PyMW - a Python Module for Parallel Master Worker 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 provides a unified interface to multiple computation environments including multicore processors, networked clusters and the Berkeley Open Infrastructure for Network Computing (BOINC) software platform. PyMW is intended to support rapid development, testing and deployment of large scale master-worker style computations. It is also designed to allow easy extension to other computing environments with little change in the master-worker program. We demonstrate the effectiveness and scalability of PyMW by performing several master-worker style parallel computations on a multicore machine, a networked cluster and a BOINC project.

1 Introduction

In recent years, there has been a surge of interest in parallel computation as development of faster individual processing cores becomes more difficult. Because of power and heat dissipation limitations, the trend in processor design is to put multiple processing cores on a single chip rather than increase core speed. Furthermore, other multiprocessor computing environments are becoming more common, including clusters with high speed networks and Internet-connected multi-site computing grids. To perform parallel computation on these platforms, a variety of software is available. However, most parallel computing software is oriented towards a particular computing platform. For example, an OpenMP program cannot fully utilize a computing Grid, and a Grid program contains more functionality than is needed to run on a cluster.

In addition, interpreted languages such as Python are becoming popular as program development time is increasingly dominant relative to execution time. Such languages allow for fast program development with runtime error checking and the support of numerous software libraries. Interpreted languages including Python, Java, PHP, and so on, are also platform independent and may be run on any machine with the proper interpreter. In response to the increased interest in parallel computing and Python, several software packages have been developed for parallel computation in Python. However, a significant drawback to many of these packages is that they are designed for only one type of computing platform, usually multicore shared memory processors.

We developed PyMW in response to the need for better Python based parallel computing tools. 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. The worker 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, ray tracing, and other work which is easily divisible with little or no dependencies between divisions. A classic example of large scale master worker computation is the SETI@home [5] project.

The goal of PyMW is to provide functionality for master-worker computation in a Python module that can be executed in a wide variety of computing environments. The module is designed to be as simple and general as possible, with the goal of letting users write a single program that can run on a multicore machine, a networked cluster of computers, or a worldwide computing Grid. However, at the same time the module should allow the user to interact with platform specific features (e.g. using half the cores in a multicore machine).

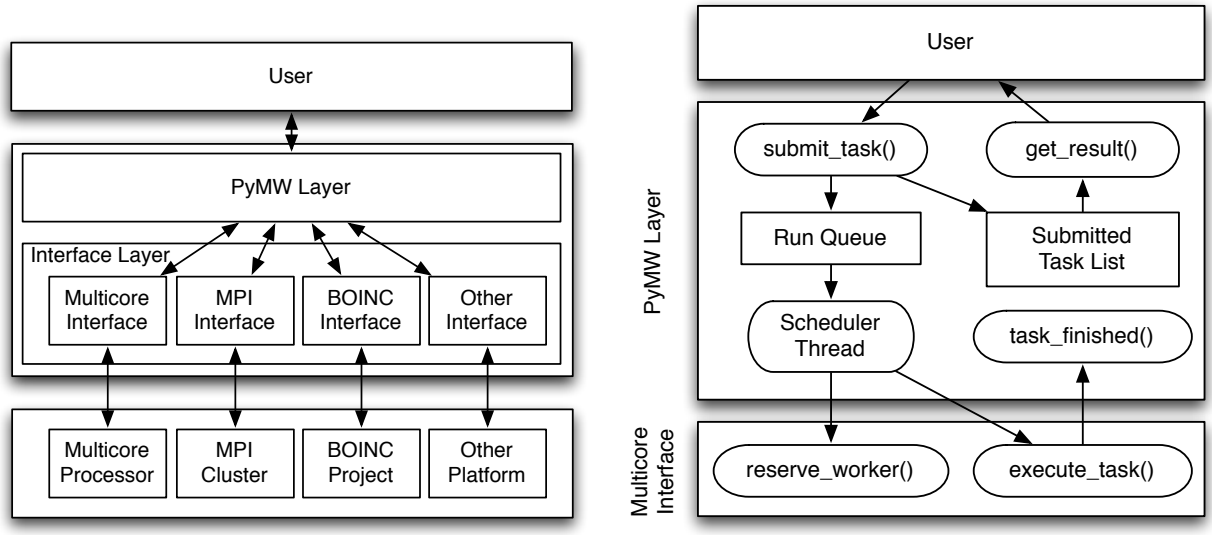
The remainder of the paper is organized as follows. In Section 0.2 we describe the PyMW module, including the functionality it provides and how programs interact with it. Section 0.3 describes platform interfaces and the requirements for interface implementations to properly interact with PyMW. We show the effectiveness of PyMW by running two master-worker style parallel programs in Section 0.4 on multiple platforms and examining the results. In Section 0.5 we review related work and finally offer our conclusions in Section 0.6.

2 PyMW

In this section we describe the organization of the PyMW module. This includes descriptions of the master API in Section 0.2.1, the interface to underlying platforms in Section 0.2.2 and the worker API in Section 0.2.3.

Although recent versions of Python support multithreaded programs, writing parallel programs in Python is difficult. This is because the Python interpreter only supports execution of one thread at a time, regardless of the number of available processors. This limitation can be overcome by spawning multiple Python interpreter processes communicating through sockets or pipes. However, even this approach is limited because it introduces security risks in multi-machine environments, it does not easily work through firewalls, and it can be complicated to implement. Furthermore, customized solutions cannot take advantage of existing features in many computing environments such as automatic load balancing, checkpointing, computational redundancy, etc.

Rather than designing custom solutions for each computing environment, the philosophy of



(a) Abstraction layers of PyMW.

(b) Internal organization of PyMW.

Figure 1: PyMW module details.

PyMW is to utilize existing software as much as possible. For example, most computing clusters contain an implementation of MPI (the Message Passing Interface) [7] which provides fast communication between processes on different machines. For a volunteer computing environment, BOINC (Berkeley Open Infrastructure for Network Computing) [4] provides a framework for packaging and distributing tasks. PyMW uses these software packages to execute tasks on the underlying hardware for master-worker style Python programs.

In order to use existing software packages and allow extension to multiple parallel computing environments, PyMW is divided into two abstraction layers as shown in Figure 1(a). The first layer is called the "PyMW layer". It 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 layer is fully described in Section 0.2.1.

The second layer is called the "interface layer". This layer manages the interaction between the PyMW layer and the underlying software/hardware. For example, on a multicore machine this layer is responsible for spawning processes to perform tasks and notifying PyMW of 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. However, all interface layer implementations must at least accept tasks from the PyMW layer, execute them on their software/hardware and notify the PyMW layer once a task is completed. The interface layer is described in detail in Section 0.2.2.

For this paper we implemented interfaces for three types of systems - multicore machines, networked clusters running MPI and volunteer computing systems running BOINC. Users may select an interface by changing a line in their program code. By switching interfaces a user can run a program on a variety of platforms by only making minute alterations to the program code. For example, this enables program development on a multicore machine, then program testing on a cluster, and finally deployment on a Grid or volunteer computing platform. By using multiple interfaces users can employ different platforms in a single program, for example performing a

parameter sweep computation on a cluster then analyzing the results on a multicore machine.

2.1 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 which the user instantiates. The interface used by the PyMW_Master object is specified at creation. The master object accepts tasks from the user and sends them to its associated interface without blocking, allowing the user program to continue. Users later retrieve the results of task execution 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 computation system such that a user may write a single program that will run on any platform.

PyMW was designed with two users in mind: the application developer and the interface developer. To allow applications to use PyMW as simply as possible, PyMW exposes only four functions to the user. Users may directly interact with the underlying interface for greater control, but the goal of PyMW is to allow program 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 described in Section 0.3.1. Otherwise, this is an instantiation of an interface described in Section 0.2.2.

```
task_object = master.submit_task(executable, input_data=None,
        modules=(), dep_func=())
```

The `submit_task` function creates a task and submits it to the interface associated with the master. It returns an object representing the submitted task. The executable must be the path to a Python script or a user defined function in the current scope. The `input_data` argument is a tuple representing the arguments to the function or script. If `executable` is a function, the `modules` and `dep_funcs` arguments specify what modules and functions are required to execute the function. If `executable` is a script, those arguments are ignored. This function returns immediately.

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

The `get_result` function attempts to return a completed task and result. If `task` is `None`, this will return the next completed task, or an arbitrary task if there are multiple completed tasks. Otherwise this will return the result of a task returned by `submit_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. Passing a task object that has not been previously submitted with `submit_task` will raise a TaskException error.

```
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 keys for interface status will vary depending on the interface implementation. The keys for the master status include a list of task objects that have been submitted with `submit_task`.

The internal organization of PyMW is shown in Figure 1(b), 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. If the user passes a function as the executable, the source of the function and dependent functions are written to a temporary Python script file. This file includes PyMW specific functions for input and output handling. This scheme allows interfaces to treat all tasks in the same manner and lets users ignore PyMW details when calling functions in parallel. To prevent blocking, calls to `submit_task` return immediately. 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 and handle any errors that occurred during processing. The results of the task are later returned to the user through `get_result`.

2.2 Interface Layer

The PyMW layer described in Section 0.2.1 interacts with the underlying computational platform through an interface layer. The interface layer 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 layer implementation is required to 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 functions provided by the interface layer are described below.

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

An interface implementation must at minimum 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 executed by the PyMW scheduler in a separate thread and therefore need not return immediately. Upon completion of the task, it is the responsibility of the interface to call the `task_finished` function. If an error occurred during task execution, the interface layer must pass an Exception object to `task_finished` describing the error. For interfaces that expect many long running tasks, it is best to exit `execute_task` immediately and use a separate thread to periodically check all tasks and report any that have completed.

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

This function returns an object representing the worker to use when executing the task. This object is interface-specific, and is not used in the PyMW layer. 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` functions specific to their program.

```
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.

2.3 Application Module

To allow workers to receive input data and return output data, an application layer is provided in the `pymw_app` module. This is intended to allow seamless integration of a worker application with the computing platform it is run on. This module provides two functions to the worker:

```
input = pymw_get_input()
```

Returns the object passed as `input_data` to `submit_task` in the PyMW layer.

```
pymw_return_output(output)
```

Writes an object back to PyMW. This will only return the last object passed to it, so consecutive calls will overwrite previous objects.

3 Interfaces

As described in Section 0.2.2, the interface layer of PyMW has different implementations depending on the underlying platform. In this section, we describe the interface implementation for three platforms - desktop machines with multicore processors (Section 0.3.1), cluster systems containing tens or hundreds of networked machines running MPI (Section 0.3.2) and global scale systems using BOINC running on thousands or millions of machines (Section 0.3.3).

3.1 Multicore

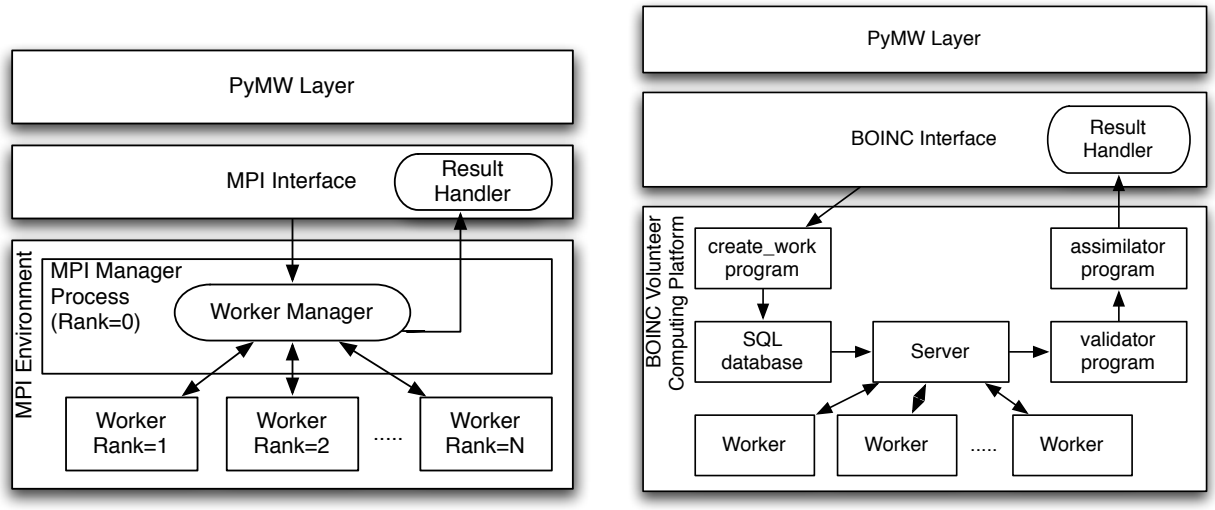
The multicore interface is the default interface for PyMW, and is for use on single machines with one or more processors. This interface is used by default in PyMW_Master when no interface is specified by the user. The multicore interface performs master-worker computation on a single machine through processes spawned by the Python `subprocess` module. The implementation of this interface is roughly 70 lines of Python code.

When initializing the interface, the user specifies the number of workers to use, which may be more than the number of processors in the machine. Because processors in a multicore machine are homogeneous, the `reserve_worker` function serves only to ensure that at most one task is run for each worker. The `execute_task` function spawns new workers with the Python command `subprocess.Popen`, waits for them to complete and reports the task as finished. Load balancing of worker processes on multiple cores is automatically performed by the operating system. Any Python or OS errors are caught and reported to the user through `task_finished` called from `execute_task`.

3.2 MPI

The MPI interface for PyMW is intended for use with clusters running an MPI implementation compatible with the MPI-enabled Python interpreter pyMPI [3]. The interface requires at least pyMPI 2.4 and Python 2.4 to be installed on the cluster.

The MPI interface was difficult to implement because of limitations in our MPI environment (SCore 6.0.2.1). Initially, we planned to spawn workers through multiple calls to `mpirun`. However, in our environment `mpirun` is a wrapper to the SCore `scrun`, which does not allocate new



(a) MPI Interface and task flow.

(b) BOINC Interface and task flow.

Figure 2: PyMW interface details.

jobs in a round robin fashion and would execute all tasks on a single machine. Some versions of `mpirun` allow users to specify the host for execution, however ours did not support this feature. Therefore, an interface for MPI implementations that support host choice or round robin allocation should be much simpler. However, by launching `mpirun` only once (during initialization) we avoid overhead that would become problematic for large numbers of tasks.

The organization of the MPI interface layer implementation is shown in Figure 2(a), with arrows representing the flow of tasks through the interface and cluster. When initializing the interface, the user specifies the number of workers (MPI processes) to start. Rather than running all of PyMW in MPI, the MPI interface starts an MPI program with pyMPI and communicates with it through a socket. The MPI interface sends task information through the socket to the manager process (MPI rank 0). The task information consists of the location of the executable, and the input and output files to be used for the task.

The manager process consists of a worker manager written in Python. The worker manager keeps a list of workers not performing a task. Upon receiving a task from the MPI interface, the manager forwards the task to an available worker using MPI. If the communication socket closes, the manager also notifies all workers to quit computation. The manager automatically balances the computational load since a worker will receive another task when the worker is free. To coordinate multiple workers, the manager uses the non-blocking functions `MPI_Isend` for task distribution and `MPI_Irecv` for task completion notification. This is necessary in pyMPI because the normal `MPI_Send` and `MPI_Recv` functions are not compatible with Python multithreading.

The worker processes (MPI ranks 1 to N) wait for new tasks from the manager process using `MPI_Irecv`. Upon receiving task information, a process uses the Python `subprocess.Popen` function to execute the task, and returns notice of task completion to the manager process with `MPI_Isend`. The manager process parses the task completion notice, replaces the worker on the list of available workers, and sends a notice back to the MPI interface through the socket. At initialization, the MPI interface spawns a separate thread to handle such task completion notifications. This thread calls `task_finished` for completed tasks upon receiving a notification.

The MPI processes are terminated when the interface communication socket is closed. This handles both normal program completion and errors in the user program. When the socket is closed, the manager process sends a `None` task to each worker, causing them to quit.

3.3 BOINC

PyMW was originally conceived to support BOINC application development by providing a simple and intuitive Python interface for the BOINC platform. A barrier to common usage of the BOINC platform is the special preparation required to run an application in volunteer computing environments. In BOINC the full master-worker cycle is supported by multiple programs including a work generator, a server, a result validator and an result assimilator. Each of these must be modified by the application developer. The goal of the PyMW BOINC interface is to automatically manage as many of these as possible, so developers can focus exclusively on the 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 to be used with PyMW, and installs BOINC components which handle interaction with PyMW. The result is that users can write Python programs with the PyMW module that will execute transparently in the BOINC environment. One goal of PyMW is to significantly increase the number of applications using BOINC by simplifying application development and deployment.

The flow of tasks through the PyMW BOINC interface and BOINC software platform is shown in Figure 2(b). The user submits tasks to PyMW using the `submit_task` function. The PyMW scheduler passes the tasks to the BOINC interface, which uses the BOINC create_work program to generate BOINC specific tasks. This involves copying the input data to a BOINC specified location, creating input and output template files and finally executing the create_work program to insert the task into the BOINC database. Later, the BOINC 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 required by BOINC. The BOINC 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 BOINC 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, `task_finished` is called for the corresponding task object.

4 Experiments

To test the effectiveness of PyMW in performing master-worker computations, we wrote two Python programs utilizing the PyMW module. These programs represent two possible applications for PyMW - a Monte Carlo simulation and a parameter sweep. Each of these programs was run on three platforms - a multicore machine, a networked cluster with MPI 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 0.4.1 and the results in Section 0.4.2.

4.1 Experiment Setup

The first program is a simple Monte Carlo program which estimates the value of pi. This program estimates pi by randomly selecting points in a unit square. If the total number of points is n and the number of points satisfying $x^2 + y^2 \leq 1$ is m, 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. Each worker is given a different initial random seed to avoid identical Monte Carlo runs. For the multicore interface 100 million (10^8) points were tested, for the cluster MPI and BOINC interfaces 1 billion (10^9) points were tested. The program is a total of 50 lines of Python code, which includes creating the PyMW master and interface objects, submitting tasks, performing the Monte Carlo simulation and gathering the results. A simplified version of the code is shown in Listings 1. This is identical to the test program except for measuring execution time.

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 [9] (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. Tasks are divided into equal sizes, with three tasks for each worker. For all three interfaces, the range [1, 100000] was tested. The program is 80 lines of Python code (including code to measure execution time).

For each platform, we performed experiments to determine the effect of number of workers on total run time. The experiments for the multicore interface were run on a dual processor quad core 2.83 GHz Intel Xeon (8 cores total) with 4 GB RAM. The cluster experiments were run on a 31 node cluster, with each node using a 2.8GHz dual Xeon processor and 2 GB RAM, connected with a Gigabit ethernet hub for a total of 62 processors. The BOINC experiments were run on a BOINC project with a dozen workers of varying hardware running Linux and Windows.

4.2 Experiment Results

The results of the experiments on the multicore platform are shown in Figure 3, and the results for the cluster platform experiments are shown in Figures 4(a) and 4(b). These figures show the total execution time given a number of workers. Each time shown is the mean value from 3 experiment runs. For the multicore interface the standard deviation of the run times for all programs was less than 5% of the mean, and for the cluster interface less than 10% of the mean.

Listing 1: Monte Carlo Python code.

```
def throw_dart():
    pt = math.pow(random.random(),2) + math.pow(random.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 += throw_dart()
    return [num_hits, num_tests]

workers = 4
num_tests = 100000000
num_hits = 0

interface_obj = MulticoreInterface(num_workers=workers)
pymw_master = pymw.PyMW_Master(interface=interface_obj)

tasks = [pymw_master.submit_task(monte_pi,
    input_data=(random.random(),num_tests/workers,), modules=("random", "math"),
    dep_funcs=(throw_dart,)) for i in range(workers)]

for task in tasks:
    res_task, result = pymw_master.get_result(task)
    num_hits += result[0]

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

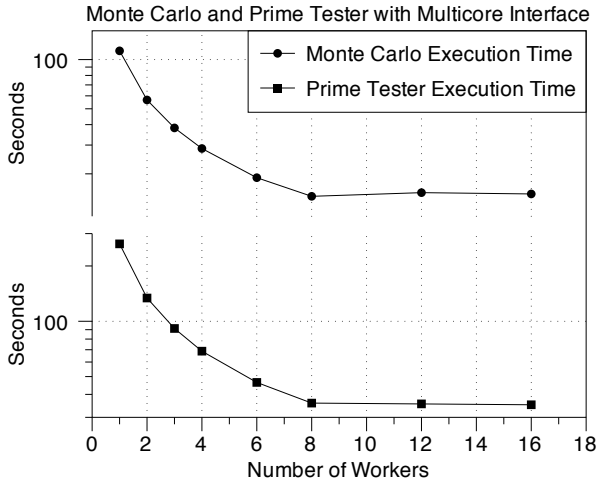
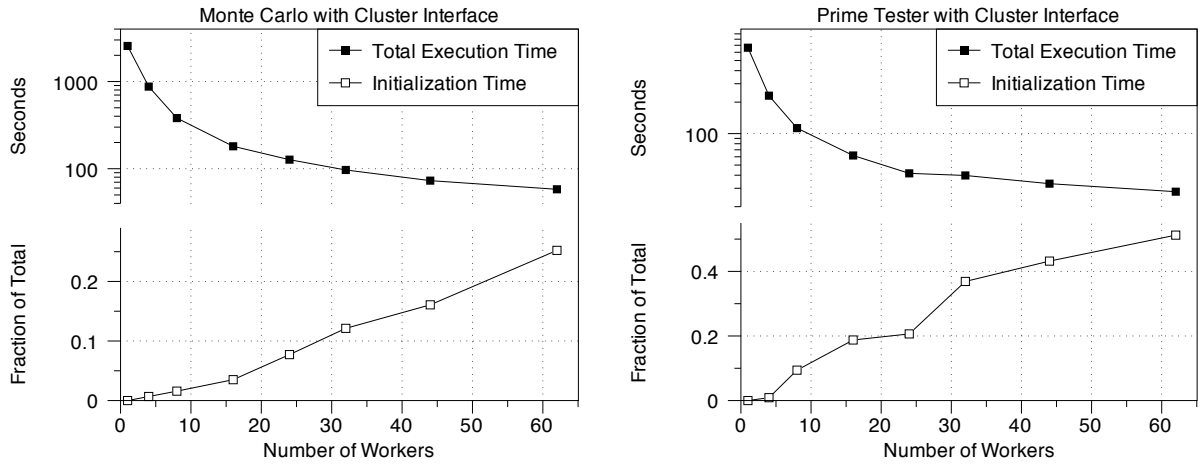


Figure 3: Multicore experiment results.

Table 1: Speedup on 2x4-core machine.

Workers	Monte Carlo	% max	Prime	% max
1	1	100	1	100
2	2.00	100	1.97	98.6
3	2.96	98.7	2.89	96.3
4	3.96	99	3.84	95.9
6	5.97	99.5	5.68	94.6
8	7.76	97	7.36	92.0
12	7.37	92.1	7.43	92.9
16	7.50	93.8	7.51	93.9



(a) Monte Carlo program results.

(b) Prime tester program results.

Figure 4: Cluster MPI interface results.

One important aspect of parallel systems is the initialization time and overhead. In PyMW the initialization time is the time required to instantiate and initialize an interface object and master object. With the multicore interface, initialization time was measured to be less than 0.01 seconds for all experiments and therefore is not distinguished from total run time in Figure 3. However, initialization time in the cluster interface was a significant fraction of total execution time, and is included in Figures 4(a) and 4(b). During initialization the cluster interface uses the `mpirun` command to start multiple MPI processes which can take a significant amount of time. As shown in both figures, the fraction of total time taken by initialization increases with the number of workers. The absolute time for initialization also increased with respect to the number of workers. Therefore, users of this interface must be careful to select a number of workers suitable to the expected time for computation. It is worth noting that this initialization time is a result of MPI startup in SCore rather than overhead caused by PyMW.

Figures 3 and 4 show a clear decrease in execution time by using more workers. Figure 3 shows little change between using 8, 12 or 16 workers because it was run on an 8-core machine. Although additional workers do not improve execution time, neither do they significantly worsen it. This indicates that PyMW does not add significant overhead when using the multicore interface. In Figures 4(a) and 4(b) we see improvements in execution time from additional workers, though the rate of improvement decreases. This is mostly due to the initialization cost mentioned above.

Table 2: Speedup with cluster interface.

	Monte Carlo				Prime Tester			
Workers	Total	% max	Non-init	% max	Total	% max	Non-init	% max
1	1	100	1	100	1	100	1	100
4	2.9	97.7	2.9	98	2.9	95.9	2.9	96.8
8	6.7	95.7	6.8	97.1	5.9	83.9	6.5	92.7
16	14.2	94.7	14.6	97.3	10.7	71.6	13.2	88.1
24	20.2	87.8	21.8	94.8	15.9	69.2	20.1	87.2
32	26.5	85.5	30.0	96.8	16.6	53.8	26.4	85.2
44	35.0	81.4	41.7	97.0	20.0	45.5	35.2	81.8
62	44.1	72.3	58.8	96.4	23.8	39.1	48.9	80.1

Detailed results regarding the scalability of PyMW are shown in Tables 1 and 2. These tables show the speedup of program execution for varying numbers of workers. For example, Table 1 shows a speedup of 3.96x for the Monte Carlo program when using 4 workers. This is 99% of the theoretical maximum speedup of 4.00x. From this table we see that the Monte Carlo program achieves excellent parallelism using the multicore interface, even when there are more workers than processors. The prime program also scales well on multicore, but does not have the same performance gains as Monte Carlo. This is because the prime program generates more tasks, and the time for PyMW to start workers and collect results introduces more overhead. Also, the prime program is non-deterministic in its execution time. Determining whether n is composite takes from 1 to 50 test repetitions, which means the final task may delay completion of the whole program.

Table 2 shows the speedup of program execution using the cluster interface. In this case, we distinguish between the total execution time and the calculation time (total time minus initialization time). This is because initialization time becomes a significant fraction of total time as the number of workers increases, and we want to examine the overhead generated only by PyMW. Similar to the multicore interface, the cluster interface performs well for the Monte Carlo program. This is evident when looking at the speedup of only the calculation, which is almost always 95% or more of the theoretical maximum. Including initialization time, the speedup becomes less impressive as workers are added. This is because initialization time becomes a greater fraction of total time for more workers. Table 2 also shows the performance of the prime tester program with the cluster interface. In this case, the performance improvement is not as good as with the Monte Carlo program, and also falls behind the prime tester on the multicore interface. This is likely due to the overhead of the MPI interface to transfer tasks to the worker manager and then the worker.

The initialization time and overhead introduced by the BOINC interface can widely vary depending on how many tasks are submitted. Upon submission the BOINC interface must copy the input data of the task to the appropriate BOINC project directory and register the task in the BOINC database. This introduces overhead from system calls and database operations. Table 3 shows the overhead for different numbers of submitted tasks, as well as total exection 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 kilo-

Table 3: Task times for the BOINC interface.

Tasks	BOINC Interface Overhead	Monte Carlo Total	Prime Tester Total
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 4: 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

byte 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 4 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 a variety of platforms. The overhead of PyMW varies depending on the platform, but in general is very low relative to the expected total task runtime.

5 Related Work

The master-worker computation model has been used for many years in a variety of parallel computing environments. An older example of a system using this model is Marionette [10], which was designed for operation on heterogeneous networks of workstations (NOW). Similar to PyMW, this aimed to provide a simple interface, but for the C programming language rather than Python. However, Marionette was intended only for use on NOWs, and was not suitable for use on large scale platforms nor for multiprocessor machines.

For master-worker style computing in Grids, the Condor [11] and Globus [8] software packages are commonly used. The IBIS project [6] is also developing 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 [4] and the Cosm platform. However, neither of these uses an interpreted language, and generally require customized programs for submitting tasks and collecting results.

There are a variety of tools for parallel programming in Python. MPI implementations for Python include pyMPI [3], mpi4py [1] and MMPI. These have the same functionality as a standard MPI implementation and thus are not well suited to master-worker computation on large scale computing platforms. There is also a variety of modules supporting master-worker style parallel computing with Python, usually on a single machine. These include Parallel Python [2], seppo

(simple embarrassingly parallel python), and the `pprocess` and `processing` Python modules.

6 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 multicores, clusters and volunteer computing systems by inexperienced scientists and programmers. At the same time, it is intended to be extendable through interfaces to support additional platforms and functionality. We demonstrated through experiment 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 running computations, PyMW will support state freezing and restoring, allowing master programs to be restarted from the middle. We also plan 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.

References

[1] mpi4py - http://mpi4py.scipy.org/.

[2] Parallel python website - http://www.parallelpython.com/.

[3] Pympi - http://pympi.sourceforge.net/.

[4] D. P. Anderson. BOINC: A system for public-resource computing and storage. In R. Buyya, editor, *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*, pages 4–10. IEEE Computer Society, 2004.

[5] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[6] O. Aumage, R. Hofman, and H. E. Bal. NetIbis: An efficient and dynamic communication system for heterogeneous grids. In *Proc. of 5rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, Cardiff, UK, May 2005.

[7] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[8] I. T. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin, D. A. Reed, and W. Jiang, editors, *NPC*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

[9] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[10] M. P. Sullivan and D. P. Anderson. Marionette: a system for parallel distributed programming using a master/slave model. Technical Report UCB/CSD-88-460, EECS Department, University of California, Berkeley, Nov 1988.

[11] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.