Applications for 68 Node Raspberry Pi 3 Education Cluster

Kelley A. Deuso, Gregory Simpson, Alok Hota, Mohammad Raji, Aaron R.Young, J. Parker Mitchell, Jordan Sangid, and Mark Dean

The University of Tennessee, Knoxville TN 37996, USA
markdean@utk.edu

Abstract. In order to provide students access to a low-cost high performance computing cluster, the Big Orange Bramble (BOB) project was created. The project involved the design and implementation of a 68 node Raspberry Pi 3 cluster, for which Pi calculation, Monte Carlo simulation, fire dynamics simulation, and Spark applications were developed and installed to demonstrate the educational advantages associated with distributed and parallel computing education.

Keywords: cluster · HPC · ARM · education · Spark

1 Introduction

There is an increasing need for computational sciences education to include a focus on distributed and parallel computing, particularly with the rise of cloud computing and big data processing. For students, the prevailing limitations of hands-on access to high performance computing clusters are their associated cost, housing, and maintenance. In order to address this need, the Big Orange Bramble (BOB) project was conceived to design a low-cost, easy-to-setup Beowulf cluster that students could develop, maintain, and utilize to further their parallel computing education. The project involved the design and construction of a cluster composed of 68 quad-core ARMv8 64-bit Raspberry Pi 3s. Aside from building the cluster, the primary goals of the project were to establish the operating environment, communication structure, application frameworks, application development tools, and libraries to support a high performance cluster. Additionally, open documentation was created to provide information to anyone wishing to copy or expand upon BOB's e orts. This paper seeks to expound the development, analysis, and educational impact of the initial applications created on BOB.

1.1 Hardware Overview

In order to understand the types of applications which could be developed, it is important to note the major hardware components and their relevant limitations.

The cluster is composed of 68 Raspberry Pi 3 Model B computers whose main attributes include a 1.2GHz 64-bit quad-core ARMv8 CPU, 1GB of LPDDR2 RAM, a microSDHC slot, and on board 10/100 Mbit/s Ethernet, 802.11n wireless, and Bluetooth 4.1 capabilities. The nodes are connected through two 48-port gigabit switches with 96 Gbps switching capacity. However, the architecture of the Raspberry Pis limits the networking throughput to 100 Mbps. There is a single master node, two storage nodes, and a monitor node which is used in tandem with a touchscreen display. The master node is also connected to a display via HDMI in order to output any graphics and to allow for direct access to the cluster. A daughter card was custom designed for the Raspberry Pis to monitor supply voltage and current for each node, and this information is displayed on the monitor node's touchscreen display.

1.2 Systems Overview

All of the Raspberry Pis use Raspbian Jessie for their operating system, as this was a customized Linux environment for the Pis. However, the Broadcom BCM2837 SoC does not have support for a 64-bit kernel, so the Raspbian Jessie kernel is 32-bit ARMv7. The limitations created by this are that the memory utilization is lower than if a 64-bit kernel could be used, and the ISA and performance improvements of ARMv8 are also not able to be taken advantage of. For job scheduling and resource management, the open source Simple Linux Resource Management (SLURM) system has been implemented. For initial node setup and deployment of con guration changes, Ansible is used. There are two di erent le systems implemented on BOB: Network File System (NFS) and Orange File System (OFS). NFS is the standard le system that is used alongside Raspbian Jessie, and it is a purely distributed le system. OFS is an object-based le system which divides data and distributes it to one or more servers. Users can choose which le system they would like to use so that applications can be catered to the preferred type. Lightweight Directory Access Protocol (LDAP) has been implemented to create and maintain user accounts as well as control and log access to the cluster.

2 Applications

Various applications were installed and developed for BOB. The intent was to showcase a wide range of uses for BOB as well as to test the system components and communication. The following sections discuss why a particular application was chosen, how it was installed or developed, what the results of testing demonstrated, and any future work or improvements for the application.

2.1 Parallel Pi

The degree to which the mathematical constant Pi can be accurately calculated is often used to demonstrate the performance of modern computers. One of the simplest methods of approximating the value of Pi is through the use of a convergent series. This method is particularly applicable to the testing of BOB because subsections of the chosen summation can be assigned to any number of available worker nodes in parallel, as the resulting sum of each subsection is independent of the others. Therefore, each worker node should be able to simply compute the sum of their subsection and return the result to the master node, where all results are then summed for a nal approximation of Pi. If the cluster is operating as intended, a linear increase should be observed in runtime performance of the approximation as the number of available worker nodes increases.

Initially, BOB's Pi approximation application utilized the Leibniz formula for calculating Pi:

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

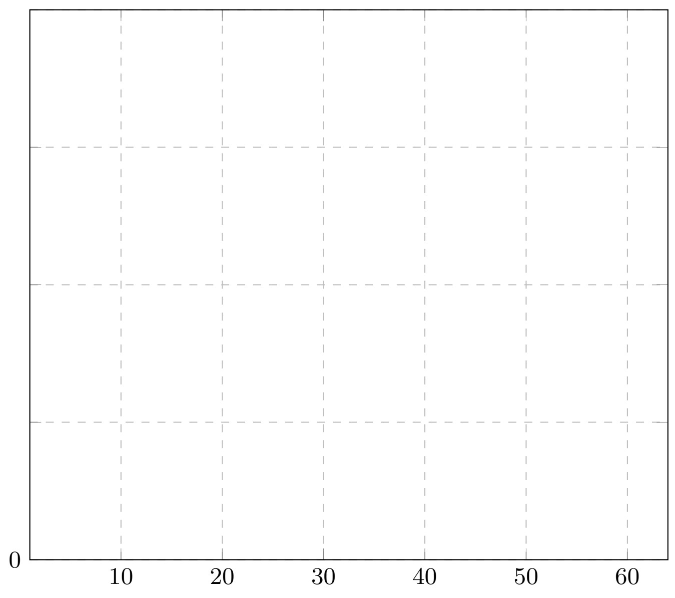
However, the formula converges extremely slowly, requiring the processing of billions of terms to achieve a correct approximation of only 10{12 digits. Eventually, the Bailey-Borwein-Plou e (BBP) formula, originally discovered by Simon Plou e, was chosen to replace the Leibniz formula due to its much faster convergence rate.

$$\pi = \sum_{i=0}^{\infty}\frac{1}{16^i}\left(\frac{4}{8i+1}-\frac{2}{8i+4}-\frac{1}{8i+5}-\frac{1}{8i+6}\right)$$

The BBP formula, in combination with Python's decimal module, which allows users to specify oating-point precision, resulted in an approximation of Pi that was accurate to thousands of decimal places in a fraction of the time required for Leibniz. Performance results will be discussed in detail in the Testing and Results section below.

Testing Results

Parallel Pi results were obtained from test runs using between 4 and 256 parallel processes, with each test using twice the number of processes of the previous test. As the division of the series into roughly equal subtasks is considered embarrassingly parallel, positive scaling results were anticipated. Test results are presented in Figure 1. Please note that the scale of the x-axis is logarithmic.


0
10
20
30
40
50
60

applications for BOB. The main goal of this application was to demonstrate proof-of-concept and test basic Python programming and SLURM features. The code for Parallel Pi is also available for any user who wishes to see a simple example of Python and SLURM functionality for BOB. There are currently no future plans for Parallel Pi. However, the successful results of this application led to the idea of an automated script program to run multiple iterations of a single program across the cluster, which has been called the Monte Carlo application due to its original purpose of running Monte Carlo simulations.

2.2 Monte Carlo

A Monte Carlo simulation is a method of running a program with random values for variable(s) in order to create an overall model of the program's output. This type of simulation is useful for many applications, speci cally for those in physics and mathematics which involve probabilistic variables, such as the Gillespie algorithm. The application can be run from either the command line or a supporting GUI and allows the user to specify the input program le, total number of runs, number of nodes to use, and maximum run time. The application supports programs written in C/C++, Python, and Octave at this time.

GNU Parallel

Initially, GNU Parallel was installed in order to launch a single Monte Carlo job which would run each iteration in parallel. GNU Parallel works with SLURM to handle any interrupts during the runs, as well as appropriately assign jobs to nodes. The idea was that one script could be created based on the user's input parameters, and using GNU Parallel, a single command could parallelize and manage all iterations. Although this seemed to be a straightforward approach, it failed to work properly during testing. For example, a batch script was created invoking the `parallel` function from GNU Parallel, but after the script would run once properly for a user, it would then not work again. In fact, all scripts invoking GNU Parallel afterwards would no longer work for that user. An example of this script is shown in Figure 2. It should also be noted that in the testing of this method, errors occurred from the Perl code present in this script. The Raspberry Pis were all using the Great Britain language packages which were the default settings during setup. Therefore, the United States standard unicode had to be updated across all nodes.

```
#!/bin/bash
#SBATCH -n 12
#SBATCH -o ot.txt
#SBATCH -t 12:00:00

srun="srun -n1 -N1 --exclusive"
parallel="parallel --delay 1 -j 30 --joblog test.log --resume"
$parallel "$srun ./a.out"
```

Fig. 2: An Example of a Batch File Using GNU Parallel

Multiple Runs in Single Batch File

Rewriting the Monte Carlo application to create multiple runs in a single batch le is the current solution to the GNU Parallel issues. This implementation assigns a single simulation per core or node as speci ed by the user, and if the desired number of runs is greater than 64, the total number of worker nodes, it then divides up the total number of runs into chunks according to the number of available cores, creates an `srun` call per chunk, and then submits the le as a job. Each `srun` appends output to a single le utilized by all batches speci ed with the `--output` ag. This is the same le that the user speci es either on the command line or in the GUI for output. The output le and the standard error le will be stored in the same directory that the script is submitted. An example of a batch le which runs 100 simulations on 8 CPUs is shown in **Figure 3**. While this method seems more brute force than the GNU Parallel solution, SLURM should still manage any interrupts. Slurm manages race conditions for the output le and only appends output as each job completes. Because of this, it should be noted that if a particular set of simulations prone to inde nite computation does not have any time constraints enforced upon it, it is likely that output will not be produced and the tasks will have to be killed manually. The maximum run time parameter becomes important in this situation as well as when the cluster is being shared with other users.

The testing of the Monte Carlo application focused on ensuring appropriate scalability. The program that acted as the user executable was a simple Python dice roll program which printed a random number between one and six. The blue line in the **Figure 4** shows the results of 100 random dice rolls on 4, 8, 16, 32, and 64 CPUs. The orange line shows the same test except with 1,000 random dice rolls. The variation using 4 CPUs for the two tests is large with the 100 rolls running approximately 3.13 seconds, while the 1,000 rolls clocks at 31.46 seconds. Both tests converge towards 64 CPUs, yet it is clear that the 1,000 runs bene ted more from additional CPUs than the 100 runs. This can be seen in **Table 1** where from 32 to 64 CPUs, the 100 runs only speeds up 8.3% compared to the previous 38.1% speedup from 16 to 32 CPUs. The total speedup from 4 to

```
#!/bin/bash

#SBATCH -N 8
#SBATCH --job-name=montecarlo_dice_roll
#SBATCH --output=dice_output.txt
#SBATCH --open-mode=append
#SBATCH --cpus-per-task=1
#SBATCH -e montecarlo_dice_roll_err.txt
#SBATCH --time="00:00:30"

srun -n 4 python dice_roll.py
srun -n 32 python dice_roll.py
srun -n 32 python dice_roll.py
srun -n 32 python dice_roll.py
```

Fig. 3: An Example of Multiple Runs in Single Batch File

64 CPUs for 100 runs is 82.2%. For the 1,000 runs, the speedup is consistently around 50% for all additional CPU tests. The speedup is 93.2% from 4 to 64 CPUs for 1,000 runs.

# Nodes	100 Dice Rolls Run Time (seconds)	% Speedup	1,000 Dice Rolls Run Time (seconds)	% Speedup
1	11.7004	-	113.4626	-
4	3.1275	73.2702	31.4619	72.2712
8	2.1206	32.1947	14.8163	52.9072
16	0.9785	53.8559	7.76318	47.6037
32	0.6057	38.1015	4.1045	47.1290
64	0.5554	8.3066	2.1351	47.9801
$1 \rightarrow 64$	-	95.2533	-	98.1182

Table 1: Test Results for Monte Carlo Application

This test data re ects other test results that have been shown on BOB. The communication bottlenecks tend to mostly a ect smaller programs, which most likely do not need to be parallelized, and programs that utilize a large number of reads and writes. With this Monte Carlo application, results are likely to vary depending on the user speci ed executable. However, the results show improvement with the addition of CPUs for these test runs.

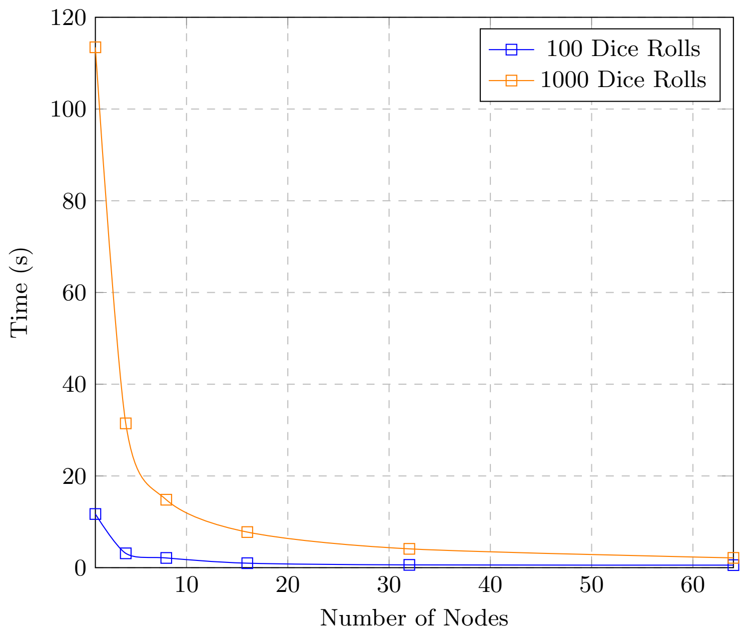


Fig. 4: Monte Carlo Scaling Results

Educational Impact

The creation of this application was targeted speci cally to make initial use of the cluster simple for any level of programmer. It eliminates the need to immediately understand SLURM commands, but it displays the output script results so that a user can ultimately see what it has created and learn these commands. Initial exposure to parallel computing can be daunting, and trying to create programs which run in distributed manner can be a challenging task for many. This application allows most any C/C++, Python, or Octave program which natively is not parallelized to produce many output results within the same real timeframe. Users can take advantage of this application for simple testing of self-contained programs, that is, those not requiring user input, or for running their own stochastic simulations.

Additionally, the application is currently being used to develop an expansion Monte Carlo application to implement the Gillespie algorithm. The Gillespie algorithm is one broadly used in biology, chemistry, and even economics for stochastic simulations. In the 1970s, Dan Gillespie and Joseph Doob created the Gillespie algorithm to simulate chemical systems under the limited computational power of the time. The algorithm's innovation lies within the discrete and stochastic method of simulation utilizing few reactants without having to iterate in real time due to its Monte Carlo steps. The Gillespie algorithm ultimately creates an accurate model of a solution for the speci ed master equation set. This expansion will allow users to create their own set of master equations to test, tweak, and analyze on BOB.

2.3 Fire Dynamics Simulator

Fire modeling and uid dynamics simulations are traditionally intensive computing applications due to the large nature of both the physical structures, such as whole houses or buildings, and the uid dynamics calculations of re ow. Because of this, new computing methods and architectures are worth looking into to determine whether the complexity can be lessened in order to reduce run time, particularly in situations where time is a factor, like life-threatening circumstances or lawsuits.

The National Institute of Standards and Technology (NIST) has developed an open source re simulation software re dynamics simulator (FDS) which aims to reduce computational and time complexities by using a technique which divides sections of the computation into meshes. A mesh is a section within a domain, e.g. a room or building, that is made up of rectilinear volumes. Then each mesh is divided into cells, the number of which depends on the speci ed resolution for the simulation. (FDS User Guide) Each mesh can then be assigned to multiple processors in parallel, which can signi cantly reduce the execution time necessary to create a complete model.

This portion of this project builds upon the work put forth by Donald Collins in his master's thesis for The University of Tennessee, Knoxville.[1] Collins' research focused on ways in which FDS modeling work could be e ciently and quickly distributed across multiple processing cores. His work included a Python script to evenly divide a single mesh into multiple meshes for insertion into an FDS le. Fundamentally, his work allowed for one complex task to be subdivided for execution in a parallel environment, ideal for an application on BOB.

Testing Results A simple FDS le which models a room with a re and an HVAC system was used for testing scalability across the cluster. The original le contains a single mesh encompassing the entire room. The mesh properties were entered into the `mesh_slicer.py` program which generates series of test les. The les generated include one mesh, two meshes, four meshes, 16 meshes, 32 meshes, 64 meshes, and 128 meshes. These numbers were chosen to make it easier to evenly divide the original mesh while still providing adequate data points to test performance scaling.

Figure 5 and Table 2 show the results of the tests that were run on the Raspberry Pi cluster above. Although the increase in performance from one to four meshes is signi cant, there is little advantage in running it on more cores on the cluster. In fact, shortly after 16 cores the networking overhead of the cluster overwhelms the performance gains of more cores.

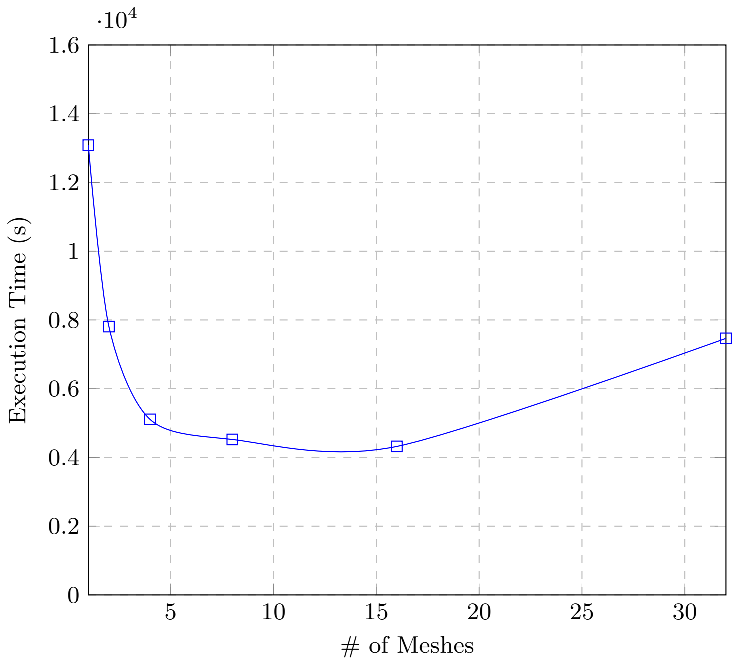


Fig. 5: Graphical results of tests dividing a simple room into multiple meshes on the Raspberry Pi cluster.

These results are not entirely unexpected, and in fact, they re ect the results that Salter experienced in his research on distributed computing with FDS on cloud-based resources. The advantage of a Raspberry Pi cluster is that it provides a large number of cores for a relatively inexpensive price. The cores themselves are computationally weak compared to traditional x86 cores. Each core adds a performance increase as well as an overhead cost, and because the Raspberry Pi cluster is limited to a 48-port switch, there is an unclear bottleneck past 48 Raspberry Pis. While the FDS overhead cost is virtually the same between x86 and ARM cores, the performance per core is not. This explains why the Raspberry Pi cluster's scaling ability plateaus more quickly than a typical multicore x86 system.

	Total Elapsed
Number of Meshes	Wall Clock Time (s)
1	13084.685
2	7807.769
4	5106.836
8	4522.845
16	4320.457
32	7464.152

Table 2: Tabular results of tests dividing a simple room into multiple meshes on a Raspberry Pi cluster.

Educational Impact

Because re and uid dynamics simulations are complex and often can take weeks to complete, students seeking to implement re and uid dynamics models have severely limited access to clusters where they can run time intensive simulations. BOB addresses this need as it is shown to handle large computations well, and because BOB is a locally owned cluster, students have the access they would need without time or monetary constraints. Unlike the Parallel Pi and Monte Carlo applications, FDS is an open source package which was installed and tested on BOB. This is crucial to the purpose and development of BOB in that students are able to install and utilize existing software, such as Octave or Spark, that is essential their distributed systems and parallel programming educations. Currently, FDS is limited to an older 32-bit version due to BOB's operating system constraints. However, potential hardware improvements on BOB may allow for a 64-bit version to be installed and tested.

2.4 Spark

Apache Spark provides an abstracted interface to parallel programming, allowing for applications to automatically scale across many nodes. Spark is commonly installed on HPC systems around the world including premiere machines, such as Titan at ORNL [5], Cori at NERSC [2], and Stampede at TACC [3]. Such machines are characterized by having many cores and a large amount of memory per compute node. However, this class of machine often has restricted access, and utilization costs may be expensive to maintain.

Spark was simple to set up and launch on BOB. To keep logins uninterrupted, the Spark environment was set up such that the rst worker node acted as the master. Any number of the remaining 63 nodes could act as Spark workers, controlled by the master. For testing the system, two Spark applications from other research projects were used. Though each node is much weaker in compute power compared to, for example, Titan nodes, good strong scaling is achievable with compute intensive tasks.

Spark Applications

Two in-house research projects were used for testing. Both projects perform segmentation on data and do a large number of comparisons among the segments. The two projects can be summarized as:

A. An image-space analysis tool that compares spatial data across time
B. An ensemble analysis tool that compares volumes across time and ensemble runs

Though both projects perform comparisons, they have vastly di erent runtime requirements. These requirements stem from the type of processing they do on segments of the data and the method in which the result is output.

Project A performs comparisons of multivariate spatio-temporal data in image space. The temporal space is represented as a stack of two-dimensional images. The master node performs a downsampling (unweighted averaging) on each image based upon a user-con gurable input parameter. Pixels in the downsampled images represent an $n \times n$ region from the original image. The downsampled images are sent to all workers. Each worker then performs a comparison between one pixel in the downsampled image space against all other pixels, similar to a Cartesian product. The comparison is based upon a user-de ned comparison function that quanti es a distance metric between partitions. The resultant value from the comparison function for each pixel is stored in a new image.

Project B performs comparisons of three-dimensional univariate ensemble data { multiple output sequences from a simulation { to remove duplicate regions. Worker nodes load a single volume into memory at a time to be segmented into equal-sized blocks. Blocks are hashed based on the values contained within them, and hash values are compared. Matching blocks are removed in favor of a single reference to the original block and receive a single block ID to identify them. This process essentially performs deduplication on volumetric data. Each worker produces a table mapping block IDs to data and a grid that de nes a blueprint for rebuilding a single volume. These are further reduced again to consolidate matching block IDs across volumes on the master.

Project A is a highly compute-intensive task. Since the images are downsampled before being distributed, they remain manageble within the Raspberry Pi's limited memory. For example, with a partition size of 4×4 pixels, each image being distributed is 16 times smaller than the original. Additionally, the output for each node is always an image of the same dimensions as the incoming downsampled image. However, the comparison function can be arbitrarily complex, and can operate over one or many time steps. Thus there is low overhead to distributing the image data to all workers and a high utilization of CPU cores per worker.

Project B on the other hand is much more limited by memory, as larger volumes need to be loaded into memory for analysis. Each volume requires a grid representation; for smaller block sizes, the grid is much denser and requires much more memory. The output for each node also includes a hash table mapping hashed block IDs to block data. While the hashes themselves are small, and the blocks are stored in a transformed state, the overhead from the data structure and the grid representation may exceed the original data's size if the degree of deduplication is low.

Scalability

The compute intensive Project A fared much better, showing good strong scaling. The image data was more manageable under limited memory conditions and required a larger amount of processing per segment. Project B failed to run on BOB due to its high memory requirements. This was somewhat expected as Project B was designed to operate on terascale datasets with large volumes.

Figure 6 shows how Project A scales based on the number of nodes used. Below 16 nodes took over two hours and did not complete within a reasonable amount of time. The parameters used were 20 time steps with a 4×4 grid size for downsampling.

There were several considerations made to run successfully with Spark within the constraints of the Raspberry Pis. The rst was Spark environment values. The driver memory and executor memory values, for the master process and worker processes, respectively, had to lowered to t within the 1 GB memory constraint. Both values were set to 512 MB. Note that for executor memory, this is the maximum amount of memory to be used per process. There are four processes per worker node utilizing the quad-core CPU, which would sum to 2 GB maximum utilization. However, Spark attempts to keep memory under a certain limit, called the $safety\ fraction$, based upon the requested maximum. Setting values lower than 512 MB per process would result in low utilization per core as the tasks become too small, while at 512 MB full utilization can be achieved without incurring an out-of-memory crash.

Second, the number of partitions was increased for the submitted Spark job. By default, Spark creates one partition per core in the set of worker nodes being used. In BOB's case, with 63 worker nodes, this defaulted to 252 partitions. However it is recommended to partition to two or three times the number of

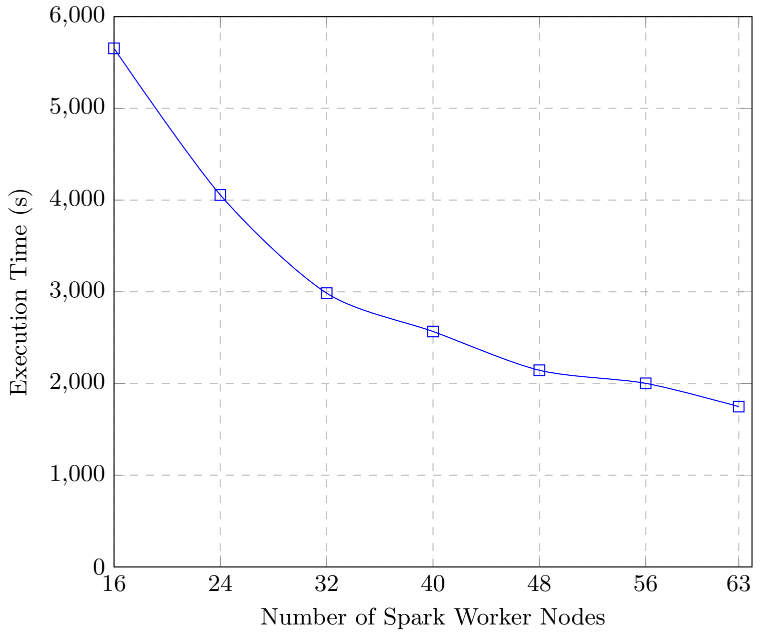


Fig. 6: Strong scaling curve for Project A running in Spark on BOB.

cores. This resulted in large tasks that did not t in memory, so the number of partitions was increased to eight times the number of cores. Note that this still showcases strong scaling when changing the number of nodes. Though each core is assigned eight tasks, the actual amount of work done per task changes because the problem size remains static.

Lastly, all calls to `collect()` or `toLocalIterator()` were removed, both of which perform a data collection operation. That is, data is transferred from worker nodes to the master node's memory. In either case, this requires a large overhead that could not be a orded. Instead, the parallel le system was taken advantage of and results from the workers nodes were saved in parallel. For Project A, this mean saving images to disk in parallel. However Project B required that the results from worker nodes be reduced, which multiplied the size of the working set.

Educational Impact

Apache Spark provides a highly abstracted interface for parallel programming. It allows for code to be written in Python, Java, or Scala using a pipeline structure. Paralellization is automated across a user-de ned number of nodes with no need for the user to manage communications. This presents an opportunity to educate students on parallel programming methods without involving any boilerplate code or environmental setup. Additionally due to the lower sunken cost of procuring a Raspberry Pi cluster, there is much wider access to high performance parallel programming platforms.

3 Conclusion

Because of the success of BOB, an expansion cluster is underway which involves combining BOB with 32 Pine64s and 12 Nvidia TX1 GPUs to create a truly heterogeneous cluster. Furthermore, applications development is now targeted at weather forecasting, facial recognition, and as previously mentioned, the Gillespie algorithm addition to the Monte Carlo application. BOB has also recently been participating in the BOINC from the University of California, Berkeley, which allows BOB to be used for scienti c research when it is not being utilized.

BOB has demonstrated a variety of capabilities through these applications, and it is with certainty that the build and setup of a BOB-like cluster is an educational task on its own. The total cost of equipment for this cluster is estimated around $5,000, a signi cant cost reduction in comparison to proprietary servers like the SuperMicro 3U, 12-node MicroCloud SuperServer, for which the chassis alone costs the same as all the components for BOB combined [4]. A parts list, all documentation, and links to BOB's code repository can be found at `http://web.eecs.utk.edu/~markdean`. The Big Orange Bramble project sets a precedent for computer and information science students to have access to local, distributed architecture. There is no longer a limitation of access to high performance clusters in education, and the bene ts of BOB are momentous for modern day computing education.

References

1. Collins, Donald Charles. “Dividing and Conquering Meshes within the NIST Fire Dynamics Simulator (FDS) on Multicore Computing Systems.” Thesis. The University of Tennessee, Knoxville, 2015. *TRACE:Tennessee Research and Creative Exchange*. Dec. 2015. Web. Dec. 2016.
2. “Cori.” *National Energy Research Scientific Computing Center*. 9 Nov. 2016. Web. 15 Dec. 2016.
3. “Stampede” *Texas Advanced Computing Center*. N.d. Web. 15 Dec. 2016.
4. “SuperMicro 3U 12 Nodes MicroCloud SuperServer - 5038ML-H12TRF.” *Rackmount Pro*. N.p., n.d. Web. 14 Dec. 2016.
5. “Titan Cray XK7.” *Oak Ridge National Laboratories*. N.d. Web. 15 Dec. 2016.

4 Acknowledgements

The authors would like to thank Caleb Williamson, Shawn Cox, Taher Naderi, Patricia J. Eckhart, Liu Liu, Chengcheng Li, and Ellias Palcu for their research and contributions to this project.