Twister2 Cross-Platform Resource Scheduler for Big Data

Ahmet Uyar*, Gurhan Gunduz†, Supun Kamburugamuve*, Pulasthi Wickramasinghe*, Chathura Widanage*, Kannan Govindarajan*, Niranda Perera*, Vibhatha Abeykoon*, Selahattin Akkas*, Geoffrey Fox*
* School of Informatics, Computing, and Engineering, Indiana University, Bloomington
{auyar, skamburu, pswickra, cdwidana, kgovind, dnperera, vlabeyko, sakkas, gcf}@indiana.edu
† Department of Computer Engineering, Faculty of Engineering, Mugla Sitki Kocman University, Turkey
{gurhangunduz}@mu.edu.tr

***Abstract*—Twister2 is an open source big data hosting environment designed to process both batch and streaming data at scale. Twister2 runs jobs in both high performance computing (HPC) and big data clusters. It provides a cross-platform resource scheduler to run jobs in diverse environments. Twister2 is designed with a layered architecture to support various clusters and big data problems. In this paper, we present the cross platform resource scheduler of Twister2. We identify required services and explain implementation details. We also present job startup delays for single jobs and multiple concurrent jobs in Kubernetes and OpenMPI clusters. We compare job startup delays for Twister2 and Spark at a Kubernetes cluster.**

***Index Terms*—HPC, big-data, resource scheduling, Kubernetes, Slurm, Nomad, OpenMPI**

I. Introduction

Today, there are many big data computing systems. These systems are generally designed to process either batch or streaming data. While Spark [1] and Hadoop are primarily designed for processing batch data, Flink [2] and Heron [3] are primarily designed for processing streaming data. Twister2 [4] is designed to process both batch and streaming data at scale. It is an open source big data hosting environment [1] providing a composable framework for high-performance data analytics.

Recently, there have been a lot of interest in converging high performance computing (HPC) and big data technologies [5, 6]. A number of projects have been trying to integrate big data technologies such as HDFS and Spark into HPC environments [7, 8]. On the other hand, there have been other projects that are trying to improve big data technologies by integrating technologies from HPC, particularly MPI [9, 10]. The layered architecture of Twister2 makes it possible both running big data applications in HPC clusters and integrating HPC technologies into the big data applications.

Twister2 proposes a 4-layer approach to big data systems: 1) Resource Scheduling, 2) Communications, 3) Task System, and 4) Distributed Data. Resource Scheduling layer handles resource acquisition, thread/process initiation and cleanup. Communications layer [11] provides MPI and dataflow style big data communications. Task system layer [12] provides task scheduling and task execution. Distributed data layer [13] offers the distributed data structures and parallel operators. Twister2 provides a polymorphic system by using these layers to produce components according to the requirements of the application types. Current big data systems are mostly designed in a monolithic approach with the above mentioned layers developed in a single project with tight integration.

In this paper, we present the resource scheduler of Twister2. It provides the following services:

1) resource acquisition, worker initiation and cleanup
2) worker scaling
3) storage provisioning
4) job package transfer
5) setting up networking
6) logging and monitoring

Since Twister2 targets both HPC and big data environments, the resource scheduler is responsible for running Twister2 jobs in many different clusters. Currently, we support Kubernetes [14], Slurm [15], Nomad [16] and Standalone MPI. Initially we also supported Mesos [17], but later on stopped supporting it. Twister2 resource scheduler is a meta resource scheduler. It talks to the resource managers in those clusters and get the requested resources.

Twister2 users write their applications using the high level API's that are provided by the framework, which expose distributed data structures and operators. Their code runs in any one of the supported clusters when Twister2 is installed. The main responsibility of the resource scheduling layer is to run Twister2 in those clusters. All cluster managers are different. Some services may not be available in all clusters and some cluster managers may provide extra services. Therefore, it may not be possible to provide uniform services in all clusters, but we try to provide uniform services as much as possible.

Job initialization costs are very important in distributed clusters. Usually big data jobs require initializing many containers or processes in different nodes. A single container initialization may take a few seconds. Therefore, it is important to minimize job initialization costs. We try to minimize job initialization delays in all supported clusters and provide performance figures in this paper.

Main contributions of the paper are:

1) Identifying the features of a cross-platform resource scheduler for big data systems and describing the solutions in popular cluster managers.

[1] https://github.com/DSC-SPIDAL/twister2

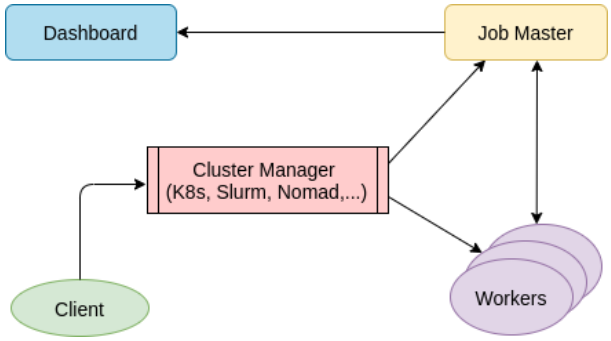

Fig. 1. Twister2 Runtime Architecture

2) Investigating resource scheduling costs in popular cluster managers.

The rest of the paper is organized as follows; Section II describes the resource scheduler in detail and Twister2 runtime architecture. Section III explains Twister2 implementation on Kubernetes and Section IV explains Twister2 implementation on MPI based schedulers. We present the experiments conducted to evaluate Twister2 job startup delays in Section V. We conclude the paper in Section VI.

II. Twister2 Resource Scheduler

Resource scheduling layer provides many services as outlined in the previous section. These services are implemented in various components in Twister2 runtime. Twister2 runtime consists of the following entities as shown in Fig. 1:

1) Job submission client
2) Job master
3) Workers
4) Twister2 Dashboard

Resource acquisition, worker initializations, and logging are implemented in the job submission client. Storage provisioning and the job package transfer are also mostly implemented in the client with some parts implemented in workers. Setting up networking is implemented in workers. Monitoring is implemented in Twister2 Dashboard. Worker scaling and cleaning up job resources are implemented in the job master.

A. Job Submission Client

Users employ the client program to submit, list and terminate jobs. The client program is installed on the user machine or on a cluster machine. It packs and uploads the job files, acquires cluster resources and asks the cluster manager to start the distributed workers and the job master. Users specify the number of workers to be started and the types of resources they will use.

The client program first packs all files related to the job including configuration files, user executables, user supplied parameters and environment specific variables. It either uploads this job package to a web server/shared file system, or transfers directly to worker machines/containers. When workers are started, they download the job package from the web server/shared file system or use already delivered local copy.

By default, workers are started in any available cluster machines. However, users can request workers to be mapped into machines with special hardware or software. Or, they can request that workers be mapped to any machines except some. In addition, they can request all workers in a job to be started on the same machine or on separate machines. Furthermore, worker binding is also supported. Workers can be bounded to the CPUs they are assigned to and they can not be moved during their lifetime.

B. Workers

Workers are the processes started in a cluster that perform actual job computations. Each worker has its own resources like CPU, memory and storage. Each worker has an IP address and a port number to communicate with other workers in the job. We also assign a unique sequential ID to each worker in a job, starting from zero and increasing sequentially to N-1 (where N is the number of workers in a job). We require that worker IDs are resilient to failures. They will be the same after relocation or failures. Communication layer uses these network identifiers to setup channels among the workers in a job.

Task System schedules the tasks in workers to execute computations [12]. It creates task execution graphs and assign tasks to workers. It also establishes communication channels between tasks (workers). Depending on the job specifications, it can schedule one or more tasks on each worker. Task system can use worker location information to provide locality aware scheduling.

C. Job Master

Job Master manages job related activities during job execution such as worker discovery, worker scaling, resource cleanup, and driver execution. In addition, it implements a barrier service to synchronize workers when needed.

There are three architectural alternatives for implementing the job master:

1) **A singleton job master**: A single job master process serves all Twister2 jobs in a cluster. It runs as a long running service. All jobs are tracked by this single job manager. Main disadvantage of this solution is that it puts too much pressure on a single process to manage all jobs. It may introduce scalability problems.
2) **A separate job master for each Job**: A separate job master process is initiated for each job. This uses more cluster resources, but provides better job isolation.
3) **Job master in the client**: When users submit a job, submitting client runs the job master as a thread. This thread has to run until the job completes. One disadvantage of this solution is that the client has to run inside the cluster, since all workers need to connect to the Job Master. Another disadvantage is that this solution is not really suitable for long running jobs.

Flink [2] implements the job master as a single long running process as in the option 1. Heron [3] and Hadoop MapReduce v2 [18] implements the option 2. Spark [1] implements both the options 2 and 3. We designed the job master as a separate process for each job. It can be initiated as a separate process on the cluster or in the submitting client. So, it covers both the options 2 and 3. Users specify the location of the job master in the configuration files when submitting the job.

D. Twister2 Dashboard

Dashboard runs in a web server. It is a singleton long running service. It presents the job related data to users. Users monitor their jobs through Dashboard. The job masters collect status data from workers and transfer them to Dashboard. Another option for users to monitor their job status is to enable client logging and checking the log files from client machines.

E. Storage Provisioning

Twister2 workers have the following four types of storage options:

1) **Memory volume**: This is a memory based file system. Its primary purpose is to provide high performance data sharing among the workers on the same pod in Kubernetes clusters.
2) **Local disk based volatile volume**: This is a volume based on the physical disk of the machine running the worker. The data on this volume will be deleted after the job completion. Therefore, only the intermediate data during the job computation should be saved to this volume.
3) **Persistent storage**: Currently we support an NFS based persistent storage volume. All workers access the same shared NFS drive. All saved data to this volume will live there after the job completion. Workers can output the results of the computations to this volume. They can also save log files to this drive by enabling persistent logging.
4) **HDFS**: Twister2 workers can access HDFS files installed on the cluster. They can read input data from HDFS files and write output data to them. Task system can also perform locality aware scheduling based on data locality of HDFS files.

III. Kubernetes Implementation

Kubernetes [14] is a popular open source container orchestration system developed by Google. It is designed to manage containers for both long running high priority applications and batch jobs on the same cluster. It deploys applications in nodes, scales up and down the deployments, provisions storage, provides security services, lets users monitor applications, etc. Its popularity and feature set attracted the attention of many big data systems. All popular big data systems such as Spark [2] and Flink [3] are already ported to Kubernetes.

[2]https://spark.apache.org/docs/latest/running-on-kubernetes.html

[3]https://ci.apache.org/projects/flink/flink-docs-stable/ops/deployment/kubernetes.html

A. Controller Selection

Kubernetes runs applications in a cluster as pods. Pods can be considered as light weight virtual machines. Each pod is assigned dedicated CPU, memory and storage resources. Each pod is also assigned a unique IP for networking. Pods run applications as containers. Each pod runs one or multiple containers. Pods on the same node are isolated from the others. Since many applications in the cloud require multiple concurrent instances, Kubernetes provides controllers to run a group of pods in parallel. Controllers initiate, scale up/down, update and clean up pods.

When implementing Twister2 on Kubernetes, first we need to decide the type of controller we will be using to manage workers and the job master. Kubernetes provides many controllers to manage pods. Possible alternatives are Jobs, Deployments, StatefulSets or plain pods [4].

Job controller is designed to manage jobs that are supposed to complete after finishing its computation. Since we would like to support both batch and long-running streaming jobs, we decided not to use the Job controller. Deployments and StatefulSets are similar controllers that are designed to support long running applications. StatefulSets provide the extra services of unique ordinal indexes. In addition, StatefulSets support persistent storage. These features are important for us to implement fault tolerance. Therefore, we decided to implement Twister2 jobs with StatefulSets.

We create two StatefulSet controllers for each job. First one manages the job master and the second one manages the workers. We create a service with cluster local IP address for the StatefulSet controller of the job master. Workers can discover the job master by using this service. The StatefulSet controller of the workers uses a headless service by default. They only use a service with an IP address when they want to provide services to external entities. In this case, we employ NodePort service.

Since StatefulSets are designed to run long running applications, they expect pods to run continually. If a StatefulSet pod completes and exits, that pod is restarted automatically. Therefore, we need to monitor the status of worker pods and terminate them explicitly. Job master handles this task. When workers finish computations, they send a message to the job master and wait for the job master to delete their pods. Job master deletes all worker pods and its own pod when the Twister2 job completes.

Spark works with plain pods. When a job is submitted, a pod is created for Spark driver. The driver creates executor pods as plain pods. It deletes the executor pods when the job completes. Flink uses Kubernetes Deployment controller to start Flink Job Manager and Task Managers.

B. Worker Implementation

There are a number of architectural choices when implementing Twister2 workers in pods:

[4]https://kubernetes.io/docs/concepts

1) **One Worker One Pod**: Exactly one worker runs in every container and every container runs in a separate pod. This is the recommended approach for long running containerized applications in Kubernetes.
2) **One Worker One Container**: Exactly one worker runs in every container, but multiple containers may run in a pod. This helps to get better performance for data exchanges among the workers on the same pod. They can share memory based volumes and physical disks on the same node. Pod failures may affect all workers on the same pod.
3) **One Worker One Process**: Each worker runs in a separate process and multiple workers may run in each container. This approach does not provide resource isolation among workers on the same pod. It also leaves process management to Twister2 runtime.

We take the second approach and let the user specify the number of workers in pods. This option also covers the first one, when the user sets the number of workers on the pods to 1. However, when we execute OpenMPI jobs, then we go with the third option. We start a single container in each pod, and OpenMPI starts MPI worker processes in those containers. OpenMPI initiates the worker processes based on the number of workers specified for pods. OpenMPI performs process management in this case.

Before starting a worker on a pod, we perform a number of initialization steps. First, we get the job package and read the configuration files. We setup logging. We calculate the worker ID by using the unique pod indexes and container indexes. We setup volatile and persistent storage if they are requested. We register the worker with the Job Master and start the worker thread.

C. OpenMPI Support

OpenMPI requires password-free-ssh enabled among the workers in a job and the hostfile generated on the mpi master pod. To enable password-free SSH, an ssh key pair is distributed to all pods in the job by using Kubernetes Secret object. This pair is mounted to the secret volume on each pod. An initialization script starts OpenSSH on each pod and uses this key pair to let other pods to have password-free ssh access. Users need to generate an ssh key pair and create a Secret object on Kubernetes master before submitting OpenMPI jobs. We provide instructions and a template for that. Each user needs to create only one Secret object and it can be reused by all subsequent job submissions.

First worker in the StatefulSet acts as the MPI master. It first watches other pods to start and get their IP addresses. It saves these IP addresses to the hostfile. Then, it executes mpirun to start Twister2 workers.

D. Uploader Implementation

We currently support two types of job package transfer to workers:

1) **HTTP Uploader**: The client program uploads the job package to a web server running in the cluster or to S3 storage in the case of AWS. A script in worker pods download the job package using HTTP with wget utility. When there are multiple workers on a pod, only the first worker downloads and extracts the job package. We designed a sample web server pod to run in the cluster as the uploader server.
2) **Direct Uploader**: The client program uses ”kubectl cp” command to upload the job package to each pod in the job. The client program watches each pod in the job and gets the pod status events. When the first container in a pod becomes ”Running”, it uploads the job package directly to that pod. This method does not require running a separate web server or accessing S3 storage, however it is not scalable. The client needs to upload the job package to all pods separately. This should only be used in jobs with fewer workers.

IV. MPI Based Schedulers: Slurm, Nomad, Standalone

MPI implementations handle worker process initialization and setting up networking among workers in distributed clusters. MPI is generally used together with Network File System, so all nodes in the cluster have the same shared file system. There is no need to transfer the job package to worker nodes. Instead, the submitting client puts the job package to a directory on NFS. MPI assigns a unique rank for each MPI process starting from 0 and increasing sequentially. We use this rank as the worker ID in Twister2 jobs.

There is no need to run Job Master in MPI clusters unless a Twister2 driver will be running in the job. If the job master is used, it can run either in the submitting client or at the first worker. When it runs at the first worker, the first worker becomes the job master and we start an extra worker in the job.

Twister2 currently supports bare metal MPI clusters, Slurm and Nomad schedulers. We construct the proper command for each scheduler and submit it for execution. They start Twister2 workers and those workers execute user provided codes. Log messages are transferred to the job submitting client.

V. Performance Tests

We measured Twister2 job startup delays in Kubernetes and bare metal OpenMPI cluster. We installed a Kubernetes cluster at AWS using Kubernetes Operations (kops) tool and installed an MPI cluster at AWS using AWS ParallelCluster tool. Compute nodes were running on a machine instance of type c5.9xlarge that has 36 cores, 72GiB of RAM and 10Gbps of network bandwidth. There were 30 compute nodes on each cluster. Each Twister2 worker is assigned 1 CPU core and 256MB of memory with no hard drive.

We measured job startup delays for single jobs and multiple concurrent jobs. Fig. 2 shows delays to start up single Twister2 jobs. OpenMPI starts up jobs much faster. Kubernetes is slower since it starts a container for each worker. Container initialization usually takes much longer. OpenMPI just starts processes in remote nodes using password-free-ssh and sets

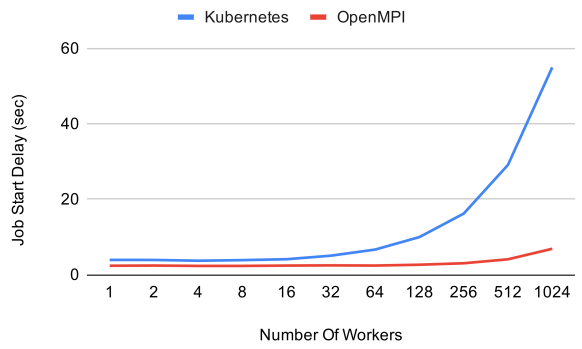

Fig. 2. Single job startup delays for Twister2

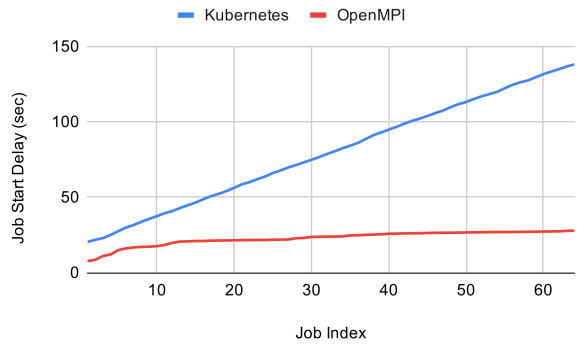

Fig. 3. 64 concurrent job startup delays each with 16 workers for Twister2

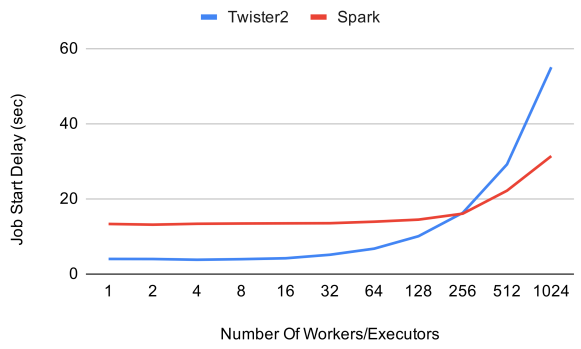

Fig. 4. Single job startup delays for Twister2 and Spark at Kubernetes

Fig. 5. 64 concurrent job startup delays for Twister2 and Spark at Kubernetes

up the networking. On the other hand, Kubernetes provides CPU and memory isolation and many other services.

It takes around 4 seconds to start up a job with one worker in Kubernetes. Since we have 30 nodes in the cluster, it takes around 5 seconds to start up 32 workers. One worker is started on each node except two nodes. Starting a job with 1024 workers takes 55 seconds in Kubernetes. 34 or 35 workers started on each node. It takes an extra 1.5 seconds to start up a worker on a node. However, starting an OpenMPI job with 1024 workers takes only 7 seconds.

Fig. 3 shows delays to start up 64 concurrent Twister2 jobs each with 16 workers. We submit 64 jobs concurrently using a 16-core machine. We measure job startup delays for each job. Delays in the figure shows job starting delays for each job sorted from fastest to slowest. The first job is started in 7.5 seconds by OpenMPI and in 20 seconds by Kubernetes. Kubernetes is much slower compared to OpenMPI. In total, it takes 138 seconds to start up all 64 jobs on Kubernetes and 28 seconds on OpenMPI.

We also measured job startup delays for Spark on Kubernetes for single jobs and multiple concurrent jobs. Comparison of Twister2 and Spark single job startup delays is shown in Fig. 4. Spark creates jobs in two steps. When a job is submitted, it first creates a driver pod in the cluster. This driver pod creates executor pods. Therefore, starting a single job even with one worker takes 13 seconds. However, it has better scalability compared to Twister2 in terms of startup costs. While Twister2 starts up jobs with fewer than 256 workers faster, Spark starts up jobs with more than 256 workers faster. This is because Spark creates plain pods while Twister2 uses StatefulSet controller. Kubernetes initiates plain pods in scale much faster than StatefulSet pods.

Comparison of Twister2 and Spark multiple concurrent job startup delays is shown in Fig. 5. Similar to single job startup case, Twister2 starts up initial jobs faster, but Spark performs better on scale.

VI. Conclusions and Future Work

In this paper, we presented Twister2 cross platform resource scheduler. We identified provided services and explained implementations details. We explored job startup delays for single jobs and multiple concurrent jobs. We compared OpenMPI and Kubernetes startup delays. The results show that OpenMPI can start up jobs much faster particularly the jobs with high number of workers. While OpenMPI can start a job with 1024 workers in 7 seconds, Kubernetes starts it in 55 seconds.

We also compared job startup delays for Twister2 and Spark. While Twister2 starts up jobs with less than 256 workers faster, Spark starts up jobs with higher number of workers faster.

We are working on implementing fault tolerance for Twister2. We have implemented worker and job master rejoins after failures. We are working on implementing fault tolerance

in all Twister2 layers to make the jobs fault tolerant. We are also working on to compare the performance of parallel computations in Kubernetes and OpenMPI environments.

ACKNOWLEDGMENTS

This work was partially supported by NSF CIF21 DIBBS 1443054 and the Indiana University Precision Health initiative. We thank Intel for their support of the Juliet and Victor systems, and extend our gratitude to the FutureSystems team for their support with the infrastructure.

REFERENCES

[1] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[3] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[4] "Twister2: Design of a big data toolkit," 2017, technical Report. [Online]. Available: http://dsc.soic.indiana.edu/publications/Twister2.pdf

[5] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big data, simulations and hpc convergence," in *Big Data Benchmarking*, T. Rabl, R. Nambiar, C. Baru, M. Bhandarkar, M. Poess, and S. Pyne, Eds. Cham: Springer International Publishing, 2016, pp. 3–17.

[6] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[7] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 35:1–35:35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389044

[8] X. Lu, D. Shankar, S. Gugnani, and D. K. D. K. Panda, "High-performance design of apache spark with RDMA and its benefits on various workloads," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 253–262.

[9] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between hpc and big data frameworks," *Proc. VLDB Endow.*, vol. 10, no. 8, pp. 901–912, Apr. 2017. [Online]. Available: https://doi.org/10.14778/3090163.3090168

[10] A. Gittens, K. Rothauge, S. Wang, M. W. Mahoney, L. Gerhardt, J. Kottalam, M. Ringenburg, K. Maschhoff *et al.*, "Accelerating large-scale data analysis by offloading to high-performance computing libraries using alchemist," *arXiv preprint arXiv:1805.11800*, 2018.

[11] S. Kamburugamuve, P. Wickramasinghe, K. Govindarajan, A. Uyar, G. Gunduz, V. Abeykoon, and G. Fox, "Twister:net - communication library for big data processing in hpc and cloud environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 383–391. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00055

[12] K. Govindarajan, S. Kamburugamuve, P. Wickramasinghe, V. Abeykoon, and G. Fox, "Task scheduling in big data - review, research challenges, and prospects," in *2017 Ninth International Conference on Advanced Computing (ICoAC)*, 2017.

[13] P. Wickramasinghe, S. Kamburugamuve, K. Govindarajan, V. Abeykoon, C. Widanage, N. Perera, A. Uyar, G. Gunduz, S. Akkas, and G. Fox, "Twister2: Tset high-performance iterative dataflow," in *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. IEEE, 2019, pp. 55–60.

[14] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, pp. 10:70–10:93, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2898442.2898444

[15] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[16] Nomad cluster. Accessed: Oct 18 2018. [Online]. Available: https://www.nomadproject.io/

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[18] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633