifies itself to the manager, and pauses itself so that it will not attempt to do any work. When the manager has received as many platforms as the experiment defines, it sends information to each Platform, including which neighbor platforms it can contact, and an initial work set. After that information has been sent, the manager sends each platform a time at which it should unpause, so that the entire experiment starts nearly at the same time. At the start point, nodes measure themselves, present that information to neighbors, rebalance work using whatever approach we are experimenting on, lately the vector scheduling approach, and execute the jobs in their individual queues. After a timeout is reached on the manager, it contacts each platform, kills it, and collects its log. At this point we can start an new experiment, terminating all processes and restarting with a new manager and new platforms.

Each configuration we test ends up being run between 20 and 30 times. These run sets are defined to test some information held in common, such as the same scheduling vector, but also contain some randomness. To reduce distinctions between experiments and improve repeatability, an experimentally defined seed is used so that things such as the graph of neighbors will be the same at the same points in different experiments. For example, experiment index 13 using a queue preferring vector should have the same neighbor graph as experiment index 13 using a performance preferring vector. Running a set of experiments is

Algorithm 4 ExpPlatformManager's primary functionality. Unlike nodes that process work, it is started with a specific experiment to run, defined by inexpdef. Worker platforms are started with the location of a ExpPlatformManager to report to, then it will communicate whatever information is necessary to those workers.

```
1: function CONSTRUCTOR(self, inexpdef)
2:     self.expdef ← inexpdef

1: function MANAGERTHREADRUN(self)
2:     self.terminate ← False
3:     while True do
4:         time.sleep(MANAGERCHECKTIME)
5:         if sent terminate then
6:             if time since term > TERM WAIT then
7:                 self.SafeStopServer()
8:                 self.dumpLogToStdOut()
9:                 break
10:        else if requested logs then
11:            if time since log request > LOG WAIT then
12:                self.compilelogs()
13:                self.SendTerminate()
14:        else if experiment started then
15:            if time since exp start > exp expected time then
16:                send gather logs request
17:        else
18:            self.ExpManagerRun()

1: function EXPMANAGERRUN(self)
2:     if exp nodes reported and not experiment started then
3:         self.expdef.startExperiment(self, reported exp node info dictionary)
4:         experiment started ← True
5:         upm ← unpause message of now + 20 seconds
6:         Send upm to all exp nodes
```

Algorithm 5 BenchDefinition's primary functionality. The BenchDefinition class must contain the information listed in the constructor to define a valid experiment. This will in reality either happen through a factory or generator method. The startExperiment method will be called from an instance of ExpPlatformManager to parameterize worker nodes and start an experiment.

```
1: function CONSTRUCTOR(self)
2:     define graph of nodes
3:     define expectedTime
4:     define work to be done
5:     define flow vectors to use
6:     define time frames for flow vectors
7:     define number of nodes
1: function STARTEXPERIMENT(self, context, expnodes)
2:     for node in expnodes do
3:         Get node's neighbor info from graph
4:         Send neighbor introduction to node
5:         Send Flow Vector information to node
6:         if node receives starting work then
7:             Send starting work to node
```

done by the script RunExpBareMetal.py, using two input files, one of which describes the cloud resources we will connect to, and one of which defines the parameters of the experiment, such as the scheduling vector to use, and the number of jobs that will be added to the system.

While we discuss the possibilities of a general purpose open source application, there are no doubt limitations to this tool due to its primary current use as a research platform. Particularly, because our test work has been fairly simplistic, we have deployed the platform and test work at one time, although this is not a restriction of the current system, and once an experiment is completed we have removed the system entirely from the hardware that it utilized. This is to allow us to use allocated cloud resources for multiple independent experiments. In a general purpose system, we would face an additional architectural problem when trying to appropriately divide the portion of code that would ideally be packaged with the hardware from that which would more correctly be packaged with the jobs.

How extensive the job wrapper will be is an open question we will face going forward. A design principle of the Organic Grid is that work itself should be the primary decision maker with regards to scheduling decisions. However, this distinction may or may not be meaningful in all systems. While work could be seen as making a request of the Platforms or hardware, it would fundamentally have to tolerate that request being unsatisfiable. Alternatively, we can see the work as providing information to the system that the system will need to utilize in order to appropriately allocate resources for that work. The question will then become what meta-information, (information about the job itself), if any, will need to be the result of active monitoring of the job as it executes, and what meta-information can simply be provided in some static form prior to execution, as is done with systems such as ProActive[5].

We can postulate a many-task job that contains some task A. When A completes, it will either add 10 or 100 new tasks to the system, with a graph dependency configuration that cannot be known accurately at the time before the job is started. Such a situation may not be common, but it serves to help us think about what type of information might be produced by a job at run time that would be unavailable before job completion and necessary to guide scheduling decisions. While understanding the range of possible outcomes would be a desirable precondition, it is our current viewpoint that the task itself should report its new requirements to the platform on which it resides at the earliest possible point. The platform can then respond to those needs as it is able. The location at which the scheduling decision takes place is most obviously the platform, but the information that guides the decision making process is a mixture of the task's requirements and the platform's capabilities.

A second issue with the notion of intelligent jobs is that a system loaded with many more jobs than can be executed at a single time will not find it useful to provide each of those jobs, particularly the ones that are not yet executing, or have completed execution, with any processor time to communicate or self analyze. In the task graph example this becomes most crucial to understand. Let us consider a set of interdependent tasks as a directed graph from some set of sources to some set of sinks. As work is completed, the tasks currently being executed can be visualized as a wave that progresses through the graph of all possible tasks. Communication to satisfy dependencies is necessary from completed tasks to those that are about to start. However, especially in a large graph with multiple dependencies to some tasks, some of a task's requirements may need to be reported significantly before that task can begin, because all requirements must first be satisfied.

Therefore, we will be visualizing the system as a set of immobile platforms that make decisions about work, and a set of mobile jobs that, along with transmitted information and measurements, provide information to those platforms to allow them to make the best decisions possible. We want the decision making apparatus to be extensible and to provide a changeable method to weigh the different metrics available to it. While work provided by an ideal programmer may be able to provide the best possible information about how it can be run, that information will be utilized by the platform system. Only the platform itself can best understand not only the job but the hardware available. Additionally, a system that is only useful to the ideal programmer is likely to suffer an unfortunate limitation on the size of its user base. As a general hypothesis we will test, more and more accurate information ought to drive better scheduling decision.

Because of this fact, as we move into task-graph computing we may consider a system of temporary ATC style leaders that are marshalled as needed in response to a many-task interdependent job, and maintain mid-level centralized control over a set of workers as long as those workers are needed for this specific type of job. These leaders and workers will be pulled from a general decentralized pool of neighbors, and will be released back into that pool as tasks complete. Node information can be used to identify ideal group members. Such information could be simply declaring which nodes are co-located and ideal communicators, to actual measurements of connection speed and bandwidth between nodes. We do not expect such a system to be ideal for an environment in which all jobs are many-task type work, but with a mix of independent and interdependent tasks we hypothesize we can occupy all nodes

and intelligently schedule work to the correct locations to complete interdependent tasks more quickly. However this leader based approach is appropriately classified as potential future work.

The downside of the ATC leaders as seen in the simulations is that they are a messaging bottleneck, and we expect that we will see the same issues here. While fully decentralizing many-task work was discussed, maintaining the locations of all dependencies as tasks moved unpredictably through a decentralized network was too burdensome and any completely decentralized solution we have imagined would also face scalability issues. We hypothesize, but will not immediately set out to prove, that we could break some many-task jobs up into portions if they are amenable to graph-cutting algorithms, allowing each portion to fall under the purview of an independent controller, and providing each controller with knowledge about its superstructure dependencies and obligations. Such a tiered system would allow scalable centralization in which controllers face manageable levels of middleman centralization at even potentially multiple tiers.

Chapter 5
Experimental Results

5.1 Simulation Results

The most relevant results of the simulation concern the job messages in the network. Figure 5.1 shows the total number of job messages that were sent by each algorithm in each network in the eight environments simulated. This includes the messages for when a subtask is moved from one node to another. There are many fragments of work for each job added, which is why for a job count of 800 one can see as many as 14000 messages under the ATC algorithm. Figure 5.1 shows that the number of job messages sent by the ATC algorithm is in all situations higher than the number sent by the Organic Grid algorithm. This is because in the ATC algorithm all jobs are relayed through leader nodes, while in the Organic Grid each node can both compute and relay jobs.

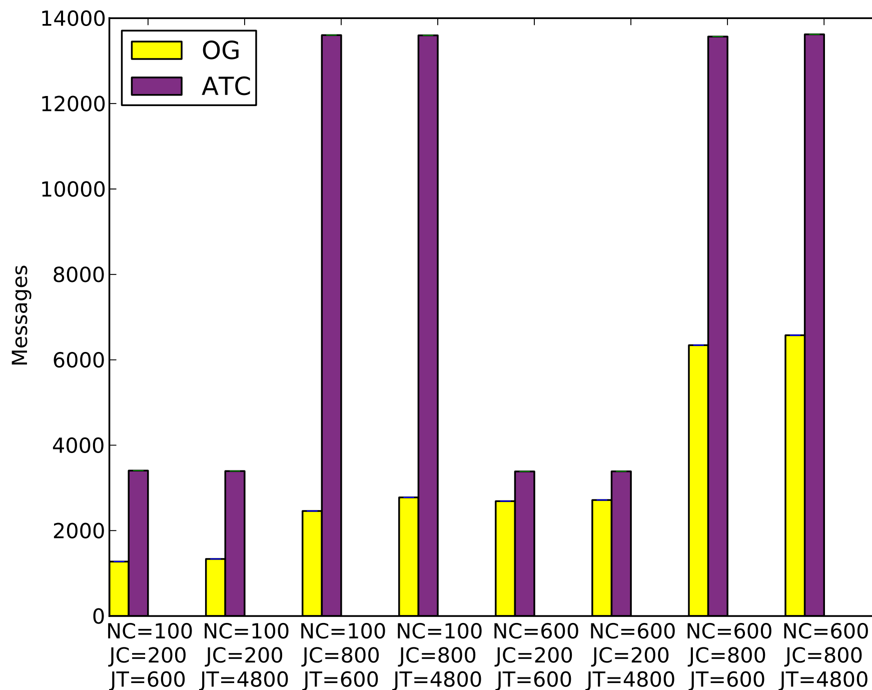

FIGURE 5.1. Sim Job Messages Total

The ATC algorithm cannot over-allocate jobs because each leader has complete knowledge of its own group. This makes the ATC algorithm better at immediately leveraging available computing nodes. We can see that, in general, the Organic Grid improves as the number of jobs and nodes in the network increases. This suggests that adopting a decentralized strategy that contained some deliberately organized grouping with defined leaders, such as the one in the ATC, would allow us to more quickly leverage large numbers of computing nodes. However, this advantage is less evident as more nodes and work are added to the network. It

is least evident in the fifth and sixth cases, which represent the simulation with the highest number of jobs and the lowest number of computational nodes, or the most-overloaded network. Still, groupings such as those defined by the ATC algorithm likely confer additional advantages when they collect nodes that work better as a group. An ideal solution should allow for nodes to be collected into groups when this produces superior results, but utilizing the grouping strategy in all situations would limit both the types of computing resources that could be managed by the network and the types of jobs that the network was capable of managing.

Figure 5.2 shows that leader nodes in the ATC algorithm can bear a disproportionally high communication load and, therefore, could become a bottleneck. Since in all cases the most-overloaded node in the ATC algorithm was the node containing the centralized job queue, we compared the number of job messages for the second-most-overloaded nodes. In six out of eight cases, the ATC approach results in a significantly larger communication burden placed on leader nodes than the Organic Grid. Only when run with the large computational pool and the smaller workload did both algorithms place similar burdens on the most used nodes. As this is the least overworked network, the problem of evenly balancing the workload becomes less difficult. Figure 5.2 also shows that in the case where the Node Count is 100 and the Job Count is 800, which results in the most-overloaded network, the ATC algorithm places the heaviest load on the leader nodes in the network and shows the worst performance relative to the Organic Grid.

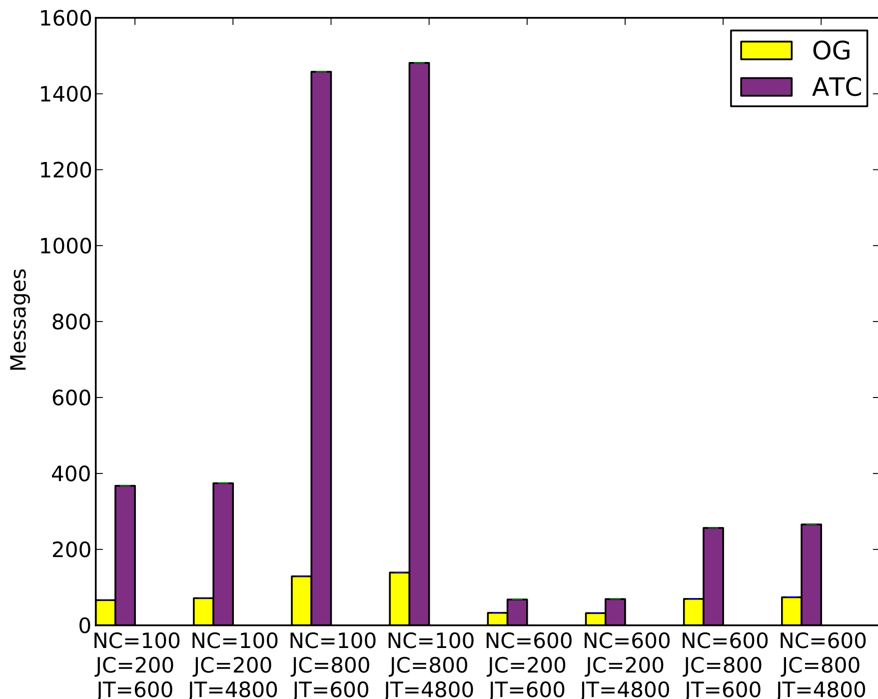

FIGURE 5.2. Sim Job Messages on 2nd Most Overloaded Nodes

This pattern is repeated in Figure 5.3, a measurement of the second-most-overloaded nodes handling messages containing job results to be collected. This suggests that the Organic Grid

approach may be superior at handling an overloaded network. If we adopt an approach that borrows from both methods, this suggests a further question. Is there a work-load point at which a leader node should fragment its own group in order to avoid becoming a bottleneck? Such a multi-layered fragmenting approach might be superior, but the Organic Grid already possesses this multi-layered aspect. On the other hand, an improved Organic Grid might vary the maximum number of children allowed to a single parent based on the load on an individual node or portion of the network, since lightly loaded nodes could afford to manage more children than heavily loaded nodes. This would result in an overlay network in a lightly loaded case that would be a flatter and shallower tree, while a more heavily loaded network might more optimally self-organize as a taller, narrower one.

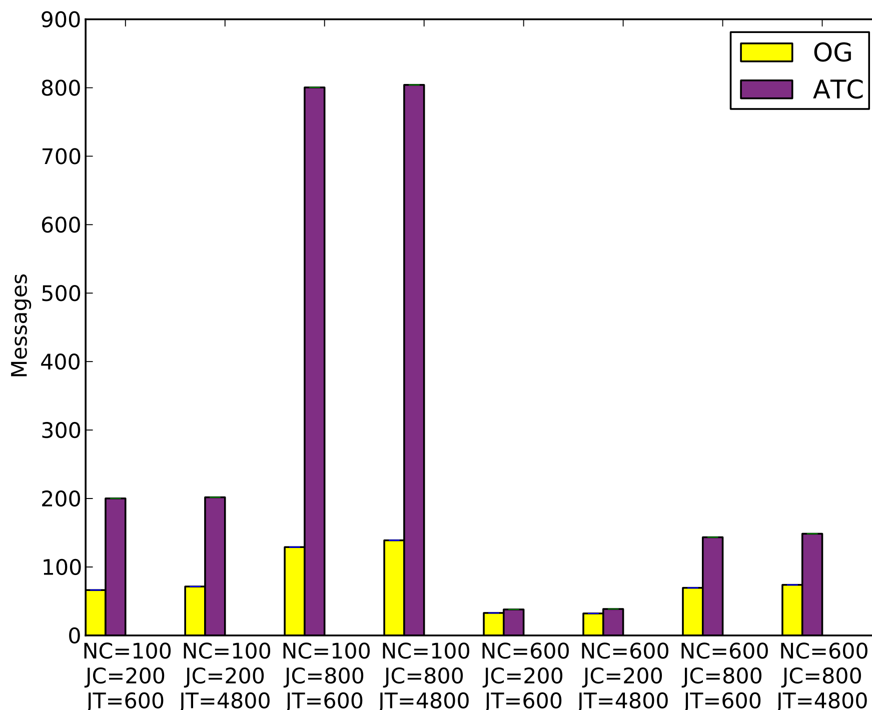

FIGURE 5.3. Sim Result Messages on 2nd-Most-Overloaded Nodes

The patterns we observed in the second-most-overloaded nodes were repeated in the most-overloaded and third-most-overloaded nodes as well. However, a different implementation of the ATC might partially decentralize the job queue, or utilize a system designed to make that bottleneck irrelevant to the work throughput. Therefore, by looking at the second-most-overloaded nodes, we can show the effects on group leader nodes, which is what we are most interested in, as they are a fundamental difference between the ATC and the OG approaches.

In IBM's patent [4], the possibility of leader bottlenecks was discussed, a potential solution that was mentioned was to simply increase the number of leader nodes in the system. This would put multiple nodes in charge of a group. However this addition might introduce further complications when making decisions for this group, or when managing resources. A multi-layered approach has the benefit of allowing one individual node to be the final manager for a group of nodes, without necessarily having knowledge of each single node that it manages.

After examining our first results we looked at what other possible variables could be changed to allow algorithms to perform more optimally. Two possible approaches to change the Organic Grid algorithm were considered, changing the initial structure of the network to be a more strongly connected graph and changing the job stealing algorithm to steal more work from parent nodes. The first approach did not result in a meaningful difference from our previous results. However, the second approach did produce a definite change in some of our metrics. Figure 5.4 shows the total jobs messages within the network with this altered approach. If the Organic Grid increases the work stolen, then it predictably moves more jobs, overtaking the ATC algorithm in the total number of job messages for the case with a Job Count of 200 and a Node Count of 600, which is the most underloaded network simulated. However, Figure 5.5 shows that while the most-overloaded nodes carry a heavier burden than in the previous scenario in Figure 5.2, the Organic Grid still does not perform worse than the ATC algorithm. Since the ATC steals work based on the size of node groups, the amount of work it steals cannot be further optimized.

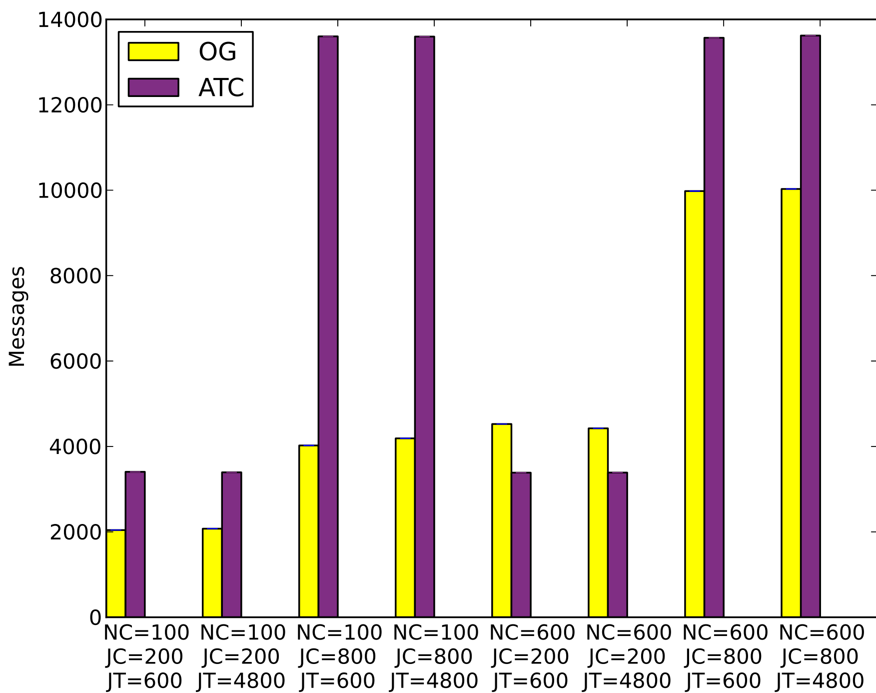

FIGURE 5.4. Sim Job Messages Total, OG Steals More Jobs

A crucial decision for a decentralized scheduler is enabling a node to determine the type and amount of work it should take from an individual work source. Since nodes in a decentralized network do not possess perfect information, and since any information we send has a cost, identifying what information is most relevant to balanced job distribution is critical. Our initial approach in the Organic Grid was to base the number of jobs pulled from a parent on the number of children possessed by a node. This would allow nodes with existing children to steal more work so that their entire subtree would have more work to pull. However, the fact that we can see an improved workload balance when the amount of work stolen is flatly increased suggests that this may not be the ideal metric to use. Since the network

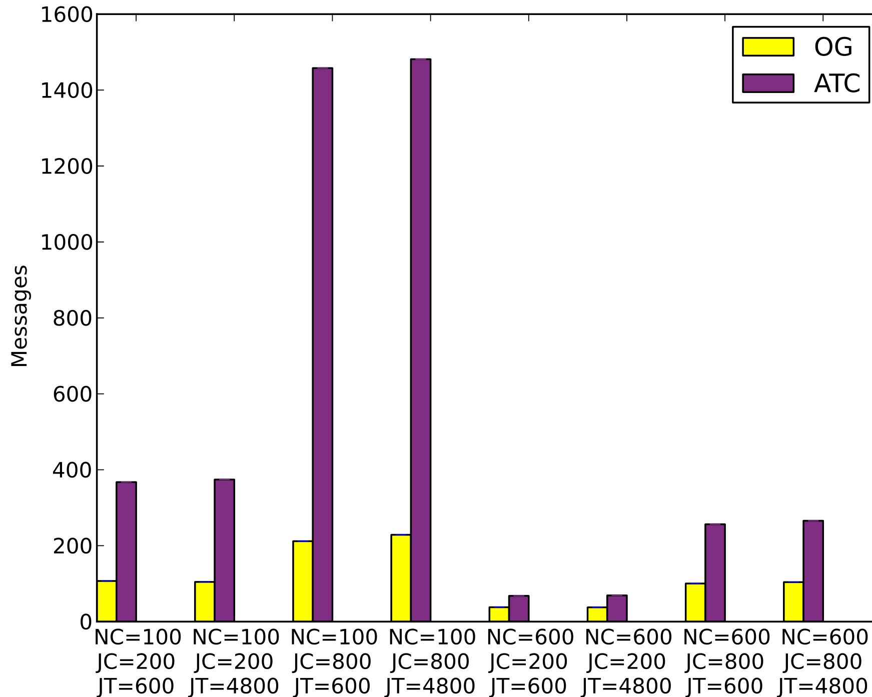

FIGURE 5.5. Sim Job Messages 2nd-Most-Overloaded Nones, OG Steals More Jobs

is dynamically altered over time, the number of child nodes possessed by any node is in flux, and pulling a larger amount simply to force the network to distribute work faster in our simulation improves workload balance. Once a node possessing work has saturated its immediate neighbors, the only way it will obtain more children is by one of its existing children being promoted to the root. Spreading work in larger pieces also more quickly builds a larger and deeper tree, as each node gives more work immediately to the first child or neighbor who requests it. This gives the Organic Grid algorithm a larger network to begin optimizing. In a combined system the approach could change as the overlay network as a whole matures, moving from a riskier strategy, stealing more work in order to generate more decentralized knowledge about node performance, to a more conservative strategy that is optimized based on the knowledge generated by a mature network.

We argue that, ideally, the best aspects of both systems should be combined. Our proposed model utilizes node grouping where appropriate, but allows a more robust system for leader nodes than the ATC, and lets work enter from any point in the network. If we extend the Organic Grid's multi-layered tree overlay network into a node leader system, we gain several advantages. One interesting difficulty with the Organic Grid is for a node to discover, using the limited knowledge available to it, how much work it should pull, keeping in mind that it will also be the path by which its children, and their children, receive work. If nodes are gathered in groups it will be possible to create a somewhat smaller overlay tree that contains more information about the total number of computational nodes available. This should combine the Air Traffic Controller's advantage in quickly disseminating work, with the Organic Grid's complete decentralization and multi-layered architecture that is more

appropriate to a variety of tasks, and is better able to handle an overloaded network in a balanced manner.

Both methods provide a mechanism by which the system can be restructured. In the ATC algorithm, leader nodes join or split their groups in order to fit the available job. In the Organic Grid, the overlay network is restructured based on measuring node performance. In a combined approach, we would maintain the Organic Grid's performance-based restructuring, but additionally allow both node groups and individual nodes to participate in the network. Dividing node groups may still be necessary, but it will no longer be required to gather nodes into artificial groups. Node gathering can be done by the overlay network structure, which can maintain smaller groups below one super leader. This would avoid a potential pitfall of the ATC approach, that over time the original groups can be dissolved not because of performance, but simply based the size of past workloads. This has the disadvantage of decoupling the overlay network from the actual hardware arrangement, which could cause performance penalties. In the proposed combined approach, we could ensure that node groups, while fragmented, can remain neighbors and will not need to be merged into other unrelated groups that might result in inferior performance over the entire network. Additionally, if the initial configuration is not optimal, a measurement-based restructuring can discard it.

5.2 Cloud ATC Results

The cloud experiment is an implementation of the ATC algorithm that was tested in the original simulation. The cloud experiment was very similar to the smaller scale simulation experiment. This experiment was run on requisitioned cloud machines and used TCP-IP for interprocess communication, while the original simulation was run on a single machine and simulated all messages. Both experiments used simulated work, with a major job consisting of a certain number of distributable sub-jobs, each of which takes a defined amount of time.

The cloud experiment measurements operate on a different implementation of the ATC algorithm than the one used in the simulation, therefore, the numbers generated are not identical. However, the same patterns are evident in the cloud experiment that were seen in the simulation. This experiment produced a similar, although not exactly the same, number of total job messages as the 100 nodes, 200 jobs and 100 nodes, 800 jobs configurations from the simulation measurements, as seen in Figure 5.6. The cloud experiment used 100 workers, five controllers, a single queue, and 200 or 800 jobs. The controllers are set by the algorithm and do not dynamically shift during the experiment. The five most overburdened nodes are all controller nodes, with the queue as the sixth. The controller nodes in the cloud experiment are more heavily burdened than those in the simulation experiment. Either experiment demonstrates the large messaging burden placed on controller nodes, as seen in Figure 5.8. Both of these experiments, therefore, point to the need for a more decentralized method for work scheduling as work scales up on Cloud systems.

In Figure 5.7 we compare the job message burdens on the controllers in the cloud and the simulation. The columns marked "1st" refer to the most-burdened controller node, those marked "2nd" refer to the second-most-burdened controller node. This comparison shows that both systems display a consistent pattern. The simulation experiments produced much smaller error bars, as the simulation is in general very consistent. In the cloud the actual performance variations on real hardware produced a much larger 95% confidence interval.

When we compare the simulation experiment and the cloud experiment, we notice some differences, but the patterns are similar. In Figure 5.6 we can see a shift in our measurements moving from simulation to cloud. However, Figure 5.7 shows that in both tests we measure a larger messaging burden on the controller nodes. We can see that the algorithm performed largely the same in the cloud environment as it did in the simulation. In Figure 5.6 the cloud experiments, as the simulation experiments, were very consistent in the compared measurement of absolute job messages, hence the essentially invisible error bars. This is largely due to the implementation of the algorithm being quite similar, and the scope of the problem being as similar as possible when migrating to a new system and implementation.

While Figure 5.6 may be used to compare the cloud approach to the simulation approach, the burdens on the individual overburdened nodes are a more interesting measurement. In the simulation, all performance distinctions were due to artificial differences in performance defined by the simulated environment. In the cloud, we are in the position to test whether we will see performance distinctions emerge from machines that should be requisitioned from one uniform system without artificial distinctions in performance. In fact, the experiment confirms the need to handle variation in performance of cloud hardware.

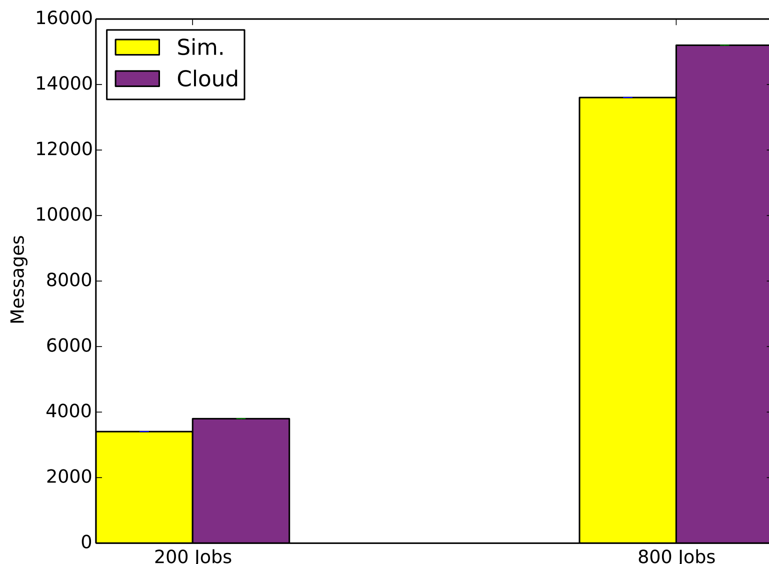

FIGURE 5.6. Sim and Cloud Experiment Job Message Total Comparison

Beyond validating the simulation, the cloud experiment is specifically useful for pointing out that even cloud hardware on a common system can show performance variation. Figure 5.8 shows that there is a clear difference between the burden placed on the different controllers. While there is some variation in the measured burdens, and the 95% confidence intervals of the most-overloaded and second-most-overloaded controllers does overlap, this is not the case for the most- and least-overloaded controller nodes. There is a definite measurable difference in burden on different controllers. This lends support to the idea that a

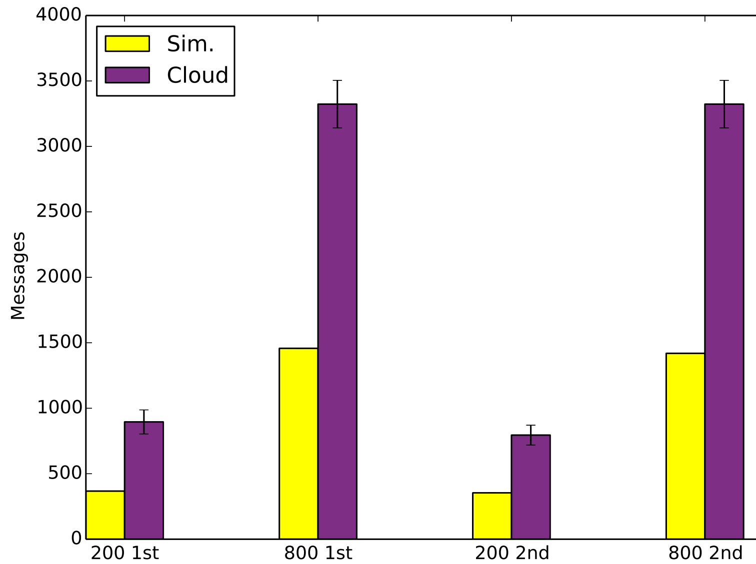

FIGURE 5.7. Sim and Cloud Job Message Burden Comparison

distributed scheduler, particularly one that measures the performance of the nodes on which work is scheduled, would be useful.

5.3 Vector Scheduling Experimental Definition

One concern with cloud scheduling is how realistic our experimental work can be. HPC problems often have communication and computation steps that are synchronized over very short time periods. Without specialized hardware with a separate scheduling mechanism, it is not possible to be competitive on that type of work. We also benefit from using work that is short enough to allow us to run many different experiments in a relatively short amount of time. However, if we make our work phases too short, then our experiment will become unrealistic because the amount of time spent on communication will dwarf computation, creating a situation in which the experiment is no longer a useful representation of any likely scenario. We have chosen to use matrix multiplication work as our fundamental computational workload. We size these workloads to take about 50 seconds of computational time on our higher performing nodes.

Organizing our experiments has allowed us to develop a cloud experiment management system. Cloud management systems, such as those used by CloudLab, have sophisticated tools in place to manage allocated resources. However, after those resources are allocated and in place, we often end up communicating with them via ssh. Even if we deploy our system in a Docker container, there must be some higher-level work done to deploy the containers, to perform the initial startup, and, crucially, to mutually identify cloud resources to one another. A general problem, of which we only attack a small part, is how to make cloud resources as automatically responsive as possible. When programming for either HPC or the cloud, there are questions of scale, of appropriate resource utilization, of ideal levels

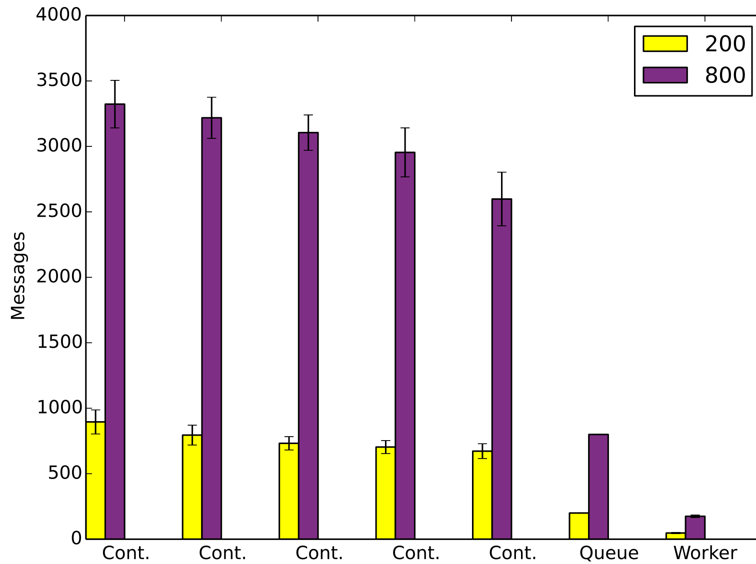

FIGURE 5.8. Cloud Messages on Most Overloaded Nodes

of parallelization, which may be solvable or at least approximately solvable automatically, but are still quite often left to be configured in a tedious and manual manner.

Building the experimental manager allowed us the opportunity to see how much of our individual cloud setup problem could be automated, how much of that automation is general purpose, and how much would be necessary to build for each new problem deployed onto the cloud. Ideally, it seems we would want cloud resources to be as transparent as possible, but this requires intelligence in place to automatically allocate and schedule as much work as possible.

Building an experiment requires not just developing the work that will be executing, but also describing the network that will execute it. Although we consider a decentralized system that will be applicable to very large scale networks, we are somewhat restricted to the actual hardware available in our own labs or on CloudLab. Therefore, most of our experiments concern how we can schedule work on heterogeneous decentralized networks, and what measurements and approaches produce measurably useful results. We can characterize a set of interconnected computing resources as a connected graph, but how do we determine what characteristics that graph should possess? We use the Erdős-Rényi algorithm for graph generation [18]. We define a probability that any two nodes in our graph of computing platforms are connected, and generate different graphs for different experiments. We examine any generated graph and ensure that it is connected by adding extra links between low ranked vertices of disconnected portions.

While this is an accepted method for graph generation, it may or may not be the best approach. It allows us to create very balanced graphs in which nodes are likely to be similar to one another. However, this may not be representative of a large scale computing network. A common real world setup might contain manager nodes with many connections, and worker

nodes that simply report via a single connection to one manager. Networks will likely be laid out along geographic terms, either along the network of a computing facility, or for a larger scale system actual geography. While graph generation might be improved, generic graphs form a useful test base. They can be strictly defined and generated in a predictable, and, as we use a defined seed for a random number generator, a repeatable fashion. We would argue that this causes our results to be more generally applicable than they would be if we utilized a set of manually defined graphs, absent a well accepted standard to apply.

Reviewing other approaches to task graph scheduling [28] [2] provides more information on the characteristics of a realistic task graph [36] than the characteristics of a realistic large scale heterogeneous cloud system. Task graph problems can be classified into static or dynamic problems. [28]. Static task graph problems are those in which the entire graph, including all dependencies, are known before scheduling. In dynamic task graph problems, dependencies and, therefore, new jobs may be generated as the result of the partial completion of the problem. We interpret this as allowing a completed task to add additional dependent jobs to the network upon completing its computation.

Based on this understanding of practical task graph problems, we organized a facsimile of a partially dependent task matrix. We generate a two dimensional matrix of tasks, with a probability, controlled by `tg_prob_connected`, that a dependency exists between directly subsequent tasks. The tasks that we view as directly subsequent are those that immediately follow in either or both of the dimensions, a total of three possible dependent tasks. If we view the matrix of tasks as shown in Figure 5.9, we see that for task A, immediate dependencies may exist only to tasks B, C, and D. In fact, in this example that tasks C and D do depend on A, but B does not. E also indirectly depends on A, but only through a direct dependency on C. This methodology allows us to create a set of tasks with a controllable level of dependency, to mix dependent and independent tasks, and to have a variety of levels of dependency. This approach was based on task graph scheduling problems used by Agrawal [2] [28], particularly the Smith-Waterman algorithm [36]. We are not trying to directly recreate this exact algorithm, we simply are mimicking the structure to provide what we consider to be a more realistic problem to optimize.

5.4 Vector Scheduling Experimental Description

When an experiment is started, we first allocate machines and then start Docker containers on each of the machines. One container functions as an experimental controller, while the rest are experimental nodes. The controller is started first and is provided with information that defines the experiment. It knows how many nodes will participate in the experiment, how long the experiment should take, and what configuration information to give to the experimental nodes. The controller builds the graph that defines node-neighbor relationships, and contains the vector(s) to use in driving vector-based scheduling. The controller starts listening for connections from nodes. When nodes start up, they are provided with the controller's location, and contact the controller to start the experiment.

Once enough nodes have contacted the controller, the controller will provide each with configuration information, potentially a set of starting jobs, and a time to start executing, which will be the start time of the experiment. At that time, the nodes will begin processing work as well as communicating with their neighbors. Communication will contain information

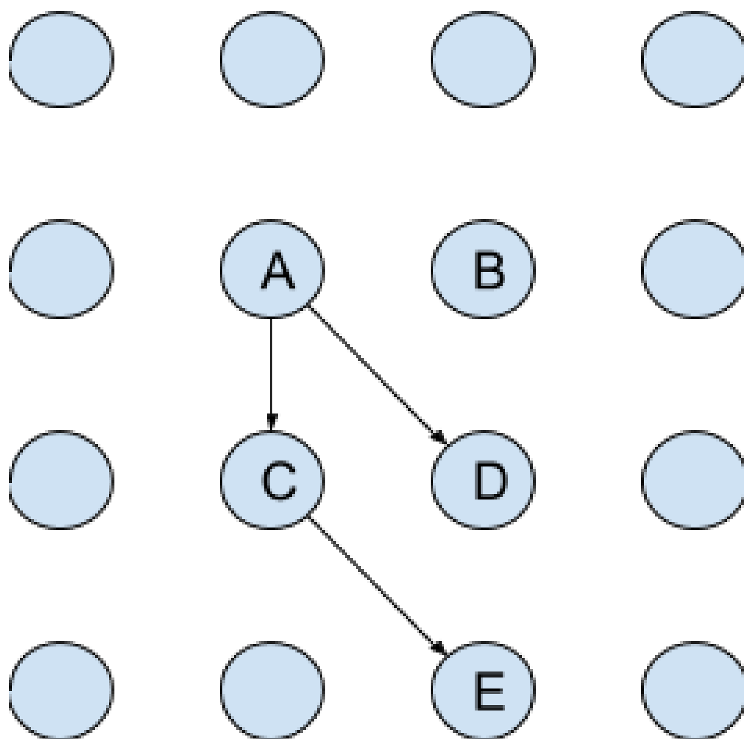

FIGURE 5.9. Partially Dependent Task Matrix

about the node's benchmark results and current work queue, as well as jobs to move from one node to another. Nodes will log all communication as well as all work they start and complete, all such logs contain time information from the perspective of the node on which the log was written. After the experiment time has elapsed, the nodes are terminated and their logs are collected. These logs form the basis of the experimental data presented in later sections.

Our first proof of concept tests concern using benchmark results to schedule independent jobs. For these tests, nodes will independently measure their own performance using a benchmark test, and then report this information to their neighbors. Additionally, we ensured that the nodes we worked with have differing performance characteristics by providing different nodes with more or fewer CPUs on which to operate. This creates what we consider to be a more realistic cloud system resource limitation than our prior simulation work that produced differing performance characteristics artificially. However, this also came with an additional effect, a suboptimal usage of those limited resources compounds a negative result across several processes that share both the same CPU and the same poor strategy. For consistent measuring and grouping, we allocated nodes such that those listening on specific ports have specific performance characteristics, as seen in Table 5.1. More detailed hardware information is available in Table 5.2.

TABLE 5.1. Node Information

Port Group	Node Count	CPU Count	Hardware Type
1100X	6	2	c8330
1200X	5	2	c8330
1300X	5	12	c8330
1400X	5	2	c6320
1500X	5	2	c6320
1600X	5	2	c6320
1700X	5	7	c6320
1800X	5	7	c6320
1900X	5	8	c6320

TABLE 5.2. CloudLab Clemson Hardware

Name	CPU	RAM
c8220	2x Intel E5-2660 v2 10 core processor	256 GB
c6320	2x Intel E5-2583 v2 14 core processors	256 GB

We measured performance to see if the hardware allocation produced the altered performance results intended, and as we can see in Figure 5.10 it did. Note that the error bars in Figure 5.10 measure one standard deviation. Notably, a decrease in a subset's performance, indicated by a higher value in Figure 5.10 is also associated with more deviation in that performance. Standard deviation is more interesting here than any kind of confidence interval, as we are observing the effects of the performance variation on the hardware that was used. We are not attempting to draw conclusions about a population from a sample. Each of these experiments was run 25 times, and within each of those individual runs multiple measurements of job completion time can be taken, depending on how many jobs an individual node completed. We also noticed that in some substandard approaches, additional burden on those low performers would exacerbate the problem and cause extremely poor performance. In Figure 5.11 we can see this issue on nodes 16000 and 16004. This figure represents a scheduling diagram of a single experiment, in which tasks' computational times are represented as bars, which allows the time consumed on each node to be easily visualized.

5.5 Initial Cloud Vector Scheduling Results

The first test we performed indicated that there were essentially three phases of the scheduling problem for a decentralized scheduler with one large application to be completed by multiple nodes. By large we mean sized to completely utilize the available computing resources for the majority of the total computation time. Phase One occurs when work moves from an extreme concentration at a single entry point to be distributed evenly throughout the network. Phase Two occurs while there is enough work to keep every computational node busy. Phase Three happens when there is still work to do, but not enough to provide work for each node. All three phases are visible in Figure 5.12. This figure, as well as several

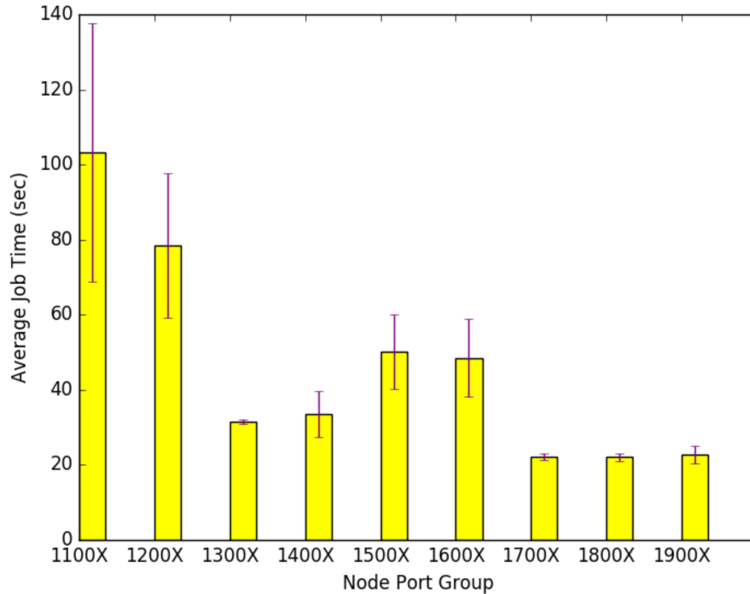

FIGURE 5.10. Node Average Performance and Standard Deviations by Port Prefix

more, show a job divided into 400 tasks that are gradually completed by a set of decentralized computing resources. Each line represents a single scheduling strategy, each of which was run through more than 20 experimental runs, allowing us to create a 95% confidence interval, indicated by a cone around the line indicating the average value. Approaches with an overall lower completion time, indicated by a lower position on the graph, are superior scheduling strategies. Note that the first and third phase each take a small portion of time, indicated by the curves at the beginning and end of each strategy.

Initial testing showed that for phases one and two, most vector-based approaches are indistinguishable in performance, except for some extremely suboptimal methods that do not strongly weigh node queue lengths, such as the black line in Figure 5.12. Phase One produced poor results originally, however this was not alleviated by using different vector scheduling strategies, but by increasing the frequency of rescheduling actions. This allowed work to spread through the network more quickly, allowing us to reach Phase Two work saturation more quickly. Phase Three proved more difficult to optimize, and the location at which the advantage of more vector-based strategies became clearer.

We also conducted experiments to determine the usefulness of our benchmark system. Initially this was a promising route, as seen in Figure 5.13. All our experiments have borne out the benchmark as a useful metric to drive scheduling at certain points, although later tests on a larger number of more diverse hardware have shown less of a relationship between a nodes average benchmark and average performance, as seen in Figure 1.1. The benchmark is a subset of the NAS benchmarks[31], specifically we utilize the cg benchmark from the Workstation class. This benchmark performs a Conjugate Gradient test, involving “irregular memory access and communication.” [31].

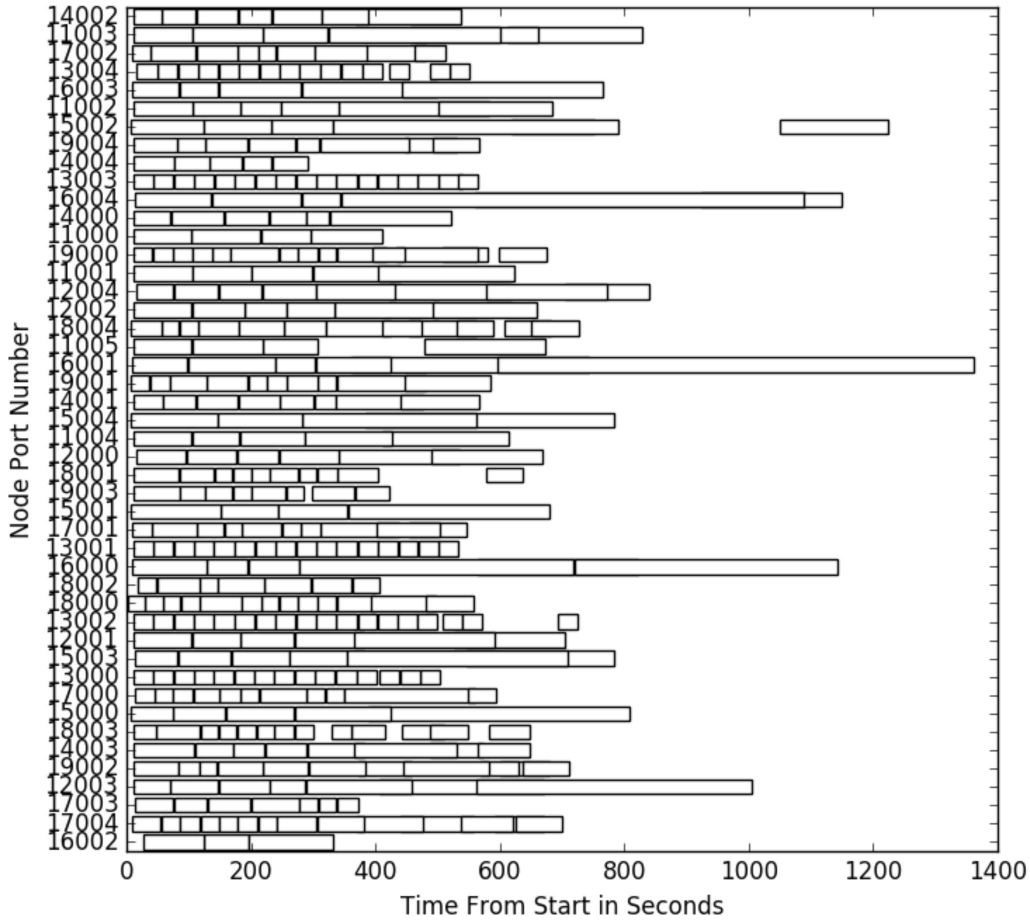

FIGURE 5.11. Schedule Diagram representing a poor approach. Extra scheduling burden is placed on stragglers, and there are some gaps where nodes become idle only to receive work at a later time.

5.6 Cloud Vector Scheduling Experimental Results

We have demonstrated the use of both benchmark values and node queue lengths as metrics to determine where work should move across computing nodes. We now examine our use of vectors to weigh the value of these metrics across several test cases on a larger platform with more performance and hardware diversity. Our first experiments are done using two different vectors, a *flow vector* and a *stasis vector*. The flow vector defines the characteristics of an ideal destination node, and all available nodes' characteristics can be compared to that vector to judge how many jobs they should be allocated. Therefore, if the flow vector defines a queue length of zero as a desirable characteristic, then the nodes with the smallest queue length will be considered the ideal targets for work to migrate towards. The stasis vector is meant to be a guard against unnecessary job movement and is used to prejudice the scheduler towards jobs remaining on the node that they already occupy. However, our experimental

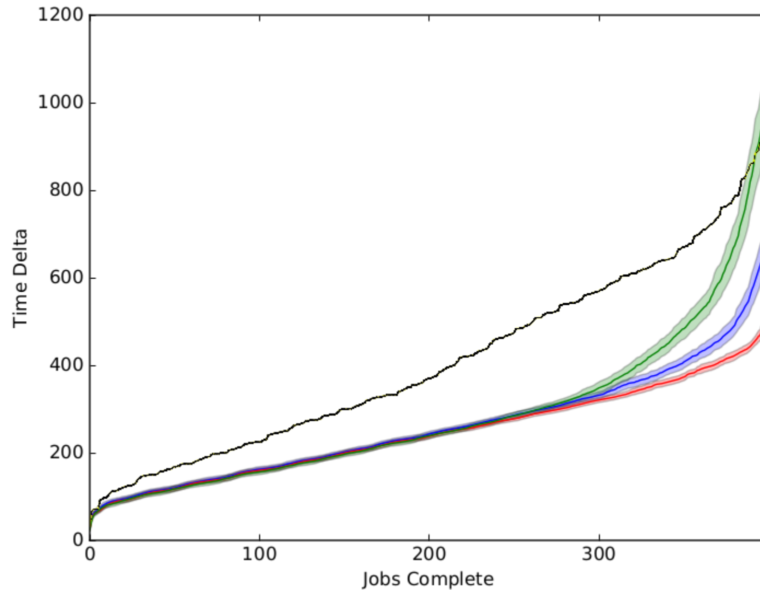

FIGURE 5.12. Initial Vector Scheduling Comparison. The red scheduling approach is superior, but of red, blue, and green none clearly diverge until after 75% or 300 tasks are completed. This early graph most clearly demonstrates three phases, but none of these approaches were used in the most up to date experiments.

results did not show that using a stasis vector was beneficial. In the most optimized scenarios it was, therefore, not used.

Our initial experiments showed that there were three primary phases and that we should utilize a different strategy to optimize each one of these phases. Speeding up the distribution of work throughout a network is absolutely crucial to improving network performance in the first phase. The adjustment we make to do this is to allow a node that exhausted its queue to request additional rescheduling tasks from its neighbors, in the hope that they might be able to provide it with more work. To avoid thrashing when the network itself ran low on work, we only allow a node to make one such request within a limited time frame of 10 seconds, and we do not allow any work requests after a certain point in the experiment.

While this achieves the desired result of improved initial performance, the overall performance of the network suffers. Even though we attempt to avoid flooding the network with reschedule requests at the end, the additional tax on the resources of the poor performers causes their last one or two jobs to take as much as three times as long as their already slower completion times. Recall that poor performance in this context is caused by sharing a CPU, therefore, a poor choice for a scheduling strategy on multiple nodes sharing the same CPU will have a strong negative influence on the performance of each individual node. We argue that this is a valuable artifact of this experimental system, as shared resources are a potential problem with low-cost cloud systems. We eliminate this problem by creating a simpler strategy that only allows rapid rescheduling operations in response to the introduction of new work into the system.

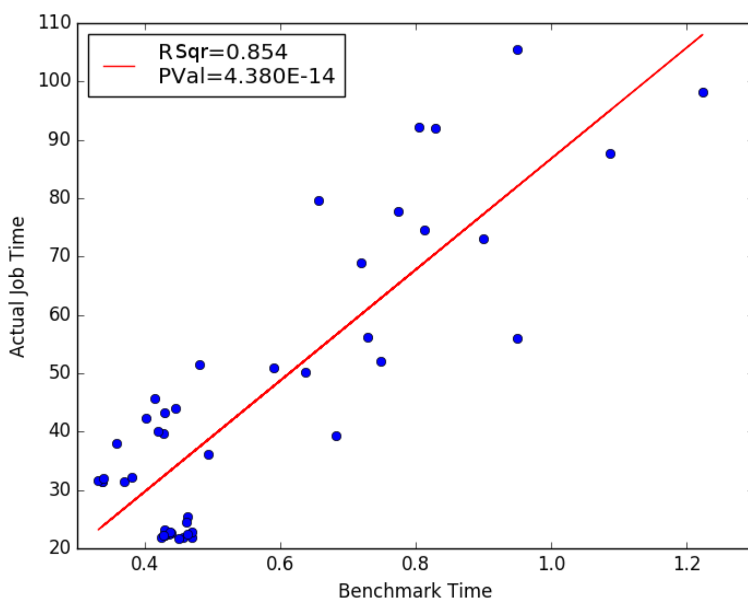

FIGURE 5.13. Initial Average Benchmark vs Average Performance. We can see that there is a fairly strong relationship between the benchmark value and the actual node's performance, producing a strong linear relationship modeled by the red line. The P value for this model is particularly low, considering the number of samples used to generate this plot. Note however that all these samples are from one site, so we do not claim that this strong relationship will necessarily hold on all hardware platforms. Nevertheless, we do use this initial test to show the benchmark was a valid approach to bring to larger scale testing on more varied hardware.

The final phase of the network, when there is no longer enough work to satisfy every node in the system, requires a different approach. From our initial experiments, we saw that we have two primary goals here. First, work must be moved away from the low performing nodes and onto the higher performing nodes. Second, any other burden on the network while the last jobs are completed should be avoided. Accomplishing the second goal in our setup simply involves turning off the functionality that allows nodes to request work once the initial job set has spread throughout the network. For future work we will look at a more sophisticated way to make this choice, for these experiments we simply turned off the functionality after a certain amount of time. After running many experiments it was fairly easy to pick a time in early Phase Two to stop allowing nodes to request more work.

Flow-vector-based scheduling was most beneficial in optimizing the final phase. For most of the duration of an experiment, any rescheduling based on relative queue length is more or less indistinguishable. As long as it does not impose a large computational burden it will spread work throughout the network, saturating the available computational resources. However, just before some nodes become idle, it is important to move work preferentially to high performing nodes, so that the last few straggling jobs are given to nodes that will complete the jobs faster. To do this we change from a flow vector of (-1.0, -0.3), which prefers

a shorter queue, and considers a better performance less relevant, to a vector of (-0.7, -0.5), which puts more emphasis on performance.

This methodology is compared to a method that simply uses the vector (-1.0, 0.0) throughout the run in Figure 5.14. Note that both of these scheduling approaches contained the optimizations to Phase One, and that neither of them produced any visible difference during the second phase. This demonstrates that the performance metric in the initial vector makes little difference until the network is no longer saturated with work. This is illustrated in the schedule graphs of two schedule sample graphs from the experimental population, Figures 5.15 and 5.16. Note that in the queue-length-only setup that there are some jobs on low performing nodes that push back the completion time of the entire system.

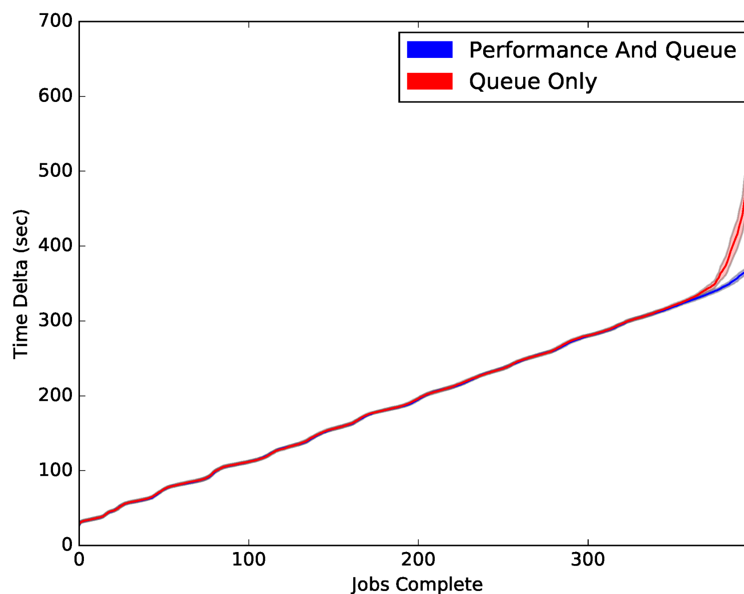

FIGURE 5.14. Job Completion Time Comparisons. Here we can see the scheduling performance of a vector that balances both queue length and performance benchmark, in blue, versus a vector that ignored the performance benchmark in red. Note the extremely poor performance of the red set near the end, exacerbated by the large performance variance of the computing nodes.

The fact that performance metrics are only useful in Phase Three, near the end of the entire application, leads to another possible improvement. Our initial data showed that the benchmark was a reasonable predictor of a node's job completion speed, see Figure 5.13. With more data as a result of completed experiments, we can better analyze the value of the CG benchmark. The data in Figure 1.1 shows a much weaker relationship between average benchmark and average performance. Regardless, experimental results have shown the most recent benchmark to be useful when driving scheduling decisions as seen in Figure 5.14. Additionally, this data shows the relationship between the average benchmark performance measurement, and the average job completion time. The relationship is not as consistent on the more diverse hardware used for the larger scale tests as it was on the original smaller scale setup. The measurement that is used for scheduling is not the overall average bench-

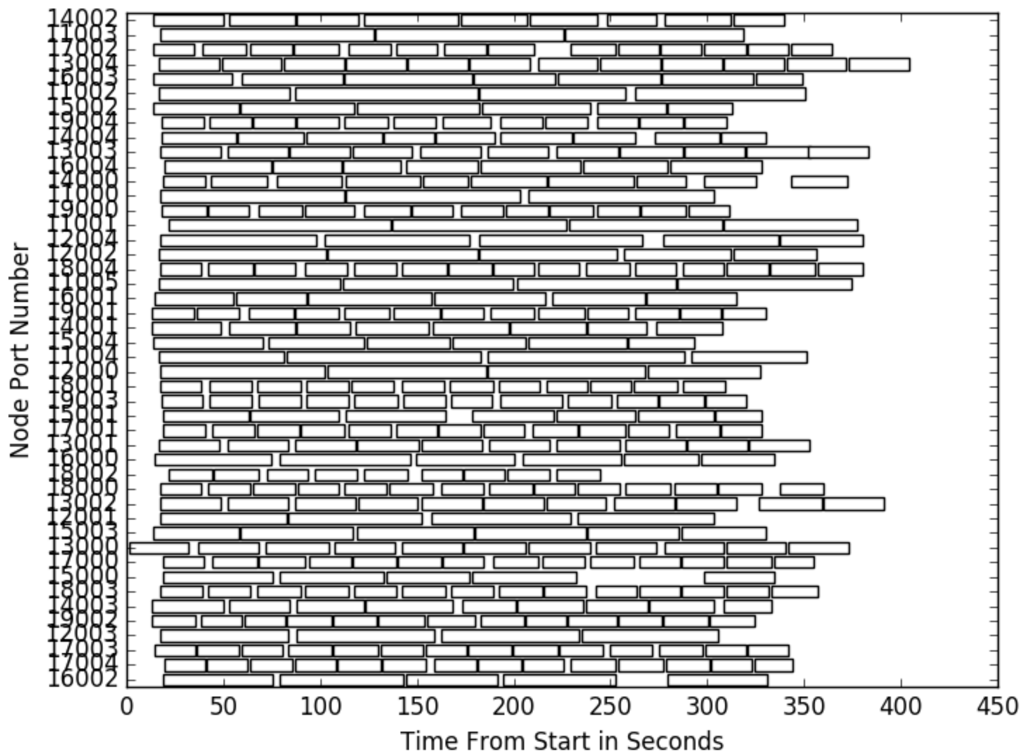

FIGURE 5.15. Performance and Queue Based Rescheduling. This is the actual schedule for each node in a single experiment using both benchmark and queue length as scheduling indicators. Notice that while it is not ideal, it is an improvement over Figure 5.16

mark measurement but the most recent benchmark measurement that a node has taken. This distinction is valuable, because we suggest the most recently taken benchmark most accurately measures the node's expected performance at that time, considering that other usage of the node's hardware is a likely cause of performance degradation.

Note that in Figure 1.1 the group that is spread along the x-axis (benchmark values), but are all consistently low on the y-axis (actual job performance). These measurements are all from nodes with a port in the 13000s. The 13000 range is one of the subsets meant to be high performers, which was born out by the actual job completion rates, but not by the average benchmark measurements. Nodes in a given port range are executed on a subset of the hardware with similar characteristics, which are meant to provide either advantages or disadvantages in performance. The compared performance of each set is shown in Figure 5.10. The error bars in this figure are simply one standard deviation, indicating that the nodes with worse performance also had more highly deviating performance. We use standard deviation here, as opposed to a 95% confidence interval, as we are more interested in seeing the result of the performance modifications on the experiments that were run, and less interested in drawing conclusions about them as samples from a larger population.

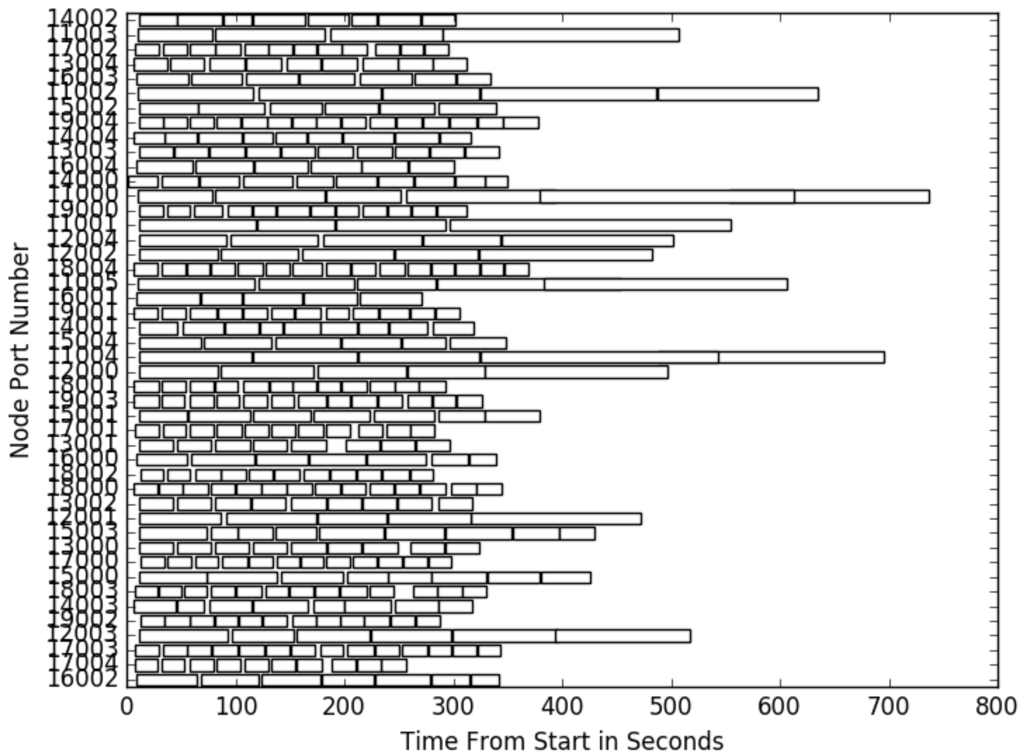

FIGURE 5.16. Queue Based Rescheduling. Here we can see a single schedule for one experiment that uses queue length only. Notice that the straggler jobs on the low performers take even longer to complete than other jobs on those already slow nodes. We suggest that this is due to these nodes bearing an additional burden of unfulfillable requests for work.

Even if the average benchmark value is not useful, the most recent performance measure may be, as it did produce the more advantageous experimental results presented originally in Figure 5.14. As a result of both the deficiencies in benchmark selection, and the result that a performance measurement is only useful near the end of work scheduling, a more simple and obvious methodology can be implemented for problems whose tasks are of consistent or predictable sizes. We considered using historical data at the beginning of the project, but initially rejected the approach because we would need some stop-gap measure to utilize before historical data was generated. A benchmark-based methodology seemed to be a better general solution, and may still be applicable given more relevant benchmarks. However, given that more than half of the work can be completed before implementing performance based scheduling, we can utilize the partial historical data available to us when some work has been completed to drive performance based scheduling, instead of using a benchmark. This result is available in Figure 5.17. The final phase curve is even less evident in these results, indicating that this strategy is superior.

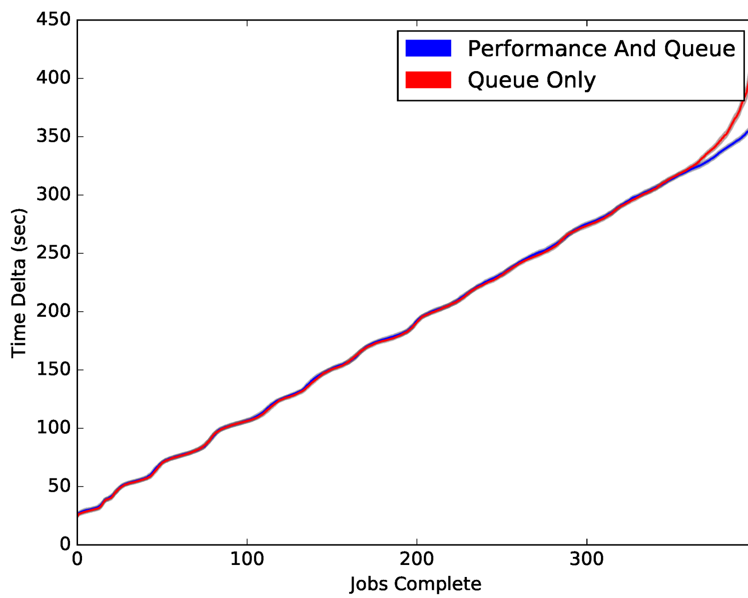

FIGURE 5.17. Historical Performance Based Scheduling. This compares a more sophisticated queue length based scheduler to one that weighs a performance metric that is constructed during the execution of the test. Note that the phase three curve on the blue line is barely evident here.

Chapter 6
Summary

Given past performance measurements [19] it is unlikely that typical cloud resources will match supercomputing resources at the type of interdependent tasks that supercomputers excel at. There is nothing to stop a cloud service provider from simply offering access to supercomputing hardware, but the optimal use of such hardware will violate the cloud computing principle of transparency. Essentially, it is our view that we are approaching a moment at which it will be necessary to expand the capabilities of the layer that exists between the programmer and the hardware on which the program is run so that cloud resources can be effectively allocated and provided work, even for sources of work that require complex interdependencies. This is arguably the goal of system such as ProActive [5], but there is an ongoing problem of a dense and arcane amount of configuration required to harness such resources.

Software written for standard multi-core processors does not often require a detailed knowledge of core hardware to harness multi-threadedness. Now we face a similar problem in the cloud, with the distinction that for some truly massive problems, fully centralized solutions may not be satisfactory. As simpler cloud solutions are driven by transparency, organizing the use of the cloud hardware is properly the role of automatic compiler or deployment system. Understanding this problem, and solving some cases of it, is the primary purpose of this work.

We have presented a framework for running high-performance and many-task applications in the cloud. Since cloud resources often have unpredictable performance characteristics, we have proposed to periodically measure the performance of cloud nodes and network connections to allow computational tasks to be placed on the most appropriate (sets of) nodes. Since centrally maintaining such detailed performance information for a large network is prohibitively expensive, we have argued that some degree of decentralization is needed. We have evaluated two candidate decentralized approaches, our Organic Grid and IBM's Air Traffic Control framework, by simulating their network traffic.

Our simulations have shown that group leaders such as those used by the ATC framework can be very beneficial in efficiently allocating work to many child nodes. However, when networks are more heavily loaded that same strategy can result in the leader nodes becoming saturated in trying to organize much more computation than can be easily handled. The Organic Grid's decentralized methods are not as effective at moving work quickly, but the burden of work distribution and management is spread more evenly through the network. Managing computation in the cloud requires utilizing a scalable approach that can dynamically adapt to variations in cloud hardware performance, and our simulation results show that our proposed cloud framework will be able to provide that scalability.

While our simulations have been run with limited numbers of nodes and jobs and have only measured communication events, they nonetheless have shown clear trends in the expected performance of the ATC and OG approaches to distributing computational tasks. They

are providing guidance in designing a better framework for deploying high-performance and many-task applications in the cloud.

We have also run further experiments in the cloud to verify both the simulation's accuracy and the existence of performance variation between cloud nodes. We did not attempt to replicate every simulated experiment in the cloud, as we want to move towards building complete functional systems as a next step, not continuing to build more and more sophisticated simulations. However, since the general patterns of the simulation have also been observed on the cloud platform, we are more confident in using the simulation to guide our future progress. Both the simulation and cloud experiments help to make the case that we need a more sophisticated, decentralized work scheduling model based on concepts from both the ATC algorithm and the Organic Grid.

The vector-based approach to scheduling has proven capable of scheduling work on a decentralized model and useful when we have to make a choice of which computing resources to use. When we are saturating all available resources, this approach reduces to a simple search for available computing power, but when we reach the end of a set of jobs, it can switch towards searching for high performers to significantly reduce the time at which all jobs are completed. In some systems this might also reduce waste.

We have observed three phases in processing of a set of jobs that fully utilizes a network of computing resources. The first phase was when work was spread throughout the network, and was optimized by performing extra scheduling tasks at that point. The second phase corresponded to the time when the entire network was occupied with computing, and behaved similarly in all of our test runs. Hypothetically, this phase could be optimized by reducing overhead traffic, but we did not test this possibility in these sets of tests. Finally, we have observed a third phase during which there was not enough work to satisfy every node in the network. We have demonstrated that our vector-based approach optimizes this phase by preferentially scheduling work to nodes with a history of high performance.

We predict that interdependent task scheduling will become more important as more sophisticated computation moves into the cloud. Additionally, making this type of computation more accessible will require cloud providers to create more intelligent and useful scheduling systems, detecting and automatically handling any exploitable parallelism the problem provides. As we improve this type of development, any automatic solutions that exploit parallelism will be beneficial in the cloud environment, even if they are approximate and not applicable in all situations. This will allow cloud systems to take on a larger set of computation than currently possible.

Scheduling work with dependencies is a necessary next step in the system. Independent work may always be best served by a MapReduce [16], however interdependent work might require a more sophisticated solution. The first step is to understand what type of work is available that fits a certain set of criteria. It must contain interdependent tasks, but the scope of the entire job must be large enough that the ideal solution will not simply be a hardware-specific supercomputer program.

Initially three approaches have been considered. The first was to schedule interdependent work within centralized subsections of the actual system. This allows us to maintain centralized information about the scheduling of interdependent tasks, and that control could be

used to ensure that links between tasks are maintained. The difficulty with a decentralized approach is that two dependent tasks may travel to a point at which a connection between them either could no longer be established or would be prohibitively expensive.

We have discussed two possible decentralized approaches. In the first, we would lock a task's immediate neighbors while that task moves, defining absolute locations for all dependent tasks before movement could occur. This was discarded as it appeared to produce a very difficult locking problem. Additionally, as one task would need to acquire many locks in order to move, the practical result for applications with large number of dependencies would be a constant tradeoff of locks while no one task was able to actually move. This starvation problem could be overcome, but the number of communications required seemed to us to make this a difficult to achieve result. Nevertheless, for certain low-dependency task groups, this may still be a viable path, as we do not have experimental results to support any conclusions on this approach.

The approach finally chosen is to use a system of forwarding addresses left behind by a task when it moves. Provided all tasks start with accurate information on the locations of all dependent tasks, and whenever those tasks move they leave behind a forwarding marker directing information to their new location, task dependencies should be able to travel freely. What we need to experimentally determine is the burden created by these forwarded messages, and how we can efficiently remove forwarding information when it is no longer necessary.

Chapter 7
Future and In Progress Work

Our goal is building a distributed system for scheduling arbitrary many-task computing problems. We are working on target problems defined by task graphs, to allow for a mixture of dependent and independent jobs, as well as jobs that may have differing hardware requirements and expected run times. We predict that some of these tasks will require tight communication between several nodes to perform optimally, while some will be more easily handled by nodes that require little to no communication. This type of problem, with a mixture of high performance requirements and tasks that can be distributed to lower performing nodes provides an ideal testing ground for a hybrid cloud platform unifying disparate resources. In the future, we plan to work with a specific application that seeks to gain an understanding of chemical properties by generating a task graph based on tensor operations.

Our experimental setup will have each task keep track of its own unique identifier, the unique identifiers of all dependencies, and an integer number that counts the number of movements that that individual task has taken. When a task moves, it leaves behind a forwarder that contains both that task's identifier, and the current movement count. The task then increments its own movement count and moves to a new location. This creates a system in which the current location of a task can always be determined by following a thread of ever more up-to-date forwarders, and provided we handle messages sent to tasks that are in-transit, should always allow a message to find its target, barring outages and network disconnections.

The next problem to face will be cleaning up forwarders, and how long they should be allowed to linger in the system. This presents an interesting problem, and one that we do not attempt to solve in our first baseline tests, as it will first be important to know what the burden of such forwarders is, and how many will build up over time. However, there are a few obvious approaches. The simplest would be to have a task remove its own forwarders once it begins processing, as at that point any messages to that task will have by necessity already arrived. Note here that we are currently viewing individual tasks as autonomous, and not requiring communication during computation, only as a prerequisite for it. This would be a safe approach, but not necessarily the most efficient. A more aggressive approach would be to remove forwarders as they become redundant. A forwarder could be enhanced with the ability to determine how many tasks should contact it and how many have contacted it so far. When it exhausts that number it could self-terminate without input from the task that left it behind. This is likely the approach that will be taken, however it might run the risk of leaving behind some forwarders in exceptional situations.

While we intend to start with similar amounts of computational work as our original experiments, we will also be on the lookout for any indications that interdependent tasks also follow the three-phase breakdown we saw in our independent task tests. If so, we will likely move towards experiments that only partially utilize a network, or that fully utilize it for shorter periods of time. It is clear that for independent tasks only the last phase is a good

target for comparative scheduling efforts as long as performance differences are not egregious. Additionally, suppose we have multiple job sets and our goal is not to reach the end of all of them, but have each complete within its own deadline. As a deadline approaches, provided the job has reached levels of completion, we should test whether a performance-based metric works for just the tasks of that particular job while other jobs are still being scheduled using a simple queue length metric.

Having group leaders can improve the efficient assignment of groups of tasks to groups of nodes, but it can also result in those same leaders becoming bottlenecks. While using an overlay network instead of a central job queue performs and scales better overall, it may be beneficial to use group leaders similar to the ones in the ATC framework near the leaves of the overlay network.[1]

There are many other characteristics of jobs and computational nodes that might produce interesting information to guide job scheduling. Developing an extensible system would allow an intelligent platform in a decentralized network to utilize many different parameters to judge the correct method to match work to processors. We need to ensure that rather than limiting our system to only the attributes that we identify as useful, our system can allow the growth, identification, and use of as-yet unknown parameters, the measurement and monitoring of which could result in better solutions to the distributed scheduling problem. While we may not be able to build a system that can adjust fully to the unknown, we intend to make it as extensible as possible. Since our approach does not rely on any specific cloud vendor or technology, it could be used to construct a hybrid cloud to coordinate a combination of multiple academic and commercial clouds as well as spare desktop machines and local supercomputers.

Additionally, it will be necessary to look at a variety of methods to move tasks through the network. A basic starting point is simply moving atomic tasks onto specific machines, but many cloud models are built on virtual machines that may migrate between hardware locations while processes are running. Work done on self-organizing overlay networks of VMs [21] has demonstrated the possibility of extending our approach to handle VM migration but also the extreme cost in communication latency between migrating virtual machines. Work has been done to optimize migration itself [23], but if our work is fine-grained enough it may be necessary to use a more fine-grained method for work migration such as using mobile agents with strong mobility [13] as containers for groups of tasks.

[1] We could think of those leaders as the approach and tower controllers who organize the flow of traffic in the vicinity of an airport, while the overlay network is used for traveling between airports, as in actual air traffic control.

Bibliography

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer Application for the Global Grid? In *Proc. Intl. Parallel and Distributed Processing Symp.*, pages 520–528, May 2000.

[2] Kunal Agrawal, Charles E Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[3] Apt Utah technical information. `http://docs.cloudlab.us/hardware.html`, 2016.

[4] E.L. Barsness, D.L. Darrington, R.L. Lucas, and J.M. Santosuosso. *Distributed job scheduling in a multi-nodal environment.* `http://www.google.com/patents/US8645745`, February 4 2014. US Patent 8,645,745.

[5] Françoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssìere. Communicating mobile active objects in Java. In Marian Bubak, Hamideh Afsarmanesh, Roy Williams, and Bob Hertzberger, editors, *Proceedings of HPCN Europe 2000*, volume 1823 of *Lecture Notes in Computer Science*, pages 633–643. Springer Verlag, May 2000.

[6] Berkeley Open Infrastructure for Network Computing (BOINC). http://boinc.berkeley.edu/.

[7] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[8] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.

[9] D. Buaklee, G. Tracy, M. K. Vernon, and S. Wright. Near-optimal adaptive control of a large Grid application. In *Proceedings of the International Conference on Supercomputing*, pages 315–326, June 2002.

[10] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[11] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. The Organic Grid: Self-organizing computation on a peer-to-peer network. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(3):373–384, May 2005.

[12] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. Self-organizing scheduling on the Organic Grid. *International Journal on High-Performance Computing Applications*, 20(1):115–130, January 2006.

[13] Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, and Gerald Baumgartner. Implementation of strong mobility for multi-threaded agents in Java. In *Proceedings of the International Conference on Parallel Processing*, pages 321–330. IEEE Computer Society, October 2003.

[14] Andrew A. Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: architecture and performance of an enterprise desktop Grid system. *J. Parallel and Distributed Computing*, 63(5):597–610, 2003.

[15] Cloudlab. `https://www.cloudlab.us/`, 2014.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[18] Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[19] Constantinos Evangelinos and Chris N. Hill. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. In *In The 1st Workshop on Cloud Computing and its Applications (CCA*, 2008.

[20] folding@home. http://folding.stanford.edu.

[21] Arijit Ganguly, Abhishek Agrawal, P Oscar Boykin, and Renato Figueiredo. WOW: Self-organizing wide area overlay networks of virtual workstations. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 30–42. IEEE, 2006.

[22] Andrew S. Grimshaw and W. A. Wulf. The Legion vision of a worldwide virtual computer. *Comm. of the ACM*, 40(1):39–45, January 1997.

[23] Stuart Hacking and Benoît Hudzia. Improving the live migration process of large enterprise applications. In *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '09, pages 51–58, New York, NY, USA, 2009. ACM.

[24] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive scheduling for master-worker applications on the computational Grid. In *Proc. of the First Intl. Workshop on Grid Computing*, pages 214–227, 2000.

[25] IBM. World Community Grid. http://www.worldcommunitygrid.org/, 2015.

[26] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[27] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive parallelism under Equus. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, pages 172–184, March 1994.

[28] Mehmet Can Kurt, Sriram Krishnamoorthy, Kunal Agrawal, and Gagan Agrawal. Fault-tolerant dynamic task graph scheduling. In *Supercomputing (SC '14)*, New Orleans, LA, November 2014.

[29] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[30] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra A. Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings of the 8th Heterogeneous Computing Workshop*, pages 30–44, April 1999.

[31] Nas benchmarks. `https://www.nas.nasa.gov/publications/npb.html`, 2016.

[32] Ioan Raicu, Ian T. Foster, and Yong Zhao. Many-task computing for Grids and supercomputers. In *Proceedings of the 2008 Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS 2008)*, pages 1–11, Austin, TX, November 2008. IEEE.

[33] Samyam Rajbhandari, Akshay Nikam, Pai-Wei Lai, Kevin Stock, Sriram Krishnamoorthy, and P. Sadayappan. A communication-optimal framework for contracting distributed tensors. In *Proceedings of SC14, The International Conference on High Performance Computing, Networking, Storage, and Analysis*, New Orleans, LA, 16–21 November 2014.

[34] Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb, Jonathon Duerig, Keith Downie, and Mike Hibler. Apt: A platform for repeatable research in computer science. *SIGOPS Oper. Syst. Rev.*, 49(1):100–107, January 2015.

[35] SETI@home. http://setiathome.ssl.berkeley.edu.

[36] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[37] Ian Taylor, Matthew Shields, and Ian Wang. *Grid Resource Management*, chapter 1 — Resource Management of Triana P2P Services. Kluwer, June 2003.

[38] Edward Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN*, 33(5):18–23, October 2008.

[39] G. Woltman. GIMPS: The great internet Mersenne prime search. http://www.mersenne.org/prime.htm.

[40] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[41] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

Chapter 8
Vita

Brian Peterson received a Bachelor's in Software Engineering from Auburn University in 2005. His interests are in cloud and scientific computing, as well as improving software testing and lifecycle management.