Evaluation of Python-based Tools for Distributed Computing on the Raspberry Pi

Jacob Morra
Department of Electrical, Computer and Software Engineering
University of Ontario Institute of Technology
Oshawa, ON, L1H 7K4 Canada
jacob.morra@uoit.ca

Qusay H. Mahmoud
Department of Electrical, Computer and Software Engineering
University of Ontario Institute of Technology
Oshawa, ON, L1H 7K4 Canada
qusay.mahmoud@uoit.ca

ABSTRACT
This paper presents the results of comparing the efficiency of four Python-based modules and libraries with regards to their respective ability to run distributed tasks across a given cluster of Raspberry Pi's. The four Python-based tools considered are the following: PyRO v4.45, DCM v1.0.0, PP v1.6.4.4, and Mpi4py v2.0.0. Three tests are used to compare run time efficiency for distributed tasks: The first test proposed is prime factorization of composite values n, where n is the product of two prime numbers, and p and q are the respective prime factors. The second test proposed is the approximation of π to a desired point of precision in a set number of iterations. The third test proposed is the approximation of π in a fixed number of iterations with variation in the cluster size used. The overall purpose of these tests is to provide a basis of comparison in regards to the efficiency of each Python-based tool.

CCS Concepts
• **Software and its engineering~Message oriented middleware.**
• *Computing methodologies~Distributed algorithms.*
• *Hardware~Testing with distributed and parallel systems.*

Keywords
Application Programming Interface (API); Python tools for distributed computing; Raspberry Pi; run time evaluation; master-slave architecture

1. INTRODUCTION
With the advent of inexpensive computers such as the Raspberry Pi Model B, experimentation with distributed tasks across clusters of machines has become more accessible than ever before for communities of hobbyists, educators, and researchers. Such experimentation is becoming increasingly important as the definition of "computer" evolves to include smart phones and tablets, and as the cost of computing power continues to decline [2]. For effective development of distributed applications, any efficiency drawn from the choice of tool used is desirable.

CASCON 2016, Oct 31–Nov 2, 2016, Markham, ON, Canada.

DOI: http://dx.doi.org/10.1145/12345.67890

Although efficiency in the execution of small tasks in parallel gains little to no benefit from using an optimized module or library, more computationally expensive computing applications would benefit from optimal module selection; to consider an example, if a particular tool could be shown to provide a scalable 5% increase in run time efficiency over another similar tool, hours of potential run time could be saved for each developer, totaling to hundreds of hours saved when scaled to a the development community as a whole.

In an attempt to support the current and future development of distributed computing projects on the Raspberry Pi, this paper presents an evaluation of Python-based tools. Python-based tools were exclusively selected for two primary reasons: first, Python is one of the most popular languages on the Raspberry Pi and therefore features plenty of available tools; second, Python can be said to be one of the easiest languages to learn due to its simple syntax and use of modules [2].

Six Python-based tools were originally considered for this research: PyRO, DCM, PP, Mpi4py, Celery, and IPython; however, due to similarities between Mpi4py, Celery, and IPython – i.e. all are based on the Message Passing Interface (MPI) standard – and existing run time comparisons between these tools in [3], only Mpi4py (the most efficient tool in [3]) was selected of the three. Tool diversity was desired for the purpose of providing a more comprehensive evaluation such that developers could weigh run time comparisons against their need to use particular features.

PyRO (Python Remote Objects) is a collection of modules which allow for method calls on remote objects over a network. PyRO features distribution of resources or tasks between machine clusters; remote control of applications from a control client; and separation of privileges using PyRO objects. Remote objects created generate a proxy which intercepts local method calls and applies them. PyRO uses serpent for serialization [4].

DCM (Distributed Computing Middleware) is a remote object API meant for setup of machine clusters in a master-slave architecture: one master distributes and executes remote tasks to slave machines. DCM builds upon the modules provided by PyRO and netifaces and includes methods for cluster-based parallel task execution. DCM assumes that nodes within the cluster are all connected via a reliable Ethernet switch, or over an encrypted channel (VPN, SSH) [1].

PP (Parallel Python) is a lightweight python module which allows for parallel execution of python scripts in a cluster architecture. Some of

the key features include automatic detection of the optimal processor and/or cluster setup, cross-platform and cross-architecture operation, and dynamic load balancing [2].

Mpi4py is a message-passing library meant to function on a variety of parallel machines concurrently. It builds on top of the established MPI standard. Mpi4py can be used to create portable programs in Fortran, C, or C++; though, it has been adapted for Python. Some similar message passing packages include Celery and IPython. Celery is an MPI-based job queue, whereas IPython is an MPI-based CMS (Content Management System) [5].

In addition to tool selection, a selection of the programming language used for parallelization is also important; however, this is beyond the scope of this paper, and may instead be considered in future work. See Table 1 for a breakdown of popular tools used for each of the three major object-oriented languages: Python, Java, and C++.

Table 1. List of available tools for Python, Java, and C++

Language	Tool			
	Name	**Creator(s)**	**Last Updated**	**Open Source? (Y/N)**
Python	PyRO v4.45	Irmen de Jong	05-24-2016	Y
	DCM v.1.0.0	M. Lescisin & Q. H. Mahmoud	08-26-2015	N
	PP v1.6.4.4	Vitalii Vanovschi	01-14-2015	Y
	Mpi4py v2.0.0	Lisandro Dalcin	10-18-2015	Y
	IPython v. 4.0.1	Fernando Perez	11-25-2015	Y
	Celery v. 3.1.23	Ask Solem	03-10-2016	Y
Java	RMI	William Grosso	2015	N
	MPJ Express v0.44	Guillermo Lopez Taboada	04-18-2015	N
	JPPF v5.1	Laurent Cohen	09-12-2015	N
C++	OpenMP	Blaise Barney	11-15-2015	N
	AMP	Kate Gregory	12-08-2013	N
	TBB v4.4	James Reinders	02-11-2016	Y

The breakdown of the remaining sections of this document is the following: In section 2, relevant points from related work are discussed. In section 3, an outline is provided for procedures and conditions for testing. In section 4, specific implementation details are provided for setup of the hardware and software used. In section 5, results of the run time tests detailed in section 3 are plotted and discussed. Finally, section 6 recommends a tool based on the criteria in section 3 and results in section 5, and presents ideas for future work.

2. RELATED WORK

The motivations, methodologies, and proposed testing of this research paper are not entirely unique. Others have conducted run time or performance tests for various machine clusters. Some of the most similar of these works have been selected for comparison.

In [6], Dye explores the performance capabilities of an eight node Raspberry Pi cluster organized into a local area network (LAN) with installed distributed computing tools (libraries, modules, APIs), much like the setup of Raspberry Pi's in this research paper. Dye performs various benchmarking tests with the cluster to determine whether or not parallelization improves run times. In one such test, Dye runs OpenFOAM – an open source collection of Computational Fluid Dynamics solvers – and observes a clear decrease in run times from a node size of C=1 to C=4; however, due to what Dye refers to as a "communication step" [6], there is an increase in run times between a cluster size of C=4 to C=8. Dye explains that this slowdown is related to OpenFOAM specifically, which indicates that the same slowdown after 4 worker nodes should not be expected in this paper.

In [3], Lunacek et al perform run time tests for Python-based tools. The tools considered include Mpi4py, IPython, and Celery. Although performance comparisons are performed with Many-Task computing (MTC) in mind – for machines with up to 12288 cores –Mpi4py showed significantly lower run times overall [3]. Moreover, Lunacek et al observed that Mpi4py has a significantly lower initialization time than the other Python packages tested; with 12 cores, Mpi4py yielded an initialization time of approximately 0.4% of IPython and 2% of Celery. The implication in [3] could be broadened by the results of this paper with respect to run time comparisons between Mpi4py, PyRO, DCM, and PP.

3. METHODOLOGY

3.1 Test Procedure

All tests are conducted in a master-slave architecture, with one master node distributing tasks across a given cluster size of slave nodes. One Raspberry Pi 2 Model B functions as the master node and four Raspberry Pi Model B's function as the slave nodes. A fifth slave node is created by a local process on the master node. Testing begins with the master sending out a distributed task to all slaves, followed by slaves executing the assigned sub-tasks, and concluding when all slaves have reported to the master node.

Prior to all tests, an access console – a Dell Latitude E5550 with Intel Core i5-5300 CPU at 2.30 GHz and 8.00 GB RAM running Windows 8 64-bit – creates a TightVNC Viewer session with each Raspberry Pi. No background programs other than TightVNC Server are running. From the access console, a timer starts once master node executes its first script – within the script, time starts immediately after all modules are imported and ends immediately after the test result is printed to the console. Note that after the timer starts, no keyboard interrupts are performed.

During all tests conducted, each Raspberry Pi has an identical image with Raspbian Jessie version 4.4, with the following additional packages: Python 2.7.3, netifaces 0.10.4, python-matplotlib, python-pyro4, python-pp, python-mpi4py, and TightVNC 2.7.10 – these packages are available through the Debian repository. All machines are connected within a local area network (LAN) and each Raspberry Pi is assigned a unique static IP address.

3.2 Test Details

Two brute-force algorithms were considered for conducting run time testing with Python-based tools on the Raspberry Pi's. For each tool considered and for each value tested, 10 run times were recorded; the average of each set of run times was recorded in addition to the standard deviation.

It is important to note that the implementation of each algorithm is by no means state-of-the art; rather, each has been implemented with the goal of comparing run times between the tools considered. It is also important to note that although the significance of load-balanced and load-unbalanced testing is not thoroughly explored in this paper, a

decision was made to include tests with both unequal work (Test 1) and equal work (Test 2) between slave nodes.

Test 1: Prime Factorization

The Prime Factorization algorithm includes two simple functions: pFactorSearch (code sample 1) and isPrime (code sample 2).

Code sample 1:

```
def pFactorSearch(start_value,end_value,composite_num):
    pFactors = []
    while start_value<=end_value:
        if composite_num%i==0:
         if isPrime(i):
            q=c/i
              pFactors.append(i)
              pFactors.append(q)
          i=c+1
       else:
          i+=1
    return pFactors
```

Code sample 2:

```
def isPrime(i):
  k=2
  while k<i:
    if (i%k==0):
      return 0
    else:
      k+=1
  else:
    return 1
```

On a given sub-interval of [start_value, end_value], each slave will test for all factors of composite_num; these factors are then tested for being prime numbers with the isPrime function. Based on the idea that composite_num is the product of two unique prime numbers, the candidate which is both a factor and a prime number is a direct match – the slave which finds this number finishes earlier than the others, which makes this a load-unbalanced implementation. The numbers considered in testing are 5977007, 26842183, 42500489, 112150573, and 285876629 – there is no specific intention behind these values, although a run time range of a few seconds to a few hundred seconds was desired.

Test 2: Pi Determination

The Pi Determination algorithm includes the piCalc function.

Code sample 3:

```
def piCalc():
        inside=0
        n=1000000
        for i in range(0,n):
                x=random()
                y=random()
                if sqrt(x*x+y*y)<=1:
                        print x
                        print y
                        print inside
                        inside+=1
        pi=4*inside/n
        return '%.2f'%(pi)
```

For a fixed number of iterations, each slave calls the piCalc function, which considers random values within the square range of (0,1) x (0,1) in a Cartesian coordinate system; the function then determines whether or not the random value is within a semicircle of radius 1 centered at the origin. To approximate pi, the area obtained – through counting the proportion of random occurrences inside the semicircle portion and dividing by the total number of random occurrences – is multiplied by four. As in Test 1, the numbers considered in testing (1000000, 5000000, 10000000, and 50000000) were solely considered for the magnitude of run times they generated (a range of a few to a few hundred seconds was desired).

Test 3: Pi Determination, Varying Node Size

As a final test, Test 2 is applied again, but in this case the number of iterations is held constant at 10000000 and the cluster size is varied from one to five worker nodes. Note that PP is not used in this test, as it features dynamic job loading – i.e. if a cluster size of 3 is intended for testing, PP automatically allocates more jobs to local processes.

3.3 Potential Sources of Error

This sub-section focuses on issues with specific choices in testing methodology which have the most potential to cause errors in the final results. These issues are highlighted in the following paragraphs.

One potential source of error in the methodology is the choice to include a fifth slave node from a local process on the master node. This decision was originally inspired by the desire to increase the cluster size with the resources available and within the time constraints of the research project. To improve the robustness of this research, future work should include a comparison between results found in this paper and results from a true five-node setup.

Another potential source of error in the methodology is the choice to use a LAN connected to larger shared LAN (see section IV-A) – shared with three other lab members. This setup was originally deemed to have negligible interference with testing; however, an inconsistency in run times was first observed during Test 1 (n=112150573). Specifically, run times of 33 seconds were followed by a decrease of approximately 3 seconds within four hours of the initial recording. To highlight the inconsistencies in communication time, bandwidth between the master node and a slave node were recorded at various times. Table 2 illustrates this.

Table 2. Bandwidth observed between nodes at various times

Date	Time	Bandwidth Observed
July 14, 2016	5:00 PM	56.0 Mbits/sec
July 14, 2016	5:40 PM	54.4 MBits/sec
July 14, 2016	6:20 PM	52.5 MBits/sec
July 15, 216	12:00 PM	47.1 MBits/sec
July 15, 216	12:40 PM	51.4 MBits/sec

As observed in Table 2, a 16% bandwidth delay is observed from July 14th at 5:00 PM and July 15th at 12:00 PM, indicating that some interference from local traffic is possible. To negate this source of error in an extension of this work or in other works, a private LAN setup would be preferred.

4. IMPLEMENTATION

4.1 Hardware

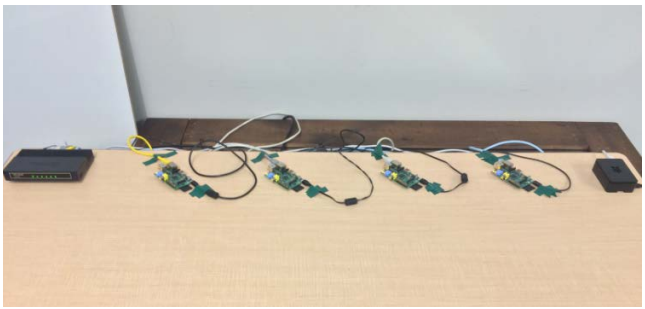

Figure 1. One Raspberry Pi 2 Model B (far right) and four Raspberry Pi Model B's – with a five-port switch connecting the router to the four Pi's – comprise the setup for all conducted tests.

The hardware setup used for testing includes four Raspberry Pi Model B's with static IP addresses connected by Ethernet to a TP-LINK 5-Port Gigabit Desktop Switch (TL-SG1005D). Four ports of the switch are occupied by the Raspberry Pi's; the fifth port is connected to another 5-Port Switch: a D-Link DGS-1005G. The D-Link switch connects to the Raspberry Pi 2 Model B as well as the 5-Port TP-LINK Switch.

4.2 Software Setup

The software setup used for testing includes all items installed as mentioned in section 3.1. For the specific tools tested, the following paragraphs describe respective setup highlights:

4.2.1 PyRO

For full implementation of PyRO and the features used for testing, the python threading module and the python logging module are required. In order for the nodes in the Raspberry Pi cluster to communicate with one another and complete tasks in parallel, PyRO natively allows for the use of a **name server** – a registry which maps URIs of remote objects to logical names. PyRO 4 has a naming module which allows for simple creation of a name server: calling the module and specifying the host IP address allows for interfacing with remote objects. The name server can be printed to check which nodes are connected. To run each test, the name server is initiated; next, each worker node runs a slave script which contains methods as well as remote object creation using a Pyro4 Daemon – a Pyro Daemon is created, the Daemon locates the name server, and the Daemon registers the object to the name server under an alias. Finally, a script is run on the master node which calls upon all remote objects and performs a method call for each node.

4.2.2 DCM

Implementation of DCM requires all PyRO dependencies in addition to python-netifaces. DCM utilizes the PyRO name server in order to broadcast and call upon slave node processes. As with PyRO, the name server is started, slave nodes connect to the name server, and a master node calls upon slaves to execute a given task. However, unlike with PyRO, DCM adds a "slave tasks list" [1], which allows for single line calling of tasks to run in parallel. DCM also includes a round robin task scheduler, which allows for the running of tasks concurrently.

4.2.3 PP

There are no additional dependencies with PP. A major difference between PP and all other modules tested is that PP does not require slave files to be distributed onto the remote Raspberry Pis; rather, a generic "pp-server" file is executed on every slave node to allow projection of tasks onto the Pi's remotely. In testing, the master node has a script which calls for tasks to be run on the remote slave machines using a dedicated function. After dividing work into jobs and allocating the jobs to the job server, PP dynamically selects which nodes should receive which jobs, based on how many nodes are available in the cluster [2].

4.2.4 Mpi4py

There are no additional dependencies with Mpi4py. Uniquely for Mpi4py, an identical copy of a single python file (containing both master and slave instructions) must be present in the exact same location on all of the Raspberry Pi's. Then, from the master node only, a script may be executed as in the case of the other modules. To control which worker node does a selected job, a control statement can be used to specify a condition for the "rank" of the machine; for example, the local node by default is assigned a rank of 0 by Mpi4py; the second node in the cluster is given rank 1, and furthermore the remaining nodes are assigned unique ranks.

5. RESULTS

5.1 Results from Test 1: Prime Factorization

Figures 2-6 show the results of each set of 40 run time tests for given composite values and Python-based tools.

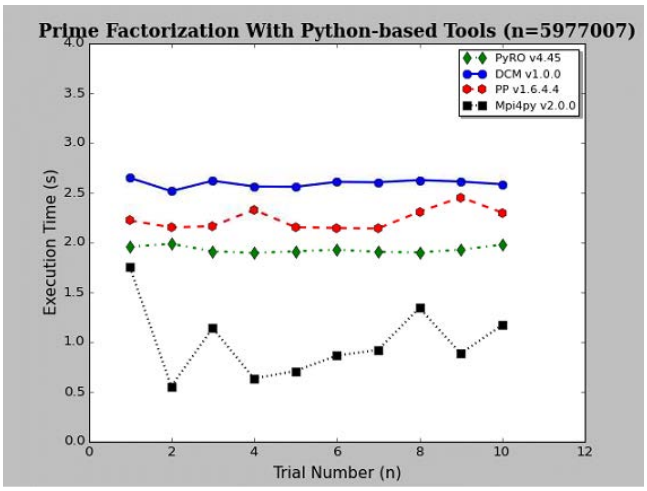

Figure 2. Prime Factorization for n=5977007 (10 run time tests/ tool).

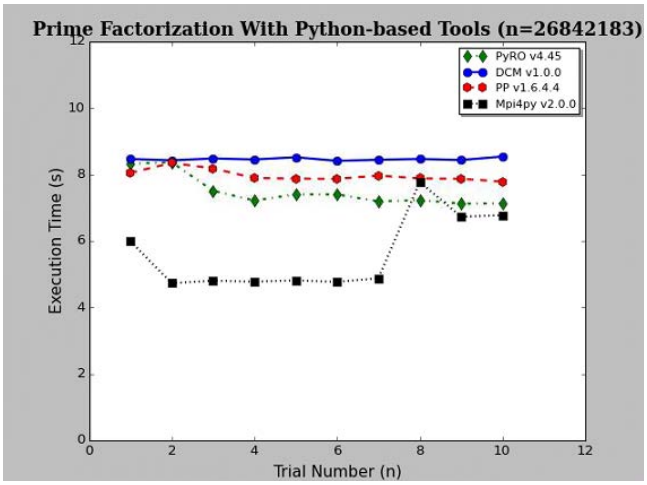

Figure 3. Prime Factorization for n=26842183 (10 run time tests/ tool).

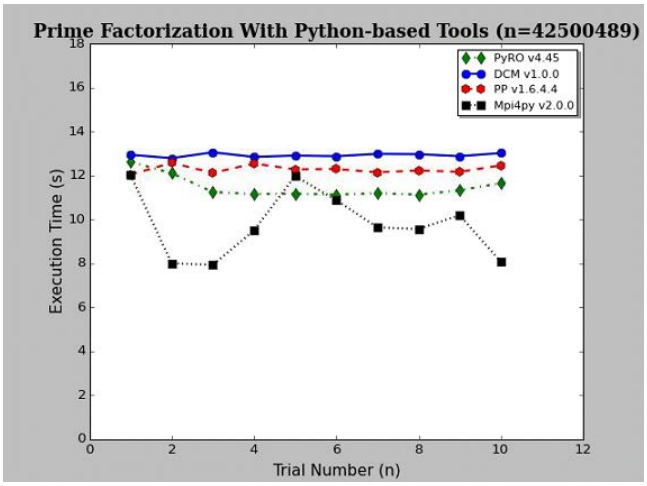

Figure 4. Prime Factorization for n=42500489 (10 run time tests/ tool).

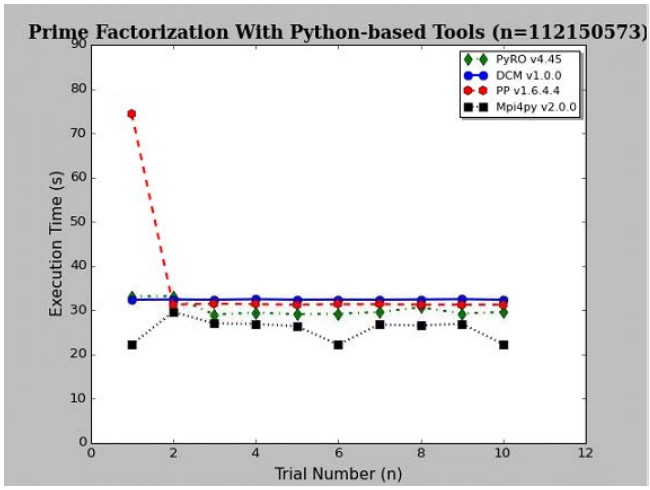

Figure 5. Prime Factorization for n=112150573 (10 run time tests/ tool).

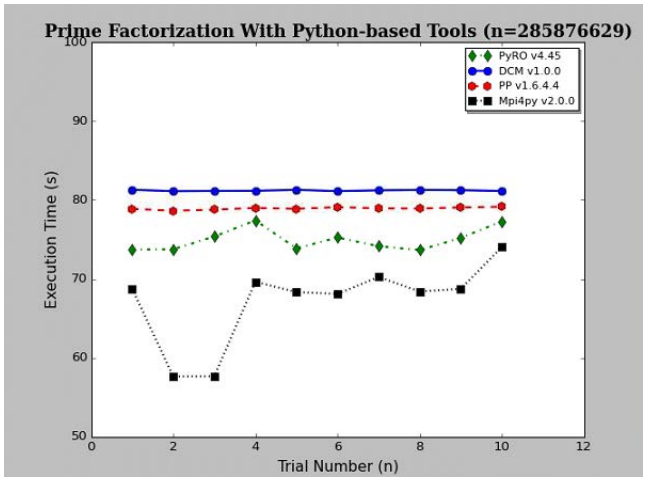

Figure 6. Prime Factorization for n=285876629 (10 run time tests/ tool).

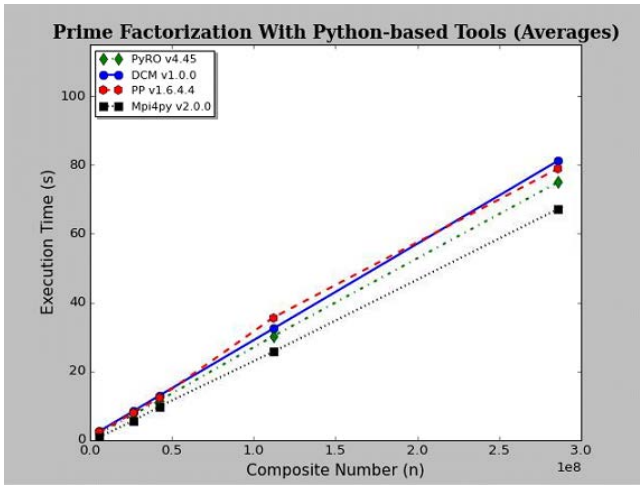

Figure 7. Prime Factorization averages for all composite values tested.

For n=5977007 (Figure 2), n=112150573 (Figure 5), and n=285876629 (Figure 6), Mpi4py consistently yields the shortest run times. Mpi4py yields the second-shortest run times for trial 8 where n=26842183 (Figure 3) and trial 5 where n=42500489 (Figure 4) – PyRO yields the shortest run time in both cases. In all cases except where n=112150573 (Figure 5), DCM exhibits the longest run times. Where n=112150573 (Figure 5), PP shows an uncharacteristically high run time in the first of 10 trials: specifically, PP yields a runtime of 73.76 s. This run time differs so significantly from the other run time values as to be considered an outlier. Using Grubb's Test for Outliers (two-sided), with a mean of 35.634, a standard deviation of 12.929, and 10 values, the value of 73.76 seconds at trial 1 is considered to be an outlier.

Table 3 showcases the standard deviations for each run time set (with the outlier from n=112150573 removed). In all tests, Mpi4py exhibits the greatest standard deviation; this suggests that times generated for load-unbalanced applications will vary the most with Mpi4py – however it is important to also note that even with high variation almost all run times are the lowest out of all tools.

Table 3. Standard Deviation for each tool and composite value

n	Standard Deviation (μ)			
	PyRO	DCM	PP	Mpi4py
5977007	0.031	0.037	0.101	0.344
26842183	0.445	0.039	0.164	1.070
42500489	0.482	0.079	0.173	1.443
112150573	1.532	0.045	0.077	2.399
285876629	1.352	0.073	0.144	5.027

Figure 7 averages all results from Figures 2-6. As observed previously, Mpi4py yields the shortest overall run times for all composite values tested and DCM yields the longest overall run times for all composite values tested except n=112150573. An approximately linear relationship is observed between composite number magnitude and run time magnitude, which suggests that these results are scalable to more computationally expensive distributed applications: future work is needed for verification.

5.2 Results from Test 2: Pi Determination

Figures 8-11 show the results of each set of 40 run time tests for specified numbers of run time iterations and Python-based tools.

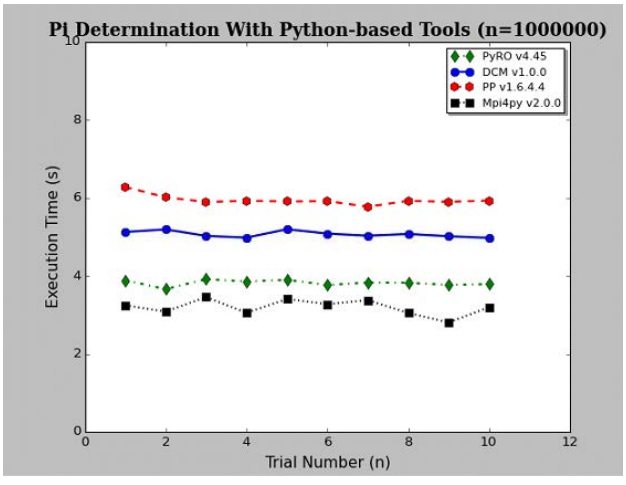

Figure 8. Pi Determination in 1000000 iterations (10 run time tests/ tool).

For all run time trials except trial 6 where n=50000000 (Figure 11), Mpi4py consistently yields the lowest run times. PP is shown to yield the longest run time in all trials and for all values of n; this differs overall from Test 1, and suggests that PP would yield the greatest run time in load-balanced tests. Furthermore, it is also suggested that DCM would yield the greatest run time in load-unbalanced tests.

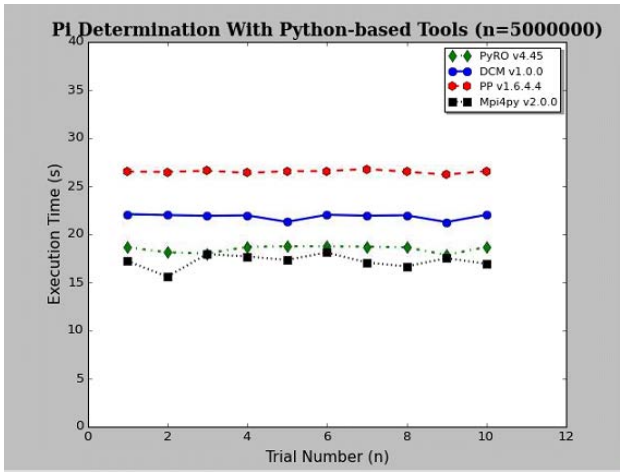

Figure 9. Pi Determination in 5000000 iterations (10 run time tests/ tool).

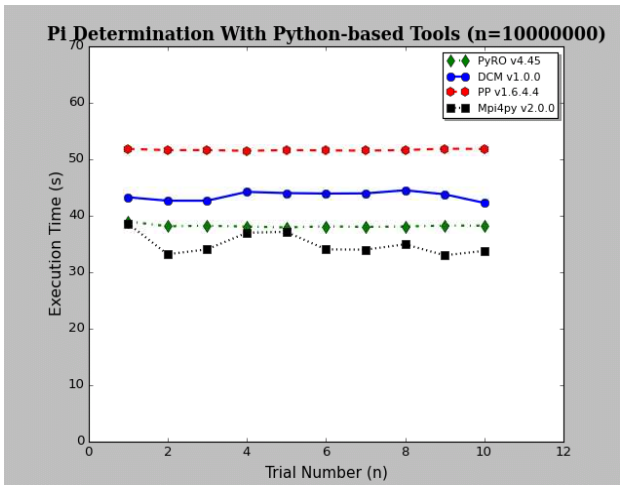

Figure 10. Pi Determination in 10000000 iterations (10 run time tests/ tool).

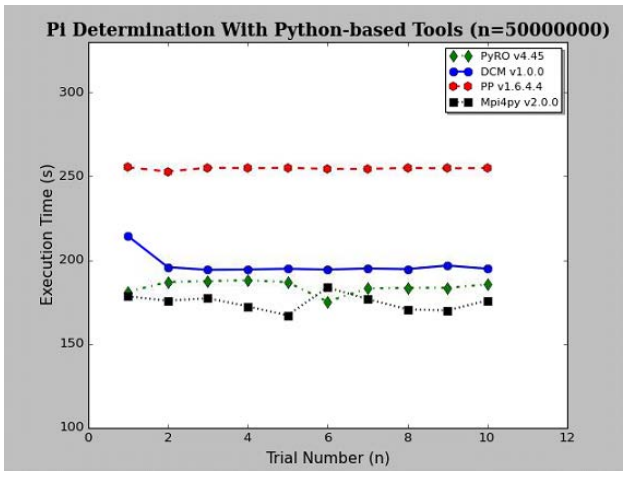

Figure 11. Pi Determination in 1000000 iterations (10 run time tests/ tool).

Table 4 showcases the standard deviation for each tool and number of iterations. In all run time tests except where n=50000000, Mpi4py shows the greatest standard deviation. DCM yields the greatest standard deviation in the set of run time tests for n=50000000 – this is due to the first trial running significantly longer than other trials in the set, and is most likely caused by the communication delay as mentioned in section 3.3 of this paper.

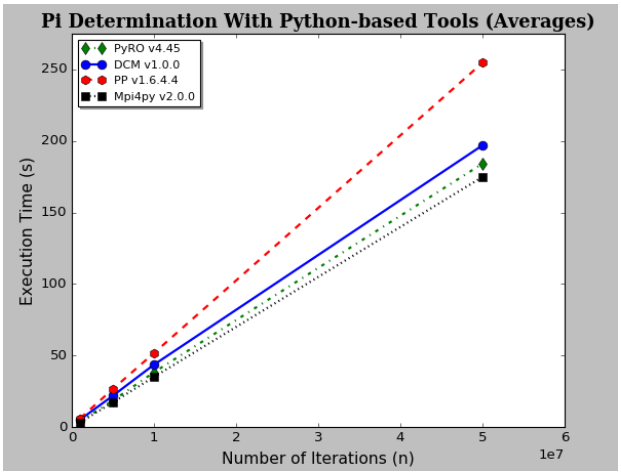

Figure 12. Pi Determination averages for all composite values tested.

Table 4. Standard Deviation for each tool and number of iterations

n	Standard Deviation (μ)			
	PyRO	DCM	PP	Mpi4py
1000000	0.072	0.076	0.124	0.190
5000000	0.331	0.289	0.143	0.687
10000000	0.277	0.723	0.128	1.830
50000000	3.649	5.859	0.669	4.579

Figure 12 averages all results from Figures 8-11. Mpi4py and PP are shown to yield the overall shortest and longest run times for all composite values tested, respectively. As observed in Test 1, an approximately linear relationship is observed between number of iterations and run time magnitude, which suggests that these results are scalable to more computationally expensive distributed applications: as with Test 1, future work is needed for verification.

5.3 Results from Test 3: Pi Determination, Varying Node Size

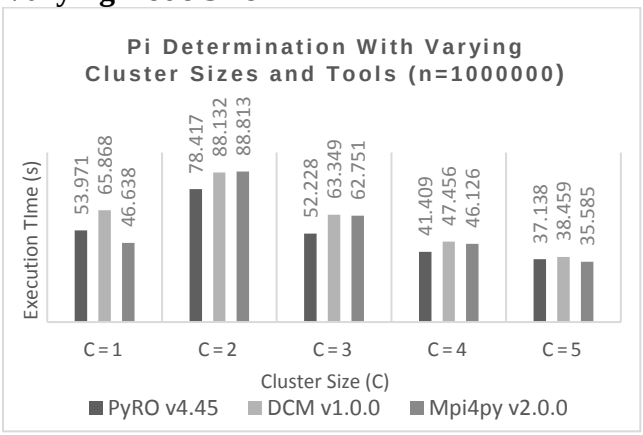

Figure 13. A summary of run time results for n=1000000 with varying cluster sizes (C=1 to C=5, where C=5 is a pseudo five-node setup).

Figure 13 shows an initial increase for all tools from a cluster size of C=1 to C=2, followed by a gradual reduction in run time as C continues to increase to 5; this phenomenon is similar to the "communication step" Dye describes in [6] (although, Dye noticed this from C=4 to C=8) – more testing is required beyond this paper to validate these results and explore the nature of this concept.

5.4 Learning Curve for Each Tool

From the experience of a student having no prior experience with distributed libraries or modules, a brief discussion on the nature of the subjective learning curve for each Python-based tool could provide an additional criterion with which to evaluate all of the tools considered. The following paragraphs describe the subjective experience for each tool.

5.4.1 PyRO

PyRO provides the most comprehensive documentation of the tools considered in our work. It contains a breadth of background theory on the major components of the library – including what a PyRO object is, and what a name server is – in addition to a number of tutorials to get started. After installing python-pyro4 through the Debian repository, it was relatively straight-forward to implement the tutorial examples. One small issue arose with a syntactical requirement which did not appear within the tutorial documentation; specifically, it was only discovered through trial-and-error that every method call for daemon initialization required the host IP address as an argument.

5.4.2 DCM

Although DCM does not provide as many samples of code or background documentation as PyRO, the samples which were provided were much more directly related to the testing which was conducted in this work – i.e. creation of a master-slave setup. Therefore, although DCM was the simplest tool to implement, it has a much narrower use case than other tools such as Mpi4py and PyRO, and thus could only be recommended for developers creating setups similar to the one featured in this paper.

5.4.3 PP

PP's learning curve was arguably the highest of the tools implemented for this research, largely because it has the least comprehensive documentation and automates many features which developers might want to customize for their projects – for example, no documentation was found for manually allocating tasks to specific slaves. Although the examples included within the documentation were relevant to the testing used in this research, when issues appeared there were very few resources to use for troubleshooting. Some resources for PP were found within [2], however.

5.4.4 Mpi4py

A difficulty with Mpi4py is that there are significantly more resources for MPI-related tools than there are for any of the other tools covered in this research; as such, when an issue with implementation of a test came up and the documentation was not sufficient for troubleshooting, there were many guides to sort through which were not found to be useful for the particular setup for this research. For the most part, however, Mpi4py has an intermediate learning curve.

6. CONCLUSION AND FUTURE WORK

Based on the results of Test 1 and Test 2, Mpi4py consistently runs more efficiently than the other three Python-based tools considered for both load-unbalanced testing and load-balanced testing. Mpi4py is therefore recommended as the optimal tool for distributed computing; this extends the work in [3] and ranks Mpi4py as the most efficient overall of the six Python-based tools featured in Table 1. It should be noted, however, that some drawbacks with the tool are apparent: First, Test 1 and Test 2 both showed that Mpi4py yields the greatest standard deviation of all tools tested; Furthermore, Mpi4py was found to have an intermediate learning curve during implementation of tests, whereas other tools such as PyRO and DCM were simpler to use.

A secondary recommendation is made for PyRO. Although DCM was found to have the lowest learning curve of the tools, PyRO offers a broader set of functionalities (DCM is meant for setups similar to the one featured in this paper); Moreover, PyRO has been shown to consistently yield lower run times than DCM in all tests conducted.

The following tasks would be prioritized in future work:

The most immediate future work to be conducted would attempt to eliminate the potential sources of error featured in this research. For example, since a pseudo five-node setup was used for testing, future work should focus on validating the results with a true five-node setup. Moreover, future work should incorporate a private LAN setup as opposed to the LAN setup used in testing and evaluation.

Other future work which could benefit this paper could include applying the methodologies used in this paper to a larger cluster size of Raspberry Pi's or to a cluster of more powerful machines; alternatively, the methodologies used in this paper could be applied to Java-based tools or C++-based tools to broaden the evaluation.

7. REFERENCES

[1] Lescisin, M., and Mahmoud, Q.H.: Middleware for Writing Distributed Applications on Physical Computing Devices. In Proceedings of the 3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft 2016), Austin, TX, USA, May 16 - 17, 2016.

[2] J. Palach, "Contextualizing Parallel, Concurrent, and Distributed Programming" in *Parallel Programming with Python*: *Develop efficient parallel systems using the robust Python environment*, 1st ed. Birmingham, UK, Packt Publishing, 2014, ch. 1, pp. 7-14.

[3] M. Lunacek, J. Braden and T. Hauser, "The scaling of many-task computing approaches in python on cluster supercomputers," *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Indianapolis, IN, 2013, pp. 1-8.

[4] I. De Jong. (2016). *Pyro – Python Remote Objects – 4.45* [Online]. Available HTTP: https://pythonhosted.org/Pyro4/index.html.

[5] L. Dalcin. (2016). MPI for Python. [Online]. Available HTTP: http://mpi4py.readthedocs.io/en/stable.

[6] B. Dye, "Distributed Computing with the Raspberry Pi," M.S. thesis, Dept of Computing and Information Sciences, College of Engineering, Kansas State University, Manhattan, Kansas, 2014.