Regular Research Paper - The Abaco Platform: A Performance and Scalability Study on the Jetstream Cloud

Christian R. Garcia, Joe Stubbs, Julia Looney
Anagha Jamthe, and Mike Packard
Texas Advanced Computing Center
University of Texas at Austin
Austin, TX 78758
Email: [cgarcia, jstubbs, jlooney, ajamthe, mpackard]@tacc.utexas.edu

Kreshel Nguyen
Computational Engineering
Dept of Aerospace Engineering
University of Texas, Austin
Austin, TX 78712
Email: kreshel@utexas.edu

***Abstract*—Abaco is an open source, distributed cloud-computing platform based on the Actor Model of Concurrent Computation and Linux containers funded by the National Science Foundation and hosted at the Texas Advanced Computing Center. Abaco recently implemented an autoscaler feature that allows for automatic scaling of an actor's worker pool based on an actor's mailbox queue length. In this paper, we address several research questions related to the performance of the Abaco platform with manual and autoscaler functionality. Performance and stability is tested by systematically studying the aggregate FLOPS and hashrate throughput of Abaco in various scenarios. From testing we establish that Abaco correctly scales to 100 JetStream "m1.medium" instances and achieves over 19 TFLOPS.**

***Keywords*—functions-as-a-service, autoscaling, containers, cloud computing, performance engineering.**

I. Introduction

Abaco (Actor BAsed COntainers) [1] is a hosted, functions-as-a-service Application Programming Interface (API) which combines Linux container technology with the Actor Model of Concurrent Computation[2]. Also referred to as a "serverless" platform, Abaco enables the rapid development of reactive, event driven architectures requiring minimal long-term maintenance. Funded since 2017 by the National Science Foundation, Abaco serves the national research community with the ability to run high throughput, low latency workloads concurrently for faster execution. In its first two years in production, Abaco has been adopted by several projects and has supported the invocation of over 70,000 functions collectively running for more than 20 million seconds.

Users register new actors in Abaco by making an API request that includes a reference to a publicly available Docker image to use for the actor; in response, Abaco generates and returns a URI associated with the new actor. Users can then use the URI to send messages to the actor. For each message sent to an actor, Abaco puts the message on an internal queue assigned to the actor, referred to as the actor's "mailbox" or "inbox". For each queued execution, as Abaco's internal compute resources become available, the system launches a container from the actor's image and injects the original message data into the container. Most Abaco executions start within one or two seconds of a message being sent; however, in some cases, users will queue tens of thousands of messages in a short period of time for a single actor, in which case, the executions will run over several hours or days.

The resource requirements of workloads on Abaco can vary greatly depending on the nature of the computation being performed. Abaco provides users with different ways of interacting with the API to facilitate executions. In particular, users have the option of manually scaling the resources assigned to an actor, or using Abaco's autoscaling feature to dynamically scale resources based on the actor's mailbox size. In this paper, we set out to experimentally determine the differences in performance between manual scaling and autoscaling under two different work loads, and metrics: "FLOPs", which determines the amount of work that can be achieved by a system at a given time and "hashrate", which is a measure of performance that became popular with the emergence of Bitcoin [3] technology.

A. Abaco Background

The Abaco system uses internal agents referred to as "workers" to facilitate the processing of actor messages. When a worker is created, it is assigned to exactly one actor, and it subscribes to the internal Abaco queue corresponding to the actor's mailbox, defined in RabbitMQ[4]. In response to receiving a message intended for its assigned actor, a worker starts an actor container from the defined image associated with the actor and injects the message data into the container, either in an environment variable in the case of text data, or a unix domain socket in the case of binary data. The worker then supervises the actor execution, monitoring for the actor process to exit, and collecting resource usage and log data along the way. A given worker does not retrieve a new message from the actor's queue until the current execution is finalized. It follows that, for a given actor, the number of messages being processed concurrently at any given time is no more than the number of workers assigned to the actor. In particular, an actor with one worker only processes one message at a time.

In many applications, it is critical that messages are processed sequentially, and Abaco formalizes this notion through a property called "stateless" provided at actor registration. If an actor is registered with the "stateless" property set to False, Abaco will never start more than one worker for the actor. This feature distinguishes Abaco from many other functions-

as-a-service offerings and is an important aspect of the Actor Model more generally.

However, for actors registered with the stateless property set to True, Abaco can increase or decrease the number of workers associated with an actor. Abaco provides an endpoint in its HTTP API that users can use to create additional workers or shutdown existing workers for actor's they have access to. This approach is referred to as "manual scaling".

B. Autoscaling

The Abaco autoscaler will automatically scale the number of workers associated with an actor according to the size of the actor's mailbox. The autoscaling will scale up when an actor has at least 1 message in its inbox. Similarly, when the actor has 0 messages in its inbox, the autoscaler will shut down workers for that actor that are not currently overseeing a running execution. There is also a limit to the number of workers an actor can have. If this limit is reached, then Abaco will stop the scale-up process until the number of workers for that actor has reduced below the limit.

The autoscaling in Abaco has been developed on top of Prometheus, an open-source time series database and monitoring server [5]. Prometheus functions by scraping plain-text values that it converts into time series data. Prometheus can be configured to repeatedly scrape values from specific APIs on a given time interval. The algorithm for autoscaling is as follows. The Abaco metrics API endpoint produces plain text values of the current actor inbox sizes, as well as how many workers are assigned to each actor. Prometheus scrapes the instantaneous mailbox sizes using the Abaco metrics endpoint every 5 seconds. The autoscaler then uses this data to determine whether it should scale-up, scale-down, or neither. This allows Abaco to provide more resources to those actors which require it, while removing resources once they are no longer needed. If an actor has at least one message, but it has reached its limit of workers, then neither scale-up or scale-down will occur.

II. Experiment Design

Harnessing the full potential of high performance computing at a user level has become increasingly difficult as the field has grown in scale and complexity. The Abaco platform attempts to alleviate this issue by giving users an additional tool in HPC. This study seeks to answer the research questions below by analyzing various performance metrics in order to better understand the use, potential, and limitations of the Abaco platform.

A. Research Questions

- What is the difference in worker creation rate between a manually scaled worker pool and using the autoscaler?
- What is the performance of Abaco at different node sizes and how does it compare to the theoretical limit of the hardware utilized?
- What are the scaling limits of the Abaco platform and what are the causes to those limits?

B. Experiment Overview

Our experiment compares the performance of the Abaco autoscaler versus manually scaling actor's worker pools. Each test run consists of three main tasks. First we must create an actor, second we must create workers for that actor, and third we must run the actual computations for that actor. Actor creation is identical no matter whether the autoscaler capability is used or not. Worker creation however varies and can be measured separately in order to quantify the scaling rate between manual scaling and autoscaling. The actual computations are also possible dependent variables which can be measured using two sets of performance metrics and allows us to measure any differences due to workload or work type. Additionally, we measure how different hardware allocations affects the results by making use of Jetstream to run tests on different node sizes.

C. Performance Metrics

In this section we describe the performance metrics utilized, FLOPS and hashrate, along with the theoretical bounds used in test setup.

1) FLOPS: The four basic mathematical floating point operations performed by computers are: addition, subtraction, multiplication, and division. The amount of above operations performed on a per second basis is referred to as floating-point operations per second, abbreviated as FLOPS. FLOPS of a computer system acts as a metric for the amount of work a computer can perform in a given time. In this experiment we use the amount of FLOPS to compare different testing scenarios and gauge system overheads, slowdowns, and most importantly, scalability.

There are two parts to the FLOPS evaluation experiment. First, a large scale distribution of relatively quick work is set up to test the autoscaling performance of the platform. Second, a smaller scale distribution of relatively slow work is set up to achieve peak FLOPS by removing overhead in worker deployment.

The "work" mentioned above is a Python script that when executed takes three inputs: the amount of threads the script should utilize, standard deviation for randomizing a square matrix, and the size of the two square matrices. The script then calculates the dot product of these two random square matrices as the "work of the experiment". This script is packaged as a Docker Hub image at `abacosamples/abaco_perf_flops`[6].

To calculate the amount of FLOP in each execution we use equation 1, where S is the input size, described above. This equation calculates the amount of FLOP in a dot product operation between two square matrices[7].

$$\text{FLOP} = (S + S - 1) * S^2 \tag{1}$$

From there, in order to calculate the speed of one whole trial we use equation 2 to calculate FLOPS. We do this by taking the amount of FLOP per executions, multiplying it by

the amount of executions, E, of each trial, and dividing by the amount of elapsed time, t, from trial start to end.

$$\mathrm{FLOPS}_{\mathrm{real}} = \frac{(2S^3 - S^2) * E}{t} \tag{2}$$

2) Hashrate: The second performance metric used in our experiment is hashrate. Hashrate is the number of hashes per second, where a hash is one iteration of a SHA256 hashing function, which creates a hashed block [8]. This measure of performance became popular with the rise of Bitcoin[3], where mining a block with certain parameters gave a reward in the form of bitcoins. Without using an identical hashing function, language, and algorithm to Bitcoin, which is proprietary, or some other proof-of-work function, we are unable to compare our hashrate with the results from other computers. Instead we will hash a set amount of blocks and compare results gathered through the testing.

To actually calculate hashrate of the system, we register an actor using the `abacosamples/abaco_perf_hashrate`[9] image, available on Docker Hub. This image contains a script that runs Python's `hashlib.sha256()` function a given number of times. Each iteration of this `hashlib.sha256()` function is a hash of a block. Our experiment runs each execution running until three million hashes are calculated. Hashrate is given in equation 3, where Hr is hashrate, h is hashes, and t is elapsed time from trial start to end.

$$\mathrm{Hr} = \frac{h}{t} \tag{3}$$

D. Obtaining Theoretical Bounds

While we have an established method of obtaining FLOPS and hashrate, we're still in need for a result to compare our data against. With an addition of a comparative result, we gain access to additional insights on system overhead and the real limits of our testing.

We use equation 4 to obtain the theoretical limit of FLOPS for a single server [10]. In this equation N is the number of cores per CPU, F is the average frequency of these cores, and O is the operations per cycle for each CPU.

$$\mathrm{FLOPS}_{\mathrm{theory}} = \mathrm{N}_{\mathrm{cores}} * \mathrm{F}_{\mathrm{avg}} * \mathrm{O}_{\mathrm{cycle}} \tag{4}$$

Plugging into the above equation knowing that the Jetstream server cluster is using 6 cores of a v3 Intel Xeon E5-2680 turbo-boosted to 3.30GHz and running 16 operations per cycle gives us a theoretical max speed of 316.8 GFLOPS per node. This number now acts as the theoretical fastest speed that our servers could ever possibly achieve.

Along with a theoretical limit for FLOPS, we also have a practical limit, which is the speed of our experiment script running solely on a Jetstream compute node. This is the practical limit for our experiment and any change from that number would be caused by overhead from Abaco, Docker, Python, networking, etc.

Hashrate on the other hand is determined empirically and thus does not have a theoretical max that we can calculate. Instead, similar to FLOPS, we run our hashrate script solely on a Jetstream compute node, which allows us to compare and view the effects of system overhead.

III. Experimental Setup

We divide the experiment setup into two parts: deployment and validation. We first deploy the servers that run Abaco, it's components, and the compute nodes that Abaco will utilize to spawn new workers on. Secondly, we run the test suite to conduct the performance studies. Our entire testing repository, along with Docker images is hosted on Github, `TACC/abaco-autoscaling`, with READMEs detailing the exact instructions to reproduce the experiment. In this section we describe an overview of the experimental process so that users can implement this method in their workflows.

A. Resource Configuration

We begin with configuring the resources needed for this experiment. All servers are hosted by the NSF's scalable cloud system for XSEDE, Jetstream [11]. This service allows for the creation and configuration of virtual machines (VMs), or "nodes", which gives our experiment the ability to scale. Jetstream uses Openstack [12] for resource management and gives us the ability to deploy servers through the command line interface (CLI). Although this paper will continue to reference the resources used, it's important to note that the Abaco platform is capable of running on any cluster of Linux nodes.

All nodes have the following specifications: CentOS Linux version 7.6.1810, kernel version 3.10.0-957.5.1.el7.x86_64, Docker version 18.09.5, and Docker Compose version 1.24.0. All Abaco nodes are Jetstream m1.quad nodes which have 10 GB RAM, 20 GB SSD storage, and 4 vCPUs. A vCPU in this case is one core of a Intel Xeon E5-2680 v3, which can turbo-boost from 2.50GHz to 3.30GHz.

B. Resource Deployment

In this section we describe the automated deployment process used by the test program to create the cluster of nodes and install the Abaco software exercised by the test trial. At a high-level, deployment consists of 1) creating 5 OpenStack instances for each of the dedicated Abaco components (MongoDB, Redis, RabbitMQ, Prometheus, and the Abaco web services), 2) creating a number of Openstack instances corresponding to the cluster size of the trial (these are the Abaco "compute nodes"), and 3) installing and starting the services.

All scripts used by the test program for automating the deployment are maintained in the Github repository for this project [13], within the `/deployment` folder. The `README.md` file, included in that directory, provides an in-depth description of all of the scripts available for the users.

To simplify the process of running multiple trials, the performance test suite git repository was designed to be cloned

to a single persistent instance running within the Openstack network where the node clusters for the different trials will be provisioned. Scripts that make use of the OpenStack CLI to start or stop instances require an OpenStack authentication. The root directory of the repository contains a shell script, `openrc-script`, which prompts the user for a password when run using command `./openrc-script`. This script sets environment variables that will authenticate a user with OpenStack. It's worth noting that an entire Abaco installation can run correctly on just one node, but for the experiment, we chose to separate Abaco's components across five separate nodes for resource distribution and debugging purposes.

Additionally, the test scripts make use of a few pre-defined Openstack instance images. Using pre-built images that included base software such as the Docker container runtime, docker-compose, etc., significantly reduced the overall rntime of the test suite across multiple trials, as the instances were burned down and recreated between runs. The images used by the tests are available on the JetStream system, and we can make them available to import to other OpenStack clusters upon request.

The Abaco system is not ready for use until the compute nodes are created and ready for use. Creation and management of the compute nodes was also automated with scripts in the `/deployment` folder. For example, the `.up_instances` script creates a specified input number of OpenStack `m1.medium` compute nodes using the prebuilt `perf-abaco-compute` Openstack image. The `up_abaco` script starts the Abaco services (packaged as Docker containers) on the various instance. The `up_abaco` script is a convenience wrapper around an Ansible playbook designed to interact with sets of instances running on an OpenStack cluster, also included in the repository.

C. Validation Setup

The test suite itself was also automated and requires little human intervention once the Abaco system against which it will run has been instantiated. The `/test_suite/tests` folder within the Github repository includes all the Python tests for this experiment. The master test script, named `run_tests.sh`, runs the test for all specified node sizes, each time deleting nodes to reach the specified amount of nodes and re-initializing all containers.

To run the tests with the autoscaler `ON`, the `run_tests.sh` script must be modified to uncomment the Python scripts that begin with `scaling`, and the instances must be redeployed with an updated version of the `abaco.conf` file, included in the repository. It's also worth noting that the Prometheus component is only needed when running with the autoscaler turned on. Complete details are provided in the project repository on Github.

D. Validation

The execution of our experiment suite has five trials of six jobs - three using manual scaling and three using autoscaler - at ten different node sizes. The first of the three jobs with manual scaling is a quick work FLOPS test. The test is setup with six Abaco actors per node to have one actor per node core. Each of these actors are given 5 executions to complete and each of these executions consists of doing the dot product of two square matrices of dimensions 8000 by 8000. The second test using manual scaling is the hash test and is setup similarly to quick work FLOPS except for the fact that each actor is given 6 executions and each execution is meant to complete 3,000,000 SHA256 hashes. The third test using manual scaling is the slow work FLOPS test. This test has one Abaco actor per node so that each actor gets 6 cores to run on. Each of these actors is given 5 executions to complete and each of these executions consists of doing the dot product of two square matrices of dimensions 25,000 by 25,000. The last three tests are exactly the same as the first three, but they utilize the Abaco autoscaler for worker management.

TABLE I
WORKER CONFIGURATIONS PER TEST TYPE

Test Type	max workers per host	max workers per actor
Quick FLOPS	6	30
Slow FLOPS	1	1
Hashrate	6	36

For this paper, we use the Abaco system configurations described in table I, for each test type. These parameters are set in the `abaco.conf` file found in the `deployment/abaco_files` folder of the repository. These parameters act as the upper bounds of available workers in each test. These bounds resemble configurations in TACC's existing production deployment of Abaco and ensure that we don't ruinously scale up our workers and cause bottlenecks due to inefficient CPU distribution.

IV. FINDINGS

We conduct experiments to answer the research questions stated in section 2.1. From the first set of experiments, we gather data on the rate of worker creation at different node sizes, manual and autoscaler to scale workers. The second set of experiments do the same, except we measures two performance metrics in order to visualize any falloff in performance.

Our experimental findings broadly follow the patterns that we expect in terms of worker creation rate and performance drop off. Using these insights we propose to answer the research questions posed by this paper in the following sections.

A. Difference in worker scaling rate between scaling types

The largest difference between manually scaled worker pool tests and autoscaler tests is obviously the rate of worker creation. By making this part of the experiment independent of any work done we can see the difference in time when creating workers.

Figures 1 and 2 both are split into two subplots for easier viewing and analysis. Experiments on node sizes 1 through 40 are on the left plot of each figure and node sizes 50 through 100 are on the right plot of each figure.

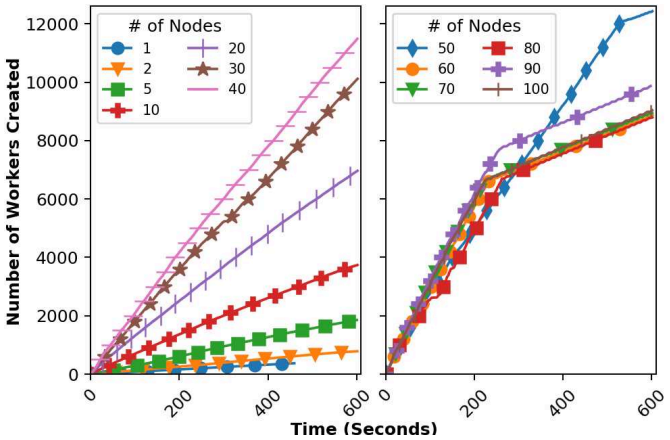

Fig. 1. Rate of worker creation in the manually scaled case

In figure 1 we see two important elements. First, from 1 to 40 nodes, the manually scaled worker creation rate increases linearly. Secondly from 50 to 100 nodes, there seems to be a worker creation slowdown at around 200 seconds, when around 8000 workers are created. While this particular set of circumstances is unlikely to repeat itself in a real workflow, we can see that the rate of worker creation reaches some upper limit that results in a slowdown of worker creation.

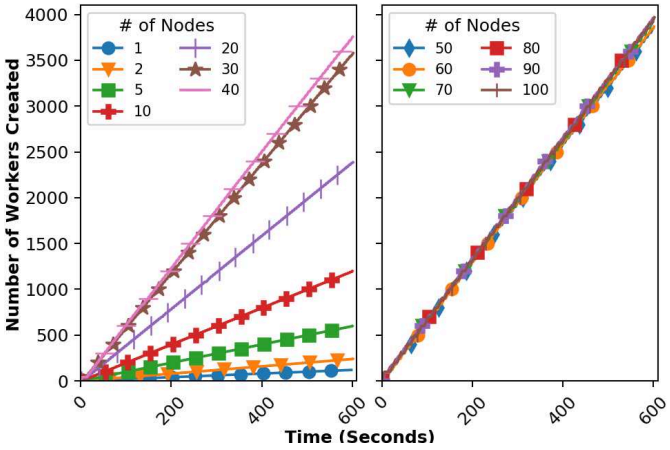

Fig. 2. Rate of worker creation in the autoscaler case

In figure 2 we can see that from 1 to 40 nodes worker creation rate increases in a relatively regular fashion and reaches a ceiling from 50 to 100 nodes. This can likely be attributed to the way Abaco queues up and creates workers once they are requested. While there may be resources to create more workers, the autoscaler requests those resources at set time increments and caps off the worker creation rate.

Figure 3 shows us the overall average worker creation rate for both manual and autoscaler cases and compares them. Here we can see that the worker creation rate slows down in the manually scaled case at around a node size of 70 when overall average rate begins to drop off. We can also see the rate gradually increasing in the autoscaling case as worker creation rate rises with additional nodes and then reaches a plateau.

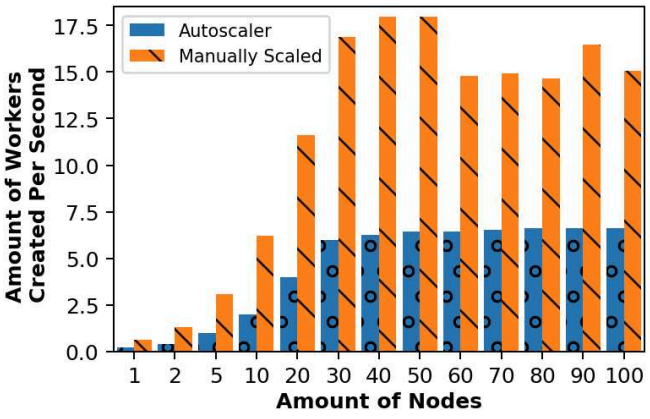

Fig. 3. Comparison of worker creation rates

The data gathered gives useful information to users about the Abaco platform. We see that the manually scaled case has a higher rate of worker creation. The necessity of this performance is somewhat negligible however due to a use case of this variety being particular rare. The autoscaler rate has a ceiling of about 6.5 workers created every second. Over the course of one minute, any node size greater than around thirty would product around 390 workers. Ten minutes would produce around 3900 workers. This amount of workers per node would create a bottleneck on resources solely due to having so many docker containers running in parallel. Even more resource hungry would be manually scaling workers at a rate of 15 workers per node which is possible at a node size above 30. While possible, this would produce a prohibitively large amount of workers on a single node.

Overall, we see that Abaco performs exceptionally well during worker creation. Assuming a user is running demanding tests, worker creation time would be a small fraction of total run time and will far exceed the performance necessary for most users. While in the manually scaled case we do run into performance issues, it's important to note that they arise at around 7000 workers and are likely due to the rate of worker creation being too fast.

B. Analysis of performance at different node sizes

One of the most important questions to ask when using Abaco is how much performance is a user ready to trade in for the ease of use and accessibility of the Abaco autoscaler.

Column 1 of table II is the ratio of autoscaler performance to manually scaled performance. From the results we can see that the overall trade-off when an execution is running is nearly indistinguishable with the autoscaler even being marginally better than the manually scaled performance in the quick work tests. This statistic follows our expected patterns as the autoscaler and manually scaled tests should only differ in how workers are created, not performance in executions.

In the second column of table II we see that the quick work tests are 65.1% of theoretical performance, the slow work tests are 67.7% of theoretical performance, and the hashrate tests does not have a theoretical performance due to

TABLE II
PERFORMANCE RATIOS AT A NODE SIZE OF 89.

	Ratio of Autoscaler Speed to Manually Scaled Speed	Ratio of Manually Scaled Speed to Theoretical Speed	Ratio of Jetstream Speed to Theoretical Speed	Ratio of Manually Scaled Speed to Jetstream Speed
Quick Work FLOPS Test	102.4%	65.1%	71.5%	80.2%
Slow Work FLOPS Test	99.1%	67.7%	81.2%	94.7%
Hashrate Test	99.5%	N/A	N/A	92.5%

the empirical nature of hashrate testing. The third column of table II gleans perspective on these numbers by comparing the theoretical hardware performance to our Jetstream performance. In one case, Jetstream is 71.5% of the theoretical performance, while in the other it is 81.2% of the theoretical performance. In essence the theoretical hardware performance is an unreasonable metric to compare against when our testing hardware could only practically achieve around 75% of that performance. Thus the important metric to compare to is the practical result, which is Jetstream performance.

In the fourth column of table II we see the manually scaled speed as a percentage of Jetstream speed. In the quick work tests we see 80.2% of Jetstream speed. This is due to a combination of Docker overhead, Abaco overhead, and overhead due to the node. For instance, running a multitude of quick tests means that the node's CPU is constantly going in and out of working and must constantly change clock speed from resting to turbo. In the other two tests performance is 92.5% and 94.7% of the Jetstream speed, which again can be attributed to the same causes.

Figure 4 visualizes the results of table II. With this figure we see that performance increases linearly as more nodes are added to the tests. and there's not any irregularities when executions are being ran. This gives us a sense of scale in regards to the amount of work that the Abaco platform is capable of running. At 89 nodes, the slow FLOPS test reaches 19 TFLOPS of performance. This is equivalent to the theoretical performance of 13 Stampede2[14] nodes.

To answer the research question, overall the performance trade-off of using the Abaco platform is minor as compared to the Jetstream performance and the trade-off of using the Abaco autoscaler is non-existent once executions are actually being ran.

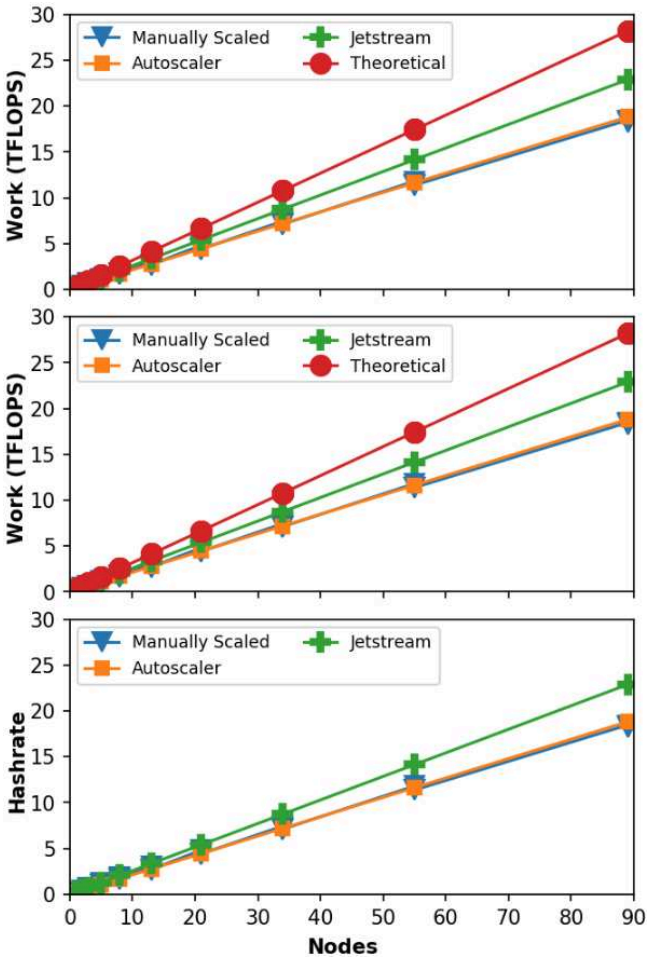

Fig. 4. TFLOPS of different tests based on node size
Top: Lots of Quick Work Tests
Center: Slow Work Tests
Bottom: Hashrate Tests

C. The scaling limits of Abaco

A question that must be asked for a researcher is, how far can the Abaco platform scale out before issues arise and potentially affect usage? In our test setup we ran into several hurdles, many were easily fixed, while others will need future improvements made to the Abaco system to ensure peak performance no matter the case.

The first issue that we faced came from Docker Hub. When testing the performance of the platform, if too many workers were requested at once, Abaco would eventually begin receiving HTTP status codes 429, "Too Many Requests". This would result in a platform error that attempted to scale back the number of workers to clear platform congestion. To alleviate this issue going forward, Abaco should either set a capped Docker requests rate or add an actor flag that bypasses pulling worker images unless they are missing.

Two more issues seen were related to node resources. The first was RabbitMQ restricting any incoming messages when a node's RAM was sufficiently in use. This was alleviated by setting the `RABBITMQ_VM_MEMORY_HIGH_WATERMARK` environment variable to a higher percentage. The second resource issue came from MongoDB, also consuming a large

amount of node RAM. This issue resulted in node slowdown and eventual slow responsiveness, and was most likely the cause of worker creation slowdowns in the case of very large node sizes.

Another major issue that was observed involved Abaco's use of it's Redis database. Abaco stores all runtime information for a given actor under a single key in the Redis database. Initial versions of the experiment sent all messages in a given test to a single actor. At very high node counts, performance suffered as a result of Redis optimistic locking when information was being written to the actor record. This was alleviated by changing the tests to use multiple actors (e.g., one actor per node) and dividing the work evenly across the actors in the experiment. Abaco is nearing completion of a set of changes to it's database usage to improve query and write performance in the rare cases that one actor doing so much work could be more beneficial.

Overall, the primary symptoms of scaling issues did not come in any fatal outcomes, but in slowdowns of the Abaco systems.

V. Related Work

Abaco draws comparison to and inspiration from a number of existing software systems.

A. Functions-as-a-service

There are commercial offerings from AWS Lambda [15], Google Cloud Functions [16], Microsoft Azure Functions [17], and IBM Apache OpenWhisk Functions [18]. However, many of these offerings have limits on memory allocation and runtime duration. Some also have limitations on which languages can be used. AWS Lambda allows many different languages, but limits users to 1536 MB of memory allocation and 300 seconds of runtime, while Google Cloud Functions only allows Node.js and has a maximum duration of 540 seconds, and Microsoft Azure functions has no limit on execution time limit but limits the amount of functions running at once to 10.

The closest in spirit to Abaco would be the open source project OpenFAAS [19], which provides functions-as-a-service based on Docker images. Unlike Abaco, however, it requires the function container to run an HTTP server. It also does not include the Actor model and several other features that are part of Abaco.

B. Containers-as-a-service

Commercial examples of containers-as-a-service include Amazon's Elastic Container Service (ECS) [20]and Google's Container Engine [21]. Although these services allow the use of arbitrary container images, they lack the Actor-based architecture that is part of Abaco's design, making them better suited for long-running server daemons.

C. Distributed Computing Platforms

Platforms such as Apache Spark [22], Apache Storm [23], iPython parallels [24]and AWS Kinesis [25] provide features similar to Abaco's scientific functions. These systems even support additional paradigms such as inter-process communication (IPC) and provide better performance. For Abaco, scientific functions only ever attempt to achieve pleasantly parallel compute jobs and its goal is to make them more accessible.

VI. Conclusion

In this study, we tested the actual performance of Abaco to compare it with the theoretical bounds of the system. We measured the differences in performance and worker creation rate between manually scaled workers and the autoscaler, and evaluated the scaling limits of Abaco and their causes. From our findings we conclude that the Abaco platform greatly automatizes additional set up and frees up valuable user time at little to no extra cost compared to a manually scaled setup. If we were to improve upon this experiment we would want to overhaul Abaco in order to optimize scaling to more compute nodes. Before this experiment the scalability of Abaco was unknown. However, with the fixes mentioned before and hardening of some systems, Abaco has the potential to grow to an even greater extent and give researchers an even better tool to make use of HPC resources. Overall the study demonstrates the practicality of using Abaco to simplify workflow and using the Abaco autoscaler to further reduce user time needed.

Acknowledgment

This material is based upon work supported by the National Science Foundation Office of Advanced CyberInfrastructure, award number 1740288. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Jetstream resource at the TACC through allocation CCR190017.

References

[1] J. Stubbs *et al.*, "Rapid development of scalable, distributed computation with Abaco," Science Gateways Community Institute. 10th International Workshop on Science Gateways, 2018.

[2] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[4] A. Wood, *Rabbit MQ: For Starters*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016.

[5] Prometheus, "Prometheus," Feb. [Online]. Available: https://github.com/prometheus/prometheus

[6] (2020) Docker hub flops image. [Online]. Available: https://hub.docker.com/repository/docker/abacosamples/abaco_perf_flops

[7] (2020) Numpy dot product. [Online]. Available: https://www.tutorialspoint.com/numpy/numpy_dot.htm

[8] (2020) What is hashrate? [Online]. Available: https://www.buybitcoinworldwide.com/mining/hash-rate/

[9] (2020) Docker hub hashrate image. [Online]. Available: https://hub.docker.com/repository/docker/abacosamples/abaco_perf_hashrate

[10] D. M. R. Fernandez. (2020) Nodes, sockets, cores and flops, oh, my. [Online]. Available: https://www.shorturl.at/kUZ12

[11] C. A. Stewart, T. M. Cockerill, I. Foster, D. Hancock, N. Merchant, E. Skidmore, D. Stanzione, J. Taylor, S. Tuecke, G. Turner, M. Vaughn, and N. I. Gaffney, "Jetstream: A self-provisioned, scalable science and engineering cloud environment," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: ACM, 2015, pp. 29:1–29:8. [Online]. Available: http://doi.acm.org/10.1145/2792745.2792774

[12] A. Shrivastwa, S. Sarat, K. Jackson, C. Bunch, E. Sigler, and T. Campbell, *OpenStack: Building a Cloud Environment*. Packt Publishing, 2016.
[13] (2020) Github - abaco autoscaling. [Online]. Available: https://github.com/tacc/abaco-autoscaling
[14] (2020) Stampede2. [Online]. Available: https://www.tacc.utexas.edu/systems/stampede2
[15] (2020) Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/
[16] (2020) Google cloud foundation. [Online]. Available: https://cloud.google.com/foundation-toolkit/
[17] (2020) Microsoft azure. [Online]. Available: https://azure.microsoft.com/
[18] (2020) Apache openwhisk. [Online]. Available: https://openwhisk.apache.org
[19] (2020) Openfaas. [Online]. Available: https://www.openfaas.com
[20] (2020) Amazon elastic container service. [Online]. Available: https://aws.amazon.com/ecs/
[21] (2020) What is gke? [Online]. Available: https://www.aquasec.com/wiki/display/containers/Google+Container+Engine
[22] (2020) Apache spark. [Online]. Available: https://spark.apache.org
[23] (2019) Apache storm. [Online]. Available: http://storm.apache.org
[24] (2020) Ipython parallel. [Online]. Available: https://ipython.org/ipython-doc/stable/parallel/parallel_intro.html
[25] (2020) Amazon kinesis. [Online]. Available: https://aws.amazon.com/kinesis/