Overcoming Hadoop Scaling Limitations through Distributed Task Execution

Ke Wang*, Ning Liu*, Iman Sadooghi*, Xi Yang*, Xiaobing Zhou§,
Tonglin Li*, Michael Lang†, Xian-He Sun*, Ioan Raicu*‡
*Illinois Institute of Technology, §Hortonworks Inc.
†Los Alamos National Laboratory, ‡Argonne National Laboratory
{kwang22, nliu8, isadoogh, xyang34}@hawk.iit.edu, xzhou@hortonworks.com,
tli13@hawk.iit.edu, mlang@lanl.gov, sun@iit.edu, iraicu@cs.iit.edu

***Abstract*—Data driven programming models like MapReduce have gained the popularity in large-scale data processing. Although great efforts through the Hadoop implementation and framework decoupling (e.g. YARN, Mesos) have allowed Hadoop to scale to tens of thousands of commodity cluster processors, the centralized designs of the resource manager, task scheduler and metadata management of HDFS file system adversely affect Hadoop's scalability to tomorrow's extreme-scale data centers. This paper aims to address the YARN scaling issues through a distributed task execution framework, MATRIX, which was originally designed to schedule the executions of data-intensive scientific applications of many-task computing on supercomputers. We propose to leverage the distributed design wisdoms of MATRIX to schedule arbitrary data processing applications in cloud. We compare MATRIX with YARN in processing typical Hadoop workloads, such as WordCount, TeraSort, Grep and RandomWriter, and the Ligand application in Bioinformatics on the Amazon Cloud. Experimental results show that MATRIX outperforms YARN by 1.27X for the typical workloads, and by 2.04X for the real application. We also run and simulate MATRIX with fine-grained sub-second workloads. With the simulation results giving the efficiency of 86.8% at 64K cores for the 150ms workload, we show that MATRIX has the potential to enable Hadoop to scale to extreme-scale data centers for fine-grained workloads.**

Keywords—data driven programming model; MapReduce; task execution framework; scheduling; extreme scales

I. INTRODUCTION

Applications in the Cloud domain (e.g. Yahoo! weather [1], Google Search Index [2], Amazon Online Streaming [3], and Facebook Photo Gallery [4]) are evolving to be data-intensive that process large volumes of data for interactive tasks. This trend has led to the programming paradigm shifting from the compute-centric to the data driven. Data driven programming models [5], in the most cases, decompose applications to embarrassingly parallel tasks that are structured as Direct Acyclic Graph (DAG) [6]. In an application DAG, the vertices are the discrete tasks, and the edges represent the data flows from one task to another.

MapReduce [7] is the representative of the data driven programming model that aims at processing large-scale data-intensive applications in Cloud on commodity processors (either an enterprise cluster, or private/public Cloud [61]). In MapReduce, applications are divided into two phases (i.e. Map and Reduce) with an intermediate shuffling procedure, and the data is formatted as unstructured (key, value) pairs. The programming framework is comprised of three major components: the *resource manager* manages the global compute nodes, the *task scheduler* places a task (either a map task or a reduce task) on the most suitable compute node, and the *file system* stores the application data and metadata.

The first generation Hadoop [8] (Hadoop_v1, circa 2005) was the open-source implementation of the MapReduce. In Hadoop_v1, the centralized *job tracker* plays the roles of both resource manager and task scheduler; the HDFS is the file system [9] to store the application data; and the centralized *namenode* is the file metadata server. In order to promote Hadoop to be not only the implementation of MapReduce, but one standard programming model for a generic Hadoop cluster, the Apache Hadoop community developed the next generation Hadoop, YARN [10] (circa 2013), by decoupling the resource management infrastructure with the programming model. From this point, when we refer to Hadoop, we mean YARN.

YARN utilizes a centralized *resource manager* (RM) to monitor and allocate resources. Each application delegates a centralized *per-application master* (AM) to schedule tasks to resource *containers* managed by the *node manager* (NM) on the computing nodes. The HDFS file system and centralized metadata management remain the same. Decoupling the resource management infrastructure with the programming model enables Hadoop to run many application frameworks (e.g. MapReduce, Iterative application, MPI, and scientific workflows) and eases the resource sharing of the Hadoop cluster. Besides, as the scheduler is separated from the RM with the implementation of the per-application AM, the Hadoop has achieved unprecedented scalability. Similarly, the Mesos [11] resource sharing platform is another example of the scalable Hadoop programming frameworks.

However, there are inevitable design issues that prevent Hadoop from scaling to extreme scales, the scales that are 2 to 3 orders of magnitude larger than that of today's distributed systems; similarly, today's scales do not support several orders of magnitude fine grained workloads (e.g. sub-second tasks). The first category of issues come from the centralized paradigm. Firstly, the centralized RM of YARN is a bottleneck. Although the RM is lightweight due to the framework separation, it would cap the number of applications supported concurrently as the RM has limited processing capacity. Secondly, the centralized per-application AM may limit the task placement speed when the task parallelism grows enormously for the applications in

certain domains. Thirdly, the centralized metadata management of HDFS is hampering the metadata query speed that will have side effects on the task placement throughput for data-locality aware scheduling. The other issue comes from the fixed division of Map and Reduce phases of the Hadoop jobs. This division is simple and works well for many applications, but not so much for more complex applications, such as iterative MapReduce [12] that supports different levels of task parallelism, and the irregular applications with random DAGs. Finally, the Hadoop framework is not well suited for running fine-grained workloads with task durations of sub-seconds, such as the lower-latency interactive data processing applications [25]. The reason is twofold. One is that the Hadoop employs a pull-based mechanism. The free containers pull tasks from the scheduler; this causes at least one extra Ping-Pong overhead per-request in scheduling. The other one is that the HDFS suggests a relatively large block size (e.g. 64MB) when partitioning the data, in order to maintain efficient metadata management. This confines the workload's granularity to be tens of seconds. Although the administrators of HDFS can easily tune the block size, it involves manual intervention. Furthermore, too small block sizes can easily saturate the metadata server.

This work proposes to utilize an existing distributed task execution framework, MATRIX [34][44][48], to do scalable task placement for Hadoop workloads, with the goal of addressing the Hadoop scaling issues. MATRIX was originally developed to schedule the fine-grained data-intensive many-task computing (MTC) [45] applications on supercomputers. MATRIX delegates a scheduler on each compute node to manage local resources and schedule tasks, and utilizes a data-aware work stealing technique to optimize task placement for the best load balancing and exploitation of data-locality. A distributed key-value store, namely ZHT [15][16][17][18], is applied to store task metadata in a scalable and fault tolerant way. We leverage the distributed design wisdoms of MATRIX in scheduling data processing applications in clouds. We compare MATRIX with YARN using typical Hadoop workloads, such as WordCount, TeraSort, RandomWriter, and Grep, as well as an application in Bioinformatics. We also run and simulate MATRIX with fine-grained sub-second workloads and MATRIX shows the potential to enable Hadoop to scale to extreme scales. **The contributions of this paper are highlighted as follows:**

- Proposed to address scalability issues of Hadoop through decentralized scheduling with MATRIX
- An inclusive comparison between MATRIX and YARN with both benchmarking and real application workloads, up to 256 cores on the AWS Cloud
- An evaluation of the scalability of MATRIX for fine-grained sub-second workloads through both real systems and simulations at extreme scales

The rest of this paper is organized as follows. Section II presents the related work. Section III analyzes the Hadoop design issues, introduces MATRIX, and shows how MATRIX can address the Hadoop scaling issues. Section IV presents the evaluation results. Section V concludes the paper and lists the future work.

II. Related Work

Ever since the emergence of the MapReduce and Cloud computing, the Apache community disclosed the Hadoop_v1 [8] implementation. As the system scale is growing exponentially and the applications are experiencing data explosion, there are extensive research efforts that aimed at addressing the scalability issues, such as resource managers, task schedulers and metadata management, to keep Hadoop scalable with the same pace of the growth of distributed systems and data volumes in data processing applications.

YARN [10] and Mesos [11] are two frameworks that decouple the resource management infrastructure from the task scheduler of the programming model to enable efficient resource sharing in general commodity Hadoop clusters for different data-intensive applications. Both of them apply a centralized RM to allocate resources to applications. The AM then will be in charge of scheduling tasks onto the allocated compute nodes. The difference between them is that Mesos employs an AM for one category of applications, while YARN is much finer grained in that it uses an AM per application, which, in theory, should be more scalable. Although they have improved the scalability and efficiency of the resource sharing in Hadoop clusters significantly with the separation, the centralized RM is still a barrier towards extreme scales or of the support for fine-grained workloads. Omega [43] is a distributed scheduling framework for Google's data-intensive production workloads. Omega deploys multiple schedulers, and each one maintains a private resource state of the whole cluster to claim resources and make scheduling decisions through an atomic operation. The private states are synchronized with a master copy of the global state. This design eliminates the bottleneck of the centralized resource allocator. However, the global state synchronization introduces considerable overheads. In addition, Omega is not a system in the public domain that Hadoop can take advantage of. The Hadoop coupled with MATRIX is a step towards a practical system integration that can accelerate Hadoop's scalability.

Another research aims to improve the Hadoop schedulers. Most of work focuses on optimizing the scheduling policies to meet different requirements in a centralized task scheduler. The Hadoop default schedulers include the Capacity Scheduler (CS) [19], the Fair Scheduler (FS) [20] and the Hadoop On Demand (HOD) Scheduler (HS) [21]. Each of them has a different design goal: the CS aims at offering resource sharing to multiple tenants with the individual capacity and performance SLA; the FS divides resources fairly among job pools to ensure that the jobs get an equal share of resources over time; the HS relies on the Torque resource manager to allocate nodes, and allows users to easily setup Hadoop by provisioning tasks and HDFS instances on the nodes. Rasooli and Down proposed a hybrid scheduling approach [22] that can dynamically select the best scheduling algorithm (e.g. FIFO, FS, and COSHH [23]) for heterogeneous systems. To optimize fairness and locality, Zaharia et. al proposed a delay scheduling algorithm [24] that delays the scheduling of a job for a limited time until highly possible to schedule the job to where the data resides.

These efforts have limited advancement to the scalability because they work within a single scheduler. Some early work towards distributed resource management was GRUBER [51], which focused on distributed brokering of Grid resources. Sparrow [25] is a distributed task scheduler that applies multiple schedulers with each one knowing all the nodes to schedule fine-grained sub-second tasks. Each scheduler probes multiple nodes and implements a pushing mechanism to place tasks on the least overloaded node with early binding. This is not scalable under heterogeneous workloads that have wide distribution of task length due to the fact that tasks cannot be moved after submission for load balancing's purpose. Furthermore, Sparrow works in tandem with Spark [38], and hence vanilla Hadoop applications cannot be executed with Sparrow easily.

Another related research direction is scalable metadata management. To optimize the metadata management and usage for small files, Machey et al. provided a mechanism that utilizes the Hadoop "harballing" compression method to reduce the metadata memory footprint [26]. Zhao et al. presented a metadata-aware storage architecture [27] that utilizes the classification algorithm of merge module and efficient indexing mechanism to merge multiple small files into Sequence File, aiming to solve the namenode memory bottleneck. However, neither work touched the base of addressing the scalability issues of the centralized namenode in processing the tremendously increased large amount of metadata access operations. Haceph [29] is a project that aims to replace the HDFS by the Ceph file system [30] integration with the Hadoop POSIX IO interfaces. Ceph uses a dynamic subtree partitioning technique to divide the name space onto multiple metadata servers. This work is more scalable, but may not function well under failures due to the difficulty of re-building a tree under failures.

III. Distributed Data Driven Task Placement

In this section, we first analyze the design issues that result in the scalability problems of Hadoop. We then introduce the MATRIX task execution framework, and propose to apply its distributed designs to address the issues.

A. Hadoop Design Issues

Although YARN has improved the Hadoop scalability significantly, there are fundamental design issues that are capping the scalability of Hadoop towards extreme scales.

1) Centralized resource manager: The resource manager (RM) is a core component of the Hadoop framework. It offers the functionalities of managing, provisioning, and monitoring the resources (e.g. CPU, memory, network bandwidth) of the compute nodes of a Hadoop cluster. Although YARN decouples the RM from the task scheduler to enable Hadoop running different frameworks, the RM is still centralized. One may argue that the centralized design should be scalable as the processing ability of a single compute node is increasing exponentially. However, the achieved network bandwith from a single node to all the compute nodes is bounded.

2) Application task scheduler: YARN delegates the scheduling of tasks of different applications to each individual application master (AM), which makes decisions to schedule tasks among the allocated resources (in the form of containers managed by the node manager on each compute node). Task scheduling component is distributed in the sense of scheduling different applications. However, from per-application's perspective, the task scheduler in the AM still has a centralized design that has too limited scheduling ability to meet the evergrowing task amount and granularity. In addition, Hadoop employs a pull-based mechanism, in which, the free containers pull tasks from the scheduler. This causes at least one extra Ping-Pong overhead in scheduling. One may have doubt about the exsistance of an application which can be decomposed as so many tasks that needs a resource allocation of all the compute nodes. But, it is surely happening given the exponential growth of the application data sizes.

3) Centralized metadata management: The HDFS is the default file system that stores all the data files in the datanodes through random distributions of data blocks, and keeps all the file/block metadata in a centralized namenode. The namenode monitors all the datanodes. As the data volumes of applications are growing in a fast rate, the number of data files are increasing significantly, leading to much higher demands of the memory footprint and metadata access rate that can easily overwelm the centralized metadata management. Things could be much worse for abundant small data files. In order to maintain an efficient metadata management in the centralized namenode, the HDFS suggests a relatively large block size (e.g. 64MB) when partitioning the data files. This is not well suited for the fine-grained lower latency workloads.

4) Limited data flow pattern: The Hadoop jobs have the fixed two-layer data flow pattern. Each job is decomposed as embarasingly parallel map-phase tasks with each one processing a partition of the input data. The map tasks generate intermediate data which is then aggregated by the reduce-phase tasks. Although many applications follow this simple pattern, there are still quite a few applications that are decomposed with much more complex workload DAGs. One example is the category of irregular parallel applications that have unpredictable data flow patterns. These applications need dynamic scalable scheduling echniques, such as work stealing, to achive distributed load balancing. Another category of the applications with complex data flow patterns are the iterative applications that aim to find an optimal solution through the convergence after iterative computing steps. Though one can divide such an application as multiple steps of Hadoop jobs, Hadoop isn't able to run these application steps seamlessly.

B. MATRIX Task Execution Framework

MATRIX is a distributed task execution framework designed to schedule MTC scientific applications on tightly-coupled cluster and supercomputers. MATRIX achieved 87% efficiency at 200 cores in scheduling two data-intensive applications, Image stacking in Astronomy and All-pairs in Bioinformatics, with average task lengths of 100ms [31][53].

Figure 1 shows the MATRIX architecture. MATRIX is fully distributed by delegating one scheduler on each

compute node. On the same compute node, there is an executor and a key-value store (KVS) server. The scheduler is responsible for managing local resource and placing tasks onto the most suitable compute node, for optimizing both load balancing and data-locality. The executor executes the tasks that have been scheduled in local with multiple (usually equals to the number of cores of a compute node) parallel executing threads. MATRIX utilizes the KVS, ZHT [52][15], to keep the metadata of tasks and data files in a scalable way. Each scheduler is initialized as a ZHT client that has a global knowledge of all the ZHT servers, and can query and update the task metadata transparently through the APIs (lookup, insert, remove) that hash the record key to direct the requests to correct server. ZHT scaled up to 32K cores with high throughput of 18M ops/sec and low latency of 1.5ms [15].

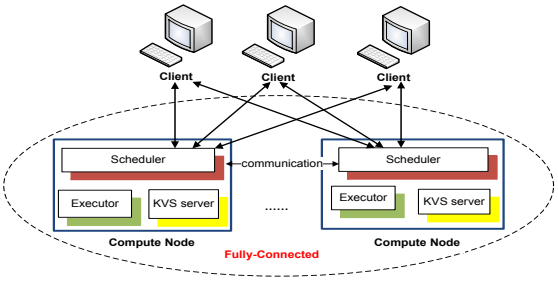

Figure 1: MATRIX architecture

The schedulers implemented a data-aware work stealing technique to optimize both load balancing and data locality. Each scheduler has four task queues, namely waiting queue (*WaitQ*), dedicated ready queue (*LReadyQ*), shared ready queue (*SReadyQ*), and complete queue (*CompleteQ*), which store tasks in different states. The *WaitQ* keeps tasks that wait for their parents to be done. The *LReadyQ* stores ready tasks whose majority of required data is local; these tasks are only executed locally. The ready tasks in the *SReadyQ* can be migrated to any node through the random work stealing technique (the idle schedulers steal tasks from the overloaded randomly chosen neighbors) for the purpose of load balancing, because the required data size of the tasks is so small that transferring it involves little overhead. The finished task is moved to the *CompleteQ*. The scheduler then updates the metadata of each child to notify the completion of this parent by querying ZHT.

C. Leveraging MATRIX Distributed Design Wisdoms

Although MATRIX was designed for scheduling fine-grained MTC data-intensive applications on supercomputers [65][67], we show how to leverage the MATRIX distributed design wisdoms to overcome the Hadoop scaling limitations for arbitrary data processing applications.

1) Distributed Resource Management: Instead of employing a single RM to manage all the resources as the Hadoop framework does, in MATRIX, each scheduler maintains a local view of the resources of an individual node. The per-node resource manager is demanded, because the physical computing and storage units are not only increasing in terms of sizes, but are becoming more complex, such as hetergeneous cores and various types of storages (e.g. NVRAM, spinning Hard Disk, and SSD). The demand is more urgent in a virtualized Cloud environment, in which, the RM also needs to conduct resource binding and monitoring, leading to more workloads.

2) Distributed Task Scheduling: The schedulers are not only in charge of resource management, but responsible for making scheduling decisions through the data-aware work stealing technique. The distributed scheduling architecture is scalable than a centralized one, since all the schedulers participate in making scheduling decisions. In addition, MATRIX can tune the work stealing parameters (e.g. *number of tasks to steal*, *number of neighbors*, *polling interval*) at runtime to reduce the network communication overheads of distributed scheduling. The distributed architecture enables the system to achieve roughly the linear scalability as the system and workload scale up.

3) Distriuted Metadata Management: We can leverage distributed KVS to offer a flat namespace in managing the task and file metadata. Comparing with other ways, such as sub-tree partitioning [32] and consistent hashing [33], flat namespace hashing has the ability to achieve both good load balancing, and faster metadata accessing rate with zero-hop routing. In sub-tree partitioning, the namespace is organized in sub trees rooted as individual directory, and the sub trees are managed by multiple metadata servers in a distributed way. As the directories may have wide distribution of number of files and file sizes, the partitioning may result in poor load balancing. In consistency hashing, each metadata server has partial knowledge of the others, leading to extra routing hops needed to find the right server that can satisfy a query. For example, MATRIX combines the task and file metadata as (*key*, *value*) pairs that represent the data dependencies and data file locations and sizes of all the tasks in a workload. For each task, the *key* is the "taskId", and the *value* specifies the "parent" and "child" tasks, as well as the names, locations, and sizes of the required data files. One may argue that the full connectivity of the KVS will be an issue at extreme scales. However, our pervious simulation results [13][14] showed that in a fully connected architecture, the number of communication messages required to maintain a reliable service is trivial when comparing with the number of request-processing messages.

4) Fault Tolerance: Fault tolerance is an important design concerns, especially for extreme-scale distributed systems that have high failure rates. MATRIX can tolerate failures with a minimum effort due to the distributed nature, the stateless feature of the schedulers, and the integration of ZHT. Failures of a node only affect the tasks, data files, and metadata on that node, and can be resolved easily as follows. The affected tasks can be acknowledged and resumitted to other schedulers; A part of the data files were copied and cached in other nodes when they were transmitted for executing tasks. In the future, we will rely on the underneath file system to handle the affected files; As ZHT stores the metadata, and ZHT has implemented failure/recovery, replication and consistency mechanisms, MATRIX needs to worry little about the affected metadata.

5) Elastic Property: MATRIX allows the resources to be dynamically expanded and shrinked in the elastic Cloud

environment. The resource shrinking is regarded as resource failure in terms of consequences, and can be resolved through the same techniques of handing failures. When adding an extra compute node, a new scheduler and ZHT server will also be introduced. ZHT has already implemented a dynamic membership mechanism to undertake the newly added server. This mechanism can also be used in MATRIX to notify all the existing schedulers about the extra added scheduler.

6) Support of arbitrary application DAG: MATRIX can support much broader categories of data-intensive applications with various data flow patterns, such as the Hadoop jobs, the iterative applications, and the irregular parallel applications. We will show how MATRIX performs for typical Hadoop applications. The MATRIX clients take any arbitrary application DAG as input. Before submitting the tasks, the clients insert the initial task dependency information into ZHT. Later, the schedulers update the dependency with the added data size and locality information when executing tasks, through the ZHT interfaces. We believe that MATRIX could be used to accelerate a large number of parallel programming systems, such as Swift [46], Pegasus [47], and Charm++ [49].

IV. EVALUATION

In this section, we evaluate MATRIX by comparing it with YARN in processing typical Hadoop workloads, such as WordCount, TeraSort, RandomWriter and Grep, as well as an application in Bioinformatics, on the Amazon cloud up to 256 cores. We first run YARN with these workloads and obtain the trace files. Through the trace files, we generate workload DAGs that become the inputs of MATRIX. We also evaluate the scalability of MATRIX through simulations up to extreme scales with sub-second workloads. We aim to show that MATRIX is not only able to perform better than YARN for the workloads that are tailored for YARN, but also has the ability to enable Hadoop to scale to extreme scales for finer-grained sub-second workloads.

A. YARN Configuration

In the experiments, we use YARN version 2.5.1. Although newer version of YARN is released very frequently, we argue that it does not have a perceivable impact on what we are trying to study and present in this paper. For example, the centralized design and implementation are not going to change significantly in the coming releases in the near future. Here, we configure the HDFS block size to be 16MB. The usual Hadoop cluster configuration is from 32MB to 256MB, but we believe this is a reasonable change as we focus on studying the scheduling overhead of the frameworks. Increasing the block size will only increase the task length (or execution time) of the map tasks and decrease the total number of map tasks. Our conclusions on the scheduling performance improvement do not vary with the different HDFS block sizes. To fairly capture the traces of each task, we use the default map and reduce logging service. The logging mode is INFO, which is lightweight comparing with DEBUG or ALL. This is to minimize the impact of logging on Hadoop performance. To best characterize the overheads of the centralized scheduling and management, we use a stand-alone instance to hold the NameNode and ResourceManager daemons. The isolation of master and slaves guarantees that the performance of the Hadoop master is not compromised by co-located NodeManager or DataNode daemons.

B. Experiment Environment

We run experiments on Amazon Cloud using the "m3.large" instances up to 256 CPUs (128 instances). Each instance has 2 CPUs, 6.5ECUs, 7.5GB memory and 16GB storage of SSD, and uses the Ubuntu 14.04LTS distribution. For both YARN and MATRIX, on one instance, we control the number of tasks executed in parallel to be 2 (equals to the number of CPUs).

C. Definitions of the Metrics

We define two metrics to evaluate the performance:

1) Efficiency: The Efficiency is the percentage of the ideal running time (T_{ideal}) to the actual running time (T_{actual}) of a workload, which quantifies the average utilization of the system. The higher efficiency ($(100 \times T_{ideal}/T_{actual})\%$) indicates less scheduling overheads.

Given a workload that has p phases and one phase cannot be started until the previous one has been finished, we can compute the ideal running time, $T_{idal} = \sum_{\lambda=1}^{p} T_{ideal(\lambda)}$, in which $T_{ideal(\lambda)}$ is the ideal running time of the λth phase. Assuming in the λth phase, on average, each core is executing k tasks with an average length of l. Therefore, $T_{ideal(\lambda)} = k \times l$.

2) Average Task-Delay Ratio: The **average Task-Delay (*td*) Ratio**, $\overline{r_{td}}$, is computed as the normalized difference between the average ideal task turnaround (*itt*) time, $\overline{T_{itt}}$, and the average actual task turnaround (*att*) time $\overline{T_{att}}$, which is $\overline{r_{td}} = (\overline{T_{att}} - \overline{T_{itt}})/\overline{T_{itt}}$. For each task n, the turnaround time (*tt*), denoted as $T_{tt(n)}$ is the time interval between the time when the task is launched and the time when the task is finished. Both MATRIX and YARN can record the detailed timestamps of each task, from which, we can know the turnaround time of each task. Therefore, we can compute $\overline{T_{att}}$ after running a workload that include k tasks: $\overline{T_{att}} = \frac{\sum_{n=1}^{k} T_{tt(n)}}{k}$. Assuming on average, each core in the system is executing k tasks with an average length of l. Therefore, the nth task needs to wait $(n-1) \times l$ time before being executed, meaning that $T_{tt(n)} = (n-1) \times l + l = nl$.

$$\overline{\mathrm{T_{ltt}}} = \frac{\sum_{n=1}^{k}(nl)}{k} = \frac{n+1}{2} \times l$$

This metric measures how fast a framework can response from each task's perspective. The smaller $\overline{\mathrm{r_{td}}}$ means faster response time and lower scheduling overheads.

We do not use the throughput as a metric, because we cannot tell how good the performance is for a given workload directly from the throughput number. These two metrics are more explicit in expressing the performance.

D. Benchmarking Hadoop Workloads

The first set of experiments run typical Hadoop workloads, such as WordCount, Terasort, RandomWriter, and Grep. The input is a 10GB data file extracted from the Wikipedia pages. We do weak-scaling experiments that process 256MB data per instance. At 128 instances, the data size is 32GB including 3.2 copies of the 10GB data file.

1) WordCount

The WordCount is a typical two-phase Hadoop workload. The map task count the frequency of each individual word in a subset data file, while the reduce task shuffles and collects the frequency of all the words. Figure 2 and Figure 3 show the performance comparisons between MATRIX and YARN.

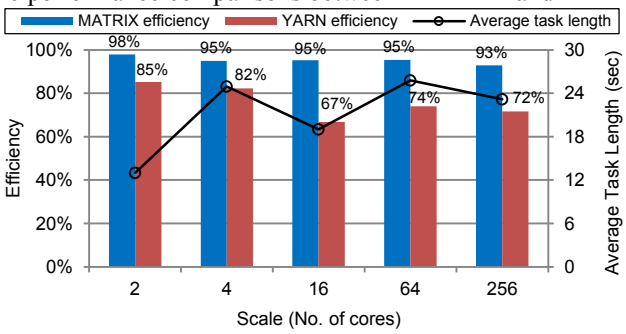

Figure 2: Efficiency for WordCount

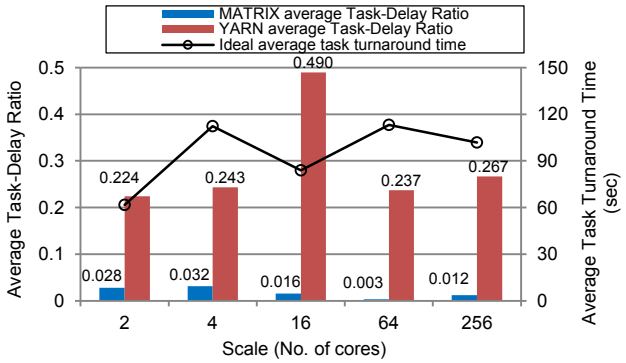

Figure 3: Average Task-Delay Ratio for WordCount

From Figure 2, we see that at all scales, MATRIX outperforms YARN by 1.26X on average for tasks with average lengths ranging from 13 to 26 sec. As scale increases from 2 cores to 256 cores, YARN's efficiency drops by 13%, while MATRIX's drops by only 5% and maintains 93%. These results indicate that MATRIX is more scalable than YARN, due to the distributed architecture and technique that optimizes both load balancing and data locality.

Figure 3 compares the average Task-Delay Ratio between MATRIX and YARN, and shows the ideal average task turnaround time for all the scales. MATRIX achieves performance that is quite close to the ideal case. The added overheads (quantified by the average Task-Delay Ratio) of MATRIX are much more trivial (20X less on average) than that of YARN. This is because each scheduler in MATRIX maintains task queues, and all the ready tasks are put in task ready queues as fast as possible. On the contrary, YARN applies a pull-based model that lets the free containers pull tasks from the application master, incurring significant Ping-Pong overheads and poor data-locality.

2) TeraSort

TeraSort is another two-phase Hadoop workload that performs in-place sort of all the words of a given data file. Figure 4 and Figure 5 present the comparison results between MATRIX and YARN.

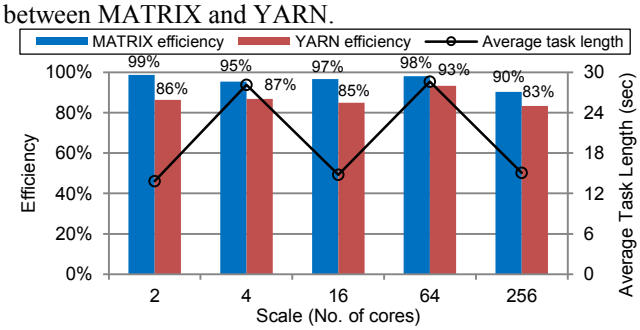

Figure 4: Efficiency for TeraSort

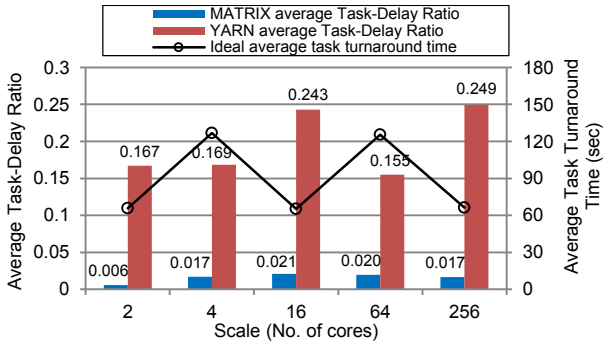

Figure 5: Average Task-Delay Ratio for TeraSort

Figure 4 illustrates the efficiency comparisons. We see that YARN can achieve performance that is close to MATRIX, however, there is still a 10% discrepancy on average. This is because in TeraSort, the time spent in the reduce phase dominates the whole process. The final output data volume is as large as the initial input one, but the number of reduce tasks is much less than that of the map tasks (In our configurations, there are 8 reduce tasks at 256 cores, and 1 reduce task at all other scales). Therefore, load balancing is less important.

However, in terms of the task turnaround time, MATRIX still achieves much faster responding time with much smaller overheads (10X less on average), according to Figure 5.

3) RandomWriter

The RandomWriter workload is consist of only map tasks, and each task writes an amount of random data to the HDFS with a summation of 10GB data per instance. Figure 6 and Figure 7 give the performance comparison results.

Figure 6 shows that at all scales, MATRIX achieves much better performance (19.5% higher efficiency on average) than YARN. In addition, as the scale increases, YARN's efficiency drops dramatically, from 95% at 2 cores to only 66% at 256 cores. The trend indicates that at larger scales, YARN efficiency would continue to decrease. On the contrary, MATRIX can maintain high efficiency at large scales and the efficiency-decreasing rate is much slower

comparing with YARN. We believe that as the scales keep increasing to extreme-scales, the performance gap between MATRIX and YARN would be getting bigger and bigger.

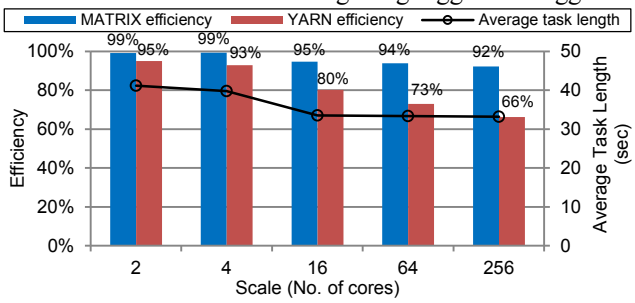

Figure 6: Efficiency for RandomWriter

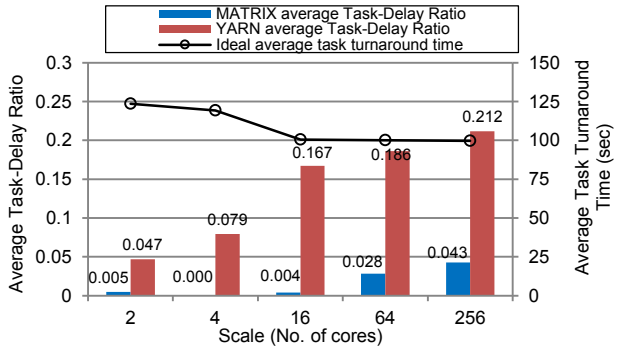

Figure 7: Average Task-Delay Ratio for RandomWriter

The reason that MATRIX can significantly beat YARN for the RandomWriter workload is not only because the distributed scheduling architecture and technique can perform better than the centralized ones of YARN, but also because MATRIX writes all the data locally while YARN writes all the data to the HDFS that may distribute the data to the remote data nodes.

In terms of the average Task-Delay Ratio presented in Figure 7, again, MATRIX can response to per-task much faster than YARN, due to the pushing mechanism used in the MATRIX scheduler that eagerly pushes all the ready tasks to the task ready queues.

4) Grep

The last Hadoop benchmark is the Grep workload that searches texts to match the given pattern in a data file. In YARN, the Grep workload is divided into 2 Hadoop jobs, namely search and sort. Both jobs have a two-phase MapReduce data pattern. However, MATRIX converts the entire Grep workload to one application DAG that has a four-phase data flow pattern. The output of the reduce phase of the search job is the input of the map phase of the sort job. The comparison results between MATRIX and YARN with the Grep workload are shown in Figure 8 and Figure 9.

Figure 8 shows that MATRIX performs much better than YARN with a performance gain of 53% on average (1.53X speedup). YARN achieves relatively low efficiency, even at 2-core scale. The reasons are two-folds. First, the Grep workload has a wide distribution of task lengths. Based on the targeting text pattern, map tasks may execute significantly different amounts of times and generate results ranging from empty to large volumes of data when given different parts of the data file. This huge heterogeneity of task length leads to poor load balancing in YARN. The other reason is that YARN needs to launch 2 Hadoop jobs for the Grep workload, which doubles the job launching overheads.

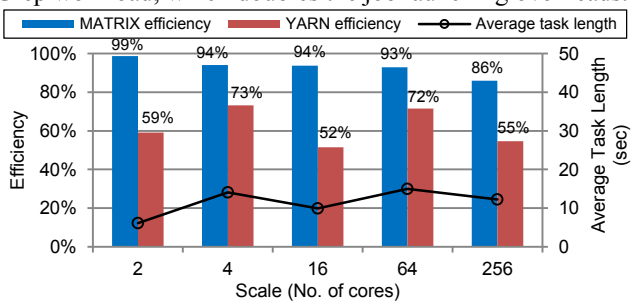

Figure 8: Efficiency for Grep

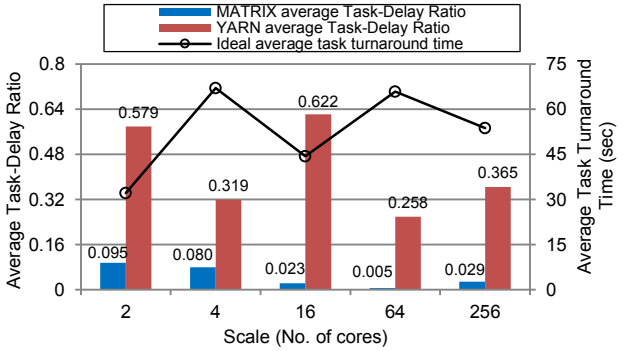

Figure 9: Average Task-Delay Ratio for Grep

However, MATRIX can optimize both load balancing and data-locality through the work stealing technique. This is preferable for heterogeneous workloads. Besides, MATRIX decomposes the workload as one DAG and launches all tasks once as fast as possible, introducing much less overheads. Figure 9 validates this justification by showing that MATRIX responses 20X faster on average than YARN.

E. Ligand Clustering Application in Bioinformatics

The previous comparisons use benchmarking workloads, and MATRIX has shown better scalability than YARN for all the workloads. In this section, we show how they perform on a real data-intensive application in bioinformatics, namely the Ligand Clustering application [37].

Large dataset clustering is a challenging problem in the field of bioinformatics, where many researchers resort to MapReduce for a viable solution. The real-world application experimented in this study is an octree-based clustering algorithm for classifying protein-ligand binding geometries, which has been developed in the University of Delaware. The application is implemented in Hadoop and is divided into iterative Hadoop jobs. In the first job, the map tasks read the input datasets that contain the protein geometry information. Depending on the size of the problem, the input dataset size varies from giga bytes to tera bytes and the workloads are considered as both data-intensive and compute-intensive. The output of the first job is the input of the second one; this applies iteratively. The output data size is about 1% of the input data size in the first job. Thus, the map tasks of the first job dominate the processing.

Like the benchmarking workloads, in this application, the input data size is 256MB per instance based on the 59.7MB real application data. We apply 5 iterative Hadoop jobs, including 10 phases of map and reduce tasks. We run the application in both MATRIX and YARN, and the performance results are given in Figure 10 and Figure 11.

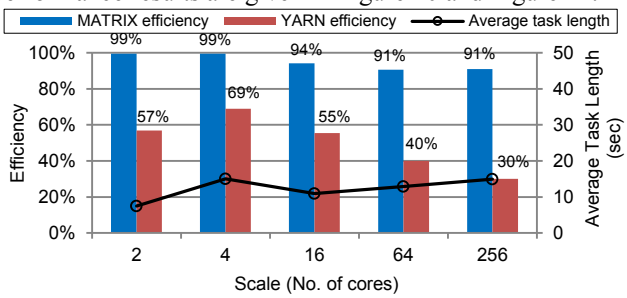

Figure 10: Efficiency for the Bioinformatics application

Figure 10 shows that as the scale increases, the efficiency of YARN is decreasing significantly. At 256-core scale, YARN only achieves 30% efficiency, which is one third of that achieved (91%) through MATRIX. The decreasing trend is likely to hold towards extreme-scales for YARN. On the other hand, the efficiency of MATRIX has a much slower decreasing trend and is becoming stable at 64-core scale. These results show the potential extreme scalability of MATRIX, which we will explore through simulations later.

The contributing factors of the large performance gain of MATRIX include the distributed architectures of the resource management, task scheduling, and metadata management; the distributed scheduling technique; as well as the general DAG decomposition of any application with arbitrary data flow pattern. All these design choices of MATRIX are radically different from those of YARN.

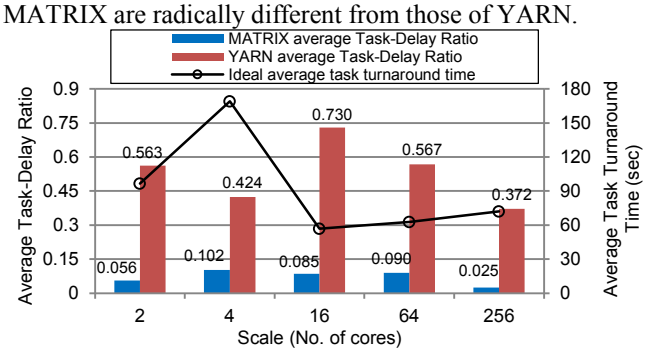

Figure 11: Average Task-Delay Ratio for the application

Like the Grep workload, YARN launches the 5 iterative Hadoop jobs one by one, incurring large amount of launching and Ping-Pong overheads, whilst MATRIX launches the whole application DAG once. This difference is another factor that causes the speed gap between YARN and MATRIX in responding to per-task shown in Figure 11 (MATRIX achieves 9X faster than YARN on average).

F. Fine-grained Data Processing Workloads

We have shown that MATRIX is more scalable than YARN in processing both typical benchmarking workloads and a real application in Bioinformatics. All the workloads evaluated so far are relatively coarse-grained (e.g. average task length of tens of seconds), comparing with the Berkeley Spark MapReduce stack [38] that targets finer-grained workloads (e.g. average task length of hundreds of milliseconds). We have identified that YARN is not well suited for running fine-grained workloads, due to the pull-based task scheduling mechanism and the HDFS centralized metadata management. To validate this, we run a fine-grained workload test of YARN at 64 cores for the Terasort workload by reducing the block size of HDFS to 256KB, 100X smaller than the previous course-grained experiments. The average task length should be only 1/100 of that of the course-grained workloads (286ms according to Figure 4). However, YARN's logs show that the average task execution time decreases only by half with about 14 sec, leading to the efficiency as low as 2.04% (286/14000).

On the contrary, MATRIX is designed to process the fine-grained sub-second workloads [28]. In [36], MATRIX achieved 85%+ efficiency for 64ms workload at 4K-core scales on an IBM BG/P supercomputer. In addition, in [39], we have compared MATRIX with the Spark sparrow scheduler with NOOP sleep 0 tasks and MATRIX was 9X faster than sparrow in executing the tasks.

To further justify that MATRIX has the potential to enable MapReduce to scale to extreme-scales, we explore the scalability of MATRIX through both real system (256-core scale) and simulations (64K-core scale) for fine-grained workloads. We choose the Ligand real application, reduce the task length of each task by 100X, increase the number of map tasks in the first iteration (there are 5 iterations of jobs in total) by 100X, and keep the task dependencies the same. The average task length in this workload ranges from 80ms to 166ms at different scales (refers to Figure 10 about the coarse-grained workload ranging from 8sec to 16.6sec).

We run MATRIX with this workload up to 256 cores. Besides, we conduct simulations of MATRIX for this workload at 64K-core scale, through the SimMatrix simulator [41]. SimMatrix is a lightweight discrete event simulator of task execution fabric, developed in Java. SimMatrix simulates the distributed task scheduling architecture, the distributed scheduling technique, and the KVS metadata management. We feed the fine-grained workloads in SimMatrix. Beyond 256 cores, we increase the number of tasks linearly with respect to the system scales by repeating the workloads at 256 cores. We show the efficiency results of MATRIX running both the fine-grained and coarse-grained workloads, as well as SimMatrix running the fine-grained workloads in Figure 12.

When the granularity increases by 100X, the efficiency only drops about 1.5% on average up to 256 cores (blue line vs. the solid red line). This shows great scalability of MATRIX in processing the fine-grained workloads. From the simulation's perspective, we run SimMatrix up to 64K cores and validate SimMatrix against MATRIX within 256-core scales. The normalized difference (black line) between SimMatrix and MATRIX is only 4.4% on average, which shows that SimMatrix is accurate and the simulation results are convincible. At the 64K cores, the efficiency maintains 86.8% for the workload of average task length of 150ms. According to the efficiency trend, we can predict that at the

1M-core scale, the efficiency will be 85.4% for this fine-grained workload. These results show that MATRIX has the potential to enable Hadoop to scale to extreme scales, even for the fine-grained sub-second workloads.

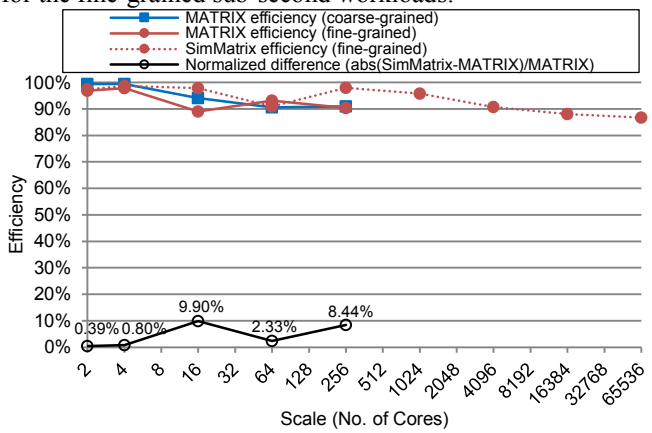

Figure 12: MATRIX for sub-second Bioinformatics workloads

V. Conclusions & Future Work

Large-scale Internet applications are processing large amount of data on the commodity cluster processors. Although the Hadoop framework has been prevalent for running these applications, there are inherent design issues that prevent Hadoop from scaling to extreme scales, given the fact that both of the data volumes and the system scale are increasing exponentially. This paper proposed to leverage the distributed design wisdoms of the MATRIX task execution framework to overcome the scaling limitations of Hadoop towards extreme scales. MATRIX addressed the scaling issues of YARN by employing distributed resource management, distributed data-aware task scheduling, and distributed metadata management using key-value stores.

We compared MATRIX with YARN using typical Hadoop workloads and the application in Bioinformatics up to 256 cores. Table 1 summarizes the average performance results of both MATRIX and YARN for different workloads for all scales. We see on average of all the Hadoop workloads, MATRIX outperforms YARN by 1.27X at all scales on average. For the application in Bioinformatics (BioApp), MATRIX outperforms YARN by 2.04X. We also explored the scalability of MATRIX through both real system and simulations at extreme scales [42] for fine-grained sub-second workloads. The simulations indicate 86.8% efficiency at 64K-core scale for 150ms workloads.

In the future, we will continue to compare SimMatrix with the YARN simulator, YARNsim [55][56][57], at extreme scales. We predict that MATRIX has the potential to enable MapReduce to scale to extreme-scale distributed systems. We will integrate MATRIX with the FusionFS [35] distributed file system. FusionFS assumes that each compute node will have local storage, and all the file servers are writing the data files locally to eliminate the data movement overheads introduced by stripping the data files as the HDFS does. FusionFS supports the POSIX interface and applies ZHT for distributed file metadata management. FusionFS will also explore the burst buffer technique [40] as a caching layer to enable faster operation speed and asynchronous check pointing. MATRIX will rely on FusionFS for scalable data management, which could enable the support of general scientific data formats [59][60][62][63], as opposed to the specialized HDFS one in Hadoop.

Table 1: Efficiency results summary of MATRIX and YARN

Workloads	Task Length	MATRIX Efficiency	YARN Efficiency	Average Speedup (MATRIX/YARN)
WordCount	21.2sec	95.2%	76.0%	1.26
TeraSort	20.1sec	95.9%	87.0%	1.10
Random Writer	36.2sec	95.8%	81.4%	1.20
Grep	11.5sec	93.0%	62.0%	1.53
BioApp	12.3sec	94.9%	50.2%	2.04
Fine-grained BioApp	128.5 ms	93.5%	N/A	N/A

Another future work will be integrating MATRIX with the Hadoop application layer, so that the application codes need not to be changed [54]. Currently, in MATRIX, the applications need to be re-programmed as the workload DAGs, so that the MATRIX client can submit them. In the future, we will develop interfaces between the Hadoop applications and MATRIX, and expose the same APIs as YARN does. To enable MATRIX to support broader categories of applications, we will investigate to change the MATRIX application interfaces to cater to the general Pilot-abstractions of jobs [50], in-situ applications [58], so that MATRIX can run a general category of data-intensive applications on all of HPC [66][68], Hadoop and Cloud infrastructures [64].

ACKNOWLEDGMENT

This work was supported by the U.S. Department of Energy contract DE-FC02-06ER25750, and in part by the National Science Foundation (NSF) under awards CNS-1042537 and NSF-1054974. This work was also possible in part due to the Amazon AWS Research Grant.

References

[1] Yahoo! Labs, avaible online: http://labs.yahoo.com/news/researchers-explain-the-science-powering-yahoo-weather/, 2014.

[2] L. Barroso, et al. “Web search for a planet: The Google cluster architecture,” IEEE Micro, vol.23, no.2, pp.22,28, March-April 2003.

[3] Coats, W. Sloan, et al. "Streaming into the Future: Music and Video Online." Loy. LA Ent. L. Rev. 20 (2000): 285..

[4] Z. Stone, et al. “Autotagging facebook: Social network context improves photo annotation,” IEEE workshop on CVPRW, 2008.

[5] F. Darema. “Dynamic data driven applications systems: A new paradigm for application simulations and measurements,” Computational Science-ICCS 2004. Springer Berlin Heidelberg, 2004.

[6] L. He, et al. “Mapping DAG-based applications to multiclusters with background workload,” in Proc. Of CCGrind, 2005.

[7] J. Dean, S. Ghemawat. “MapReduce: simplified data processing on large clusters,” Communications of the ACM 51.1 (2008): 107-113.

[8] A. Bialecki, et al. “Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware”, http://lucene.apache.org/hadoop/, 2005.

[9] K. Shvachko, H. Huang, et al. “The hadoop distributed file system”, in: 26th IEEE Symposium on MSST, May, 2010.

[10] V. Vavilapalli, et al. “Apache Hadoop YARN: yet another resource negotiator,” SOCC '13.

[11] B. Hindman, et al. “Mesos: a platform for fine-grained resource sharing in the data center,” NSDI'11.

[12] T. Gunarathne, et al. “Scalable parallel computing on clouds using Twister4Azure iterative MapReduce,” Future Gener. Comput. Syst., 2013.

[13] K. Wang, et al. “Using simulation to explore distributed key-value stores for extreme-scale system services,” SC '13.

[14] K. Wang, et al. “Exploring the Design Tradeoffs for Extreme-Scale High-Performance Computing System Software,” IEEE TPDS, 2015.

[15] T. Li, et al. “ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table,” IPDPS '13.

[16] K. Wang, et al. “Towards Scalable Distributed Workload Manager with Monitoring-Based Weakly Consistent Resource Stealing,” ACM HPDC 2015.

[17] K. Wang, et al. “Next Generation Job Management Systems for Extreme Scale Ensemble Computing,” ACM HPDC 2014.

[18] T. Li, et al. “A Dynamically Scalable Cloud Data Infrastructure for Sensor Networks,” Invited Paper, ACM ScienceCloud 2015.

[19] A. Raj, et al. “Enhancement of Hadoop Clusters with Virtualization Using the Capacity Scheduler,” International Conf. on ICSEM, 2012.

[20] Apache Hadoop 1.2.1 Documentation. 2014. [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.

[21] Apache Hadoop on Demand (HOD). 2014. [Online]. Available: http://hadoop.apache.org/common/docs/r0.21.0/hod-scheduler.html.

[22] A. Rasooli and D. Down. “A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems,” In Proc. of the SC Companion (SCC), 2012.

[23] A. Rasooli and D. Down. “An adaptive scheduling algorithm for dynamic heterogeneous Hadoop systems,” CASCON, 2011.

[24] M. Zaharia, et al. “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” EuroSys, 2010.

[25] K. Ousterhout, et al. “Sparrow: distributed, low latency scheduling”, in Proc. of SOSP, 2013.

[26] G. Mackey, et al. “Improving metadata management for small files in HDFS,” Cluster Computing and Workshops, CLUSTER, 2009.

[27] X. Zhao, et al. “Metadata-Aware small files storage architecture on hadoop,” In Proc. of the conf. on WISM, 2012.

[28] K. Wang, et al. “Modeling Many-Task Computing Workloads on a Petaflop IBM BlueGene/P Supercomputer,” IEEE CloudFlow 2013.

[29] E. M. Estolano, et al. “Haceph: Scalable Meta- data Management for Hadoop using Ceph,” Post Session at NSDI, 2010.

[30] S. A. Weil, et al. “Ceph: a scalable, high-performance distributed file system,” In Proc. of OSDI, 2006.

[31] K. Wang, et al. “Optimizing Load Balancing and Data-Locality with Data-aware Scheduling,” IEEE International Conf. on Big Data, 2014.

[32] S. A. Weil, et al. “Dynamic Metadata Management for Petabyte-Scale File Systems,” SC '04.

[33] I. Stoica, et al. “Chord: A scalable peer-to-peer lookup service for Internet applications,” in Proc. ACM SIGCOMM 2001.

[34] K. Wang, et al. “Paving the Road to Exascale with Many-Task Computing,” Doctoral Showcase, SC’12.

[35] D. Zhao, et al. “FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems,” IEEE International Conf. on Big Data, 2014.

[36] K. Wang, et al. “MATRIX: Many-Task Computing Execution Fabric for Extreme Scales,” Tech report, CS Dept., IIT, 2013.

[37] B. Zhang, et al. “Enhancement of Accuracy and Efficiency for RNA Secondary Structure Prediction by Sequence Segmentation and MapReduce,” BMC Structural Biology Journal, 13(Suppl 1):S3, 2013.

[38] M. Zaharia, et al. “Spark: cluster computing with working sets,” in Proc. of the 2nd USENIX conf. on HotCloud, 2010.

[39] I. Sadooghi, et al. “Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing,” CCGrid, 2014.

[40] N. Liu, et al. “On the role of burst buffers in leadership-class storage systems,” in Proc. of MSST, 2012.

[41] K. Wang, et al. “SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascales,” ACM HPC 2013.

[42] D. Zhao, et al. “Exploring Reliability of Exascale Systems through Simulations,” ACM HPC 2013.

[43] M. Schwarzkopf, et al. “Omega: flexible, scalable schedulers for large compute clusters,” In Proc. of EuroSys, 2013.

[44] K. Wang, I. Raicu. “Scheduling Data-intensive Many-task Computing Applications in the Cloud," NSFCloud Workshop, 2014.

[45] I. Raicu, I. Foster. “Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing,” PhD Dissertation, CS Depart., UChicago, 2009.

[46] M. Wilde, et al. “Swift: A language for distributed parallel scripting,” Parallel Comput. 37, 9 (September 2011), 633-652.

[47] R. Knöll and M. Mezini. “Pegasus: first steps toward a naturalistic programming language,” ACM SIGPLAN OOPSLA 2006.

[48] K. Wang, I. Raicu. "Towards Next Generation Resource Management at Extreme-Scales", PhD Proposal, CS Dept., IIT, 2014.

[49] L. V. Kale and S. Krishnan. “CHARM++: a portable concurrent object oriented system based on C++,” SIGPLAN Not. 28, 10, 1993.

[50] P. Mantha, et al. “P*: A model of pilot-abstractions,” IEEE Int. Conf. on E-Science, 2012.

[51] C. Dumitrescu, et al. “The Design, Usage, and Performance of GRUBER: A Grid uSLA-based Brokering Infrastructure,” International Journal of Grid Computing, 2007.

[52] T. Li, et al. “A Convergence of Key-Value Storage Systems from Clouds to Supercomputers,” Journal of CCPE, 2015.

[53] K. Wang, et al. “Load-balanced and locality-aware scheduling for data-intensive workloads at extreme-scales,” Journal of CCPE, 2015.

[54] K. Wang. “Scalable Resource Management System Software for Extreme-Scale Distributed Systems,” PhD Dissertation, CS Depart., IIT, 2015.

[55] N. Liu, et al. “YARNsim: Hadoop YARN Simulation System,” in Proc. of the 15th IEEE/ACM CCGrid, 2015.

[56] N. Liu, et al. “FatTreeSim: Modeling a Large-scale Fat-Tree Network for HPC Systems and Data Centers Using Parallel and Discrete Event Simulation,” in Proc. of the 29th ACM SIGSIM PADS, 2015.

[57] N. Liu, et al. “Model and simulation of exascale communication networks,” Journal of Simulation, 2012.

[58] Y. Wang, et al. “Smart: A mapreduce-like framework for in-situ scientific analytics,” Tech. rep., OSU-CISRC-4/15-TR05, OSU, 2015.

[59] Y. Wang, et al. “SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats,” in Proc. of CCGrid, 2012.

[60] Y. Wang, et al. “Supporting a Light-Weight Data Management Layer Over HDF5,” in Proc. of the 13th IEEE/ACM CCGrid, 2013.

[61] S. Zhang, et al. “After We Knew It: Empirical Study and Modeling of Cost-effectiveness of Exploiting Prevalent Known Vulnerabilities Across IaaS Cloud,” In Proc. of the 9th ACM ASIACCS, 2014.

[62] Y. Wang, et al. “A Novel Approach for Approximate Aggregations Over Arrays,” in Proc. of the SSDBM, 2015.

[63] Y. Wang, et al. “SAGA: Array Storage as a DB with Support for Structural Aggregations,” in Proc. Of the SSDBM, 2014.

[64] Z. Lv, et al. “Game on, science - how video game technology may help biologists tackle visualization challenges,” PLoS One 8 e57990.

[65] X. Yang, et al. “Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems,” SC '13.

[66] X. Yang, et al. “Balancing Job Performance with System Performance via Locality-Aware Scheduling on Torus-Connected Systems,” in Proc. of IEEE Cluster'14 , 2014.

[67] Z. Zhou, et al. “Reducing Energy Costs for IBM Blue Gene/P via Power-Aware Job Scheduling", Workshop on JSSPP, 2013.

[68] Z. Zhou, et al. “Improving Batch Scheduling on Blue Gene/Q by Relaxing 5D Torus Network Allocation Constraints,” IPDPS, 2015.