PyMW - a Python Module for Desktop Grid and Volunteer Computing

Eric M. Heien, Yusuke Takata, Kenichi Hagihara
Graduate School of Information Science and Technology
Osaka University, Toyonaka, Osaka 560-8531, Japan
{e-heien, y-takata, hagihara}@ist.osaka-u.ac.jp

Adam Kornafeld
Laboratory of Parallel and Distributed Systems
Computer and Automation Research Institute
Hungarian Academy of Sciences
H-1132 Victor Hugo u. 18-22, Budapest, Hungary
kadam@sztaki.hu

***Abstract*—We describe a general purpose master-worker parallel computation Python module called PyMW. PyMW is intended to support rapid development, testing and deployment of large scale master-worker style computations on a desktop grid or volunteer computing environment. This module targets non-expert computer users by hiding complicated task submission and result retrieval procedures behind a simple interface. PyMW also provides a unified interface to multiple computing environments with easy extension to support additional environments. In this paper, we describe the internal structure and external interface to the PyMW module and its support for the Condor computing environment and the Berkeley Open Infrastructure for Network Computing (BOINC) platform. We demonstrate the effectiveness and scalability of PyMW by performing master-worker style computations on a desktop grid using Condor and a BOINC volunteer computing project.**

I. Introduction

In recent years, desktop grid and volunteer computing research has increased thanks to the success of large scale projects such as SETI@home [1] and Folding@home [2]. However, despite the success of these projects desktop grid (DG) and volunteer computing (VC) remain uncommon in most institutions when compared to traditional clusters. For example, there are currently less than 50 active public VC projects running BOINC [3], whereas there are well over 1000 clusters using the ROCKS cluster software [4]. This is partly due to the complexity of starting and maintaining a DG or VC environment, especially for users without computer expertise. We developed PyMW to help increase the use of desktop grid and volunteer computing by providing a simple Python interface targeting non-computer specialists such as scientists and researchers.

Python, Ruby, PHP and other such interpreted languages are becoming increasingly popular as program development time dominates relative to execution time. These languages allow fast development combined with runtime error checking, libraries for computationally intensive functions and platform independence. Software packages are available for both small and large scale parallel computation in Python. However, there are few packages that support desktop grids, and to our knowledge none that support volunteer computing.

Packages for multicore shared memory processors include Parallel Python [5] and seppo (simple embarrassingly parallel python). Rudimentary Python cluster computing is supported in Parallel Python [5], while MPI implementations for Python include pyMPI [6] and mpi4py [7]. For master-worker style computing in Grids, the Condor [8] and Globus [9] software packages are commonly used, though no Python interface exists for Condor. The IBIS project [10] offers software oriented towards multiplatform Grid computing in Java that allows sophisticated interprocess communication. For large scale computations on volunteer computing platforms, two common software platforms are BOINC [11] and the Cosm platform. However, neither of these uses an interpreted language, and often require customized programs for submitting tasks and collecting results.

We developed PyMW in response to the need for better Python based DG and VC tools, in hopes of further opening these computing environments to scientists and researchers. PyMW is a Python module designed for master-worker style computations on a wide variety of parallel computing platforms. Master-worker parallel computation involves a master process which sends computational tasks to worker processes. These processes often run on separate machines such as in a cluster or Grid. The worker processes perform their assigned tasks and return the results to the master. This is repeated until all tasks are complete. Examples of common master-worker style computations include parameter sweeps, Monte Carlo simulations, genetic algorithms, and other work which is easily divisible with little or no dependencies between pieces. An older example of such a system is Marionette [12], which was designed for heterogeneous networks of workstations (NOW). Similar to PyMW, Marionette aimed to provide a simple interface for master-worker computing, though as a C library rather than a Python module. However, Marionette was intended only for use on NOWs, and was not suitable for use on large scale platforms nor for multiprocessor machines.

PyMW aims to provide functionality for master-worker computation in a Python module that can be executed in a wide variety of computing environments. In particular, PyMW targets desktop grid and volunteer computing systems performing large numbers of tasks. The module is designed to be as simple and flexible as possible, with the goal of providing an intuitive interface for users.

The remainder of the paper is organized as follows. In

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

Section II we describe the PyMW module internal structure, including the functionality it provides and how programs interact with it. Section III describes platform interfaces for Condor and BOINC and how interface implementations for a given platform should interact with PyMW. We show the effectiveness of PyMW by running some master-worker style parallel programs in Section IV on a desktop grid and volunteer computing platform. Finally we offer our conclusions in Section V.

II. THE PYMW MODULE

In this section we describe the organization and usage of the PyMW module. This includes descriptions of the master API used by a users program in Section II-A and the interface to different computing platforms in Section II-B.

Rather than designing custom solutions from scratch for each computing environment, the philosophy of PyMW is to utilize existing software as much as possible. Customized solutions require extra work to implement features already used in many computing environments such as load balancing, checkpointing, computational redundancy, etc. For example, the Condor [8] computing environment provides tools to distribute and execute tasks on a desktop grid as well as checkpoint and move tasks between machines. BOINC (Berkeley Open Infrastructure for Network Computing) [11] provides a framework for packaging and distributing tasks in a volunteer computing environment, automatically handling task failures and redundant computation. PyMW uses these software packages to execute tasks on the underlying hardware.

In order to use existing software packages and allow extension to multiple computing environments, PyMW is divided into abstraction layers as shown in Figure 1. The first layer is the user program which loads the PyMW module and calls PyMW to perform master-worker computations. The second layer is the "PyMW layer" and contains functions for the user to submit tasks and retrieve the results of task execution. This layer also manages task time accounting, error handling and other platform independent functionality. The PyMW layer is fully described in Section II-A.

The third layer is called the "interface layer". This layer manages the interaction between the PyMW layer and the underlying software/hardware. For example, with a desktop grid running Condor this layer is responsible for formatting and submitting tasks then notifying PyMW on their completion. Depending on what platform the user wishes to use, they will select a different implementation of the interface layer. The functionality of this layer may vary depending on the underlying hardware. The interface layer is described in detail in Section II-B.

PyMW currently offers interfaces for four types of systems - multicore/multiprocessor computers, clusters running MPI, desktop grids running Condor and volunteer computing systems running BOINC. Users may change interfaces by altering one line in their program code, allowing them to run a single program in different parallel environments. This enables program development on a multicore machine, testing on a cluster, and finally deployment on a Grid or volunteer computing platform. Combining multiple interfaces allows users to employ different platforms in a single program, for example, performing a parameter sweep on a grid then analyzing the results on a multicore machine.

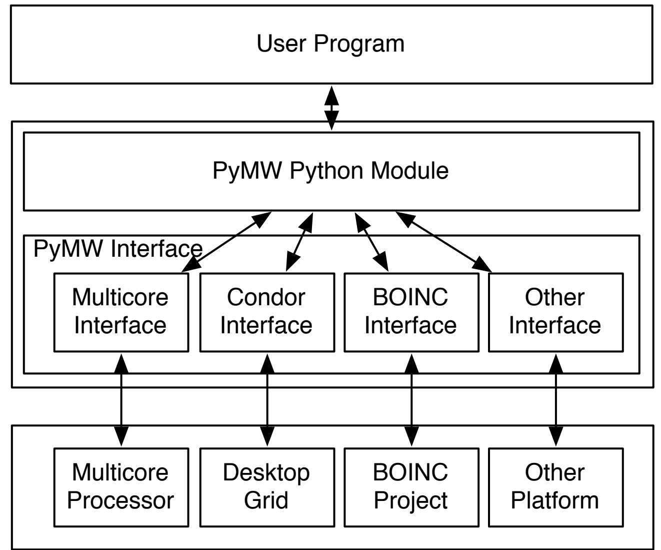


Fig. 1. Abstraction layers of PyMW.

A. PyMW Layer

The abstraction layer of PyMW with which the user most directly interacts is the PyMW layer. This layer is accessed through a PyMW_Master object instantiated by the user. The interface used by the PyMW_Master object is specified at creation. The master object accepts tasks from the user and sends them to the interface. This is done without blocking, allowing the user program to continue and possibly submit more tasks. Users later retrieve the results through the master object. The PyMW layer provides functions for executing tasks, retrieving the result of finished tasks and checking the status of the system. The functions provided by this layer are independent of the underlying interface such that a user may write a single program that will run on any platform.

PyMW was designed with two users in mind: the program developer and the interface developer. PyMW exposes only four functions to the user to allow quick and easy program development. Users may directly interact with the underlying interface for greater control, but the goal of PyMW is to allow development with only the four functions listed below.

```
master = PyMW_Master(interface=None)
```

This function creates a new PyMW master object associated with a specified interface. An interface of `None` indicates the default multicore interface. Otherwise, this is an instantiation of an interface described in Section II-B. Tasks are submitted and results are retrieved using this master object. Master objects are independent, so a given program can use multiple platforms if desired.

```
task = master.submit_task(executable,
        input_data=None,
        modules=(), dep_funcs=())
```

The `submit_task` function creates a task, puts it on a queue for later scheduling and immediately returns an object representing the submitted task. The executable must be a Python script or function. The `input_data` argument may be a Python tuple object, or `None` if the program requires no input. This will be passed as an set of arguments to the executable. If `executable` is a function, additional dependencies such as modules or functions are specified by the `modules` and `dep_funcs` arguments.

```
task, result = master.get_result(task=None,
        blocking=True)
```

The `get_result` function returns a completed task and its associated result. If `task` is `None`, this will return the next completed task, or an arbitrary task if there are multiple completed tasks. If `task` is not `None`, this will return the result of the specified task. If `blocking` is `True`, `get_result` will wait until the task has completed before returning. If `blocking` is not `True` and there are no completed tasks, `get_result` will return None. If there was an error performing the associated task, calling `get_result` will raise a Python exception describing the problem. Passing an object that has not been previously submitted with `submit_task` will cause an exception.

```
status_dict = master.get_status()
```

The `get_status` function returns a Python dictionary with keys specifying the current status of the master and its associated interface. The status of an interface will vary depending on the interface implementation. The keys for master status include a list of task objects that have been submitted with `submit_task`.

The internal organization of PyMW is shown in Figure 2, with the flow of tasks in the system represented by arrows. First, the call to `submit_task` creates a task object, which is put on a run queue and added to the submitted task list. The scheduler thread removes tasks from the run queue and attempts to match them with a worker from the interface function `reserve_worker`. After finding a suitable worker, the scheduler thread calls `execute_task` in the interface to execute the task. Once the task is completed, the interface is responsible for calling `task_finished` on the task object. This causes PyMW to parse the resulting output data into a Python object, handle any processing errors, perform other accounting functions and mark the task as complete. The results of the task are returned to the user through the `get_result` function.

B. Interface Layer

The PyMW layer described in Section II-A interacts with an underlying desktop grid, volunteer computing project or other platform through an interface layer. The interface layer

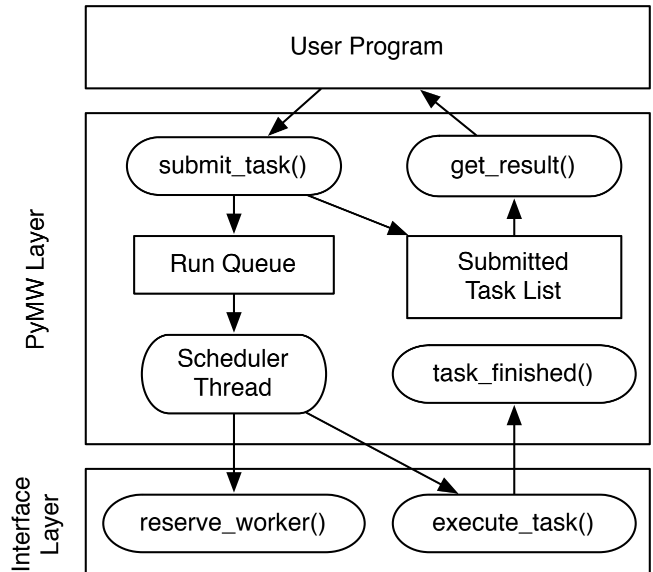


Fig. 2. The internal organization of PyMW and a generic interface. The arrows represent the flow of tasks through PyMW.

implementation will vary depending on the platform it supports. This section only describes functionality common to all interface layer implementations. To properly interact with the PyMW layer, an interface implementation must expose certain functions to the PyMW layer. To simplify development of new interfaces, we tried to make these functions as simple as possible yet provide flexibility in accommodating different interface characteristics. The required interface functions are described below.

```
interface.execute_task(task, worker)
```

At minimum, all interface implementations must provide the `execute_task` function. This function receives a task object representing the task to be executed, and an interface specific object representing the worker to execute the task on. If the worker object is `None` then any worker may be used. The `execute_task` function is called by the PyMW scheduler in a separate thread and therefore need not return immediately. This allows multiple tasks to be performed simultaneously. Upon completion of the task, the interface must call the `task_finished` function. If an interface error occurred the interface layer must pass a Python Exception object to `task_finished` describing the error, for example, failing to call the grid task submission program. For interfaces that expect many long running tasks such as with Condor and BOINC, active thread limits may cause problems. In this case, it is best to exit `execute_task` immediately and use a single separate thread to periodically check through all tasks and report any that have completed.

```
worker = interface.reserve_worker(task)
```

This function returns an interface-specific object representing the worker to use when executing the task. The

`reserve_worker` function may block if a worker is not available for the task, for example, if the worker is being used for another task. If the interface does not implement this function, the worker object is set to `None`. Future versions of PyMW will allow users to implement `reserve_worker` functionality specific to their program, such as specifying minimum memory or disk space requirements.

```
status_dict = get_status()
```

This function returns a dictionary of keys with information specific to the underlying platform. If not implemented, the dictionary is assumed to be empty.

```
pymw_write_location(selfobj, my_obj, loc)
my_obj = pymw_read_location(selfobj, loc)
```

These functions allow interface-specific data reading and writing functionality. For example, input and output for tasks on a multicore machine can be done quickly using stdin and stdout, whereas a task executed on BOINC requires these to be provided as files. For each function, `loc` represents an interface specific location and `selfobj` is used to determine whether the function is being called in the task or in PyMW. When implemented in an interface, `pymw_write_location` should store the `output` object in the specified location, possibly in a pickled format. `pymw_read_location` reads the data at this location, and returns the corresponding Python object. If these functions are undefined, input/output is read/written from files.

III. PyMW Interfaces

As described in Section II-B, the interface layer of PyMW has different implementations depending on the underlying platform. PyMW currently offers four interfaces: multicore processors, networked clusters running MPI, desktop grids using Condor and volunteer computing systems running BOINC. Interfaces for other platforms can be added following the guidelines in Section II-B. In this section, we describe the interface implementation for two platforms - desktop grid systems containing tens or hundreds of networked machines running Condor (Section III-A) and global scale systems using BOINC running on thousands or millions of machines (Section III-B). Details of the multicore and cluster interfaces are available in [13].

A. Condor

Condor is a workload management system aimed at performing compute-intensive jobs on clusters [14] and desktop grids [15] or a mix of the two. Users install the Condor application on machines they wish to make available for computation, then submit tasks to the system.

When submitting a task via `condor_submit`, it is necessary to create a description file. This is automatically handled by the interface, which specifies the proper data input and output locations and error/log files. Furthermore, the interface ensures that Python is copied to remote machines. This allows tasks to execute on machines without a shared filesystem or

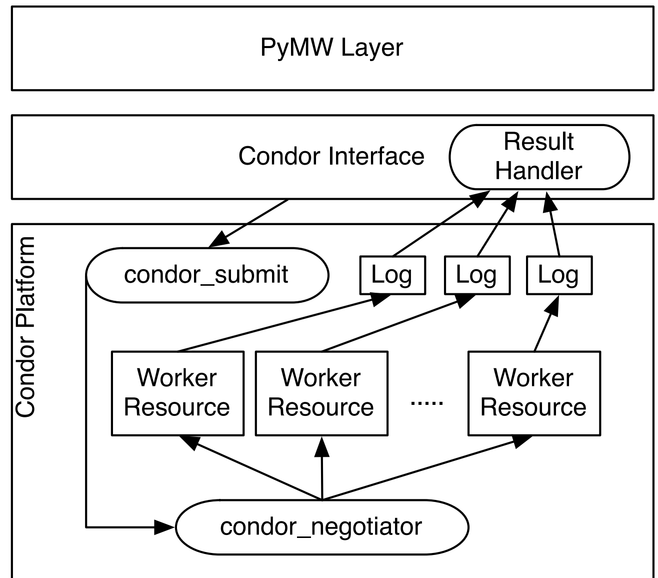


Fig. 3. Condor Interface and task flow.

a Python installation. Tasks submitted to `condor_submit` are then passed to the `condor_negotiator` which matches them to an appropriate resource. The current version of PyMW matches a task with any available resource, but future versions will allow user specified resource requirements.

The input (`pymw_write_location`) and output (`pymw_read_location`) functions for the Condor interface read or write to files when called from PyMW. This is because input and output is required to be in file format when submitting tasks through `condor_submit`. However, when executing the tasks these functions read/write standard input and output. This demonstrates how PyMW allows the user program to function in multiple environments with varying input/output methods.

After the task is completed on a worker, Condor updates a task specific log file. Because there may be hundreds or thousands of simultaneously executing tasks, result checking is performed by a single separate thread. The result handler thread periodically checks the log files and notifies completed tasks. The log and error files are removed after task completion, and any Condor errors are relayed to the user.

B. BOINC

PyMW was originally conceived to support BOINC application development by providing a simple Python interface for the BOINC platform. As previously discussed, a barrier to common usage of the BOINC platform and other volunteer computing environments is the preparation required to run an application. In BOINC, multiple programs must be modified by the developer, including a work generator, server, result validator and a result assimilator. The goal of this interface is to automatically manage these, so developers can focus on the

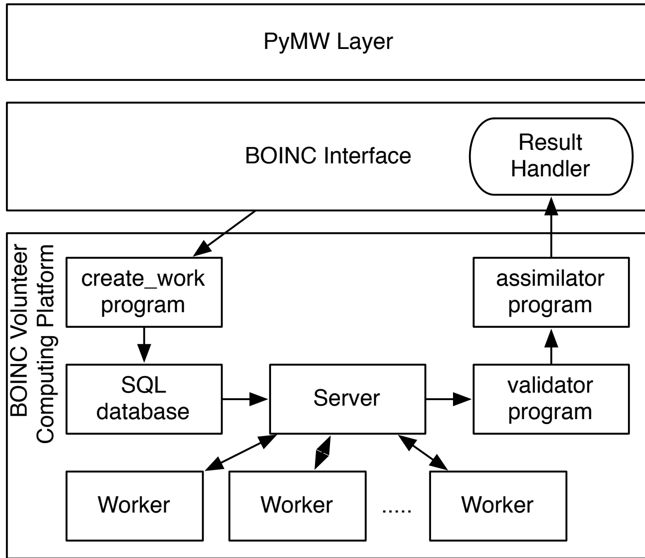


Fig. 4. BOINC Interface and task flow.

application. The BOINC interface of PyMW is accompanied with a setup script that configures a BOINC project to handle PyMW applications. This script creates and registers a special BOINC worker application for PyMW, and installs BOINC components to interact with PyMW. The result is that users can write programs with PyMW that will execute transparently in the BOINC environment.

The flow of tasks through the PyMW BOINC interface and BOINC platform is shown in Figure III-B. The user submits tasks to PyMW using the `submit_task` function. The BOINC interface uses the BOINC create_work program to generate tasks. This involves copying the input data to a BOINC specified location, creating input and output template files and executing the create_work program to insert the task into the database. Later, the server reads the task from the database and distributes the program and data to the workers.

The workers execute a copy of the BOINC worker, which periodically contacts the server and requests tasks. The server sends tasks to the workers, which download the program and input data from the appropriate location. Because many computers do not have Python installed, the program is a Python interpreter and the input data is the user program and task data. The interpreter executes the user program with the data file and takes care of initialization and cleanup. The worker then uploads the result to a specified location and notifies the BOINC server.

To ensure result correctness, some projects execute the same program on different workers and compare the results using a validator program. A validator for PyMW is automatically installed by the setup script mentioned above to handle the validation of PyMW tasks. After successful task validation PyMW must be notified of the available results. The PyMW setup script installs an assimilator component for BOINC, which copies result output files of PyMW applications to the PyMW working directory. The BOINC interface spawns a result handler thread during initialization, which periodically checks the PyMW working directory for new result output files. When an output file appears, the corresponding task object is notified through `task_finished`.

IV. Experiments

To test the effectiveness of PyMW in performing master-worker computations, we wrote two Python programs using the PyMW module. These programs represent example applications for PyMW - a Monte Carlo style simulation and a test for prime numbers. These programs were run on two platforms - a small desktop grid running Condor and a BOINC project. Except for changes to select the interface, the programs were identical across all platforms. Each program was executed as a Python script with no special environment variables or optimizations. We describe the programs and experimental setup in Section IV-A and the results in Section IV-B.

A. Experiment Setup

The first program is a simple Monte Carlo program [16] which estimates the value of pi by randomly selecting points in a unit square. For n selected points with m of the points satisfying $x^2 + y^2 \leq 1$, the value of pi is estimated as $4m/n$. This uses the master-worker model by giving p workers each a task to test n/p points and return m. For the Condor interface 10^8 points were tested and for the BOINC interface 10^9 points. The total program size is 70 lines of Python code. Listing 1 shows a version the Monte Carlo PyMW code for the Condor interface. This program performs master-worker style computation by submitting a set of tasks to the master, then retrieving the results as they are completed. As shown in this example, PyMW can execute complex programs in parallel on arbitrary platforms with minimal additional code.

The second program finds prime numbers of the form n^2+1 for integers in the range $[1, n]$. The input to the worker program is the range of integers $[i, j]$ to search. This program uses the Miller-Rabin test [17] (k=50) to probabilistically guarantee the primality of each number. A value of $k = 50$ means each integer is checked at most 50 times by the Miller-Rabin test and that any number stated to be prime by the test has a probability no greater than 4^{-50} of actually being composite. For both interfaces, the range [1, 100000] was tested. The total program size is 90 lines of Python code.

For each platform, we performed experiments to determine the effect of number of workers on total run time. The experiments for the Condor interface were run on a desktop grid containing a total of 12 cores running around 2.8 GHz each with Windows XP and Vista. The BOINC experiments were run on a BOINC project with a dozen workers of varying hardware running Linux and Windows.

B. Experiment Results

The results for the Condor platform are shown in Figure 5. These figures show the total execution time and per-task

Listing 1. Monte Carlo PyMW code using the Condor interface.

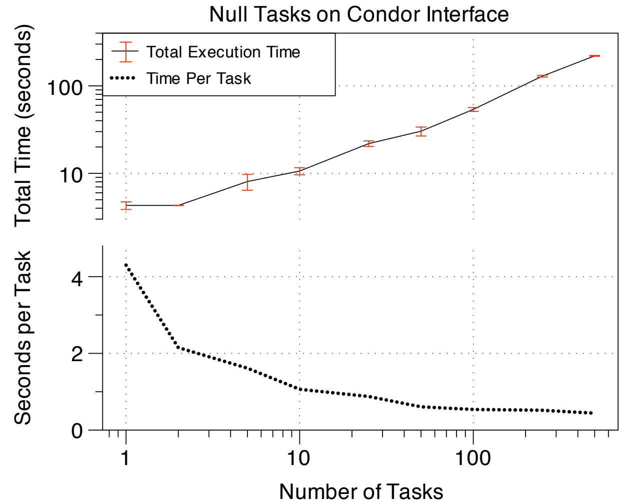
```
def choose_point():
    pt = math.pow(random(),2) + math.pow(random(),2)
    if pt <= 1: return 1
    else: return 0

def monte_pi(rand_seed, num_tests):
        random.seed(rand_seed)
        num_hits = 0
        for i in xrange(num_tests):
                num_hits += choose_point()
        return num_hits

n_tasks, n_tests = 10, 1000000000

interface_obj = CondorInterface()
pymw_master = PyMW_Master(interface=interface_obj)

tasks = [pymw_master.submit_task(monte_pi,
        input_data=(random.random(),n_tests/n_tasks),
        modules=("random","math"),
        dep_funcs=(choose_point,))
    for i in range(n_tasks)]

num_hits = 0
for i in range(n_tasks):
        res_task, result = pymw_master.get_result()
        num_hits += result

print 4 * float(num_hits)/num_tests
```

execution time for a number of tasks. The upper graph in each figure shows the mean and standard deviation of total execution time from 3 experiment runs.

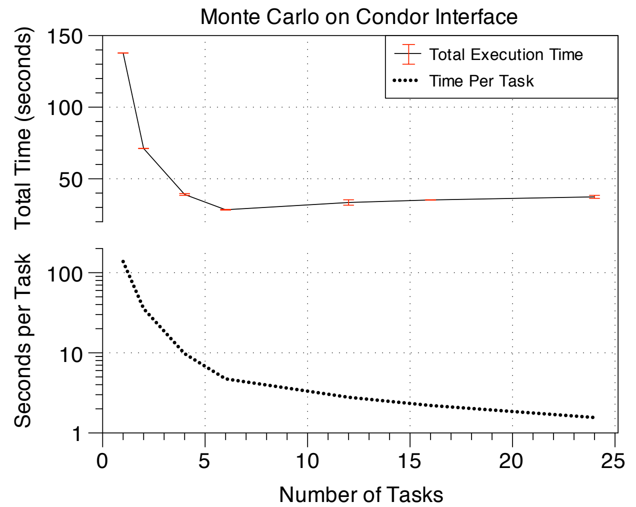
One important aspect of desktop grid systems is the initialization time and task submission overhead. Figure 5(a) shows the time for null tasks to be processed by the Condor system. In this case, a null task is a task that does no calculation and immediately returns the input. As seen in the lower half of Figure 5(a), the initialization time for submitting tasks with the Condor PyMW interface is around 4 seconds. The average overhead per task when submitting large numbers of tasks is less than 1 second, demonstrating the effectiveness of PyMW for performing large batches of tasks on desktop grids.

Figures 5(b) and 5(c) show the execution times for the Monte Carlo and prime tester applications. In both applications, total execution time decreases to a certain point, but additional tasks do not significantly affect total runtime. The Monte Carlo program has a maximum speedup of around 4.9x while the prime tester has a speedup of around 5.4x. For large numbers of tasks with longer computation time, this speedup should be nearly linear. These results demonstrate the scalability of PyMW of performing applications on a Condor based desktop grid platform.

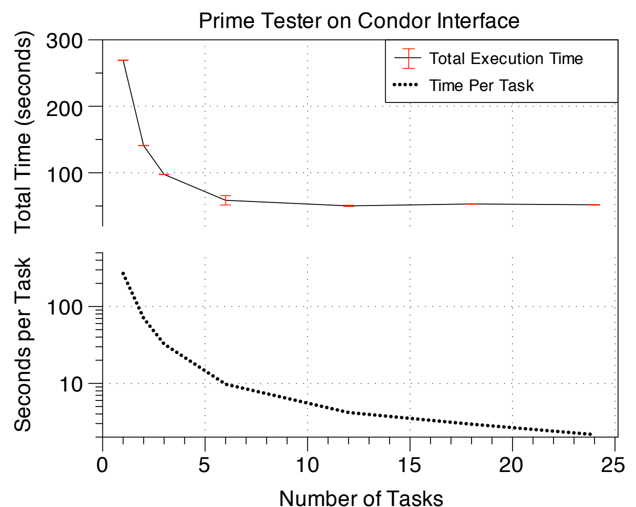
The initialization time and overhead in the BOINC interface varies depending on how many tasks are submitted. Upon submission the BOINC interface copies the task input data to the appropriate BOINC project directory and registers the task in the BOINC database. This introduces overhead from system calls and database operations. Table I shows the overhead for different numbers of submitted tasks, as well as total exection



(a) Null task test results.



(b) Monte Carlo program results.



(c) Prime tester program results.

Fig. 5. Condor interface results.

TABLE I
TASK TIMES FOR THE BOINC INTERFACE.

Tasks	Interface Overhead	Monte Carlo	Prime Tester
10^2	5 seconds	4 mins	16 mins
10^3	2 minutes	6 mins	13 mins
10^4	15 minutes	22 mins	22 mins

TABLE II
TIME TO COMPLETE 10 TASKS PER WORKER (INCLUDING UPLOAD/DOWNLOAD)

Task size	Total tasks	Monte Carlo	Prime Tester
Large	10^2	3.5 mins	15 mins
Medium	10^3	30 secs	1.1 mins
Small	10^4	15 secs	15 secs

times of the three tests. Although job submission can get slower if large numbers of tasks are submitted, it should be noted that BOINC can simultaneously handle task submission and task processing, meaning that as soon as the first tasks are submitted the workers can download and process them.

The worker side of BOINC downloads input files and uploads output files to the server, before and after processing respectively. Input and output files of the tested programs are less than a kilobyte in size, so the overhead is only a few seconds. Each worker downloaded ten tasks per session, adding approximately ten seconds of upload and download overhead to the actual processor times. The process of result validation and assimilation also introduces a few seconds overhead. The three tests we ran for the programs consisted of 10^2, 10^3 and 10^4 tasks. Table II shows the time spent by one worker to process ten tasks of varying size. Because the workers do not exclusively perform BOINC tasks, total execution time can vary greatly. Although these results show that BOINC adds significant overhead, BOINC oriented tasks often take hours or days to complete, meaning this overhead will be negligible for most BOINC programs.

From these experiments, we see that PyMW provides a scalable interface for running master-worker style parallel Python programs on the Condor and BOINC platforms. The overhead of PyMW varies depending on the platform, but in general is very low relative to the expected total task runtime. We also see that PyMW can handle large numbers of tasks without significant slowdown.

V. CONCLUSIONS AND FUTURE WORK

In this paper we introduced the PyMW Python module for parallel distributed master-worker style computation. The module is designed to allow easy usage of various computing platforms by inexperienced scientists and programmers, with emphasis on desktop grids and volunteer computing systems. At the same time, it is intended to be extendable through interfaces to support additional platforms and functionality. We demonstrated through experiments that PyMW can be used to write parallel programs that run on a variety of platforms and scale well.

In future work, we hope to extend the functionality of PyMW. In addition to supporting more computing platforms, PyMW will also give the user more control to determine task status, kill tasks and run non-Python worker programs. To support very long computations, PyMW will allow state freezing and restoring, enabling programs to be restarted without significant recomputation in the case of a crash. We also plan to use PyMW to introduce a new type of BOINC project – called PyBOINC – to support users needing massive computing power for master-worker style computing. The current version of PyMW is available at http://pymw.sourceforge.net/

ACKNOWLEDGMENTS

This work was supported in part by Research Fellowship (19·55401) and Grant-in-Aid for Scientific Research (A)20240002 from the Japan Society for the Promotion of Science, and by the Global COE Program "in silico medicine" at Osaka University. The research and development published in this paper is also supported by the European Commission under contract numbers LSHC-CT-2006-037559 (FP6/CancerGrid) and RI-211727 (FP7/EDGeS).

REFERENCES

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.
[2] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande, "Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology," *Modern Methods in Computational Biology, Horizon Press*, 2003.
[3] "Boinc projects list - http://boinc.berkeley.edu/projects.php."
[4] "Rocks cluster register - http://www.rocksclusters.org/rocks-register/."
[5] Parallel python website - http://www.parallelpython.com/.
[6] Pympi - http://pympi.sourceforge.net/.
[7] mpi4py - http://mpi4py.scipy.org/.
[8] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency and Computation: Practice & Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.
[9] I. T. Foster, "Globus toolkit version 4: Software for service-oriented systems." in *NPC*, ser. Lecture Notes in Computer Science, H. Jin, D. A. Reed, and W. Jiang, Eds., vol. 3779. Springer, 2005, pp. 2–13.
[10] O. Aumage, R. Hofman, and H. Bal, "Netibis: an efficient and dynamic communication system for heterogeneous grids," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 2, pp. 1101–1108, 2005.
[11] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.
[12] M. P. Sullivan and D. P. Anderson, "Marionette: a system for parallel distributed programming using a master/slave model," Tech. Rep. UCB/CSD-88-460, Nov 1988.
[13] E. Heien, A. Kornafeld, Y. Takata, and K. Hagihara, "Pymw - a python module for parallel master worker computing," Tech. Rep., 2008.
[14] J. Basney and M. Livny, "Deploying a high throughput computing cluster," in *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, 1999.
[15] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., Dec 2002.
[16] J. Amar, "The monte carlo method in science and engineering," *Computing in Science & Engineering*, vol. 8, no. 2, pp. 9 – 19, Mar 2006.
[17] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128–138, 1980.