Image Processing using Task Parallel and Data Parallel Frameworks

William Cheng, Ioannis Paraskevakos, Mateo Turilli, Shantenu Jha

***Abstract*—Focused on imaging algorithms, we compare three frameworks, RADICAL-Pilot, Spark, and Dask. We do this by measuring the weak scaling and strong scaling of these frameworks on the Comet High Performance Supercomputer and a Jetstream cloud VM.**

CONTENTS

***Index Terms*—RADICAL-Pilot, Spark, Dask, Parallel Task Frameworks, Image Processing, Watershed, Blob Detector**

I. INTRODUCTION

IMAGE processing has become increasingly computational heavy as images are growing in resolution and size. Not only that, but there are a plethora of images to be analyzed, and that situation might not ever change. There are many people in the world that own cell phones with cameras, and many more civilian or government devices that such as video cameras and satellites that take output lots of data in video or image format. It is difficult to efficiently analyze lots of images without smart algorithms or certain levels of parallelism; this report focuses on the latter.

We will be focusing on three frameworks: RADICAL-Pilot [1], Spark [2], and Dask [3]. RADICAL-Pilot is considered a task-parallel framework as it focuses on running tasks or in RADICAL's vernacular, compute units, parallel. PySpark and Dask are considered data-parallel frameworks as they focus more on analyzing certain amounts of data in parallel. One interesting thing to note is that these frameworks are written in pure Python, with the exception that PySpark being an interface to the Spark programming model.

It is interesting to observe and compare the performance of these frameworks together. We do so by taking a data set of images that will vary in size to compare the capabilities of these frameworks. We will do so by running various experiments, observing the total time to completion for different imaging algorithms as well as data and resource sizes, and looking at the performance and weak and strong scaling that the frameworks return.

II. MATERIALS AND METHODS

A. Image Algorithms

Watershed Algorithm

We use the watershed algorithm from Ref. [4]. We choose this algorithm to observe and compare the performance and scaling of additional big data frameworks.

Blobdetector Algorithm We use the Blob Detector algorithm created from Nivethia.

B. Images Datasets

The images used in our analysis are taken from Anastasios's [4] report. There are three images used in this case.

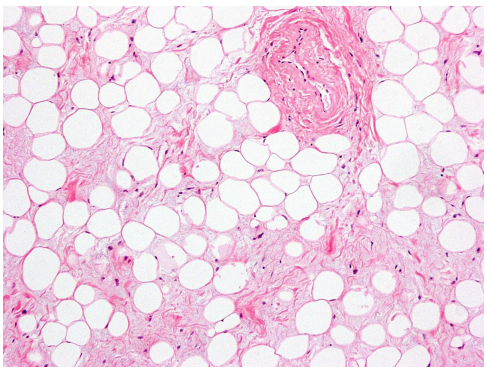

Fig. 1. Image 1: 213KB

Fig. 2. Image 2: 14KB

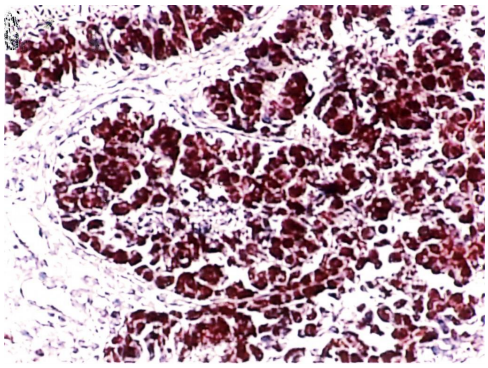

Fig. 3. Image 3: 166KB

C. Computer Systems

Comet

is a high performance computer within the San Diego Supercomputer Center. It has 1944 standard compute nodes, each which has Intel Xeon E5-2680v3 2.5Ghz dual socket, 12 cores/socket, 320 GB flash memory, and 120 GB/s memory bandwidth. It uses the 7.6 PB Lustre-based parallel file system as well as hybrid fat-tree topology, and 56 Gb/s bidirectional link bandwidth.

The images shown in section II-B were stored on the Lustre parallel distributed file system. Since the data-set were repeats of those three images, the data-set was created by creating 4096 soft links to those images.

Jet Stream

is a cloud environment that is user friendly and meant for interactive research. Indiana University (IU) and Texas Advanced Computing Center (TACC) share the two homogeneous production computing clusters and University of Arizona holds the testing computing cluster. Each of the production computing cluster has 320 nodes, 12 cores per node, 128GB RAM, 2TB local storage, and 10 Gb/s network speed to XSEDE resources.

D. Frameworks

RADICAL-Pilot (RP)

is a scalable pilot system that provides a simple and versatile approach for executing concurrent simulations and their data requirements on clusters, grids, and clouds. It offers users a lightweight API capable of handling a variety of workloads and scheduling O(10k) tasks and marshalling O(10k) distributed cores. RP is written in pure python and was created and currently being maintained by RADICAL Labs.

RP provides an API which abstracts away a lot of the work necessary for resource access and task management. RP itself is abstracted away from the resource access as it is built on top of Simple API for Grid Applications (SAGA), which provides common distributed systems components such as file transfer, job schedulers, and resource provisioning. RP's abstraction of a unit of work is called a "Compute Unit (CU)", and the details of a task that the CUs is given is a "Compute Unit Description (CUD)", which contains the *executable*, *arguments*, and *data staging directives*. RP will then launch "Pilot(s)" to manage a requested remote resource on behalf of the user. The Pilot then launches agent(s) to schedule a subset of CUs. There are a number of different CU scheduling algorithms that RP has under its belt: continuous, continuous-FIFO, round-robin, back-filling, etc. A MongoDB instance is utilized as a communication channel between the client and server; CUDs will be pulled from the Pilot and CUs will share their state information on MongoDB.

Spark

is an open source distributed computing framework created by Berkeley and maintained by the Apache Software Foundation which is written in Scala and runs on the Java Virtual Machine. **PySpark** is a python interface to Spark.

Spark was built based on the map-reduce programming model so that parallel operations such as map, reduce, and filter are invoked on the data. These operations are scheduled by creating a direct acyclic graph (DAG), where each node in the DAG is considered a task, and then this DAG gets submitted to the DAG scheduler, which will execute tasks that have their dependencies fulfilled.

We use the *spark-submit* script [5] to set up the network between our requested compute clusters.

Dask

is a parallel computing library that is made to scale from single-machine to distributed clusters. It is written in pure python as well and builds upon Numpy arrays and Pandas Dataframes.

Dask schedules tasks in a similar fashion to Spark: it creates DAGs and executes the tasks within the DAG lazily.

We use *dask-ssh* [5] to set up the network between our requested compute clusters.

E. Implementations

Environments The environments and package versions were maintained using Anaconda Python and kept in sub-

sequent *environment.yml* files. The Linux distribution Spark V2.3.1 was installed within our working directory.

Degrees of Freedom

We tune two parameters to analyze the frameworks within this report: (1) number of nodes (24 cores per node) and (2) number of images to analyze.

Weak Scaling

Weak Scaling experiments were performed for each algorithm by starting out with 512 images and 1 node (24 tasks), which comes out to be 512 images per node. We scaled the total number of images and nodes but kept the ratio the same all the way up to 4096 images and 8 nodes. This averages out to be 21 images per task. The full range of experiment parameters are shown in table II-E.

TABLE I
WEAK SCALING EXPERIMENTS

# Images	# Nodes
512	1
1024	2
2048	4
4096	8

Strong Scaling

Strong scaling experiments were performed for each algorithm by starting out with 4096 images and 1 node (24 tasks). We then kept the total number of images constant but scaled up the number of cores from 1 to 8 nodes. The full range of experiment parameters are shown in table II-E.

TABLE II
STRONG SCALING EXPERIMENTS

# Images	# Nodes
4096	1
4096	2
4096	4
4096	8

RADICAL-Pilot

As mentioned previously in Section II-D, RP requires that the user has a live client connection while the server side computations are running. We utilize a Jetstream VM instance running on Ubuntu 14.04.4. A ”medium” VM specification was used with 6 cores, 16GB memory, and 60GB storage. We used conda virtual environments and used RP V0.47.10 for our experiments.

A MongoDB instance on MLab was created as the communication chancel mentioned in section II-D. We used a free MLab MongoDB instance which included 500MB of free storage.

The watershed algorithm executable was placed in the same directory as where we start the RP client, so the executable has to staged to the Comet file system.

There wasn't anything that we had to configure RP for other than staging source, target, and actions.

It was simple to use the RP API as it will automate a majority of the setup such as the server side including environments and data and file transfer.

PySpark

We used the Linux distribution of Spark V2.3.1 and used PySpark V2.4.0 at the time of these experiments. The creation and setup of the PySpark implementation python environments were done in Comet's interactive debug mode.

The Spark configurations is listed in table II-E. The SPARK_WORKER_CORE is the number of workers (processes) per node, which is set to 24 because that is the number of cores Comet has per node. MaxResult is set to 30g (30GB) to prevent the driver from experiencing out of memory errors. ExecMem is the max memory that each executor process can use up to. DriverMem is the max memory that the driver, where the spark configuration is initialized, can use.

TABLE III
SPARK CONFIG

Configuration	Value
SPARK_WORKER_CORE	24
MaxResult	30g
ExecMem	60g
DriverMem	40g

We had to run PySpark with an extra node for its Driver, however. Thus, the PySpark weak-scaling and strong-scaling experiments will each have an extra node.

To run a PySpark experiment, we created an sbatch script to request the appropriate resources from Comet. Other pre-experiment steps were to echo the addresses and ports of available compute nodes given to us from Comet into a file and setting environment variable for the Spark environments. Then the .start-all.sh bash script was called from within the directory of the installed Spark V2.3.1 folder.

Once the Spark driver is initialized, we use spark-submit to submit our PySpark python application to the cluster that was set up.

Dask

We used Dask V0.19.3. The creation and setup of the Dask implementation python environments were done in Comet's interactive debug mode.

The Dask configurations is listed in table II-E. The nprocs configuration is the number of processes per node, just like SPARK_WORKER_CORE in II-E. The configuration nthreads is the max number of threads per worker process.

TABLE IV
DASK CONFIG

Configuration	Value
nprocs	24
nthreads	1

To run a Dask experiment, we created an sbatch script, similar to the sbatch script for the PySpark experiments in section II-E. We first requested resources from Comet, and once that step is done, we run dask-ssh and give it the addresses and ports of the compute units given to us so that dask-ssh can initialize the network accordingly by opening several SSH connections.

Once dask-ssh sets up the network and scheduler, we execute our Dask application and feed it experiment parameters which includes the address and port of the Dask scheduler,

which then uses this to create a connection to send tasks through.

III. Results

A. Blob Detector

Weak-Scaling

Our weak scale experiments of the blob detector using the frameworks, RADICAL-Pilot, Spark, and Dask, are shown in figure 4.

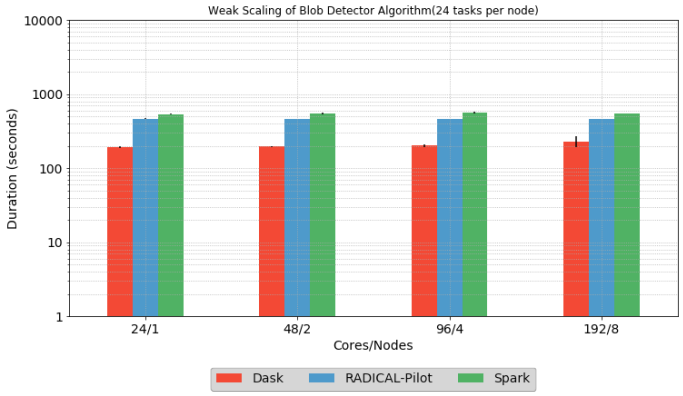

Fig. 4. Weak Scaling of Blob Detector Algorithm

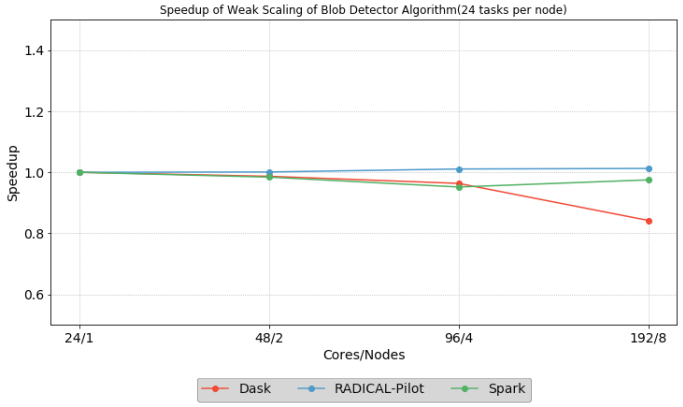

Fig. 5. Speedup of Weak Scaling of Blob Detector Algorithm

TABLE V
Results of Weak Scaling of Blob Detector Algorithm

Nodes	Dask		RADICAL-Pilot		Spark	
	Time	Speedup	Time	Speedup	Time	Speedup
1	192.52	1	466.32	1	534.59	1
2	195.09	0.98	465.86	1.00	543.23	0.98
4	199.79	0.96	461.34	1.01	561.65	0.95
8	228.55	0.84	460.55	1.01	548.25	0.97

Strong-Scaling

Our strong scale experiments of the blob detector using the frameworks, RADICAL-Pilot, Spark, and Dask, are shown in figure 6.

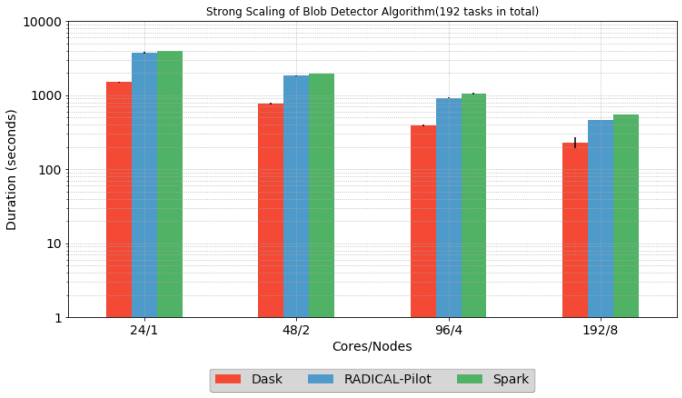

Fig. 6. Strong Scaling of Blob Detector Algorithm

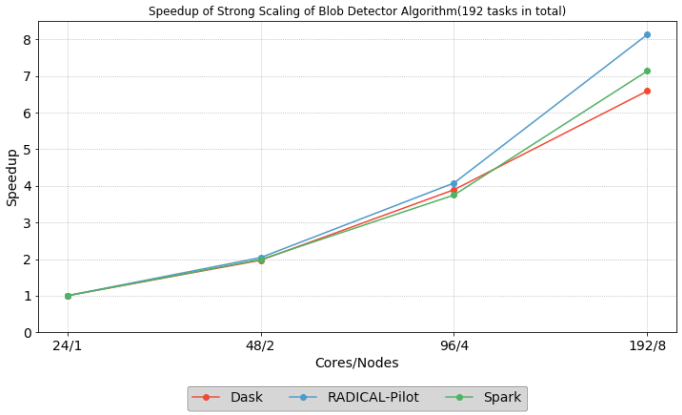

Fig. 7. Results of Strong Scaling of Blob Detector Algorithm

TABLE VI
Results of Strong Scaling of Blob Detector Algorithm

Nodes	Dask		RADICAL-Pilot		Spark	
	Time	Speedup	Time	Speedup	Time	Speedup
1	1506.46	1	3744.40	1	3912.24	1
2	765.24	1.97	1835.50	2.04	1969.08	1.99
4	387.01	3.89	918.62	4.08	1043.86	3.75
8	228.55	6.59	460.55	8.13	548.25	7.14

Blob Director Result

We discuss the results the weak and strong scaling of the blob detector algorithm on a data set consisting of the images shown in section II-B.

With these frameworks, we notice that the duration of the blob detector experiments in the weak scale experiments in Fig. 4 are near constant as the number of resources and data is scaled up. This is expected from a weak scaling experiment as the optimal scaling taken from Eq. 1. This shows that the three frameworks do really well in scaling the amount of data being processed while keeping the work per resource the same. However, when there are 4096 images and 192 cores, the performance of Dask starts to suffer and the weak scale speedup drops to roughly *0.8x*. RP stays the most consistent in this weak-scale experiment and maintains a nearly *1x* speedup.

Spark is also consistent and maintains a close to *1x* speedup, but not as much as RP does.

We also notice that the strong scaling experiments show promising results, too, because the scaling decreases log-linearly as the amount of data being processed stays constant and the number of resources increases as described by Eq. 2. Consistent with the weak scale results, we notice that RP keeps near-perfect strong scaling as the number of cores is increased to 192. However, Spark and Dask's speedups reduce down to *7x* and *6.5x*, respectively.

Even though we observe that the RP's performance in strong and weak scaling is better than that of Spark and Dask, it is also fair to note that Dask has a faster *Total Time to Completion (TTC)* by almost *3-4x*. On the other hand, RP scales the best as it maintains a weak scale close to 1x and expected strong scaling factors, and Dask and Spark scales very well initially up to four nodes, but their weak and strong scaling factors drop off when eight nodes are used.

B. Watershed

Weak-Scaling

Our weak scale experiments of the watershed using the frameworks, RADICAL-Pilot, Spark, and Dask, are shown in figure 8.

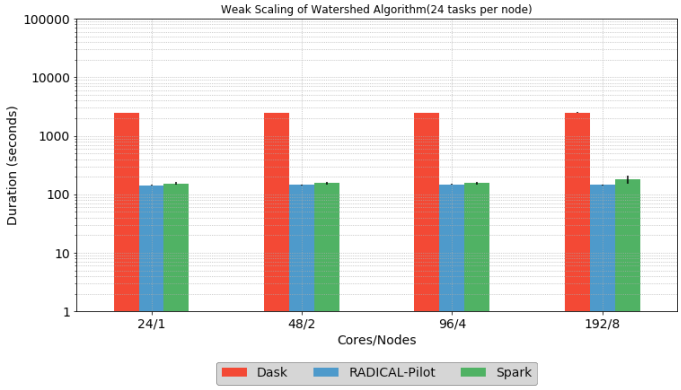

Fig. 8. Weak Scaling of Watershed Algorithm

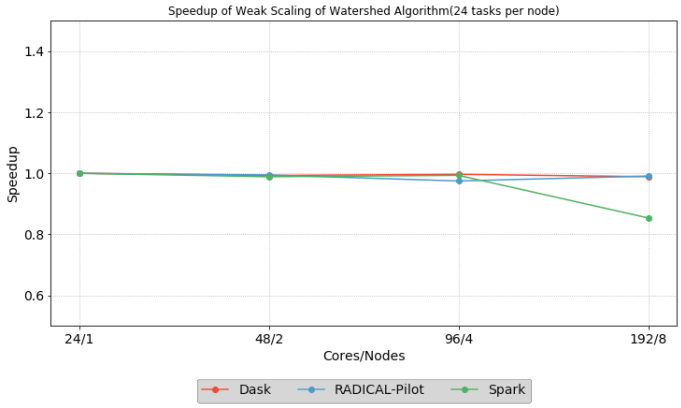

Fig. 9. Speedup of Weak Scaling of Watershed Algorithm

TABLE VII
RESULTS OF WEAK SCALING OF WATERSHED ALGORITHM

Nodes	Dask		RADICAL-Pilot		Spark	
	Time	Speedup	Time	Speedup	Time	Speedup
1	2478.46	1	142.80	1	152.90	1
2	2496.26	0.99	143.60	0.99	154.65	0.98
4	2486.96	0.99	146.50	0.97	153.94	0.99
8	2506.66	0.98	144.10	0.99	179.09	0.85

Strong-Scaling

Our strong scale experiments of the watershed using the frameworks, RADICAL-Pilot, Spark, and Dask, are shown in figure 10.

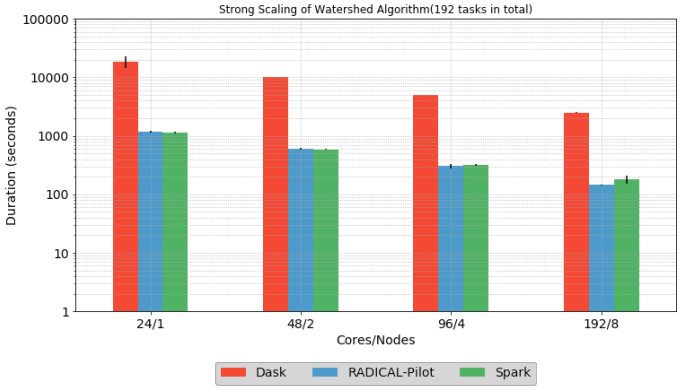

Fig. 10. Strong Scaling of Watershed Algorithm

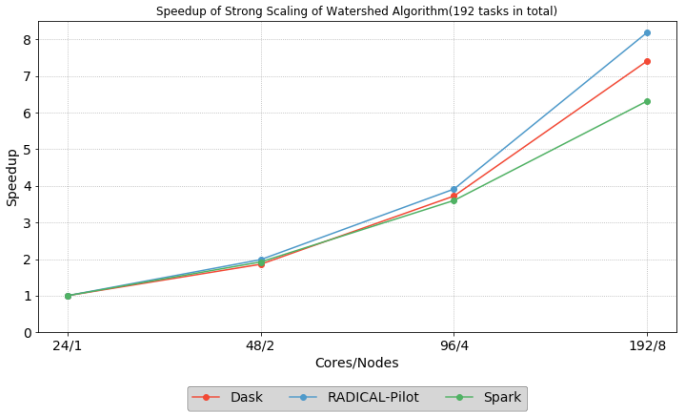

Fig. 11. Speedup of Strong Scaling of Watershed Algorithm

TABLE VIII
RESULTS OF STRONG SCALING OF WATERSHED ALGORITHM

Nodes	Dask		RADICAL-Pilot		Spark	
	Time	Speedup	Time	Speedup	Time	Speedup
1	18578.93	1.00	1180.40	1.00	1131.31	1.00
2	10000.55	1.85	594.90	1.98	589.01	1.92
4	4993.17	3.72	301.80	3.91	314.24	3.60
8	2506.66	7.41	144.10	8.19	179.09	6.31

Watershed Result

We discuss the results the weak and strong scaling of the watershed algorithm on a data set consisting of the images shown in section II-B.

The scaling for the watershed algorithm using the frameworks are as expected as well. The duration of the watershed experiments in the weak scale experiments in Fig. 8 are near constant as the number of resources and data is scaled up. Both RP and Dask have very good weak scaling, nearly *1x* throughout all the weak scale experiments. On the other hand, Spark drops off to *0.85x* speedup in the experiment without the largest number of resources and nodes: 4096 images and 192 cores.

Also, the frameworks do well in the strong scaling as well, as the scaling decreases log-linearly as the amount of data being processed stays constant at 4096 images and the number of resources increases. RP scales very well in this experiment with a speedup of *8x* when using 192 cores and Dask is close with a speedup of *7.2x*. Spark's performance goes down to *6x*.

Although the scaling of the watershed algorithm implemented with the three frameworks perform well, we observe unexpected behavior of the duration associated with the Dask framework compared to the duration that RP and Spark took. The TTC of Dask using the watershed algorithm takes roughly more than *15x* longer than the TTC of RP and Spark. This is contrary to what we observed with the blob detector algorithm results in Fig. **??** and **??**, where the TCC of Dask was actually less than that of RP and Spark.

We looked further into this discrepancy and observe the the start and finish times of the workers between Dask and Spark. We take the first weak scale experimental where we constrain 24 workers to 1 node (24 cores).

We see that there the workers for both frameworks are created almost instantly. However, all of the Dask workers finish at the same time at the 2500s boundary in Fig. 12, and all of the Spark workers finish near the 150s boundary in Fig. 13.

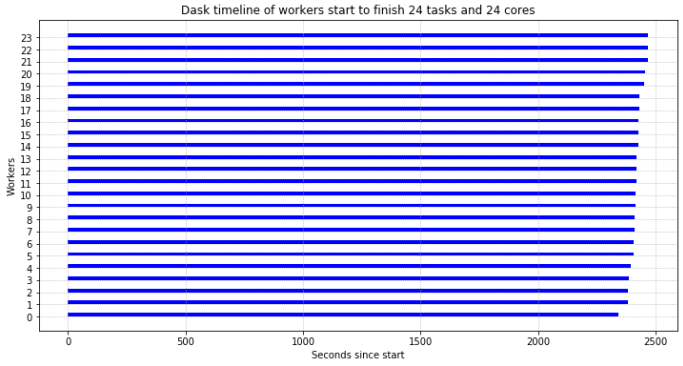

Fig. 12. Dask Worker Timelines

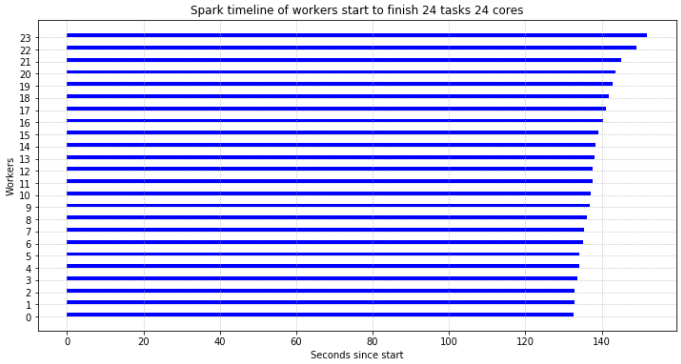

Fig. 13. Spark Worker Timelines

IV. Discussion and Summary

We utilized XSEDE resources, Comet and Jetstream, to conduct these experiments.

We were able to observe expected results from the weak and strong scaling of the blob detector algorithms using RP, Spark, and Dask. On the other hand, we also observed expected results from the weak and strong scaling of the watershed algorithm using RP, Spark, and Dask, but the duration of the Dask results were nearly an order of magnitude higher than what was expected. We hypothesize that there is a resource competition between the workers in Dask that is resulting in the increased TTC in our results. To further investigate, we are currently running various experiments using the watershed algorithm to capture what the main issue of the watershed Dask results. We will profile the CPU and memory usage to evaluate our hypothesis of resource competition.

Appendix A
Scaling Equations

Weak Scaling Equation

$$\frac{T_1}{T_N} * 100\% \tag{1}$$

Strong Scaling Equation

$$\frac{T_1}{N * T_N} * 100\% \tag{2}$$

Acknowledgment

The authors would like to thank...

References

[1] A. Merzky, M. Turilli, M. Maldonado, M. Santcroos, and S. Jha, "Using pilot systems to execute many task workloads on supercomputers," in *Job Scheduling Strategies for Parallel Processing*, D. Klusáček, W. Cirne, and N. Desai, Eds. Cham: Springer International Publishing, 2019, pp. 61–82.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10.

Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[3] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130 – 136.

[4] A. Stathopoulos, "Integrating image segmentation algorithm with midas," 2016, draft.

[5] I. Paraskevakos, A. Luckow, M. Khoshlessan, G. Chantzialexiou, T. E. Cheatham, O. Beckstein, G. C. Fox, and S. Jha, "Task-parallel analysis of molecular dynamics trajectories," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 49:1–49:10. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225128