Optimizing Load Balancing and Data-Locality with Data-aware Scheduling

Ke Wang[*], Xiaobing Zhou[§], Tonglin Li[*], Dongfang Zhao[*], Michael Lang[†], Ioan Raicu[*‡]
[*]Illinois Institute of Technology, [§]Hortonworks Inc., [†]Los Alamos National Laboratory, [‡]Argonne National Laboratory
kwang22@hawk.iit.edu, xzhou@hortonworks.com, {tli13, dzhao8}@hawk.iit.edu, mlang@lanl.gov, iraicu@cs.iit.edu

***Abstract*—Load balancing techniques (e.g. work stealing) are important to obtain the best performance for distributed task scheduling systems that have multiple schedulers making scheduling decisions. In work stealing, tasks are randomly migrated from heavy-loaded schedulers to idle ones. However, for data-intensive applications where tasks are dependent and task execution involves processing a large amount of data, migrating tasks blindly yields poor data-locality and incurs significant data-transferring overhead. This work improves work stealing by using both dedicated and shared queues. Tasks are organized in queues based on task data size and location. We implement our technique in MATRIX, a distributed task scheduler for many-task computing. We leverage distributed key-value store to organize and scale the task metadata, task dependency, and data-locality. We evaluate the improved work stealing technique with both applications and micro-benchmarks structured as direct acyclic graphs. Results show that the proposed data-aware work stealing technique performs well.**

Keywords—data-intensive computing; data-aware scheduling; work stealing; key-value stores; many-task computing

I. INTRODUCTION

There is a growing set of large-scale scientific applications which are loosely-coupled in nature containing many small jobs/tasks (e.g. per-core) with shorter durations (e.g. sub-second), along with large volumes of data – these applications include those from data analytics, bioinformatics, data mining, astronomy, astrophysics, and MPI ensembles [1]. As systems are growing exponentially in parallelism approaching billion way concurrency at exascale [2], we argue that future programming models will likely employ over-decomposition generating even many more fined-grained tasks than available parallelism. While over-decomposition has been shown to improve utilization at extreme scales as well to make fault tolerance more efficient [3][4], it poses significant challenges on task scheduling system to make extremely fast scheduling decisions (e.g. millions/sec), in order to achieve the highest throughput and utilization. This requirement is far beyond the capability of today's centralized scheduling systems, such as SLURM [5], Condor [6], SGE [7], Cobalt [8], and Falkon [9].

The Many-task computing (MTC) [10] paradigm aims to define and address the challenges of scheduling fine grained data-intensive workloads [11]. MTC applications are usually structured as direct acyclic graphs (DAG) of small discrete tasks, with data dependencies forming the graph edges. The tasks are per-core and do not require strict coordination of processes at job launch as the traditional HPC workloads.

We propose the task scheduling system for MTC will need to be fully-distributed to achieve extremely high throughput and utilization. In this design, each compute node runs one scheduler and one or more executors/workers. All the schedulers are fully-connected, and receive workloads to schedule tasks to local executors. Therefore, ideally, the throughput would gain near-optimal linear speedup as the system scales. We abandon the centralized architecture due to the limited processing capacity and the single-point-of-failure issue. We bypass the hierarchical architecture as it is difficult to maintain a tree under failures, and a task may experience longer latency because it needs to go through multiple hops as opposed to only one in a fully-distributed architecture.

Load balancing [12] is challenging for fully-distributed architectures as a scheduler only has knowledge of its own state, and therefore must be done with limited partial state. Load balancing refers to distributing workloads as evenly as possible across all the schedulers/workers, and it is important given that a single heavily-loaded scheduler would lower the system utilization significantly. This work adopts the work stealing technique [13] at the node/scheduler (instead of core/thread) level. In work stealing, the idle schedulers communicate with neighbors to balance their loads. Our previous work [14] has explored the parameter space (e.g. number of tasks to steal, number of static/dynamic neighbors, poll interval) extensively through a simulator, SimMatrix [15][16], up to millions of nodes and billions of cores.

However, as more applications are becoming data-intensive and experiencing data explosion [17] such that tasks are dependent and task execution involves processing large amount of data, data-aware scheduling and load balancing are two indispensable yet orthogonal needs. Migrating tasks randomly through work stealing would compromise the data-locality and incur significant data-transferring overhead. On the other hand, struggling to map each task to the location where the data resides is infeasible due to the complexity of the computation (this mapping is a NP-complete problem [18]). Furthermore, this mapping may cause poor load balancing due to the potential unbalanced data distribution. Therefore, in this work, we investigate scheduling methods that satisfy both needs and still achieves good performance.

In this paper, we propose a data-aware work stealing technique that is able to achieve good load balancing, and yet still tries to best exploit data-locality. Basically, each scheduler would maintain both dedicated and shared task ready queues (implemented as descending priority queues based on the data size a task requires). Tasks in the dedicated queue are only scheduled and executed locally unless special policy is applied, while tasks in the shared queue could be migrated among schedulers for balancing loads through work stealing. A ready task will be put in either queue based on the size and location of the data demanded by the task. In addition, a scheduler may push a task to others if the majority of data demanded by the task is remote. This is much better than most work stealing work that employs only one locality-oblivious task ready queue (either implemented as normal queue, or double-ended deque [20]). We apply a distributed key-value store (DKVS), ZHT [21][52], as a meta-data service that stores important data-locality information for all the tasks. Our data-aware work stealing technique works in both homogeneous and heterogeneous distributed systems [53]. We implement our technique in MATRIX [22], a task execution fabric for MTC.

This paper has the following main contributions:

- Propose a data-aware work stealing technique that combines load balancing with data-aware scheduling.
- Apply a distributed key-value store as a meta-data service to store important data dependency and locality information.
- Evaluate the proposed technique up to hundreds of nodes showing good performance using different applications under different scheduling policies.

The rest of this paper is organized as follows. Section II introduces the MATRIX task execution fabric. Section III presents our proposed technique and the implementation details, which is followed by an extensive evaluation in Section IV. We list the related work in Section V. Finally, Section VI draws the conclusions and envisions our future work.

II. MATRIX: LIGHT WEIGHT TASK EXECUTION FABRIC

By using simulation in previous work [23], we have concluded that for extreme-scale systems, a distributed architecture is required. With this conclusion and the justification we have made about fully-distributed architecture, we built a light-weight task execution fabric, MATRIX, from scratch. We have evaluated MATRIX using micro-benchmarks on an IBM Blue Gene/P supercomputer up to 4K-core scale. MATRIX maintains throughput as high as 13K tasks/sec, and 85%+ efficiency with fine-grained sub-second tasks (64ms), and shows efficiency speedup of 4X compared to Falkon [9] with task granularity of 1-second and 9X compared to Sparrow [35] with NOOP sleep tasks (sleep 0). We implement the proposed technique in MATRIX in this work.

The basic architecture of MATRIX is shown in Figure 1. MATRIX is comprised of three components: client, scheduler and executor. Each compute node runs a scheduler, an executor and a ZHT server. All the schedulers are fully-connected. The client is a benchmarking tool that issues requests to generate a set of tasks, submits the tasks to any scheduler, and monitors the task execution progress. Each scheduler schedules tasks to local executor. Whenever a scheduler has no more ready tasks, it communicates with other schedulers to pull ready tasks through load balancing techniques (e.g. work stealing). Each executor forks several (usually equals to number of physical cores of a machine) threads to execute ready tasks concurrently.

ZHT is used to monitor the execution progress by MATRIX client, and to keep the system state meta-data by MATRIX scheduler in a distributed, scalable, and fault tolerant way. ZHT is a zero-hop persistent DKVS where each ZHT client has a global view of all the servers. For each operation of ZHT server (insert, lookup, remove, append, compare and swap), there is a corresponding client API. The client calls the API, which sends a request to the exact server that is responsible for the request by hashing the key. Upon receiving a request, the ZHT server converts it to the corresponding operation type and executes the operation. ZHT serves as a data management building block for extreme scale system services, and has been tested up to 8K nodes (32K cores) on a IBM Blue Gene /P supercomputer [21].

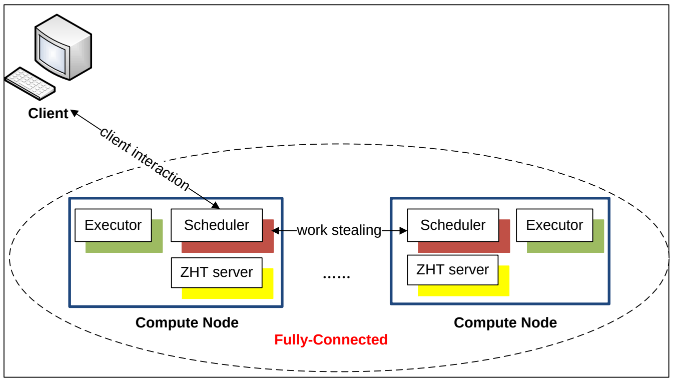

Figure 1: MATRIX architecture overview

Both MATRIX client and schedulers are initialized as ZHT clients. Each MATRIX scheduler records local state information (e.g. number of complete tasks, number of idle executing threads) to ZHT periodically, and the client keeps polling this information until all tasks are completed.

III. DATA-AWARE WORK STEALING

This section covers our proposed data-aware work stealing technique. First, we describe the adaptive work stealing technique to achieve load balancing. Then we present how we combined a data-aware scheduling strategy with work stealing.

A. *Adapative Work Stealing*

Work stealing has been proven as an efficient load balancing technique at thread/core level in multi-core shared memory machine [12]. It is a pull-based method in which the idle processors try to steal tasks from the overloaded ones that are randomly selected. This work adopts work stealing at the node level in distributed environment, and modifies it to be adaptive according to the system states at runtime.

Work stealing is proceeded by the idle schedulers stealing workloads from neighbors. As in fully-distributed architecture, every scheduler has a global membership list, therefore, the selection of candidate neighbors (victims) from which an idle scheduler (thief) would steal tasks could be static or dynamic/random. In static mechanism, the neighbors are pre-determined and will not change; this has limitation that every scheduler is confined to communicate with part of other schedulers. In dynamic case, whenever a scheduler is idle, it randomly chooses some candidate neighbors. The traditional work stealing randomly selects one neighbor to steal tasks [25] yielding poor performance at extreme scales, because the chance that one neighbor would have queued ready task is low at large scales. In the end, we choose to have a multiple-random neighbor selection strategy that randomly selects several neighbors instead of only one.

After an idle scheduler selects candidate neighbors, it goes through each neighbor in sequential to ask for "***load information***" (number of ready tasks), and tries to steal tasks from the most heavily-overloaded one. When a scheduler fails to steal tasks from the selected neighbors, because either all selected neighbors have no more tasks, or the reported extra ready tasks have been executed when the stealing happens, the scheduler waits for a period of time and then does work stealing again. We call this wait time the **polling interval**.

If the polling interval is fixed, there would be difficulties to set the value with right granularity. If the polling interval is set to be too small, at the final stage when there are not many tasks left, many schedulers would poll neighbors to do work stealing, which would ultimately fail and lead to more work stealing communications. If the polling interval was set large enough to limit the number of work stealing events, work stealing would not respond quickly to change conditions, and lead to poor load balancing. Therefore, we implement an adaptive polling interval strategy, in which, the polling interval of a scheduler is changed dynamically similar to the exponential back-off approach in the TCP networking protocol [26]. The default polling interval is set to be a small value (e.g. 1 ms). Once a scheduler successfully steals tasks, the polling interval of that scheduler is set back to the initial small value. If a scheduler fails to steal tasks, it waits the time of polling interval and doubles the polling interval and tries to do work stealing again. In addition, we set an upper bound for the polling interval. Whenever the polling interval hits the upper bound, a scheduler would not do work stealing anymore. This would reduce the amount of failing work stealing at the final stage.

The parameters of work stealing are the number of dynamic neighbors, the number of tasks to steal, and the polling interval. Our previous simulation work [14][24] studied the parameter space extensively and found the optimal configurations: **the number of tasks to steal is half; the number of dynamic neighbors is square root of the number of all schedulers; and an exponential back-off polling interval.** These optimal parameters have been used in MATRIX for scalable work stealing to achieve distributed load balancing.

One thing to notice is that the optimal parameters are unnecessary to hold always for all the workloads. In fact, it is unlikely to get an optimal parameter space for all the workloads. Our work stealing technique has the ability to change these parameters dynamically at runtime in a decentralized way. Each scheduler observes the system state at runtime according to whether the current work stealing succeeds or not. If a scheduler steals tasks successfully, then it keeps the optimal parameter configuration; otherwise, if a scheduler experiences several failures in a row, it can change the suggested parameters. For example, a scheduler may increase the number of dynamic neighbors by 1 or 2 to reach further so that it has higher chance to steal tasks.

B. Data-Aware Scheduling

The work stealing technique does not employ any data locality information. This is harmful when executing data-intensive workloads in which tasks are dependent and tasks take a large volume of data as input and generate output with more data. This subsection presents the proposed ideals to combine work stealing with data-aware scheduling.

1) DKVS Used as a Meta-Data Service

As our previous work [23] has claimed that DKVS could be used as a building block for distributed system services, we apply ZHT to store all the important information, such as data dependency conditions, and data locality information of tasks transparently. The "key" is task id, and the "value" is the important meta-data of the task that is defined as the following data structure in Figure 2. Upon task submission, the client takes an application workload (represented as a DAG), sends the task meta-data (specifically "num_wait_parent" and "children") to ZHT, and submits the tasks to MATRIX for scheduling. The meta-data is later updated as tasks make progress through the various states.

```
typedef    TaskMetaData
{
    int    num_wait_parent;    // number of waiting parents
    vector<string> parent_list;    // schedulers that run each parent task
    vector<string> data_object;    // data object name produced by each parent
    vector<long> data_size;    // data object size (byte) produced by each parent
    long all_data_size;    // all data object size (byte) produced by all parents
    vector<string> children;    // children of this tasks
} TMD;
```

Figure 2: Data structure of task metadata

2) Distributed Queues in Matrix

Each scheduler would maintain four local task queues: task waiting queue (*WaitQ*), dedicated local task ready queue (*LReadyQ*), shared work stealing task ready queue (*SReadyQ*), and task complete queue (*CompleteQ*). These queues hold tasks in different states that are stored as meta-data in ZHT. A task is moved from one queue to another when state changes. With these queues, the scheduler can support scheduling tasks with data dependencies specified by certain DAGs.

a) WaitQ

Initially, the scheduler would put all the incoming tasks from the client to the *WaitQ*. A thread keeps checking every task in the *WaitQ* to see whether the dependency conditions for that task are satisfied by querying the meta-data from ZHT. The task meta-data has been inserted into ZHT by MATRIX client before submitting tasks to schedulers. Specifically, only if the value of the field of "num_wait_parent" in the meta-data is equal to 0 would the task be ready to run.

b) LReadyQ and SReadyQ

When a task is ready to run, the scheduler makes decision to put it in either the *LReadyQ* or the *SReadyQ*, according to the size and location of the data required by the task. The *LReadyQ* stores the ready tasks that usually require large volume of data and the majority of the required data is located at the current node; these tasks could only be scheduled and executed locally unless special policy (explained later) is used. The *SReadyQ* stores the tasks that could be migrated to any scheduler for load balancing's purpose; these tasks either don't need any input data or the demanded data volume is so small that the transferring overhead is negligible. The "***load information***" queried by work stealing is the length of the *SReadyQ* (number of tasks in the *SReadyQ*). The pseudo-code for decision making is given in Algorithm 1.

The threshold (*t*) defines the upper bound of the data transferring rate that is achieved when transmitting the data for the task. The value equals to a small percentage (e.g. 10%) multiplying the theoretical network bandwidth (e.g. 10Gbps) of the system. The percentage also means the ratio between the data-transferring overhead and the estimated task execution length (*est_task_length*). The smaller the percentage is, the more likely the tasks could be migrated. As we don't know ahead how long a task runs, we predict the *est_task_length* as the average length of the previous tasks that have been finished.

ALGORITHM 1. Decision Making to Put a Task in the Right Ready Queue

Input: a ready task (*task*), TMD (*tm*), a threshold (*t*), current scheduler id (*id*), *LReadyQ*, *SReadyQ*, estimated length of the task in second (*est_task_length*)
Output: void.

```
1   if (tm.all_data_size / est_task_length <= t) then
2       SReadyQ.push(task);
3   else
4       long max_data_size = tm.data_size.at(0);
5       int max_data_scheduler_idx = 0;
6       for each i in 1 to tm.data_size.size() - 1; do
7           if tm.data_size.at(i) > max_data_size; then
8               max_data_size = tm.data_size.at(i);
9               max_data_scheduler_idx = i;
10          end
11      end
12      if (max_data_size / est_task_length <= t); then
13          SReadyQ.push(task);
14      else if tm.parent_list.at(max_data_scheduler_idx) == id; then
15          LReadyQ.push(task);
16      else
17          send task to: tm.parent_list.at(max_data_scheduler_idx)
18      end
19  end
20  return;
```

Lines 1-2 decide to put the task in *SReadyQ* as the data movement overhead is small. Otherwise, lines 4-11 find the maximum data object size (*max_data_size*), and the scheduler (indexed at *max_data_scheduler_idx*) that ran the parent task generating this maximum data object. As a task could have multiple parents, it is the best if it runs on the scheduler that has the largest portion of the required data. Lines 12-13 decide to put the task in *SReadyQ* because the *max_data_size* is small. Otherwise, lines 14-15 put the task in *LReadyQ* as the current scheduler is indexed at *max_data_scheduler_idx*. Otherwise, lines 16-18 push the task to the corresponding scheduler. When a scheduler receives a pushing task, it puts the task in *LReadyQ*.

An executor has several executing threads that keep pulling ready tasks to execute. Each thread would first pop tasks from *LReadyQ*, and then pop tasks from *SReadyQ* if the *LReadyQ* is empty. Both *LReadyQ* and *SReadyQ* are implemented as descending priority queue based on the required data size. The larger the required data size is, the higher priority the task would be, as a task that requires larger data size usually runs longer. When executing a task, the executing thread first queries the meta-data for the size and location of the required data produced by each parent, and then gets the data either from local or remote nodes. After collecting all the data, the task will be executed. The number of executing threads is configurable; in practice it is usually configured to be the number of physical cores (a similar strategy was used in Falkon [9] on the Blue Gene/P supercomputer). As long as both queues are empty, the scheduler would start doing work stealing, and the stolen tasks would be put in the *SReadyQ*.

c) CompleteQ

When a task is done, it is moved to the *CompleteQ*. There is another thread responsible for updating the meta-data for all the children of each completed task. It first queries the meta-data of the completed task to find out the children, and then updates each child's meta-data as follows: decreasing the "num_wait_parent" by 1; adding current scheduler id to the "parent_list"; adding the produced data object name to the "data_object"; adding the size of the produced object to the "data_size"; increasing the "all_data_size". As long as the "num_wait_parent" equals to 0, the task would be ready to run.

3) Different Scheduling Policies

We define four different scheduling policies for our data-aware scheduling technique, namely maximized load balancing (*MLB*), maximized data-locality (*MDL*), rigid load balancing and data-locality segregation (*RLDS*), and flexible load balancing and data-locality segregation (*FLDS*).

a) MLB

MLB considers only the load balancing, and all the ready tasks are put in the *SreadyQ* to be allowed for migration. We achieve the *MLB* policy by tuning the threshold (*t*) in Algorithm 1 to be the maximum possible value (i.e. LONG_MAX). This is the baseline work stealing strategy without taking into consideration the data locality.

b) MDL

MDL only considers data-locality, and all the ready tasks that require input data would be put in *LReadyQ*, no matter how big the data is. This policy is achieved by tuning the threshold (*t*) in Algorithm 1 to be 0.

c) RLDS

RLDS sets the threshold (*t*) in Algorithm 1 to be somewhere between 0 and the maximum possible value. Once a task is put in the *LReadyQ* of a scheduler, it is confined to be executed locally (this is also true for the *MDL* policy).

d) FLDS

The *RLDS* policy could have load balancing issues under situations where a task produces large volumes of data and has many children. For example, for a workload DAG shaped as a fat tree, all the tasks would be eventually executed on one scheduler that runs the root task. To avoid the hotspot problem, we relax the *RLDS* to a more flexible policy (*FLDS*) in which tasks could be moved from *LReadyQ* to *SReadyQ* under certain circumstance. We set another time threshold (*tt*) and use a monitoring thread in each scheduler to check the *LReadyQ* periodically. If the thread detects that the estimated running time (*est_run_time*) of all tasks in the *LReadyQ* is above *tt*, it would move some tasks from the bottom of *LReadyQ* to *SReadyQ* to guarantee that the *est_run_time* of the rest tasks is below *tt*. The *est_run_time* is calculated as the *LReadyQ* length divided by the overall throughput of the scheduler so far. For example, assuming 1000-tasks are finished so far and takes 10sec, the *LReadyQ* contains 5000-tasks, and *tt*=30sec. We calculate the number of moving tasks: The throughput =1000/10=100tasks/sec. The *est_run_time*=5000/100=50sec, 20sec longer than *tt*. 20sec takes 20/50=40% ratio, therefore, 40%*5000=2000-taks will be moved. As these assumed values are changing with time, the moving task count is changing.

4) Write Locality and Read Locality

Our data-aware scheduling technique ensures the best write locality, and at the meanwhile, optimizes the read locality. To ensure the best write locality, every task writes the produced data locally. We considered using a distributed key/value store (ZHT) as both a data and a meta-data service, however it would have been extremely difficult to optimize the data-locality as distributed key/value stores rely on consistent hashing to determine the ultimate location of data stored. We also considered leveraging distributed file system (e.g. HDFS [27]) underneath to manage the data. We argue that the scheduling and load balancing strategies are not affected by the actual method of data storage. We envision allowing data to be stored in a distributed file system as future work.

Read locality is optimized by pushing a task to the location where has the majority of the required data. For large data volumes, tasks are migrated to where the data resides. For small data volumes, tasks are run wherever there are available compute resources to maximize utilization.

C. List of Short Terms

To summarize and make it clear about the important short terms we use in this paper, we list and explain them in Table 1 as follows. These short terms are used in the overall paper. We explain other terms at where they appear.

Table 1: List of important short terms

Term	Description
DKVS	distributed key-value stores
WaitQ	task waiting queue
LReadyQ	dedicated local task ready queue
SReadyQ	shared work stealing task ready queue
CompleteQ	Task complete queue
t	a threshold to define the upper bound of the data transferring rate achieved when transmitting the data
MLB	maximized load balancing policy
MDL	maximized data-locality policy
RLDS	rigid load balancing and data-locality segregation policy
FLDS	flexible load balancing and data-locality segregation policy
tt	a time threshold to determine the number of tasks being moved from *LReadyQ* to *SReadyQ* for the *FLDS* policy
DAWS	Data aware work stealing

D. Implementation Details

We re-implemented MATRIX to separate it totally from the ZHT codebase in C++. The new version (version 2) of MATRIX simply uses ZHT as a black box through client APIs, ensuring easier maintainability and extensibility. Although the basic architecture did not change, many features were added, such as task dependencies and data-aware scheduling. The codebase of the MATRIX version 2 is made open source on Github: https://github.com/kwangiit/matrix_v2. It has about 3K lines of code implementing the MATRIX client, MATRIX scheduler, MATRIX executor, a network communication layer and the proposed data-aware work stealing technique, with 8K lines of ZHT codebase plus 1K lines of auto-generated code from Google Protocol Buffer [29]. The version 2 has dependencies on ZHT [21] and Google Protocol Buffer.

IV. EVALUATION

In this section, we present the performance evaluation results of the data-aware work stealing technique. We first introduce the experiment environment and the evaluation metrics. Then, we conduct experiments using workloads structured as DAGs coming from two scientific applications, namely image stacking from astronomy [30][56] and all-pairs from biometrics [31][32]. We compare our results with those achieved through data-diffusion technique in Falkon [9][11] which employed a centralized data-aware scheduler [30][32]. Next, we compare different scheduling polices using the all-pairs workload. Then, we run benchmarking DAGs.

A. Experiment Environment

We conduct all the experiments on the Kodiak cluster from the Parallel Reconfigurable Observational Environment (PROBE) [19] of Los Alamos National Laboratory. Kodiak has 1028 nodes, each node has two AMD Opteron (tm) processers 252 (2.6GHZ), and has 8GB memory. The network supports both Ethernet and InfiniBand. Our experiments use 10Gbits Ethernet (default configuration of Kodiak). We run experiments with up to 100 nodes (200 cores), similar to the data-diffusion Falkon work [30][32] we compared to.

B. Evaluation Metrics

We define important metrics as follows:

Average Time Per Task Per CPU defines the average time to execute one task from one CPU's perspective for all the tasks. Ideally, each CPU would process tasks sequentially without holding to wait tasks. therefore, the ideal average time should be equal to the average task length of all the tasks. In reality, the average time should be larger than the ideal case, and the closer they are, the better. The throughput is the reciprocal of the average time multiplying the number of CPUs. For example, assuming it takes 10sec to run 2000-tasks on 100-cores, this means 2 task-per-sec-per-cpu (2000/10/100). The "Average Time Per Task Per CPU" is 1/2=0.5sec. This metric is from the Falkon Data-Diffusion paper [30] for fair comparison. We don't use throughput, because given a workload, a throughput number cannot tell how good it is directly, the average time is more self-explanatory.

Efficiency refers to the proportion of time that the system is executing tasks. The system spends the other time doing network communications, such as moving data, moving tasks, doing work stealing. Efficiency also reflects the overall system utilization, the higher the better. It is calculated as the ideal time (production of the average task length and the average task per CPU) to finish a workload dividing the actual time.

Utilization is an instant metric that measures the ratio of busy CPUs out of all CPUs. Utilization is usually useful when doing visualization for the system state.

C. Scientific Application Evaluation

We evaluate the data-aware work stealing technique using two scientific applications: image stacking in astronomy [30] and all-pairs in biometrics [32]. These two applications represent different data-intensive patterns.

1) Image Stacking in Astronomy

This application conducts the "stacking" of image cutouts from different parts of the sky. The procedure involves re-projecting each image to a common set of pixel planes, then co-adding many images to obtain a detectable signal that can measure their average brightness/shape. The workload DAG is represented as in Figure 3. The dotted lines represent independent tasks (t_0 to t_n) fetching ROI objects in a set of image files (f_0 to f_m) that are randomly distributed, and then generate an output individually. The last task (t_{end}) waits until collecting all the outputs, and then obtains a detectable signal.

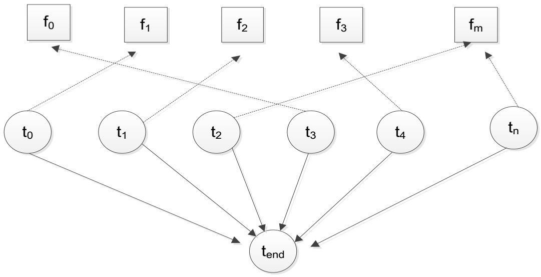

Figure 3: Image Stacking Workload DAG

Followed the workload characterization in [30], in our experiments, each task would require a file that has 2MB of data, and generates 10KB data of output. The ratio of the number of tasks to the number of files refers to **locality number**. Locality 1 means the number of tasks equals to the number of files, and each task requires an unique file. Locality n means that the number of tasks is n-times of the number of files, and each file is required by exactly n tasks. The higher the locality is, the less number of files and tasks there would be. The number of tasks and the number of files for each locality are given in [30]. We evaluate different locality values, i.e. 1, 1.38, 2, 3, 4, 5, 10, 20, 30. Each task would run for an average of 158 ms (as reported in [30]) .We run experiments up to 100 nodes (200 cores) for all locality values. The files are uniformly distributed to each node, and the tasks are randomly distributed. We use the *MDL* policy to move every task to where the required data resides, as 2MB of data is relatively large. We compare with data diffusion technique in Figure 4 at scale of 128 cores (the largest scale that data-diffusion ran).

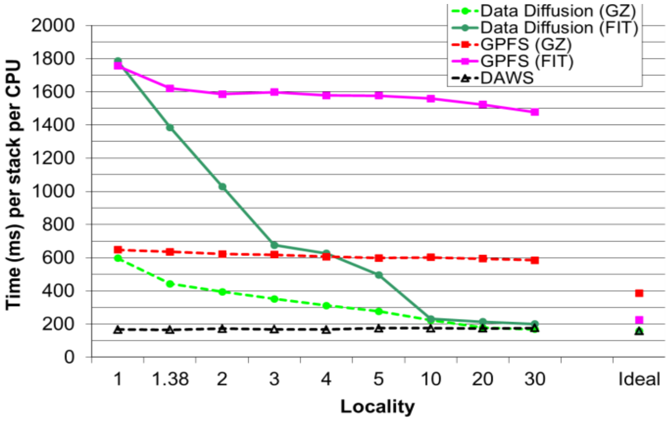

Figure 4: Comparison between DAWS and Data-Diffusion

In Figure 4, GZ indicates that the image data is in compressed format while FIT indicates that the image data is uncompressed. Data Diffusion (GZ) and Data Diffusion (FIT) mean using data diffusion technique on compressed and uncompressed data respectively. Likely, GPFS (GZ) and GPFS (FIT) mean using GPFS file system on compressed and uncompressed data respectively. DAWS represents our result, and stands for data-aware work stealing. Specifically, our experiments have the same workload configuration as Data Diffusion (GZ) compressed format (each file is 2MB) [30]. Figure 4 shows that at 128-core scale, the time per task of our DAWS technique keeps almost constant as locality increases, and it is very close to the ideal task running time (158ms). Data Diffusion (GZ) experienced very large average time when locality is small, and decreases to be close to the ideal time when locality is 30. The reason that DAWS could keep constant and perfectly close to the ideal is that all the data is uniformly distributed over all compute nodes. The only overhead is caused by the schedulers making decisions to transfer tasks in the right spots. While in Data Diffusion (GZ), as data is initially kept in a slower shared file system, the data would be copied to local disks when needed. When locality is small, the chances that the same piece of data will be reused is low therefore involving more amount of data access from the shared file system. This explains why Data Diffusion (GZ) has a large average time when locality is small. It is also noteworthy to point out that GPFS (GZ) and GPFS (FIT) remain largely constant regardless of locality, which is due to data being accessed from the shared file system remotely upon every data access; performance increases slightly with higher locality, likely due to OS-level caching.

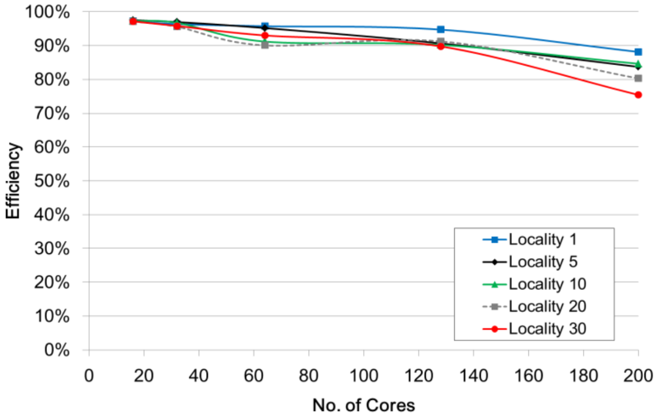

Figure 5: Efficiency of DAWS with Localities at scales

The time per task of DAWS experienced a slight increase from Locality 1 (167ms) to 30 (176ms). We explain the reason in Figure 5, which shows the efficiencies of different localities in terms of scale. From Figure 5, we see that the efficiency decreases slightly with respect to both scale and locality (but still keeps above 75% efficiency for task length of 158ms). The reason is that the number of files per compute node is decreasing as the scale and locality increase. Therefore, more tasks on a compute node could not be run locally and need to be moved to the right nodes causing significant network traffic and load imbalance. At the extreme case where the locality is infinitely large, there would be only one file on one compute node, eventually all the tasks need to be run on that node.

2) *All-Pairs in Biometrics*

All-Pairs [31] is a common benchmark for data-intensive applications that describes the behavior of a new function on sets A and sets B. For example, in Biometrics, it is very important to find out the covariance of two sequences of gene codes. In this workload, all the tasks are independent, and each task execute for 1 second to compare two 12MB files with one from each set.

Figure 6 shows the an example of the workload DAG, in which four independent tasks operate on two sets of two files, and each task requires one file from each set.

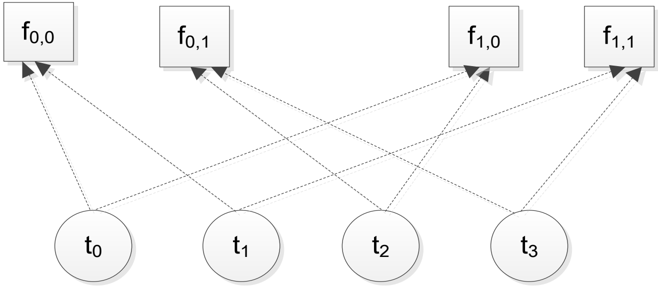

Figure 6: All-Pair Workload DAG

We run strong-scaling experiments up to 100 nodes with a 500*500 workload size. Therefore, there would be 250K tasks in total. All the 1000 files from two sets are uniformly distributed to each compute node, and all the tasks are randomly distributed. As a task needs two pieces of data files that may locate at different nodes at the worst case, one piece of data may need to be transmitted. To make the workload more data-intensive, we reduced the task running time by 10X, resulting in 100-ms running time with 24MB of data requirement. This is the same workload referenced in [32]. We use the *FLSD* policy, and at the end (80% of the workload is done) of the experiments, we set the time threshold *tt* to be 20 seconds initially, which is then decreased by half when moving ready tasks from *LReadyQ* to *SReadyQ*.

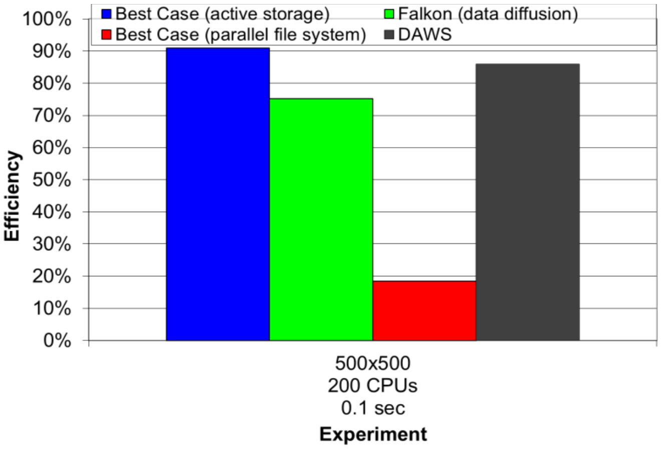

Figure 7: Comparison between Data Diffusion and DAWS

We compare DAWS with Data-Diffusion [32] in Figure 7. The "active storage" terms comes from [31] and means that all the data is stored locally in memory. The "parallel file system" means that the data is kept in the parallel file system through interfaces. We see that for 100-ms tasks at the scale of 200 cores, our DAWS technique improved Data Diffusion by 14.21% (85.9% vs 75%), and it is quite close to the best case using active storage (85.9% vs 91%). Data diffusion applies a centralized index-server for data-aware scheduling, while our DAWS technique utilizes DKVS that is much more scalable. For example, the centralized data diffusion showed good results for the all-pairs workload at 1K-cores, but it had to increase work granularity to 1 second long due to the inability of the data-aware scheduler to keep up with the increasing in scheduling throughput needed to maintain good efficiency. Based on prior results on work stealing in MATRIX [22], we anticipate the data-aware work stealing scheduler to support fine grained tasks of 10ms to 100ms with good efficiencies at 1000-node scale. It is also noteworthy to point out that without harnessing data-locality (Best Case parallel file system), the efficiency of this workload would be less than 20%, and it would only get worse with larger scales.

D. Comparisons Among Different Policies

We compare different scheduling polices using the all-pairs workloads. As all the tasks require the same amount of data (24MB), the *RLDS* is equivalent to the *MDL* policy. We compare the three scheduling polices, *MLB*, *MDL*, and *FLDS*. In the end, we will give a detailed guide on how to choose the best fit policy for different data-intensive applications.

Figure 8 shows the comparison results with different policies using the all-pairs workload. As we expected, *MLB* performs the worst because it just considers load balancing, and the required data is so large that it takes significant amount of time to transfer data. *MDL* policy performs moderately, and because all the required data has the same size (2*12MB=24MB), *MDL* policy is equivalent to *RLDS* when setting the threshold *t* to be small enough. From the load balancing's perspective, *MDL* did quite well except for the ending period. As it does not allow work stealing, load would be imbalanced at the final stage when there aren't many tasks leading to a long-tail problem. *FLDS* policy performs the best, as it has a monitoring thread that keeps polling the *LReadyQ* to move tasks to *SReadyQ* for load balancing. This is helpful at the final stage when some nodes are idle while others are busy.

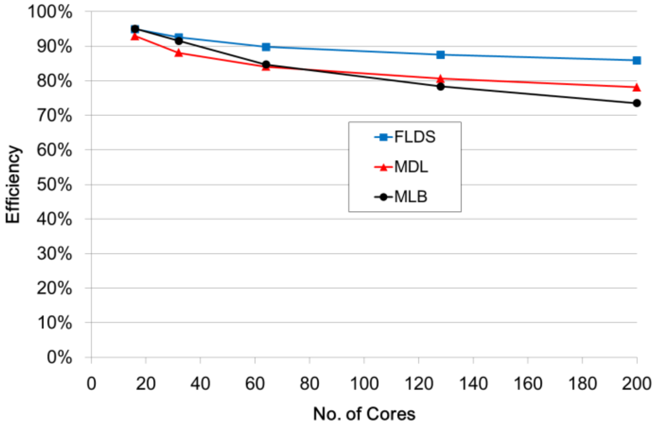

Figure 8: Comparison of different scheduling policies

Based on the above analysis, for applications that require large amount of data (e.g. several Megabytes) for each task, *FLDS* should always be the first choice. Unless the tasks require extremely large volume of data that would easily saturate the network bandwidth, the *MDL* policy should not be considered. *MLB* policy should only be used when tasks require small data pieces. *RLDS* policy should be preferable when the data pieces required by tasks have a very wide distribution in terms of size (from a few bytes to several Megabytes). In addition, our scheduler is able to adjust from one policy to another at runtime.

E. Different Benchmark DAGs

As we have shown that our proposed data-aware work stealing technique can perform well for data-intensive applications structured as simple DAGs, this section aims to evaluate more complex synthetic DAGs. We explore four different DAGs, namely Bag of Task (BOT), Fan-In, Fan-Out and Pipeline, which are represented in Figure 9.

BOT workload is used as a baseline, it includes independent tasks; Fan-In and Fan-Out DAGs are similar except that they are reverse. The performance of these workloads depends on the in-degree, out-degree and the dependent data sizes. Pipeline DAG is a collection of "pipes" where each task within a pipe is dependent on the previous task.

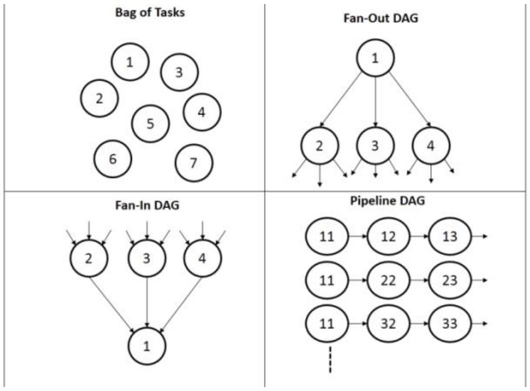

Figure 9: Various Workload DAGs representation

MATRIX client is able to generate a specific DAG given the input parameters that describe the DAG, such as DAG type (BOT, Fan-In, Fan-Out, Pipeline), DAG degree (fan-in degree, fan-out degree, and pipeline size). We run these synthetic DAGs in MATRIX up to 200 cores using *FLDS* policy. For all the DAGs, each core executes 1000 tasks on average, and each task runs an average time of 50ms (0 to 100ms) and generates an average data size of 5MB (0 to 10MB). We set the maximum data transfer rate threshold (*t*) to be 0.5*10Gbps = 5Gbps, a ratio of 0.5 between the data-transferring time and the estimated task length. We set the initial local ready queue execution time upper bound (*tt*) for *FLDS* policy to be 10 sec, and reduces it by half when moving ready tasks from *LReadyQ* to *SReadyQ*, and doubles it when work stealing fails. We set the fan-in degree, fan-out degree and pipeline size to be the same value of 10. We set the work stealing upper bound to be 50 sec, and the polling interval is changed adaptively as described in section III.A. In order to cooperate with the *FLDS* policy, after the polling interval of work stealing arrives the upper bound (no work stealing anymore), we set the polling interval back to the initial small value only if the threshold *tt* becomes too small to allow work stealing again.

Figure 10 shows the throughput results of all the DAGs up to 200 cores and 200K tasks. We see that for BOT workloads, we can achieve nearly-perfect performance, the throughput numbers imply a 90%+ efficiency for BOT workloads at all scales. This is because tasks are all run locally without requiring any piece of data. For the other three DAGs, our technique shows great scalability, as the throughput doubles when the scale and workload double. The throughput numbers are good, considering the data size and DAG complexities. Out of the three DAGs, Pipeline workloads show the highest throughput, as each task has at most one child and one parent. The data dependency condition is easy to be satisfied. For Fan-Out DAG, our experiments experienced a relatively long ramp-up period, as at the beginning, the number of ready tasks is small. Initially, only the root task is ready to run. As time increases, there would be more and more tasks that are ready, and we had better utilization. For Fan-In DAG, it is quite the opposite. At the beginning, tasks were running very fast. But it would get slower and slower, leading to a very long tail. This is not caused by load imbalance. In the end, it gets more and more difficult for a task to be ready given the Fan-In DAG shape and properties. This very long-tail has worse effect than that is caused by the slow ramp-up period for the Fan-Out DAG.

Above all, MATRIX shows great scalability running different complex benchmark DAGs. It is noteworthy that MATRIX is able to run any arbitrary workflow DAG, not just the few examples given in this paper.

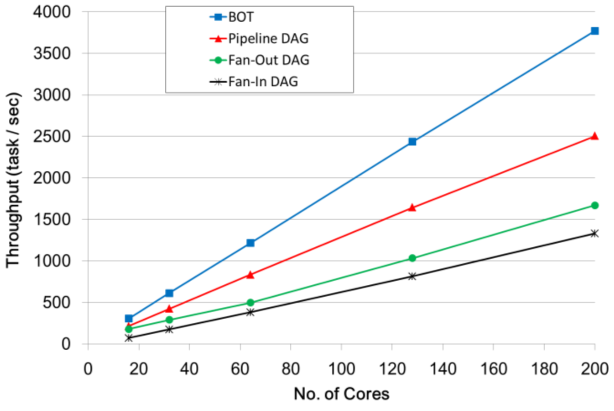

Figure 10: MATRIX with Benchmark DAGs

V. RELATED WORK

There has been a lot of research projects that are related to our work about load balancing and data-aware scheduling. This analyzes the similarities and differences comparing to our work.

Falkon [9] is a centralized task scheduler that supports naive hierarchical scheduling for MTC applications. Though Falkon scaled much better than others, it has problems to scale to even a petascale system, and the hierarchical implementation suffered from poor load balancing under unpredictable task execution times. Falkon also implemented a data diffusion approach [30] to schedule data-intensive workloads. Data diffusion acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. However, Falkon used a centralized index server to store the metadata, as opposed to our distributed key-value store, which leads to poor scalability.

Charm++ [33] is a machine independent parallel programming system, in which, load balancing can be performed in either a centralized (static), hierarchical or fully distributed (dynamic) fashion. The static approach has poor scalability (i.e. up to 3K cores [33]). The dynamic approach used the neighboring averaging schemes, which however limits the load balancing within a local space, and could yield poor load balance at extreme scales.

Sparrow [35] is similar to our work in that it implemented distributed load balancing for weighted fair shares, and supported the constraint that each task needs to be co-resident with input data, for fin-grained sub-second tasks. However, in Sparrow, each scheduler is aware of all the compute daemons, this design can cause a lot of resource contentions when the number of tasks are large. What's more, Sparrow implements pushing mechanism with early binding of tasks to workers. Each scheduler probes multiple compute nodes and assigns tasks to the least overloaded one. This mechanism suffers long-tail problem under heterogeneous workloads [34] due to early binding of tasks to worker resources. We have compared Sparrow and the basic MATRIX without data-aware scheduling technique using heterogeneous workloads in [39], and MATRIX outperforms Sparrow by 9X. Furthermore, there is an implementation barrier with Sparrow as it is developed in Java, which has little support in high-end computing systems.

Mesos [36] is platform for sharing resource between multiple diverse cluster computing frameworks to schedule tasks. Mesos allows frameworks to achieve data-locality by taking turns reading data stored on each machine. Mesos uses delay scheduling policy, and frameworks wait for a limited time to acquire nodes storing their data. However, this approach causes significant waiting time before a task could be scheduled, especially when the required data is large.

Quincy [37] is a flexible framework for scheduling concurrent distributed jobs with fine-grain resource sharing. Quincy tries to find optimal solutions of scheduling jobs under data-locality and load balancing constraints by mapping the problem to a graph data structure. Even though the motivation of Quincy is similar to our work, it takes significant amount of time to find the optimal solution of the graph that combines both load balancing and data-aware scheduling.

Dryad [38] is a general-purpose distributed execution engine for coarse-grained data-parallel applications. Dryad is similar with our work in that it supports running of applications structured as workflow DAGs. However, like the Hadoop scheduler [28], Dryad does centralized scheduling with a centralized metadata management that greedily maps tasks to the where the data resides, which is neither fair nor scalable.

CloudKon [39] has similar architecture as MATRIX, except that CloudKon focuses on the Cloud environment, and relies on the Cloud services, SQS [40] to do distributed load balancing, and DynamoDB [41] as the DKVS to keep task metadata. Relying on the Cloud services could facilitate the easier development, at the cost of potential loss of performance and control. Furthermore, CloudKon doesn't support data-aware scheduling at the current stage.

SLAW [42] is a scalable locality-aware adaptive work stealing scheduler that supports both work-first and help-first policies [43] adaptively at runtime on a per-task basis. Though SLAW aimed to address issues (e.g. locality-obliviousness, fixed task scheduling policy) that limit the scalability of work stealing, it focuses on the core/thread level. The technique would unlikely to hold for large-scale distributed systems.

Another work [44] that did data-aware work stealing is similar to us in that it uses both dedicated and share queues. However, it relies on the X10 global address space programming model [45] to statically expose the data-locality information and distinguish between location-sensitive an location-flexible tasks at beginning. Once the data-locality information of a task is defined, it remains unchanged. This is not adaptive to various data-intensive workloads.

VI. Conclusion and Future Work

Applications for extreme-scales are becoming more data-intensive and fine-grained in both task size and duration. Task schedulers for data-intensive applications at extreme-scales need to be scalable to deliver the highest system utilization, which poses urgent demands for both load balancing and data-aware scheduling. This work combined distributed load balancing with data-aware scheduling through a data-aware work stealing technique. We implement the technique in a distributed task execution fabric, MATRIX, and apply a DKVS, as a transparent meta-data service. We evaluated our technique under four different scheduling policies with different workloads, and compared our technique with the data diffusion approach. Results showed that our technique is scalable to achieve both good load balancing and high location-hit rate.

We have planned much work in the future, we will continue to scale MATRIX to the full scale of IBM BG/Q machine at ANL that has 768K cores, with 3M hardware threads. We will try to deploy MATRIX on accelerators and GPUs [53]. In order to make MATRIX versatile for extreme-scale ensemble computing [46], we will add HPC support [47] to MATRIX, so that MATRIX will be able to run HPC ensemble of workloads. We plan to integrate MATRIX with our SLURM++ project [46], which is a distributed job launch prototype developed from SLURM and ZHT, to explore different resource stealing techniques [48].

Another direction is to integrate scientific workflow engines, such as Swift [49][55], with MATRIX to enable running large-scale scientific applications. Swift will serve as the high-level parallel programming language between the applications and MATRIX. Instead of having Swift manage the DAG, the DAG would be managed in a distributed way by MATRIX. Furthermore, we will be working on extending the centralized Hadoop scheduler [28] to be distributed through MATRIX. We will extend MATRIX to support the scheduling of the MapReduce styled data-intensive workloads [50][54]. We will utilize distributed file systems, such as FusionFS [51] and HDFS [27], to help MATRIX manage data in a distributed, scalable, and reliable way.

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 CNS-1042543 (PRObE, http://www.nmc-probe.org/). This work was also in collaboration with the FusionFS project supported by the NSF grant NSF-1054974.

References

[1] I. Raicu, I. Foster, Y. Zhao. "Many-Task Computing for Grids and Supercomputers", Invited Paper, IEEE MTAGS 2008.

[2] V. Sarkar, S. Amarasinghe, et al. "ExaScale Software Study: Software Challenges in Extreme Scale Systems", ExaScale Computing Study, DARPA IPTO, 2009.

[3] X. Besseron and T. Gautier. "Impact of Over-Decomposition on Coordinated Checkpoint/Rollback Protocol", Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science Volume 7156, 2012, pp 322-332.

[4] D. Zhao, D. Zhang, K. Wang, I. Raicu. “Exploring reliability of exascale systems through simulations.” ACM HPC 2013.

[5] M. A. Jette, A. B. Yoo, M. Grondona. “SLURM: Simple Linux utility for resource management.” Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2003.

[6] D. Thain, T. Tannenbaum, M. Livny, “Distributed Computing in Practice: The Condor Experience” Concurrency and Computation: Practice and Experience 17 (2-4), pp. 323-356, 2005.

[7] W. Gentzsch, et. al. “Sun Grid Engine: Towards Creating a Compute Power Grid,” 1st International Symposium on Cluster Computing and the Grid, 2001.

[8] Cobalt: http://trac.mcs.anl.gov/projects/cobalt, 2014.

[9] I. Raicu, Y. Zhao et al. “Falkon: A Fast and Light-weight tasK executiON Framework,” IEEE/ACM SC 2007.

[10] I. Raicu. “Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing”, ISBN: 978-3-639-15614-0, VDM Verlag Dr. Muller Publisher, 2009.

[11] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. “Toward Loosely Coupled Programming on Petascale Systems”, IEEE/ACM Supercomputing 2008.

[12] M. H. Willebeek-LeMair, A. P. Reeves. “Strategies for dynamic load balancing on highly parallel computers,” In IEEE Transactions on Parallel and Distributed Systems, volume 4, September 1993.

[13] R. D. Blumofe and C. E. Leiserson. “Scheduling multithreaded computations by work stealing”, Symposium on Foundations of Computer Science 1994.

[14] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang, I. Raicu. “Paving the Road to Exascale with Many-Task Computing”, Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012.

[15] K. Wang, K. Brandstatter, I. Raicu. “SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascales,” ACM HPC 2013.

[16] K. Wang, J. Munuera, I. Raicu, H. Jin. “Centralized and Distributed Job Scheduling System Simulation at Exascale.” Tech Report, IIT, 2011.

[17] A. S. Szalay et al. “GrayWulf: Scalable Clustered Architecture for Data-Intensive Computing”, Proceedings of the 42nd Hawaii International Conference on System Sciences, Hawaii, 5 to 8 January 2009, paper no. 720; available as Microsoft Tech Report MSR-TR-2008-187 at http://research.microsoft.com/apps/pubs/default.aspx?id=79429.

[18] J. D. Ullman. “NP-complete scheduling problems”, Journal of Computer and System Sciences, Volume 10 Issue 3, June, 1975, Pages 384-393.

[19] Gibson, G., Grider, G., Jacobson, A. and Lloyd, W. “PRObE: A thousand-node experimental cluster for computer systems research.” Usenix ;login: 38, 3 (2013).

[20] D. Chase, Y. Lev. “Dynamic circular work-stealing deque”, Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA’05), 2005, pp 21 – 28.

[21] T. Li, X. Zhou, et al. “ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, IPDPS, 2013.

[22] K. Wang, A. Rajendran, I. Raicu. “MATRIX: MAny-Task computing execution fabRIc at eXascale,” tech report, IIT, 2013.

[23] K. Wang, A. Kulkarni, M. Lang, D. Arnold, I. Raicu. “Using Simulation to Explore Distributed Key-Value Stores for Extreme-Scale Systems Services,” IEEE/ACM Supercomputing/SC 2013.

[24] K. Wang, A. Kulkami, M. Lang, I. Raicu. “Exploring Design Tradeoffs for Exascale System Services through Simulation.” Tech Report, Los Alamos National Laboratory, 2013.

[25] J. Dinan, D. B. Larkins. “Scalable work stealing”, IEEE/ACM SC 2009.

[26] V. G. Cerf, R. E. Kahn. “A Protocol for Packet Network Intercommunication,” IEEE Transactions on Communications 22 (5): 637–648, May 1974.

[27] K. Shvachko, H. Huang, S. Radia, R. Chansler. “The hadoop distributed file system”, in: 26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies, May, 2010.

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

[29] Google. “Google Protocol Buffers,” available at http://code.google.com/apis/protocolbuffers/, 2014.

[30] I. Raicu, Y. Zhao, et al. “Accelerating Large-scale Data Exploration through Data Diffusion”, International Workshop on Data-Aware Distributed Computing 2008, co-locate with ACM/IEEE HPDC 2008.

[31] Christopher Moretti, Hoang Bui, Karen Hollingsworth, Brandon Rich, Patrick Flynn, and Douglas Thain. 2010. All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids. IEEE Trans. Parallel Distrib. Syst. 21, 1 (January 2010), 33-46.

[32] I. Raicu, I. Foster, et al. “The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems”, ACM HPDC 2009.

[33] G. Zhang, E. Meneses, A. Bhatele, and L. V. Kale. “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers”, Conference on Parallel Processing Workshops, ICPPW10, 2010.

[34] K. Wang, Z. Ma, I. Raicu. “Modeling Many-Task Computing Workloads on a Petaflop IBM Blue Gene/P Supercomputer.” IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) 2013.

[35] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica. “Sparrow: distributed, low latency scheduling”, Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP’13, pp. 69-84.

[36] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. “Mesos: A PlatformFor Fine-Grained Resource Sharing in the Data Center”, NSDI 2011.

[37] M. Isard, V. Prabhakaran, et al. “Quincy: fair scheduling for distributed computing clusters”, Proceedings of the ACM Symposium on Operating Systems Principles, SOSP’09, pp. 261-276.

[38] M. Isard, M. Budiu, et al. “Dryad: Distributed data-parallel programs from sequential building blocks”, In Proc. Eurosys, March 2007.

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

[40] Amazon Simple Queue Service. Avaible online: http://aws.amazon.com/documentation/sqs/. 2014.

[41] G. DeCandia, D. Hastorun, M. Jampani, et al. “Dynamo: Amazon’s highly available key-value store”, ACM SOSP, 2007.

[42] Y. Guo, J. Zhao, V. Cave, V. Sarkar. “SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems”, Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2010.

[43] Y. Guo, R. Barik, R. Raman, and V. Sarkar, “Work-first and help-first scheduling policies for async-finish task parallelism,” IPDPS, 2009.

[44] J. Paudel, O. Tardieu and J. Amaral. “On the merits of distributed work-stealing on selective locality-aware tasks”, ICPP, 2013.

[45] P. Charles, C. Grothoff, et al. “X10: An Object-oriented Approach to Non-uniform Cluster Computing,” ACM Conference on Object-oriented Programming Systems Languages and Applications(OOPSLA), 2005.

[46] K. Wang, X. Zhou, et al. “Next Generation Job Management Systems for Extreme-Scale Ensemble Computinig”, ACM HPDC 2014.

[47] K. Ramamurthy, K. Wang, I. Raicu. “Exploring Distributed HPC Scheduling in MATRIX”. Tech Report, IIT, 2013.

[48] X. Zhou, H. Chen, K. Wang, M. Lang, I. Raicu. “Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System.” Tech Report, IIT, 2013.

[49] Y. Zhao, M. Hategan, et al. “Swift: Fast, Reliable, Loosely Coupled Parallel Computation,” IEEE Workshop on Scientific Workflows 2007.

[50] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”, OSDI'04, CA, December, 2004.

[51] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. “FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems.” IEEE BigData, 2014.

[52] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu. “Exploring Distributed Hash Tables in High-End Computing”, ACM Performance Evaluation Review (PER), 2011.

[53] S. Krieder and I. Raicu. “Towards the Support for Many-Task Computing on Many-Core Computing Platforms”, Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012.

[54] H. Jin, X. Yang, X. Sun and I. Raicu. “ADAPT: Availability-aware MapReduce Data Placement in Non-Dedicated Distributed Computing Environment”, IEEE International Conference on Distributed Computing Systems (ICDCS) 2012.

[55] M. Wilde, I. Raicu et al. “Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers”, Poster Presentation, Scientific Discovery through Advanced Computing Conference (SciDAC09) 2009.

[56] I. Raicu, I. Foster, A. Szalay and G. Turcu. “AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis,” TeraGrid Conference 2006, June 2006.