Making Work Queue Cluster-Friendly for Data Intensive Scientific Applications

Michael Albrecht, Dinesh Rajan, and Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame

***Abstract*—Researchers with large-scale data-intensive applications often wish to scale up applications to run on multiple clusters, employing a middleware layer for resource management across clusters. However, at the very largest scales, such middleware is often "unfriendly" to individual clusters, which are usually designed to support communication within the cluster, not outside of it. To address this problem we have modified the Work Queue master-worker application framework to support a hierarchical configuration that more closely matches the physical architecture of existing clusters. Using a synthetic application we explore the properties of the system and evaluate its performance under multiple configurations, with varying worker reliability, network capabilities, and data requirements. We show that by matching the software and hardware architectures more closely we can gain both a modest improvement in runtime and a dramatic reduction in network footprint at the master. We then run a scalable molecular dynamics application (AWE) to examine the impact of hierarchy on performance, cost and efficiency for real scientific applications and see a 96% reduction in network footprint, making it much more palatable to system operators and opening the possibility of increasing the application scale by another order of magnitude or more.**

I. Introduction

Researchers with large-scale data-intensive applications in many fields of science and engineering often wish to harness multiple computing resources simultaneously – they may own a private cluster, access a campus shared cluster, request an allocation on a national computing resource, or purchase resources from a commercial cloud. In order to construct applications of the largest scale, middleware is needed that can harness multiple resources simultaneously. Work Queue [6] is one such example that has enabled building large applications that attack problems such as genome assembly [10], protein folding [1], and other workflow applications [2]. Using multiple clusters gained from national cyberinfrastructure, these applications have successfully scaled up to thousands of nodes.

However, at the very largest scales, Work Queue is often "unfriendly" to the individual clusters providing service. Each worker manages a single core with its own private data cache and requires a direct TCP connection to the master, wherever that master is running. Oftentimes this results in redundant data transfer, as workers on the same filesystem do not share their cache, congestion for wide-area network links, and special permissions for cluster firewalls and network devices to allow inter-cluster communication. Furthermore, Work Queue's approach to overcommitment of resources, like filesystems, is to continue operating until the resource is exhausted and then clean up everything at once. This leaves overly-ambitious applications prone to impacting other users of the system, requiring intervention by local administrators.

In this paper, we describe our modifications that make Work Queue more cluster-friendly for data intensive applications This involves adding two new components to the framework: a *multi-slot worker* which can manage multiple tasks sharing a common storage space, and a *foreman* which manages local storage and coordinates the activity of workers within a cluster. With these components we eliminate redundant data transfers to tasks running on the same node, significantly reduce the number of wide area network connections, and gain more control over resource management on a cluster-by-cluster basis. Additionally, the new design gives local administrators a point of control where they can monitor and control the participation of the cluster within the global computation.

A key technique of this solution is the coordinated management of computation and data. Work Queue does not permit applications to arbitrarily access data on demand, which typically leads to overloads of either storage or network capacity as workloads scale up. Rather, the binding between tasks and the data they access is made explicit in the system, so that data movement can be scheduled along with task assignment, throttled to conform to local policies, and localized to avoid undue impact on shared resources. The improved system is highly aggressive about garbage collection, so that the system can be easily torn down on a moment's notice.

To evaluate the improved system, we first explore the behavior of a synthetic benchmark run in a variety of network conditions using both flat and hierarchical frameworks. We perform a sensitivity study of the benchmark run under varying conditions of worker volatility, network heterogeneity, and data imbalance, observing that the improved system offers a modest improvement in execution time and particularly benefits the cluster operator by dramatically reducing wide area network usage.

Finally, we demonstrate the impact of these changes on a production molecular dynamics application by running it on several thousand cores from four distinct clusters – a shared HPC cluster, a campus Condor pool, and two FutureGrid sites. We show that using the hierarchy reduces the network footprint of the application at the master by 96%, making it much more palatable to system operators and opening the possibility of increasing the application scale by another order of magnitude.

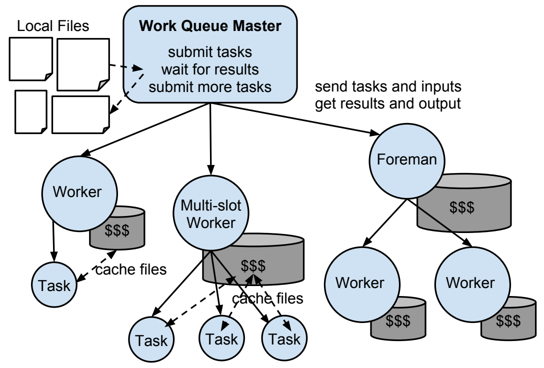


Fig. 1: Work Queue Architecture

II. Flat Architecture

In previous work we introduced the Work Queue application framework [6]. From the programmer's perspective, a Work Queue application is a program written in the fork-join style using a library interface in C, Perl, or Python. The programmer composes tasks that consist of individual applications, each annotated with the input files that they require and the output files they are expected to produce. Tasks are submitted to the queue with `work_queue_task_submit()`, and returned to the caller via `work_queue_wait()`. In pseudo-code the main loop of most Work Queue programs look like this:

```
while(not done) {
  for(each new task) {
    task = work_queue_task_create(command);
    /* specify files used by task here */
    work_queue_task_submit(queue, task);
  }
  task = work_queue_wait(queue);
  /* process the result of this task */
  work_queue_task_delete(task);
}
```

To execute the application, the end user must start a number of generic worker processes on desired machines. This can be done manually or by submitting the workers as jobs in a batch system. Each worker contacts the master process and reports its available resources, including CPU cores, disk space, and memory. To execute a task the master first identifies the best connected worker for the task using one or more resource metrics (average measured execution speed, available storage or memory, cache contents). It sends the input files necessary for the task, which the worker stores on its local disk. The master then sends the command to be run for that task which is executed by the worker in the sandbox directory where the files are stored. When the task is complete the master requests the output files, any input or output files marked as cacheable are retained on the worker for use by future tasks, and the remainder are deleted. The master keeps a list of each worker's cache contents to inform future scheduling decisions.

Work Queue is deliberately a *shared-nothing* architecture. Every task is self contained, being described entirely by its input files and a command string. Each worker relies only on the files transmitted to it by the master. This permits the system to operate correctly across a variety of computational resources without requiring a common shared filesystem or other data movement system. It also gives the master visibility into the data state of the system, so that tasks can be assigned to appropriate storage resources.

Typically, tasks executed through Work Queue have two kinds of data dependencies: *common data* consists of programs, libraries, configuration files, and other static data that is shared between multiple tasks, while *unique data* serves as input to a single task, and is never used again.

A few other elements of the system are worth a brief mention. Workers are directed to connect to masters either by providing an explicit hostname and port or by specifying a logical "project" name which is resolved through the use of an external catalog service. Security is maintained by the use of a shared key which is securely verified at the start of each network connection. Fault tolerance is achieved by tracking the tasks assigned to each worker and reassigning in the event of a failure. Garbage collection is performed by having each worker to delete its cache of data whenever its connection to the current master fails or the worker shuts down.

In a university context, it is common to have access to multiple clusters of different kinds – a single user may have access to a dedicated cluster of their own, a larger shared cluster at their own institution, an allocation on national cyberinfrastructure, and pay-as-you-go access to a commercial cloud. For a problem of very large scale one might wish to harness all of those resources simultaneously. This can be accomplished with Work Queue by running workers at each resource that report back to a single master process at whatever site contains the original data and applications to be run.

While this multi-site technique does work and has been used for real applications harnessing 3000+ cores[1], it runs into some fundamental problems as the application size scales up:

Master Bandwidth. Every task requires the master to send some amount of data to describe it, limiting the number of workers a master can support before its bandwidth is saturated.

Resource Volatility. For large-scale computations resources leave and join frequently, due to equipment failure, network disruption, system policy and other factors. Every time a worker reconnects the common data must be re-sent, consuming additional bandwidth and diverting the master's attention, slowing the computation.

Administrator Policy and System Limits. Oftentimes cluster policy or operating system configurations will enforce arbitrary limits on a user's computation that are inaccessible or even invisible to the user. Things like limits on the number of open file descriptors or simultaneous TCP connections, or firewall or network translation devices preventing inter-cluster communication become major impediments as applications reach increasingly larger scales.

III. Hierarchical Architecture

To address these problems, we have extended Work Queue into a fully hierarchical system, by adding two new components: a multi-slot worker and a foreman.

A *multi-slot worker* represents a machine capable of running multiple concurrent tasks. The master can transmit an arbitrary

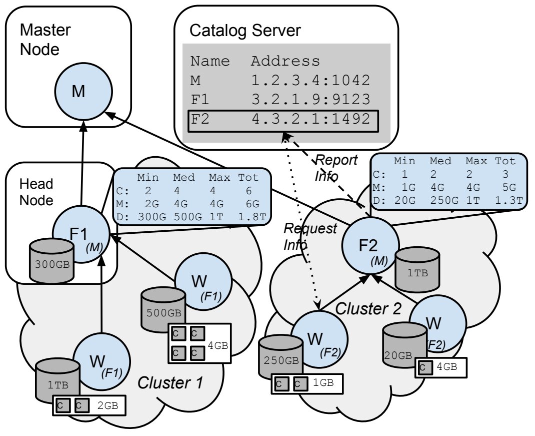


Fig. 2: Resource Discovery and Reporting

number of tasks to the multi-slot worker, which then runs as many tasks simultaneously as the available resources (cores, memory, and disk) will permit. If the master transmits more tasks than can be executed simultaneously, they are queued until the necessary resources are available.

In a similar way, a *foreman* represents a tree of workers. The foreman connects to the master and accepts tasks like a worker, while allowing workers to connect to it as a master. As the foreman receives files and tasks from its supervisor, it stores the files on its local disk and queues the tasks up for dispatch to its subordinates. Common files that are stored at the foreman can be transmitted to its subordinates without incurring any load on the master. As workers connect to the foreman, or when connected workers fail or are removed, the foreman reports these resource changes to the master, allowing the master to appropriately allocate tasks.

The system is designed to accept an arbitrary level of composition. Foremen can supervise foremen, leading to a control structure that is many layers deep. Furthermore, workers can connect directly to the master, even when foreman are also connected. From the master's perspective a worker and a foreman are nearly indistinguishable: both report their available resources for task execution, except that the resources reported by the foreman may change frequently as workers join and leave the system.

The hierarchical architecture offers solutions to many of the problems encountered when scaling up traditional master-worker applications:

Intermediate Caching. One of the methods the hierarchical Work Queue architecture offers for increasing the scale of applications is cluster-local caching. In a flat architecture, data that can be cached must be transferred from the master to every new worker that needs it. In a hierarchical system any common data shared by jobs assigned to a given foreman only needs to be transferred once from the master to the foreman where it can be accessed repeatedly by the foreman's subordinates. This allows the master to focus on transferring unique data to set up new tasks, allowing the foremen to transfer common data to multiple workers in parallel.

Furthermore, it is often the case that bandwidth into a cluster is much less than the bandwidth within a cluster, sometimes by an order of magnitude or more. In a flat system this would result in a long transfer time for the common data sent to every worker. In a hierarchical system that data is transferred only once over the slow link to a foreman located in the cluster, which can then retransfer that data using the higher intra-cluster bandwidth to each of its subordinates.

Resource Volatility. In the presence of large-scale resource volatility, where frequent disconnection and reconnection effectively multiplies the number of "new" workers seen by the system, a hierarchical architecture can help ameliorate the data transfer burden by shifting it to the foremen. When "new" workers connect, instead of diverting the master's attention from task dispatch, the foreman they connect to can handle the data transfer. This isolates any slowdown due to additional data transfer at the affected foreman, rather than impacting the entire system. The master can continue to handle tasks assigned to the unaffected foremen while the affected one brings the new worker up to speed. Foremen can also retain tasks even when the workers die, retransmitting those task descriptions to the reconnecting workers without needing any further communication from the master.

Administrator Policy and System Limits. Many of the limits encountered when scaling up Work Queue applications can be avoided or their impact can be minimized though the use of a hierarchical architecture. By splitting up the control structure, each foreman can operate well within the limits imposed on it. Once any limit is reached, a new foreman node can be allocated to control the remaining resources.

Furthermore, by introducing the foreman we added a convenient point of control for each cluster. A foreman can be configured to limit its bandwidth utilization, disk consumption, or consumed concurrency in order to ensure that it's resource consumption remains within the parameters set by the system administrator. These policies can be set on a per-cluster basis, and remove the burden for ensuring compliance with cluster limits from the master.

Network Administration. One of the major issues facing users of distributed software are the multitude of administrative restrictions and constraints encountered when using community resource managers. Oftentimes for security reasons computational clusters are isolated by firewall preventing communication from within the cluster to the outside world except via a small number of head- or submission- nodes. Furthermore, even in open clusters it is common for systems administrators to be wary of large numbers of network connections to the outside world spontaneously erupting, as that pattern looks suspiciously similar to malicious behavior.

The hierarchical architecture enables the user to run a single foreman process on the head node, having all jobs within the cluster report to that foreman, who makes a single connection to the outside world. This keeps systems administrators happy, as they have a single well-known application directing network traffic, while still allowing the user to harness whatever shared resources they might have access to.

Transfer Cost. One prevalent trend in distributed computing is the increasing reliance on pay-as-you-go cloud services. These infrastructure as a service providers offer instantaneous scaling to whatever your problem size is, often for far less than maintaining a traditional cluster would cost. However, each provider has a different set of policies for how much each resource costs and when that cost is charged. The hierarchical architecture allows the user acquiring the cloud resources to specify policy for each cloud, leaving the foreman for that provider in charge of managing the resources available to it to maximize value and minimize costs. This frees the master from having to make those decisions, and allows each foreman policy to be tailored to the cost structure of its surroundings.

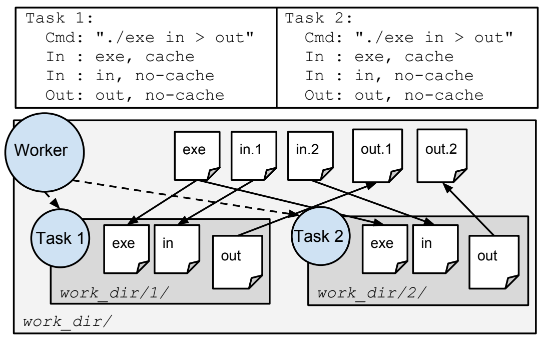
IV. Implementation

Transforming the architecture described in Section II into the system described in Section III required both adapting the worker to handle multiple tasks simultaneously and the implementation of a foreman process. For the multi-slot worker the architecture was already mostly in-place, requiring only mechanisms to assign multiple tasks to the same worker and monitor those tasks simultaneously during execution.

The foreman was implemented as an amalgamation of the worker and master components. The foreman attaches to its supervisor like a worker, accepting files and tasks via the same communication protocol. Once a task description is received, the foreman packages and submits it to an internal work_queue, where it is dispatched to any workers connected to the foreman.

Implementing and testing the multi-slot and foreman components required us to reconsider many previously well-understood aspects of the system including namespacing, resource reporting, cache management and failure recovery.

Namespacing. Under the flat Work Queue architecture, each worker manages a single task. This simplified both caching and file management, ensuring that as long as the files attached to a task had no namespace collisions, the task executing on a worker would also have no namespace collisions. Furthermore, each task could have a file with the same remote name attached to it, and as long as none were marked as "cacheable" their contents did not matter, as they would never interact.



```
Task 1:                       Task 2:
  Cmd: "./exe in > out"         Cmd: "./exe in > out"
  In : exe, cache               In : exe, cache
  In : in, no-cache             In : in, no-cache
  Out: out, no-cache            Out: out, no-cache
```

Fig. 3: Namespace Handling

When introducing the hierarchy this policy of exclusive namespaces comes into conflict with the desired benefits of common caching. Files must be uniquely identified if we wish to share data among concurrently executing tasks without re-transfer. If they are not, either incompatible versions of a file will overwrite each other, or the segmentation necessary to prevent name collisions would prevent any attempt to share caches among tasks.

To resolve this problem we have implemented task-centric file management at the foreman and worker processes. Each task gets a dedicated sandbox within the worker's active directory. Prior to task execution, the necessary input files are linked from the worker's common cache directory into the task's sandbox. Upon task completion the output files marked as cacheable are moved into the worker's cache directory, and the remainder are saved for retrieval. Once the relevant files have been retrieved, the task's sandbox is removed.

This solution has a beneficial side effect of preventing a misbehaving task from leaving untracked files polluting the workspace. Every new task is guaranteed a pristine working environment, containing only the files explicitly requested.

Resource Management and Reporting. Another major consideration in implementing hierarchy was identifying how resources attached to a project should be reported to and managed by the master. This is especially important when heterogeneous or shared homogeneous resources are attached to a foreman, as resource distribution is unlikely to be uniform and may also be unstable. The capabilities of each resource available have major implications for scheduling policy and failure profiles and must be carefully managed by the master.

Each worker reports to its supervisor the number of cores available, amount of free disk space, system memory, and its architecture and operating system. This allows the supervisor to assign tasks and avoid overcommitment.

In the case of foremen, it would be impractical to report to the master the details of every resource, as that would drown the master in data without much benefit. Instead, the foreman reports key statistics about each resource, including minimum, maximum, and total available. This gives the master enough information to ensure that a task assigned to a foreman will be able to be completed: assigning a task under or at the minimum value for a foreman ensures that every worker connected to the foreman can run the task, while assigning a task at or under the maximum ensures that *at least one* worker can run it. Finally, tracking the total resources used in comparison to the total available allows the master to avoid overcommitment.

Failure Recovery. With the addition of hierarchy to Work Queue the cost of a failure increased. Instead of losing just one task per failure the loss of a foreman or multi-slot worker means the loss of all resources it manages. When each foreman is configured to manage many workers, each failure can result in the loss of a huge fraction of the active resources.

The solution to this problem is a combination of increasing the resources and reliability of the foremen and increasing their tolerance for temporary or transitory failures. Adding additional foremen reduces the cost of each failure, while running the foremen on well-provisioned, reliable machines reduces its frequency. Increasing the tolerance of the system for transitory failures by increasing timeout values or allowing for disconnection and reconnection also reduces the failure rate, but requires a mechanism (or policy) for distinguishing between transitory and permanent failures, as the only thing worse than a system which fails often is a system which never fails but never completes.

For our implementation, where possible, we run the foremen on large, well-provisioned and fairly reliable machines, where the probability of random failure is small. We also relaxed some timeouts, allowing us to ignore brief interruptions in service and reduce observed failures to an acceptable rate.

Cache management. Under the flat Work Queue, caches were managed by a combination of master oversight in the form of explicit delete commands for files which should not be cached and disk monitoring on the worker's side. When attempting to store a file at the worker, if there was insufficient space it was considered an error and the worker would reset itself. Given the relatively minor cost of a failure this produced an acceptable method of cache management, allowing the master to intelligently schedule tasks to take advantage of cached data while minimizing undesirable interactions with uncacheable data and providing an escape valve for overfull resources.

With the addition of hierarchy and multi-slot workers the cost of resetting due to an overfull cache becomes much larger. Under this system a more proactive approach to cache management is needed. For this system we implemented a more aggressive cache clearing strategy, with the workers and foremen actively deleting any file not specifically marked as cacheable. Along with ensuring that developed applications do not cache unnecessary files and correctly provisioning the foremen nodes, this has reduced catastrophic restarts to a manageable level.

V. Sensitivity Study

The net impact of hierarchy in a master-worker system is affected by numerous task and system characteristics. To determine the role each characteristic has in the overall performance of the system we performed a sensitivity analysis. We set up a synthetic application, modeled after AWE, in which we could vary each component of the system to get an accurate picture of which elements provide the most impact. Each experiment consisted of 500 tasks that would "busy-sleep" for one minute each. For the default configuration each task consumed 400MB of Common data and 100MB of Unique data per task, with all communication between the master and its subordinates occurring on a 1Gbps network, and near-zero chances of worker failure. We ran each experiment using both flat and hierarchical setups with multiple foremen.

Resource Volatility. We investigated the behavior of the system with varying levels of resource volatility by adding a tuneable parameter to the `work_queue_worker` that introduces a chance of failure every minute. Upon failure, the worker disconnects from its supervisor and then immediately reconnects. We varied the volatility of each worker from a 0% chance of failure up to a 30% chance of failure per minute and used the system logs to determine the runtime of the system, number of failures that actually occurred, and the amount of data transferred by each component.

We expected to see a slight benefit for performance due to the isolation of failures at the foremen. We also expected to see a drastic reduction in the amount of data the master had to transfer relative to a flat architecture as volatility increased, since all error handling would be performed by the foremen.

Bandwidth. We also investigated the effect of hierarchy on the data transfer overhead when the available bandwidth at the master varies. We used the Work Queue's bandwidth limiter and varied the bandwidth available at the master from the default 1Gbps down to 200Mbps. For the hierarchical tests the bandwidth at the foremen was left at 1Gbps. We expected to see a performance benefit with the hierarchical system as the master-to-foreman bandwidth ratio increased, due to the substantial savings in transfer cost for common input data.

Data Size - Common versus Unique. Finally we investigated the impact of varying the ratio of common to unique data on the relative performance of our hierarchical versus flat systems. Since the primary benefit of hierarchy for performance in the absence of volatility is the ability to transfer common data in parallel, we expected to see improvements when the common data made up a substantial portion of the overall data transfer. When the unique data was large in comparison we expected to see the effect of the additional data transfer from master to foreman dominate, and probably see a slight penalty to performance compared to a flat system.

VI. Results/Analysis

Worker Volatility. In the presence of volatile workers the foreman nodes in a hierarchical architecture minimize the amount of redundant data the master must transfer. This can be seen in Figure 5a, where the master in the hierarchical architecture transfers the same amount of total data regardless of the amount of volatility, while the master in the flat architecture transfers a substantial amount of additional data as the volatility increases.

This data transfer difference also has a significant effect on the runtime of the workflow. By providing the foremen to handle failures, the master can focus on dispatching new tasks instead of resending data to failed workers. Figure 5b shows that as the volatility climbs the average runtime of the

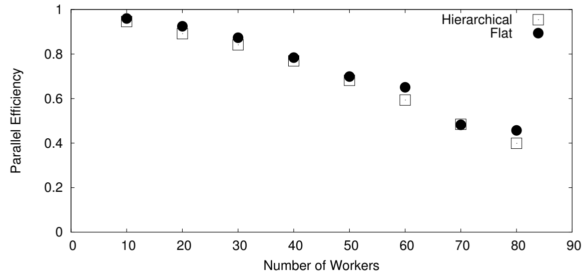
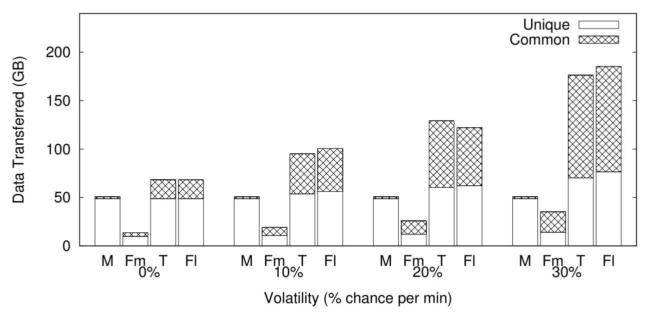
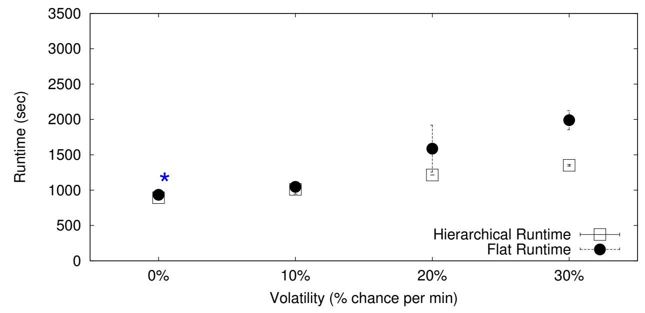


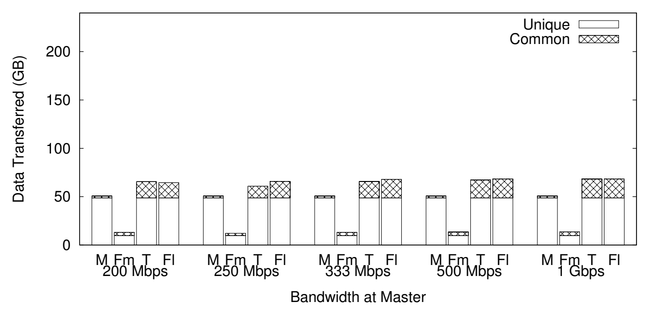
Fig. 4: System Scale



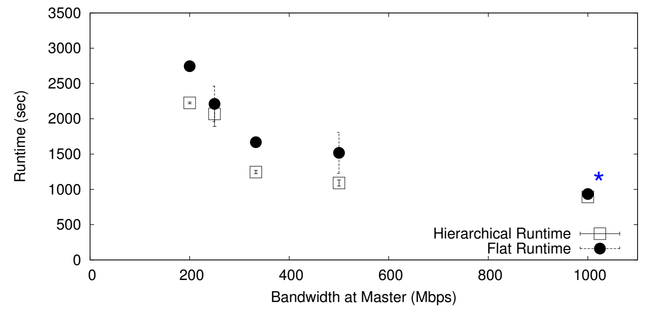
(a) Volatility - Data Usage



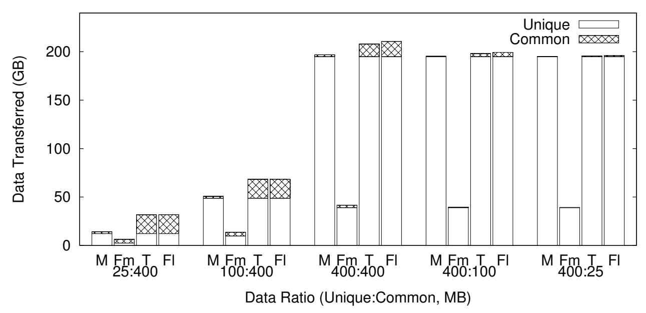
(b) Volatility - Runtime



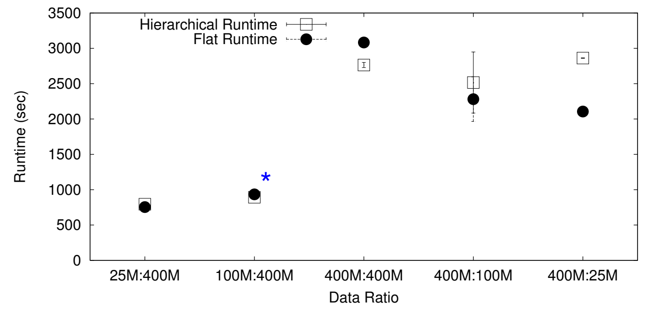
(c) Bandwidth - Data Usage



(d) Bandwidth - Runtime



(e) Varying Data Ratio - Data Usage



(f) Varying Data Ratio - Runtime

Fig. 5: Results: Volatility, Bandwidth, and Data. The data graphs show the Hierarchical Master (M), Average Foreman (Fm), Total for all Foremen (T) and Flat Master (Fl). The common cases are marked with an '*' symbol.

workflow on the hierarchical system only gradually increases, while the flat system slows down by up to a factor of two.

Bandwidth. As expected, the imbalance in available bandwidth between the master and the foremen versus the foremen and their workers did not affect the total amount of data transferred (Figure 5c). For each bandwidth constraint, the master in both architectures transferred a consistent amount of data, though the hierarchy reduced the amount of common data transmitted by the master. This reduction, coupled with the difference in bandwidth between the two sites, leads to a slight reduction in runtime as the bandwidth difference increases. This is shown in Figure 5d, where when the bandwidth at the Master is limited to 200Mbps (approximately $\frac{1}{5}$ of the bandwidth between the foremen and the workers) we see a 19% reduction in runtime.

Data Size. In the absence of worker volatility the primary effect of the hierarchical architecture on runtime is the reduction in common data transferred by the master, and the additional parallelism in transferring that common data achieved by the foremen. When the amount of data common across all tasks is large compared to the unique data per task we should see a slight reduction in runtime. The first three columns in Figures 5e and 5f reflect this.

However, as the proportion of common data versus unique data decreases, the additional latency imposed by the foreman on transmitting unique data begins to have a substantial negative effect on performance. For tasks that are almost entirely unique data, the hierarchical system can impose a performance penalty as seen at the left side of Figure 5f.

The results of this study indicate that the hierarchical work queue architecture enables a reduction in network footprint at the master in every circumstance, and that the reduction becomes substantial when the amount of common data across all transactions is large or the amount of volatility in the available resources is high. Furthermore, it demonstrates a slight performance improvement in the presence of worker volatility or network bandwidth imbalance.

VII. Case Study: Multi-Site AWE

Accelerated Weighted Ensemble (AWE) [1] is a method for enhancing the sampling accuracy of the molecular dynamics simulations of protein systems. It partitions the conformational space of a protein into cells and creates a fixed number of simulation tasks or "walkers" in each cell. Every walker is assigned a probabilistic weight such that they provide an unbiased sampling of the conformational space. The sampling efficiency is further improved by utilizing a large number of short simulation steps.

Our earlier work described the implementation of AWE using the original flat Work Queue framework and demonstrated its inherent scalability in harnessing 3500 cores from heterogeneous resources in multiple distributed computing platforms [1]. For the case study in this work, we use the improved hierarchical Work Queue to show scalability while simultaneously consuming less wide-area network bandwidth and harnessing resources in a more cluster-friendly manner.

Experimental Setup. We ran AWE on the WW protein domain using 20 walkers, resulting in 12,000 tasks per iteration. There was 40.5MB of common data and each task's unique input files totaled 75KB.

We used resources from four clusters - Notre Dame's High Performance Cluster, the ND Condor pool, FutureGrid's Sierra cluster (at San Diego Supercomputer Center), and FutureGrid's India cluster (at Indiana University, Bloomington). At each independent cluster (ND-HPC, Sierra, and India) we ran a single foreman on the head node and set up single-slot worker processes for every core in our allocation: 200 workers at ND-HPC and 800 workers across the two FutureGrid sites.

We submitted 5000 workers to the ND Condor pool of which approximately 3400 were seen over the course of the experiment. Due to an IT department policy there is a limit of 1024 simultaneous TCP connections on each Condor node, so we allocated one foremen for every thousand Condor workers. Each foreman ran on its own machine at the same data center as the majority of the Condor pool.

Scale. We ran the experiment for about 15 hours, completing three iterations. Figure 6 shows the concurrency we achieved during the experiment. The valleys correspond to synchronization barriers at the end of each iteration, while at our peak (around hour 6) we saw 3862 workers simultaneously executing AWE tasks. The drop in concurrent workers at hour 8 occured because of the termination of allocated resources in FutureGrid (due to limits on resource usage) and Condor (due to contention for resources). Table I shows our measurements of this experiment, including the number of tasks dispatched by the master and each foreman, the total failures of foreman and its workers, the average bandwidth observed between each component, and the cumulative data sent by each entity.

Failures. Over the course of a 15-hour experiment using thousands of resources spanning multiple locations some num-

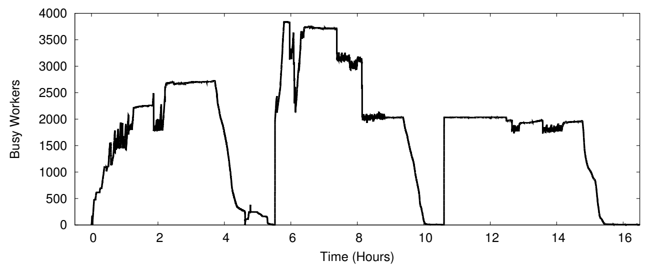


Fig. 6: Busy workers over the duration of the AWE run.

ber of failures are inevitable. Causes may include network disruptions, local disk failures, and scheduling policies. Examining the results in Table I we noticed the number of tasks dispatched by the master to a foreman was often higher than those dispatched by that foreman: this indicates that the foreman failed before being able to dispatch all the tasks assigned to it. On closer inspection we found that the foreman failed multiple times in a very short timeframe, indicating a short-term resource failure rather than transient network fluctuations. Worker failures had a much more varied set of causes, including resource failures, network disruptions, and terminations due to cluster scheduling policies.

Bandwidth and Data Usage. Table I shows the average bandwidth measured between the master and foremen, as well as between each foreman and its workers. The master to foreman bandwidth is consistently smaller than the foreman to worker bandwidth, regardless of the platform. The difference was especially pronounced in the case of our foreman running at the Sierra site in San Diego, over 1800 miles away.

Table I also shows the data sent by the master and foremen, as well as an estimate of the data a master in a flat configuration would have sent when running the same experiment. This estimate was derived by adding the data sent by each foreman and removing the data resent due to foreman failures, since those retransmissions do not exist in a flat configuration. In comparison to the hypothetical flat master configuration, we see on average a 96% reduction in data transmitted by the hierarchical master.

VIII. Related Work

While master-worker frameworks and hierarchical structures have been studied before, both independently and in combination, our contribution focuses on the data-intensive computations found in modern scientific applications. Furthermore, most studies of hierarchy discuss systems intended to replace the middleware layer at every cluster rather than work in concert with each scheduler to combine disparate resources.

Work Queue is not the first master-worker framework to look at hierarchy as a solution for scaling. Ranaldo et al present one such framework implemented in Java. Their framework uses the ProActive [7] software suite for communication and requires both a custom master application and a specially designed worker executable, both implemented as Java Classes with a programming interface reminiscent of

		Max Workers	**Tasks Dispatched**		**Resource Failures**		**Average BW (Mbps)**		**Total Data Sent (GB)**			
			M	F	F	W	M→F	F→W	M→F	F→W	Est Flat M	Savings (%)
ND-Condor	Fm 1	447	9279	9064	7	59	178	809	1.0	74.6	33.3	97.8
	Fm 2	915	14614	14033	13	294	135	900	1.7	168.9	74.6	97.7
	Fm 3	499	2451	1657	2	9	124	2163	0.3	53.5	21.4	98.6
	Fm 4	688	3146	2431	2	69	137	1481	0.4	69.0	28.3	98.6
	Fm 5	805	18715	18428	11	24	124	891	1.9	191.7	50.2	96.2
ND-HPC	Fm 1	200	7902	6855	50	6	139	492	2.6	179.4	22.5	88.4
Future-Grid	Fm Sierra	678	6793	4905	49	1120	11	449	2.6	186.4	107.6	97.5
	Fm India	88	1747	1747	3	75	267	392	0.3	16.1	12.3	97.6
Total		3862	64647	59120	141	1656	-	-	10.8	939.6	350.2	96.9

TABLE I: Statistics for the master-foreman and foreman-worker interactions from the AWE run on the WW domain.

Apache Hadoop [5]. However, their system is focused on tasks requiring minimal data, rather than the more data-intensive applications Work Queue is designed for. Similarly, Dai et all look at a system for coordinating large numbers of low-data tasks, specifically using hierarchy as a way to coordinate work stealing within and across clusters.

Many times grid middleware systems will implement hierarchy as a mechanism to coordinate resources across multiple clusters and even campuses. SZTAKI desktop grid [3] and the Hierarchical Metacomputer Middleware [9], [8] software suite are both examples of grid systems that use hierarchy for organization, specifically for coordinating access to resources behind firewalls or otherwise inaccessible due to cluster or organizational policies. However these software suites require administrator intervention to set up and are primarily concerned with enforcing fairness among many users rather than improving performance for one user.

Finally, there are multiple groups that have looked at using hierarchical frameworks or abstractions for coordinating conveniently parallel tasks. Berthold et al [4] and Priebe et al [11] explore a variety of architectures for managing large master-worker systems. Both assume minimal data and homogeneous networks, and in both systems task generation is leaf-driven, with the upper layers of the hierarchy used as conduits for load balancing instead of Work Queue's centralized control.

IX. Conclusions and Future Work

Our results demonstrate substantial benefits for data transfer, workflow runtime and/or administrative headaches under any of the following conditions, or a combination thereof.

Large Shared Data. Hierarchy substantially reduces the cost to the master of transferring common input data allowing it to concentrate on dispatching tasks while the foremen transfer that common data to their workers in parallel.

Heterogeneous Networks. Hierarchy allows common data to be transferred only once across slow inter-cluster connections, with the foreman handling distribution of the data within the cluster using the higher-speed intra-cluster network.

Volatility in the Workforce. Hierarchy minimizes the effect of volatile resources by having the foreman absorb the additional data transfer, allowing the master to focus on dispatching new tasks rather than replacing lost common data.

Hierarchy offers a mechanism for users to comply with cluster policies and system limitations while still harnessing the resources they have available, by using the foreman as both a bridge from isolated clusters to the outside world and a control point to monitor and limit resource consumption. It also benefits the end user through increased access to otherwise unusable resources and by offering performance improvements in common operating circumstances.

Acknowledgments. We thank Prof. Jesus Izaguirre and Badi Abdul-Wahid for their contributions and support for the experiments using AWE. We acknowledge funding from NSFOCI-1148330 and NSF-CNS-0643229.

References

[1] B. Abdul-Wahid, L. Yu, D. Rajan, H. Feng, E. Darve, D. Thain, and J. A. Izaguirre. Folding Proteins at 500 ns/hour with Work Queue. In *8th IEEE International Conference on eScience (eScience 2012)*, 2012.

[2] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[3] Z. Balaton, G. Gombás, P. Kacsuk, A. Kornafeld, J. Kovács, A. Marosi, G. Vida, N. Podhorszki, and T. Kiss. Sztaki desktop grid: a modular and scalable way of building large computing grids. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[4] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical master-worker skeletons. *Practical Aspects of Declarative Languages*, pages 248–264, 2008.

[5] D. Borthakur. The hadoop distributed file system: Architecture and design. http://hadoop.apache.org/, 2007.

[6] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[7] D. Caromel, C. Delbé, A. Di Costanzo, M. Leyton, et al. Proactive: an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12, 2006.

[8] M. Di Santo, F. Frattolillo, N. Ranaldo, W. Russo, and E. Zimeo. Programming metasystems with active objects. In *Proceedings of the 5th International Workshop on Java for Parallel and Distributed Computing*, 2003.

[9] M. Di Santo, F. Frattolillo, W. Russo, and E. Zimeo. A component-based approach to build a portable and flexible middleware for metacomputing. *Parallel Computing*, 28(12):1789–1810, 2002.

[10] C. Moretti, A. Thrasher, L. Yu, M. Olson, S. Emrich, and D. Thain. A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids. *IEEE Transactions on Parallel and Distributed Systems*, 23(12), 2012.

[11] S. Priebe. Dynamic task generation and transformation within a nestable workpool skeleton. *Euro-Par 2006 Parallel Processing*, pages 615–624, 2006.