# Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications

Peter Bui, Dinesh Rajan, Badi Abdul-Wahid, Jesus Izaguirre, Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{pbui,dpandiar,cabdulwa,izaguirr,dthain}@nd.edu

***Abstract*—Even with the increase in the number and variety of computer resources available to research scientists today, it is still challenging to construct scalable distributed applications. To address this issue, we developed Work Queue, a flexible master/-worker framework for building large scale scientific ensemble applications that span many machines including clusters, grids, and clouds. In this paper, we describe Work Queue and then present the Python-WorkQueue module, which enables scientists to take advantage of our Work Queue framework while using the Python programming language. To demonstrate the module's flexibility and power, we examine two distributed scientific applications, RepExWQ and Folding@work. Both of these programs were written using Python-WorkQueue and manifest the Work Queue framework's ability to scale not only to hundreds of workers, but to also enable scientists to take advantage of multiple distributed computing resources simultaneously.**

***Keywords*-python; work queue; master/worker; distributed systems; cloud computing; molecular dynamics**

## I. Introduction

Today, research scientists face the challenge of efficiently and effectively utilizing the abundance of computing resources now available to them through campus clusters, computing grids, and cloud environments. Although there are tools for building applications for each of these individual distributed environments, there are very few systems designed to harness the computing power of all of these resources simultaneously.

To address this problem, we developed Work Queue, a flexible master/worker framework for constructing large scale scientific ensemble applications that span many machines including clusters, grids, and clouds. Unlike traditional distributed programming systems such as MPI, Work Queue allows for an elastic worker pool and thus enables the user to scale the number of workers up or down as required by their application. Additionally, it provides fault tolerance for intermittent errors by gracefully handling worker failures. Moreover, Work Queue also provides data management features to support data intensive distributed applications.

We have briefly mentioned Work Queue in passing in some of our previous work [1], [2], [3]. This paper presents a detailed explanation and description of the Work Queue framework and introduces the Python-WorkQueue module, which allows research scientists to construct scalable distributed ensemble applications using the Python programming language [4].

Although there is some previous research on parallel and distributed computing frameworks for Python, Python-WorkQueue has a few distinguishing properties. First, Work Queue is designed as a framework for composing an *ensemble* or collection of scientific applications. That is, rather than composing the entire application as a single program, Work Queue applications consist of multiple executables, each responsible for a single stage or function in the application workflow. This is in contrast to modules such as PyMW [5], Mpi4Py [6], PyDoop [7], Scientific.BSP [8], which are designed for building monolithic distributed applications primarily in Python and where functions are typically embedded internally in the application code rather than as independent external executables.

Second, Work Queue provides additional distributing computing features to enable the construction of robust and scalable scientific applications. For instance, it provides workflow data management for caching data on remote workers to avoid costly data transfers. As mentioned earlier, it also supports fault tolerance for intermittent errors, and will automatically retry tasks in certain conditions. Additionally, Work Queue has multiple scheduling algorithms that can be used to optimize how tasks are dispatched to workers and it has a feature for detecting and dealing with straggler tasks. Work Queue also provides a catalog discovery service for automatically connecting workers to different masters.

Finally, Work Queue's flexible worker deployment mechanism allows Work Queue applications to take advantage of computing resources from multiple distributed environments such as a personal cluster, campus grid, or public cloud provider. Not only can Work Queue applications execute on different types of distributed systems, but they can also utilize resources from multiple platforms simultaneously to form a personal cloud consisting of hundreds to thousands of workers.

In the next section of the paper, we provide an exhaustive description of the Work Queue framework and the features it provides for developing scalable ensemble applications. After this, we present the Python-WorkQueue module, describe our implementation, and provide an example of using the API to build a distributed image conversion program. To demonstrate the power and flexibility of using Work Queue and Python together, we then examine two scientific ensemble applications, RepExWQ and Folding@work. Both of these distributed

applications manifest Python-WorkQueue's ability to scale not only to hundreds of workers, but to also enable scientists to take advantage of multiple distributed computing resources. Finally, we conclude the paper with a discussion of related work, and consider possible avenues for future research.

## II. Work Queue

The Work Queue programming model is based on the traditional master/worker pattern [9]. In this model, a *master* dispatches a series of *tasks* to a pool of *workers* to execute. Normally, the master contains all of the application logic and orchestrates the data flow, while the workers perform specific computational functions described by the tasks sent from the master. Unlike systems such as MPI, the worker pool is not fixed and can grow and shrink dynamically over the life-time of the application.

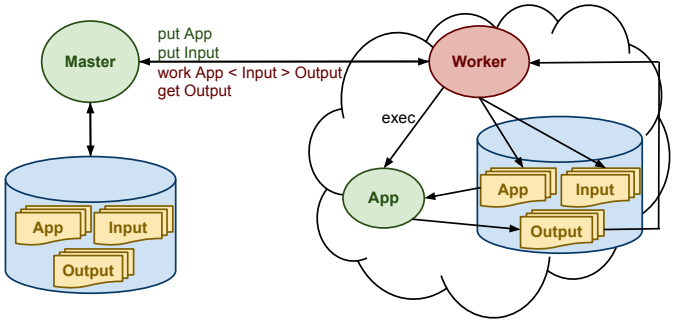


Fig. 1. Work Queue Master/Worker Pattern.

Figure 1 provides an overview of master/worker pattern utilized in Work Queue. In a typical Work Queue application, there is a single master and many workers. The master maintains a set of tasks referred to as the *workqueue*, where each task consists of a specification of the required executable and input files, and the output files to be generated by each task. Workers run as separate processes, receive these tasks, and perform the specified computation. As mentioned in the introduction, Work Queue was designed as a framework for combining and orchestrating an *ensemble* or collection of scientific applications. As such, all computational tasks are encapsulated in external executables, rather than internal functions. This is convenient as it allows for re-using existing software to develop more sophisticated scientific workflows.

To coordinate the distribution of data and the execution of applications, Work Queue utilizes a TCP-based RPC protocol for communicating between the master and worker. The protocol includes commands such as `get` and `put` to transfer data between the master and worker and `work` to have the worker execute a task. The general communication pattern for a Work Queue application is as follows:

1) The master retrieves the next pending task from its workqueue.
2) Next, the master selects a worker and transfers any data such as the application executable or input files required by the task by using the `put` RPC.
3) Once the files are on the remote worker, the master requests the worker to execute the task with the `work` command.
4) When the task is complete, the worker completes the `work` RPC previously sent by the master by responding with the output and exit status of the application executed in the task.
5) Finally, the master retrieves any output files specified in the task from the worker using the `get` RPC.

This general process is performed continuously until the workqueue is empty and the application is complete.

### A. Data Management

As noted previously, Work Queue provides a mechanism for sending and receiving data between the master and worker. This is because Work Queue does not assume any shared data storage system, and thus considers the master and workers to be operating in separate sandbox environments. In this model, the master and workers only have access to their own independent local filesystems.

This means that before a task can be executed, any necessary executable and input files must first be transfered from the master to the remote worker. These data dependencies are specified when creating a task. If a file is to be utilized in subsequent tasks, the user can tell Work Queue to cache the file on the worker. When a file is cached, it will only be transferred if the remote worker does not already have the file, which can greatly improve performance by avoiding costly data transfers. Caching is available for both input and output files, which is useful workflows that consist of a pipeline of sequential operations.

In addition to files, Work Queue also supports transmitting data that is stored in memory. Since the execution model of the framework is based on coordinating ensembles of external applications, any data stored in memory will be materialized as a file upon task execution. That is, the master may transmit data stored in a memory buffer to the worker, who will then transfer that data to a file on its local filesystem to be used by the task application. Similarly, the standard output of the task is stored in memory and transmitted back to the master upon completion. As such it is not necessary for the user to manually capture the standard output of the task applications.

To ensure that the remote environment is not littered with artifacts of the Work Queue application, workers will perform periodic garbage collection. Whenever the worker disconnects from the master or is terminated, it will also remove the contents of its sandbox environment.

### B. Fault Tolerance

Because Work Queue was designed to work in a distributed environment, it provides some measure of fault-tolerance. In particular, it is resilient to communication link failures and will automatically re-schedule a task if it detects a connection drop between the master and worker.

Figure 2 provides the state diagram of a Work Queue task. A task begins in the *ready* state and then transitions

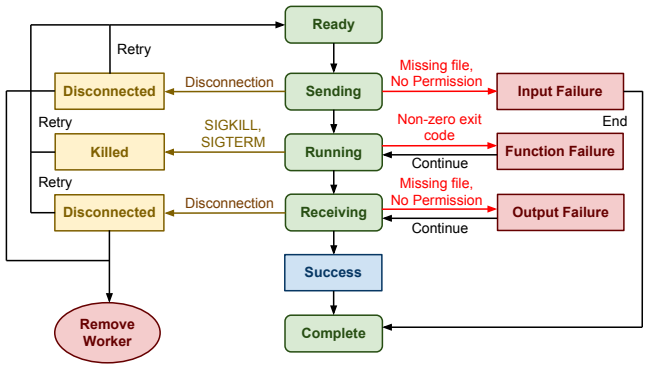


Fig. 2. Work Queue Task State Diagram.

to the *sending* state when the master begins to transfer its dependencies to a remote worker. If a failure such as not having the appropriate permissions to read a file on the master occurs, the task's status is marked as *input failure* and the task is placed in the *complete* state as nothing else can be done by Work Queue to overcome this error. If a transmission error occurs while sending, then the task is put back in the *ready* state and will be retried. Additionally, the worker will be removed since the network connection has failed.

Once the files are transferred, then the worker begins executing the task, which is then placed in the *running* state. If the task process is killed by the remote machine, Work Queue will put the task back in the *ready* state to be retried. Otherwise, the exit status of the process is checked. If the application returned abnormally, then the task's status is marked as *function failure*. Regardless of the application's exit status, the task moves to the *receiving* state.

After the task is finished executing, the master will request the output files from the worker. At this point, the task transitions to the *receiving* state. If a file is missing or has the wrong the permissions, then the task's status is set to *output failure*. Otherwise, if all files transfer successfully, then the task's status is set to *success* and the task finishes in the *complete* state.

Overall, the Work Queue task state machine is designed to allow tasks to be retried automatically when intermittent errors such as transmission failures or eviction on the remote host occur. For other types of failures, Work Queue marks the task with a description of the error and allows the user decide what to do. The user may wish to re-submit the task to the workqueue, and in that case the task will be retried again.

### C. Scheduling

Work Queue provides multiple scheduling algorithms for selecting which worker to dispatch a task to based on certain criteria. That is, when a task is ready for execution, it is sent to a worker that is determined by one of the following methods:

1) **First Come First Serve:** In this algorithm, tasks are assigned to the first ready worker in the queue. If there are more tasks than workers, then this algorithm will ensure that workers are kept busy with work. However, if there are more workers than tasks, then it is possible for some workers to starve for work if they are at the end of the queue. Fortunately, this is a rare occurrence as there are usually many more tasks then workers.
2) **Cached Files:** This algorithm takes advantage of the fact that Work Queue tracks what input and output files are stored on each worker, and will prefer the workers that already contain the required input data for the current task. Normally, Work Queue will select the worker that has the most required data and is ready. If no worker has the required files, then the first ready worker is selected. In either case, any missing input files will be transferred to the worker if necessary. This method takes advantage of data locality, which can improve the performance of data intensive applications.
3) **Fastest Time:** This scheduling option selects the host that has the fastest average turnaround time for completing tasks and transferring data, and is in the ready state. The purpose of this method is to favor the best performing workers and thus attempt to avoid any stragglers. If no host is available, then the *First-Come-First-Serve* scheduler is utilized to find a suitable worker.
4) **Preferred Hosts:** Work Queue allows users to tag tasks with a set of preferred hosts. When using this method, Work Queue will attempt to match the task to one of its preferred hosts. If none are available, then the first ready worker is selected. This option is useful for optimizing host selection based on priorities such as data locality, operating environment, hardware requirements, etc.
5) **Random:** This algorithm attempts to perform rudimentary load balancing by selecting a random ready worker.

These scheduling configurations may be set on both a per workqueue basis, or a per task basis by the user. The default scheduler is the *First-Come-First-Serve* algorithm.

### D. Fast Abort

In any distributed system, it is possible for unexpected delays to occur. For instance, in a Condor pool, a running task may be arbitrarily delayed during execution; it may be evicted by system policy, stalled due to competition for local resources, or simply caught on a very slow machine. Likewise, a task in a cloud environment like Amazon EC2 may be sent to an over-committed node or experience network issues, and thus face slow execution time. Such delays may inhibit parallelism and adversely affect overall execution throughput.

To address these problems, the Work Queue provides a *fast abort* option. By keeping statistics on the average execution time of successful jobs and the success rate of individual workers, Work Queue can determine which tasks are progressing too slowly. End users enable the *fast abort* option by setting a multiplier. When this option is set, Work Queue proactively aborts and re-assigns any tasks that have run longer than $fast\_abort\_multiplier \times average\_execution\_time$ to different workers. Previous research has shown that careful use of the *fast abort* option can increase parallelism for certain applications [1].

### E. Worker Deployment

Because Work Queue utilizes a TCP-based RPC protocol for coordinating the interaction between the master and the workers, it is possible to start a worker anywhere that has a network connection. Unlike some other systems, the deployment and activation of a worker is performed externally by the user, rather than by the Work Queue framework. That is, users must manually start workers using whatever means is convenient to them. Fortunately, this is a straightforward process as each worker is simply a statically linked executable called `work_queue_worker`. To start a worker, the user simply runs the following:

```
work_queue_worker <hostname> <port>
```

Listing 1. Start a Work Queue worker.

To simplify the activation of multiple workers on various distributed systems, we include a set of worker submission script for starting workers on Condor [10], Sun Grid Engine (SGE), and local multi-core machines. Here is an example of activating 50 workers on a local campus Condor cluster:

```
condor_submit_workers <hostname> <port> 50
```

Listing 2. Submit 50 Work Queue workers to Condor pool.

In Listings 1 and 2, the `hostname` and `port` refer to the location of the master the workers should connect to in order to retrieve tasks.

In addition to these deployment scripts, we also provide a utility called `work_queue_pool` which not only deploys workers to Condor, SGE, and local multi-core machines, but will also maintain a constant pool of workers. For instance, when using Condor, it is possible for a worker to be evicted. Unlike the submission scripts, the `work_queue_pool` utility will monitor and track the execution of the workers and if they are stopped for any reason, it will automatically resubmit a worker. This allows for the user to maintain a worker pool of constant size, which is useful for certain workflows.

### F. Catalog Discovery Service

Sometimes, it is not always known *a priori* where the master application is running, which would make the automation of starting a master and a pool of workers difficult since the workers require the location of the master. Other times, users may wish to maintain a constant pool of dedicated workers, but have them migrate to new masters as they come online. To facilitate such scenarios, Work Queue supports a *catalog discovery service*.

To utilize this feature, users simply apply a project name to their master workqueue and tell the workers to utilize this project name when looking up a master. When a Work Queue application has a project name and catalog mode is enabled, it will contact a known catalog server at startup and announce its availability. Likewise, when workers are in catalog mode, they will contact the catalog server and request projects identified by the desired project name. To avoid keeping stale information, the master will periodically send heartbeat messages to the catalog server and include information such as the number of workers attached, the number of workers busy, the number of tasks to perform, and the number of tasks completed. Similarly, the catalog server will also periodically remove master entries that have not been updated after a certain timeout. The status of Work Queue applications available on the catalog server can be queried using the `work_queue_status` utility.

### G. Application Architecture

Applications access the Work Queue framework through the use of the Work Queue Library, which is written in the C programming language. This library provides methods for creating a workqueue, specifying tasks, and submitting work to workers along with functions to access the previously described caching, scheduling, *fast abort*, and catalog features. To allow the Master process to perform computations while the workers are busy, Work Queue provides a polling mechanism that enables the master to check if results have been returned without blocking indefinitely.

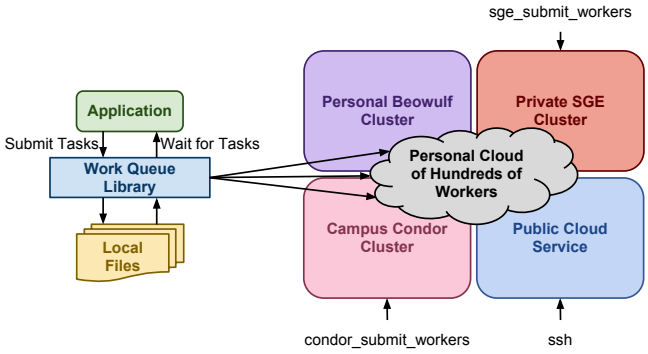


Fig. 3. Work Queue Application Architecture.

Figure 3 provides an example of the high-level architecture of a Work Queue application spanning multiple distributed systems. As can be seen, each Work Queue application utilizes the C Work Queue library to setup a workqueue and specify a set of tasks to perform. Using the worker deployment scripts described earlier, users can activate workers on a variety of distributed systems such as a personal Beowulf cluster, a campus Condor cluster, a private SGE grid, and even a public cloud service such as Amazon EC2 or Microsoft's Azure. When utilizing workers from multiple distributed environments, Work Queue applications can in effect configure and take advantage of a personal cloud of hundreds of workers for scientific ensemble applications.

## III. Python-WorkQueue

Although Work Queue has been shown to be a robust and flexible framework for constructing scalable distributed applications in previous work [1], [2], [3], its wider adoption is

slightly hindered by the fact that applications must be written in C in order to use the library. While using C is not a problem for most experienced distributed systems programmers, developing in C can be tedious and error-prone. Likewise, computational scientists who require the use of extensive computational resources and thus would benefit from using Work Queue are generally well-versed in scripting languages but not in low-level languages such as C. Therefore, to provide scientific researchers access to the Work Queue framework in a more user-friendly language, we have developed a Python binding for the Work Queue library.

### A. Python Module

The Python-WorkQueue module exposes an API that closely resembles the conventional C API, except that it converts the core Work Queue data structures into Python objects in order to make the module as *Pythonic* as possible. Specifically, the module provides two core objects, `WorkQueue` and `Task`, which have methods to perform appropriate operations such as configuring parameters, submitting tasks, and retrieving results. Once installed, the Python-WorkQueue module can be loaded by importing the `work_queue` module.

Initially, we manually wrote a Python C extension to wrap the original C Work Queue library, rather than use an auto-generation tool such as SWIG. This was done to maintain fine-grain control of the Python-WorkQueue package and to minimize the number of dependencies required for our software suite, CCTools, which now includes the Python module. Directly writing Python bindings for Work Queue by hand also allowed us to support both Python 2 and Python 3 from a single code-base. Although most scientific Python libraries only support Python 2, we made great efforts through the careful use of macros and pre-processor constructs to support both Python 2 and Python 3. This allows users to utilize whichever Python version they are most comfortable using.

The flexibility of a custom hand-written binding, however, came at the cost of maintainability. As new features were added and developed it became tedious to also have to update the Python C extension. Moreover, only a few people had experience with the Python C API, so the burden of maintenance fell to this select party. To alleviate this problem, we recently implemented a SWIG-based binding that produces a low-level or direct one-to-one binding to our Work Queue C library. We then added a second higher-level binding that builds on the low-level library and adds an object-oriented interface to the framework. This more *Pythonic* interface is what we recommend our users to utilize in constructing their applications.

Unfortunately, due to the use of shared global data structures in the original Work Queue C library, the Python Work Queue binding is not thread safe. This means that while it is possible to run multiple Work Queue masters from a single Python thread, it is just not possible to safely run different masters in multiple threads. In practice, this is not a problem as most applications only run one master.

To demonstrate the Python-WorkQueue module, the remainder of this section will provide examples of using the package to construct `WorkQueue` and `Task` objects and a small example of a complete Python-WorkQueue application.

```
# Import objects from work_queue module
from work_queue import WorkQueue
from work_queue import WORK_QUEUE_RANDOM_PORT
from work_queue import WORK_QUEUE_SCHEDULE_FILES

# Start Work Queue on first open port
wq = WorkQueue(WORK_QUEUE_RANDOM_PORT)
print "WorkQueue started on port %d" % wq.port

# Set project name
wq.specify_name('project.name')

# Select cached files scheduling algorithm
wq.specify_algorithm(WORK_QUEUE_SCHEDULE_FILES)

# Enable fast abort with multiplier
wq.activate_fast_abort(1.5)
```

Listing 3. Create WorkQueue object and specify settings.

### B. WorkQueue

The first main component of a Python-WorkQueue application is the `WorkQueue` object. Each Work Queue master application must have at least one workqueue which tracks tasks and maintains a listing of connected workers. To instantiate a *WorkQueue* object, a user simply calls the *WorkQueue* class constructor like any other Python object as demonstrated in Listing 3.

By default, if no port is specified a workqueue will be started on the default Work Queue port (9123). In Listing 3, we utilize a method provided by the module for automatically finding an open port in a given range by passing the `WORK_QUEUE_RANDOM_PORT` constant. In addition to a port argument, the `WorkQueue` object can also take optional keyword arguments for setting the project name and for enabling catalog mode.

Once we have constructed a `WorkQueue` object, we can utilize different class methods to set various workqueue parameters. Listing 3 shows how to set the workqueue project name, select the worker selection algorithm, and enable the fast abort mechanism in Python. The names of the *WorkQueue* methods and the module's constants have a direct correspondence to the C API for the most part.

During the life-time of the application, the *WorkQueue* object also maintains a set of statistics for tracking information such as the number of workers connected, workers busy, tasks waiting, tasks running, and tasks completed. To access this information, the user simply needs to get the `stats` property of the *WorkQueue* object, which will return an object with members containing all of the statistical information for the workqueue.

### C. Task

The second core component of the Python-WorkQueue module is the `Task` object. As explained previously, Work

```
from work_queue import Task
from work_queue import WORK_QUEUE_SCHEDULE_FCFS

# Input and output files
input_file  = 'input'
output_file = 'output'

# Create Task object
task = Task('cat < %s > %s' % (input_file,
                               output_file))

# Set algorithm and tag
task.algorithm = WORK_QUEUE_SCHEDULE_FCFS
task.tag = str(time.time())

# Specify input buffer and output file
task.specify_input_buffer('hello, world',
                          input_file,
                          cache=False)
task.specify_output_file(output_file, output_file)

# Submit Task to WorkQueue
wq.submit(task)
```

Listing 4. Create Work Queue Task and specify settings.

Queue is a framework for constructing scientific ensemble applications. That is, it is used to coordinate the execution of a collection of external scientific executables. Therefore, in order to perform computations on remote workers, it is necessary to specify the input and output files for each task, including any executables and libraries required for the computation.

Listing 4 provides an example of creating a *Task* object and specifying the file dependencies. In this example, we create a *Task* by passing it the shell command to execute on the remote worker. Next, we set the worker selection algorithm to use for this task. By default this is unset, which means the workqueue algorithm setting will be used. In Listing 4, we set the algorithm to *First-Come-First-Serve*. Afterwards, we give the task a tag, which is a user-defined property that can be employed by the programmer to manually track different tasks.

Then, we specify our input and output files. The file specification methods in the *Task* object take two required arguments: the name of the file on the local system, and the name of the file on the remote system. Having both arguments allows users to rename files before consumption on the remote worker. In this example, we use an in-memory string that will be materialized by Work Queue on the worker and disable caching for this input. For the output, we simply use the same file name for both the local and remote systems. After we are finished specifying the task, we submit it to the workqueue by using the *WorkQueue*'s `submit` method.

### D. Distributed Convert Example

Listing 5 is an example of a complete application written using our Python-WorkQueue bindings that converts a set of input images to another format using ImageMagick's `convert` utility. The program in takes an initial argument that specifies the new output image extension, which is then followed by a list of input images to convert. For each of these images, we determine the output filename and create a task that uses the `convert` utility to transcode the original input image to the new output image. To ensure we transfer our files properly, we mark the input and output file dependencies in the *Task* object we created and then submit the specified task to the *WorkQueue* object.

```
from workqueue import WorkQueue, Task
import os, sys

wq = WorkQueue()
output_ext = sys.argv[1]

# For each file, construct & submit a transcoding task
for input_file in sys.argv[2:]:
    output_file  = os.path.splitext(input_file)[0]
    output_file += '.' + output_ext

    task = Task('convert %s %s' % (input_file,
                                   output_file))
    task.specify_input_file(input_file, input_file)
    task.specify_output_file(output_file, output_file)
    wq.submit(task)

# While workqueue is not empty, poll for task
# and then print command and result
while not wq.empty():
    task = wq.wait(1)
    if task:
        print task.command, task.result
```

Listing 5. Python-WorkQueue Distributed Convert Example.

The main processing occurs in the `while` loop near the end of Listing 5. In this loop, we first check if the workqueue is empty. If it is not, we then trigger processing the workqueue and waiting for a task by calling the `wait` method of the *WorkQueue* object with a timeout of one second. When this method completes it will return a completed *Task* object if one is available, otherwise, it will return `None`. Once we receive a valid *Task* object, we print out the command and its result. Remember that whenever a task is returned, all of its specified output files will have been transferred already, so the user does not need to request the files manually.

As shown in the code listings in this section, the Python-WorkQueue module is a relatively simple and straightforward package for developing master/worker type distributed applications.

## IV. Applications

To demonstrate Python-WorkQueue's flexibility and power in composing scalable scientific ensemble applications, we present two active research systems used at the University of Notre Dame, RepExWQ and Folding@work. These two applications were built using Python-WorkQueue and manifest the framework's ability to scale workflows across multiple distributed systems and up to over a thousand of workers.

### A. RepExWQ

Several techniques have been proposed to sample the conformational space of biomolecules such as proteins, which is a difficult problem due to the high dimensionality of the search

space. A well-known and common technique employed in such studies is replica exchange [11], [12]. As a result, replica exchange has been built into several molecular dynamics simulation software such as ProtoMol [13], NAMD [14], CHARMM [15], and GROMACS [16].

Using this technique, simulations are run by creating replicas of a protein molecule and executing each in parallel over several Monte Carlo steps at different temperatures. At the end of every Monte Carlo step, an exchange is performed between neighboring replicas if a Metropolis Monte Carlo criterion is met. In this exchange, the temperature of the replicas are swapped and the simulations are continued in the same manner until the required number of Monte Carlo steps is reached.

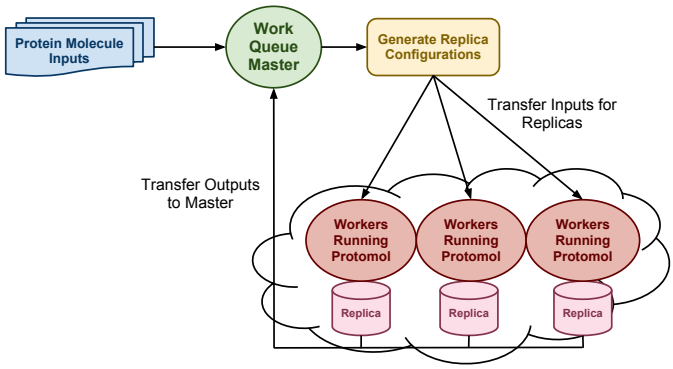


Fig. 4. Implementation of Replica Exchange using the Work Queue workflow.

Traditionally, replica exchange is implemented in parallel using MPI, which performs well when all the resources are dedicated but not when resources may be intermittently available, as is more common. Here, we describe the results of converting this application to an elastic application using Work Queue.

The implementation of replica exchange using the Work Queue framework involves a master script built in python called RepExWQ. This master script uses the Python-WorkQueue library to setup a workqueue and submit tasks corresponding to the simulations of each replica to run on the remote workers. The master achieves this by defining the configuration for each replica's simulation at the current Monte Carlo step and submitting these simulations as tasks. The configuration along with other required input are specifed as input files, and the data created at the end of simulation is specified as output files as illustrated in Listing 4. The task specification corresponding to the simulation of each replica's step consists of its input files, its output files created at the end of simulation, and the command arguments to run the simulation program executable. The simulation program used here, ProtoMol [13], is also specified as an input file so that it is transferred and cached at the workers. Similarly, any libraries and dependencies required for the execution of the simulation program can also be specified as input files in the master script.

After the tasks are submitted, the master waits for the workers to complete execution of the tasks and return the specified output files. When all scheduled tasks have finished execution, the master checks to see if a replica exchange can be attempted between two neighboring replicas. At the satisfaction of certain criteria, the temperature of the two replicas are swapped and updated. The configuration and parameters for the next step are created and the process is repeated. The master, therefore, breaks down the simulation into smaller work units (tasks) that are run at distributed workers, and processes the results of each work unit to produce the final output of the entire simulation run. Figure 4 is an illustration of this implementation of replica exchange.

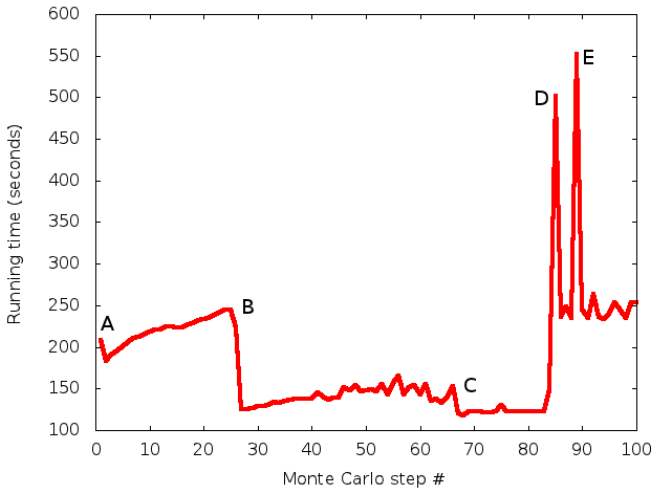


Fig. 5. Running time of Monte Carlo steps run over 400 replicas in RepExWQ with workers running on multiple cloud platforms.

| Event | Description | Total workers at end of event |
|---|---|---|
| A | Start of experiment with 100 workers in ND SGE | 100 |
| B | Addition of 150 workers in Condor | 250 |
| C | Addition of 110 workers in Condor and 40 workers in Amazon EC2 | 400 |
| D | Removal of 100 workers in ND SGE | 150 |
| E | Removal of 125 workers in Condor and 25 workers in Amazon EC2 | 250 |

TABLE I
DESCRIPTION OF THE EVENTS SHOWN IN FIGURE 5.

We proceed to experimentally study this Work Queue implementation of replica exchange. In this experiment, we run a replica exchange simulation with 400 replicas over 100 Monte Carlo steps. Figure 5 plots the time to complete the simulation of a Monte Carlo step over all replicas. The master script was run from a workstation inside the Notre Dame campus network. We note that the master can be run from any site including cloud platforms, but we chose this setup to illustrate ease of building and deploying the master.

We utilize workers on three different platforms: our 4000-core campus cluster managed by SGE, our 1200-core campus grid managed by Condor, and the Amazon EC2 service. Over the course of the experiment, the computing resources varied dynamically as compute nodes were requested, allocated, and terminated. The specific instances at which the available resources changed is labeled in the figure and described in Table I.

We make the following observations from Figure 5. The Work Queue implementation of replica exchange is elastic and dynamically adapts to resource availability. We observe this at each of the described events in Table I. At Events B and C, the addition of workers to the existing pool results in the running time of the Monte Carlo steps being lowered. At Events D and E, the termination and removal of workers only leads to a brief spike in the running time without stalling the simulation run. This timeline also shows that the Work Queue implementation is fault tolerant and recovers from failures. We observe this at Events D and E, where workers were removed while they were executing tasks corresponding to the Monte Carlo simulations of the replicas. This resulted in the failure of tasks that were being executed by the removed workers. Work Queue dynamically rescheduled these failed tasks on the remaining workers, as described in Figure 2 for handling failures resulting from tasks being killed or terminated. We attribute the spike in the running time following Events D and E to the failed work units being rerun on the remaining workers as they finish execution of their assigned tasks.

Finally, we make the observation that the simulations successfully completed with individual tasks executing simultaneously on different distributed systems. The Work Queue implementation is oblivious to the underlying execution platform and environment and thus is able to leverage multiple distributed computing platforms at the same time as part of a virtual cloud of workers. This ability to utilize multiple distributed environments is especially useful when the user is constrained by local resources or administrative policies and requires the utilization of additional computational services to perform their experiment.

### B. Folding@work

Molecular dynamics (MD) attempts to capture the behavior of biological systems at atomic resolution by solving Newton's equations of motion at small timesteps – typically at the femtosecond ($10^{-15}$s) scale. Certain biologically important mechanisms, such as protein folding, occur in microsecond ($10^{-6}$s) to second timescales. Even the most powerful supercomputers have difficulty capturing the relevant motions of large systems.

Previous work has shown that several parallel simulation trajectories can be used to approximate long trajectories [17], [18], [19]. This exposes an interface for parallelism that is exploited by the highly popular Folding@home project [17], allowing commodity hardware to capture protein folding using the code bases of existing MD software such as GROMACS [16].

Folding@work (F@w) aims to bring heterogeneous resources such as Condor, batch systems such as the Sun Grid Engine available on clusters, or the cloud such as Amazon EC2 together. The goal is to allow researchers to declaratively describe their molecular dynamics simulations and allow F@w to manage the complexity of dispatching tasks to resources and handling the failures.

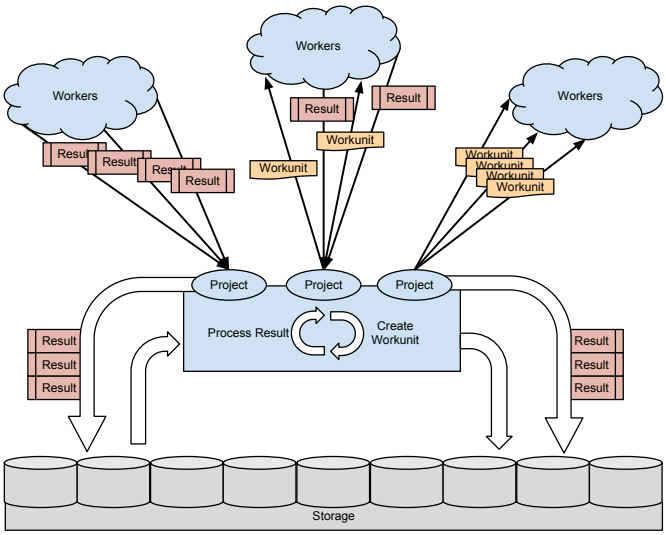


Fig. 6. Folding@work using the Work Queue Workflow.

Figure 6 illustrates the architecture of the current version of F@w. The user defines a project in terms of scientific goals by describing the experimental conditions of interest. These conditions may require different conformations, different molecules, different temperatures, and so forth. The project is then translated into task specifications called *workunits*. A workunit defines the input and output files, the executables, and the environmental setup. These workunits are then assigned to a worker using the Python-WorkQueue library. As workunits complete, the output files are returned to the project, which then processes the results to create subsequent workunits.

We ran F@w for one month using the Work Queue framework and ProtoMol [13] to capture the folding of a 32 residue mutant of the WW domain. The goal was to investigate the effects of simulation parameters using long-timestep molecular dynamics [20] as compared to the Langevin Leapfrog integrator. Five experimental conditions were run in 1,000 replicates using different initial velocities with a timestep of 50 fs.

The results of the F@w project running for a month are shown in Table II. Using five Work Queue master nodes with each node responsible for 1,000 tasks at a time, we were able to accumulate over 300 $\mu$s of aggregate simulation time using over 280,000 tasks. For reference, it would require 3,000 days for a single continuous simulation to access 300 $\mu$s. The execution time of each task averaged around 2 hours. Though we did see a large variation in execution time, this is likely due to the heterogeneity of the hardware. During

| | |
|---|---|
| Tasks assigned | 283830 |
| Results received | 122141 |
| Aggregate data gathered | 305 $\mu$s |
| Execution time average (min) | 125 |
| Execution time std. dev. (min) | 87 |
| Number of Workers | 5000 |
| Number of unique machines | 370 |

TABLE II
FOLDING@WORK DATA GATHERED OVER ONE MONTH.

this time, jobs were running on three distinct campus grids: the University of Notre Dame, Purdue University, and the University of Wisconsin-Madison. By utilizing multiple Work Queue masters in conjunction with computing resources from multiple distributed systems from three different campus grids, we were able scale up to 5,000 workers and accumulate a considerable amount of experimental data.

Using Folding@work, we were able to overcome resource availability limits, which are around 100 jobs in the SGE scheduler at Notre Dame. Even though in theory we could have done this using Condor, the length of our simulations that run continuously for several weeks would have made getting results improbable. By using Work Queue to partition each long running simulation into small chunks and managing all the complexity for the user, we have made this both practical and easy to use.

## V. Related Work

Although Work Queue was originally intended to serve as a light-weight master/worker glide-in [21] system for traditional computational grids such as Condor [10] and SGE, it quickly outgrew this limited role and developed additional features as described in this paper. In this context, Work Queue is similar to systems such as Falkon [22], in that it provides fast, low-latency, task execution on clusters and computational grids. Work Queue differs from Falkon, however, in that it focuses on robust scalability and thus allows for use across multiple distributed systems simultaneously, while Falkon focus on performance and efficiency on single highly parallel machines.

Because of Python's user friendliness and power, it has attracted a serious following in the scientific community. As such, there have been multiple projects focused on easing and simplifying the development of distributed scientific applications while using Python. Examples of such Python packages include MPI4Py [6], the Modular toolkit for Data Processing (MDP) [23], PyMW [5], Scientific.BSP [8], and Scientific.DistributedComputing.MasterSlave [24]. Like the PyMW and Scientific.DistributedComputing.MasterSlave packages, Python-WorkQueue utilizes a master/worker programming model, which permits an arbitrary number of worker processes. This flexibility allows these systems to be very fault tolerant and to scale dynamically. Unlike these packages, however, Python-WorkQueue focuses primarily on applications that consist of orchestrating ensembles of scientific applications, rather than producing a single monolithic distributed application.

Another related work is PyDoop [7], which is a Python library for creating Map-Reduce [25] applications that run on Hadoop [26]. In this case, the resulting distributed application relies on a sophisticated distributed system and is constrained by a limiting programming model. Python-WorkQueue does not depend on any particular distributed environment and thus is more portable. Moreover, because it uses the master/worker paradigm, its programming model is more general than Map-Reduce and allows for different types of workflows that would be inefficient or impractical on Hadoop.

As noted in the introduction, Work Queue has been mentioned briefly in some of our previous work [1], [2], [3]. These papers, however, focused on the applications presented in those works and did not describe the Work Queue framework in detail as this paper does. Moreover, this paper presents for the first time our Python-WorkQueue module and provides examples of Work Queue applications written in Python that scale across multiple distributed systems simultaneously and up to over a thousand workers.

## VI. Future Work

The Python-WorkQueue module is a part of the standard CCTools software distribution maintained by the Cooperative Computing Lab at the University of Notre Dame. The software collection is released under the GNU General Public License (GPLv2) and can be downloaded at http://cse.nd.edu/~ccl/software/. Work is currently underway to extend some of the other components in the CCTools software collection with new Python bindings and to build various utilities using the new modules. From our experience with Python-WorkQueue, making our existing software available for use in a user friendly language such as Python enables wider adoption of our tools. This is especially relevant when collaborating with computational scientists who may be familiar with scripting languages but not with distributed programming or our lower-level libraries.

In addition to this, due to Python-WorkQueue's ease of use and relative simplicity, we are considering methods of utilizing the module in the Programming Paradigms course at the University of Notre Dame. Thus far, we have presented lectures on Work Queue and have had students run example applications, but have not required the students to build applications using the module by themselves. We believe that Python-WorkQueue would be an effective way of introducing students to distributed computing and plan on developing educational course material around our software.

Regarding the Work Queue framework, we are currently looking at a few new avenues of research. One possibility is to extend Work Queue to support the more general fork/join [27] programming model. This would allow for the construction of hierarchical workflows with multiple levels of sub-masters, rather than a single master. Likewise, we are also investigating methods of propagating and managing resource constraints, which is something we do not currently handle as part of the Work Queue framework. This would mean a few possibilities such as throttling bandwidth, limiting the number of cores

utilized, or reserving a certain amount of memory for the application. Whatever additional features or capabilities we implement in the future, we will ensure that the Python-WorkQueue module has access to them.

## VII. Conclusions

Work Queue is a flexible and powerful framework for constructing scalable scientific ensemble applications. It provides attractive features such as fault-tolerance, data management, multiple scheduling algorithms, fast abort, and support for multiple distributed systems. With the introduction of the Python-WorkQueue module, this functionality is now available to research scientists in a user-friendly programming language.

In this paper, we present an overview of the Python-WorkQueue module and then examine two real world scientific applications that were built using Python-WorkQueue. The first is RepExWQ, which demonstrates the ability of Work Queue applications to span multiple distributed environments such as a local campus cluster, a private grid, and a public cloud service provider. The second application is the Folding@work data processing pipeline, which manifests Work Queue's ability scale to over a thousand concurrent workers spread across different distributed systems. These applications are evidence of Work Queue's effectiveness in constructing scientific ensemble applications that scale across multiple distributed systems.

## VIII. Acknowledgments

We gratefully acknowledge the support of the U.S. Department of Education through a GAANN Fellowship for Peter Bui (award P200A090044). This work was also supported by the National Science Foundation under grants NSF-CNS-0643229, NSF-CNS-0855047, and NSF-CCF-1018570.

## References

[1] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain, "Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions," *Journal of Cluster Computing*, vol. 13, no. 3, pp. 243–256, 2010.

[2] C. Moretti, M. Olson, S. Emrich, and D. Thain, "Highly Scalable Genome Assembly on Campus Grids," in *Many-Task Computing on Grids and Supercomputers (MTAGS)*, 2009.

[3] P. Bui, L. Yu, and D. Thain, "Weaver: Integrating distributed computing abstractions into scientific workflows using Python," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 636–643.

[4] Python Programming Language, http://www.python.org/, 2010. [Online]. Available: {http://www.python.org/}

[5] E. M. Heien, Y. Takata, K. Hagihara, and A. Kornafeld, "PyMW - A Python module for desktop grid and volunteer computing," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–7, 2009.

[6] L. Dalcin, R. Paz, M. Storti, and J. Delia, "MPI for Python: Performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655–662, May 2008.

[7] S. Leo and G. Zanetti, "Pydoop: a Python MapReduce and HDFS API for Hadoop," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 819–825.

[8] K. Hinsen, "High-Level Parallel Software Development with Python and BSP," *PARALLEL PROCESSING LETTERS*, vol. 13, p. 2003, 2003.

[9] M. P. Sullivan and D. P. Anderson, "Marionette: a System for Parallel Distributed Programming Using a Master/Slave Model," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-88-460, Nov 1988. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/5728.html

[10] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, A. Hey, and G. Fox, Eds. John Wiley, 2003.

[11] Y. Sugita and Y. Okamoto, "Replica-exchange molecular dynamics method for protein folding," *Chemical Physics Letters*, vol. 314, no. 1-2, pp. 141 – 151, 1999.

[12] P. Brenner, C. R. Sweet, D. VonHandorf, and J. A. Izaguirre, "Accelerating the replica exchange method through an efficient all-pairs exchange." *J. Chem. Phys.*, vol. 126, no. 7, p. 074103, Feb. 2007.

[13] T. Matthey, T. Cickovski, S. Hampton, A. Ko, Q. Ma, M. Nyerges, T. Raeder, T. Slabach, and J. A. Izaguirre, "Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics," *ACM Transactions on Mathematical Software*, vol. 30, no. 3, pp. 237–265, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1024074.1024075

[14] J. Phillips, G. Zheng, S. Kumar, and L. Kale, "Namd: Biomolecular simulation on thousands of processors," in *Supercomputing, ACM/IEEE 2002 Conference*, November 2002, p. 36.

[15] B. Brooks, R. Bruccoleri, D. Olafson, D. States, S. Swaminathan, and M. Karplus, "Charmm: A program for macromolecular energy, minimization, and dynamics calculations," *Journal of Computational Chemistry*, vol. 4, pp. 187–217, 1983.

[16] "Gromacs: A message-passing parallel molecular dynamics implementation," *Computer Physics Communications*, vol. 91, no. 1-3, pp. 43 – 56, 1995.

[17] M. Shirts and V. S. Pande, "COMPUTING: Screen Savers of the World Unite!" *Science (New York, N.Y.)*, vol. 290, no. 5498, pp. 1903–4, Dec. 2000. [Online]. Available: http://www.sciencemag.org/content/290/5498/1903.shorthttp://www.ncbi.nlm.nih.gov/pubmed/17742054

[18] M. R. Shirts and V. S. Pande, "Mathematical analysis of coupled parallel simulations." *Physical review letters*, vol. 86, no. 22, pp. 4983–7, May 2001. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevLett.86.4983http://www.ncbi.nlm.nih.gov/pubmed/11384401

[19] C. D. Snow, H. Nguyen, V. S. Pande, and M. Gruebele, "Absolute comparison of simulated and experimental protein-folding dynamics." *Nature*, vol. 420, no. 6911, pp. 102–6, Nov. 2002. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/12422224

[20] J. A. Izaguirre, C. R. Sweet, and V. S. Pande, "Multiscale dynamics of macromolecules using normal mode langevin." *Pacific Symposium On Biocomputing*, vol. 251, pp. 240–251, 2010. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/19908376

[21] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth, "An enabling framework for master-worker applications on the computational grid," *High-Performance Distributed Computing, International Symposium on*, vol. 0, p. 43, 2000.

[22] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight task execution framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 43:1–43:12.

[23] T. Zito, N. Wilbert, L. Wiskott, and P. Berkes, "Modular toolkit for Data Processing (MDP): a Python data processing framework," *Frontiers in Neuroinformatics*, vol. 2, no. 0, 2009.

[24] Scientific Python, http://dirac.cnrs-orleans.fr/plone/software/scientificpython/, 2011. [Online]. Available: {http://dirac.cnrs-orleans.fr/plone/software/scientificpython/}

[25] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Operating Systems Design and Implementation*, 2004.

[26] Hadoop, http://hadoop.apache.org/, 2007. [Online]. Available: {http://hadoop.apache.org/}

[27] F. Baccelli, W. A. Massey, and D. Towsley, "Acyclic fork-join queuing networks," *J. ACM*, vol. 36, pp. 615–642, July 1989.