CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; 00:1-29

Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales

Ke Wang[1,*,†], Kan Qiao[2], Iman Sadooghi[1],
Xiaobing Zhou[3], Tonglin Li[1], Michael Lang[4], Ioan Raicu[1,5]

[1]*Illinois Institute of Technology, USA;* [2]*Google Inc., USA;* [3]*Hortonworks Inc., USA*
[4]*Los Alamos National Laboratory, USA;* [5]*Argonne National Laboratory*

[†]E-mail: kwang22@hawk.iit.edu

SUMMARY

Data driven programming models such as many-task computing (MTC) have been prevalent for running data-intensive scientific applications. MTC applies over-decomposition to enable distributed scheduling. To achieve extreme scalability, MTC proposes a fully distributed task scheduling architecture that employs as many schedulers as the compute nodes to make scheduling decisions. Achieving distributed load balancing and best exploiting data-locality are two important goals for the best performance of distributed scheduling of data-intensive applications. Our previous research proposed a data-aware work stealing technique to optimize both load balancing and data-locality by using both dedicated and shared task ready queues in each scheduler. Tasks were organized in queues based on the input data size and location. Distributed key-value store was applied to manage task metadata. We implemented the technique in MATRIX, a distributed MTC task execution framework. In this work, we devise an analytical sub-optimal upper bound of the proposed technique; compare MATRIX with other scheduling systems; and explore the scalability of the technique at extreme scales. Results show that the technique is not only scalable, but can achieve performance within 15% of the sub-optimal solution. Copyright © 2015 John Wiley & Sons, Ltd.

Received …

KEY WORDS: data-intensive computing; data-aware scheduling; work stealing; key-value stores; many-task computing

1. INTRODUCTION

Large-scale scientific applications are ushering in the era of big data such that task execution involves consuming and producing large volumes of input and output data with data dependencies among tasks. These applications are referred to as data-intensive applications that cover a wide range of disciplines, including data analytics, bioinformatics, data mining, astronomy, astrophysics, and MPI ensembles [1]. The big data phenomenon has expedited the evolution of paradigm shifting from compute-centric model to data-centric one.

*Correspondence to: Ke Wang, Department of Computer Science, Illinois Institute of Technology, 10 W 31st St, Stuart Building, Room 002, Chicago IL, 60616, USA.

Scheduling data-intensive applications is dramatically different from scheduling the traditional high performance computing (HPC) MPI applications [54, 55, 66, 67], which are usually comprised of big tightly-coupled jobs with long durations, and the amount of jobs is not that many. A centralized batch scheduling system, such as SLURM [2], Condor [3], SGE [4], and Cobalt [5], may still work well at certain scales, as there wouldn't be many scheduling decisions to make. Load balancing and data-locality are both trivial due to the global view of the centralized scheduler. On the contrary, more and more data-intensive applications are becoming loosely coupled in nature. They contain many small jobs/tasks (e.g. per-core) that have shorter durations (e.g. sub-second) and large volumes of data dependencies [1]. As systems are approaching billion-way concurrency at exascale [6], we argue that the data driven programming models will likely employ over-decomposition to generate more fined-grained tasks than available parallelism. While over-decomposition has the ability to improve utilization and fault tolerance at extreme scales [7], it poses severe challenges on scheduling system to make fast scheduling decisions (e.g. millions/sec) and to be available, in order to achieve the best performance. These requirements are far beyond the capability of today's centralized batch scheduling systems.

The Many-task computing (MTC) [8] paradigm comes from the data driven model, and aims to address the challenges of scheduling fine-grained data-intensive workloads [9]. MTC applies over-decomposition to structure applications as Direct Acyclic Graphs (DAG), in which the vertices are small discrete tasks and the edges represent the data flows from one task to another. The tasks are embarrassingly parallel with fine granularity in both size (e.g. per-core) and durations (e.g. sub-seconds to a few seconds).

We have shown that the MTC paradigm will likely require a fully distributed task scheduling architecture (as opposed to the centralized one) that employs as many schedulers as the compute nodes to make scheduling decisions, in order to achieve high efficiency, scalability, and availability [10] for exascale machines with billion-way parallelism [6]. As at exascale, each compute node would have 2 to 3 orders of magnitude more intra-node parallelism, and the MTC data-intensive workloads contain extremely large amount of fine-grained tasks. Therefore, there would need a scheduler on one "fat" compute node forming 1:1 mapping to make full utilization of the node. All the schedulers are aware of each other and receive workloads to schedule tasks to local executors. Therefore, ideally, the throughput would gain near-optimal linear speedup as the system scales. Besides, failures only affect the tasks that are run on the failed nodes, and can be resolved by resubmitting the affected tasks to other schedulers for execution without much effort.

The two important but conflicting goals of the distributed scheduling are load balancing and data-locality [10]. Load balancing [11] refers to distributing workloads as evenly as possible across all the schedulers, and it is important given that a single heavily loaded scheduler would lower the system utilization significantly. Data-locality aware scheduling requires mapping a task to the node that has the input data. This aims to minimize the overheads of moving large volumes of data through network. To achieve dynamic load balancing, work stealing technique [12,13,14] was utilized such that the idle schedulers poll neighbors to balance their loads by migrating tasks from the overloaded schedulers. However, the action of moving tasks randomly regardless of the data-locality may incur significant data-transferring overhead. To best exploit data-locality, we need to map each task to where the data resides. This is infeasible because this mapping is an NP-complete problem [17], and is leading to poor load balancing due to the potential unbalanced data distribution.

To optimize between both goals, our previous work [10] proposed a data-aware work stealing (DAWS) technique that was able to achieve good load balancing and also tried to best exploit data-locality. Each scheduler maintained both dedicated and shared task ready queues that were implemented as max priority queues [18] based on the data size a task requires. Tasks in the dedicated queue were confined to be executed locally unless special policy was applied, while tasks in the shared queue may be migrated through work stealing among schedulers for balancing loads. A ready task was put in either queue based on the size and location of the required input data. A distributed key-value store (KVS) (i.e. ZHT [19, 62, 20, 21]) was applied as a metadata service to keep task metadata, including data dependency and locality information, for all the tasks. We implemented the technique in MATRIX [22, 64, 70], a MTC task execution framework, and evaluated up to 200 cores.

This article makes the following new contributions that extend the previous work broader in scope and more in depth:

(1) Devise an analytical model to analyze the DAWS technique. This model gives a sub-optimal upper bound of the performance of the technique and helps us understand it in depth from a theoretical perspective.

(2) Compare the experimental results with those achieved through the analytical model, and explore the scalability of the technique through the model up to extreme scales of 128K cores.

(3) Add additional fine-grained task scheduling systems (Sparrow [23], CloudKon [24]) to the performance comparison, further showing the potential broader impact of the DAWS technique on the Cloud data centers.

The rest of this article is organized as follows. Section 2 introduces the fully distributed scheduling architecture, the previously proposed data-aware work stealing technique, and the MATRIX task scheduling system. Section 3 devises an analytical model that gives a sub-optimal upper bound of the technique. Section 4 evaluates the technique through MATRIX using real applications, as well as benchmarking workloads. We list the related work in Section 5. Section 6 draws the conclusions and presents the future work.

2. LOAD-BALANCED AND LOCALITY-AWARE SCHEDULING

This section introduces the fully distributed scheduling architecture, the proposed data-aware work stealing (DAWS) technique, and the implementation details of MATRIX. The bulk of this section is from the previous paper [10] with a shortened version, along with a couple of new added subsections.

2.1. Fully Distributed Scheduling Architecture

We have shown that the MTC paradigm will likely require a fully distributed task scheduling architecture for exascale machines. The architecture is shown in Figure 1.

The system has four components: client, scheduler, executor, and key-value store (KVS). Each compute node runs a scheduler, an executor and a KVS server. The client issues requests to generate a set of tasks, puts task metadata into KVS servers, submits tasks to all schedulers, and monitors the execution progress. The schedulers are fully connected, and map tasks to either the local or the remote executors, according to the location and size of the required data of the tasks. Whenever a scheduler has no ready tasks, it communicates with other schedulers to migrate 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.

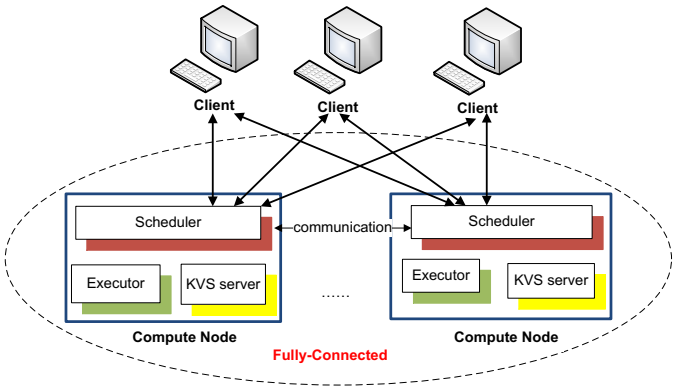


Figure 1: Fully distributed scheduling architecture

A distributed key-value store (KVS), in our case the ZHT KVS [19, 25], is used to monitor the execution progress and to keep the metadata of all the tasks and system states in a distributed, scalable, and fault tolerant way. ZHT is a zero-hop persistent distributed KVS with each ZHT client having a global view of all the ZHT servers. For each operation (e.g. *insert*, *lookup*, *remove*) of ZHT server, there is a corresponding client API. The client calls the API, which sends a request to the exact server (known to the client by hashing the *key*) that is responsible for the request. Upon receiving a request, the ZHT server executes the corresponding operation. ZHT serves as a data management building block for extreme scale system software, and has been tested up to 32K cores on an IBM Blue Gene /P supercomputer [19].

Both the system clients and schedulers are initialized as ZHT clients. The system client inserts all the task metadata information to ZHT before submitting tasks to all the schedulers. The scheduler queries and updates the task metadata when scheduling tasks, and in the meanwhile, puts local state information (e.g. number of waiting, ready, and complete tasks, number of idle executing threads) to ZHT periodically and the system client keeps monitoring this information until all tasks are completed.

2.2. Data-Aware Work Stealing

This section covers the proposed data-aware work stealing (DAWS) technique. We first describe the adaptive data-locality oblivious work stealing technique that aims to achieve load balancing. Then we put constraints on it to enable data-aware scheduling.

2.2.1. Adaptive Work Stealing

Work stealing has been proven as an efficient load balancing technique at the thread/core level in shared memory environment [11, 26]. It is a pull-based method in which the idle processors randomly steal tasks from the overloaded ones. Our fully distributed scheduling system adopts work stealing at the node level in distributed environment. When a scheduler is idle, it randomly chooses some candidate neighbors. Then, it goes through all neighbors in sequential to query the "***load information***" (*number of ready tasks*), and tries to steal tasks from the most heavily loaded neighbor.

When a work stealing operation fails because either none of the selected neighbors have ready tasks, or the reported tasks have been executed when the stealing happens, the scheduler waits for a period of time (**polling interval**) and then does work stealing again. We implemented an adaptive, exponential back-off polling interval strategy: The default polling interval is set to a small value (e.g. 1 ms). Once a scheduler

successfully steals tasks, it sets its polling interval back to the initial small value. Otherwise, the scheduler waits the time of polling interval and doubles it and tries to do work stealing again. Also, we set an upper bound to the polling interval. Whenever the value hits the upper bound, a scheduler would not do work stealing anymore. This aims at reducing the amount of failed work stealing at the final stage when there are rare tasks.

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 [13, 15, 16] studied the parameter space extensively and found the optimal configurations that can minimize the network communication overhead while achieving load balancing: *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 strategy*.

2.2.2. Data-Aware Work Stealing (DAWS)

The adaptive work stealing technique is data-locality oblivious. This may incur significant data movement overheads for data-intensive workloads. We present the ideals that combine work stealing with data-aware scheduling.

1) Distributed KVS Used as a Meta-Data Service

As distributed key-value store (KVS) has gained its popularity in serving as a building block for distributed system services [25, 61, 65], we apply ZHT (a distributed KVS) to store task metadata as (*key*, *value*) records for all the tasks. The *key* is task id, and the *value* is defined as the following data structure in Figure 2. The *value* includes the following information: the task status (e.g. queuing, being executed, and finished); data dependency conditions (*num_wait_parent*, *parent_list*, *children*); data locality information (*data_object*, *data_size*, *all_data_size*); task timestamps that record the times of different phases (e.g. submission, queued, execution, and end) of the task; and the task migrating history from one scheduler to another in the system.

```
typedef  TaskMetaData {
   byte status;          // the status of the task: queuing, being executed, finished
   int num_wait_parent;        // number of waiting parent
   vector<string> parent_list;        // schedulers that run each parent task
   vector<string> children;        // children of this tasks
   vector<string> data_object;        // data object name produced by each parent
   vector<long> data_size;        // data object size produced by each parent
   long all_data_size;        // all data object size (byte) produced by all parents
   List<long> timestamps;        // time stamps of a task of different phases
   List<string> history;        // the provenance of a task, from one node to another
} TMD;
```

Figure 2: Task metadata stored in ZHT

Job Submission: Before submitting an application workload DAG to the schedulers for scheduling, the client generates a task metadata (focusing on the "*num_wait_parent*" and "*children*") for each task and inserts all the task metadata to ZHT. The task metadata will be updated later by the schedulers when task state changes. There are different mechanisms through which the client submits the tasks to the schedulers. The first one is the worst case, in which the client submits all the tasks to only one arbitrarily chosen scheduler. This is the worst case scenario from load balancing's perspective. The tasks will be spread out among all the schedulers through the work stealing technique. The second one is the best case, in which the client submits all the tasks to all the schedulers through some load balancing method

(e.g. hashing the task ids among all the schedulers). In addition, the client is able to submit tasks to whatever groups of schedulers it wants to.

Job Removal: The client can conduct the job removal easily when it decides to remove a task in a workload after submission. The client can simply *lookup* the task metadata from ZHT, and finds out where the task is at present – the last scheduler in the *history* field of the task metadata. Then, the client sends a message to the last scheduler to request removing the task. After the scheduler receives the message, it deletes the task in one of the task queues (will explain later). If the removed task has not been finished (identified by the *status* field), the scheduler will need to remove all the tasks in the subtree rooted as the removed task, because these tasks are waiting for the removed task to be finished while this will never happen due to the removal. The scheduler can remove these tasks as follows: First, the scheduler queries ZHT for the children of the removed task. Then, for each task, it checks which scheduler is holding the task, and sends removal message to that scheduler. This procedure is continued until all the subtree tasks are removed.

2) Distributed Queues in Scheduler

Each scheduler maintained four local task queues: wait queue (*WaitQ*), dedicated local ready queue (*LReadyQ*), shared work stealing ready queue (*SReadyQ*), and complete queue (*CompleteQ*) [68], as shown in Figure 3. These queues hold tasks in different states (stored as metadata in ZHT). A task is moved from one queue to another when state changes. With these queues, the system supports scheduling tasks with data dependencies specified by an arbitrary DAG.

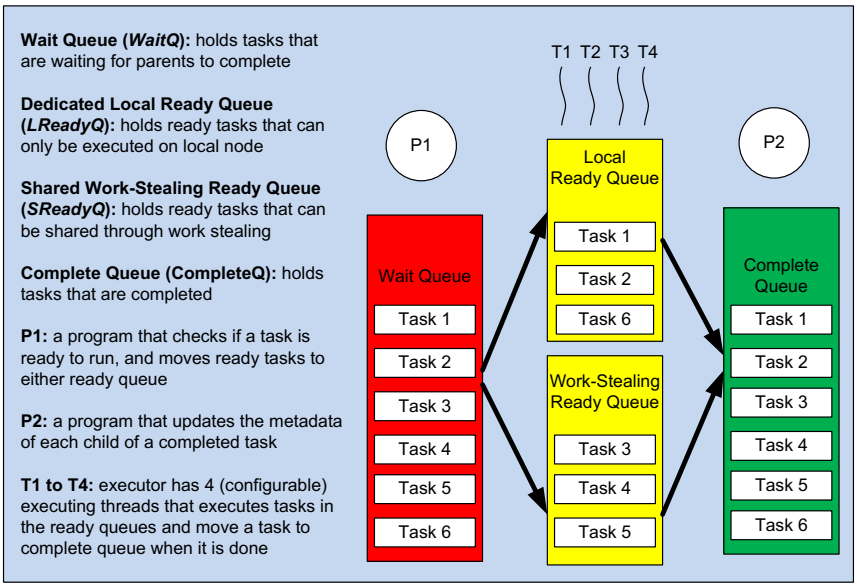


Figure 3: Specification of task queues in a scheduler

Figure 4 displays the flowchart of the execution procedure of a task, during which the task is moved from one queue to another. Initially, the scheduler puts all the incoming tasks from the client to the *WaitQ*. A program (P1 in Figure 3) keeps checking every task in the *WaitQ* to see whether the task is ready to run by querying the metadata from ZHT. The task metadata has been inserted into ZHT by the client. Specifically, only if the value of the field of "*num_wait_parent*" in the *TMD* is equal to 0 would the task be ready to run. When a task is ready to run, the scheduler makes decision to put it in either the *LReadyQ* or the *SReadyQ*, or push it to another node.

The decision making procedure is shown in the rectangle that is marked with dotted-line edges in Figure 4. We will explain the decision making algorithm later.

When a task is done, it is moved to the CompleteQ. Another program (P2 in Figure 3) is responsible for updating the metadata for all the children of each completed task. P2 first queries the metadata of the completed task to find out the children, and then updates each child's metadata 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" by the size of the produced data object.

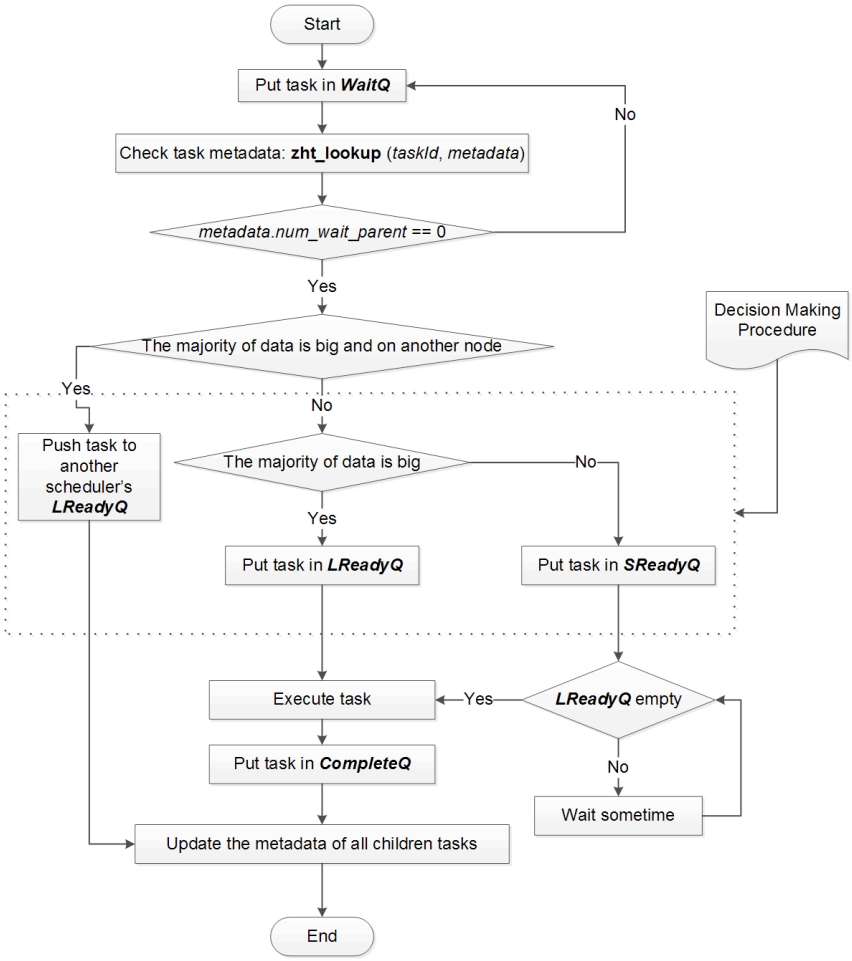


Figure 4: Flowchart of a task during the execution procedure

3) *Decision Making Algorithm*

We explain the decision making procedure (the dotted-line rectangle in Figure 4) that decides to put a ready task in either the *LReadyQ* or the *SReadyQ*, or push it to another node, given in Algorithm 1.

The *SReadyQ* stores the tasks that can be migrated to any scheduler for load balancing's purpose (lines 1 – 13), the "***load information***" queried by work stealing is the length of the *SReadyQ*; these tasks either don't need any input data or the demanded data volume is so small that the transferring overhead is negligible. The *LReadyQ* stores the tasks that require large volumes of data and the majority of the data is at the current node (lines 14 – 15); these tasks are confined to be scheduled and executed locally unless special policy is used. If the majority of the input data is large but at a different node, the scheduler then pushes the task to that node (lines 16 – 18).

When a scheduler receives a pushed task, it puts the task in the *LReadyQ*. The threshold *t* defines the upper bound of the ratio of the data-transferring overhead (data size divided by network bandwidth: *net_band*) to the estimated task execution length (*est_task_length*). The smaller *t* means the less tolerance of moving data: If *t* is smaller, in order to put the task in *SReadyQ* (meaning that the task can be migrated through work stealing technique and that moving the data is tolerable), the task's all required data size (*tm.all_data_size*) needs to be smaller. This means less tolerance of moving a decent large amount of data.

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 / net_band / 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 / net_band / 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;
```

As we don't know ahead how long a task will be running, we will need to predict the *est_task_length* in some ways. One method is to use the average execution time of the completed tasks as the *est_task_length*. This method works fine for workload of largely homogeneous tasks that have small variance of task lengths. For highly heterogeneous tasks, we can assume the task length conforms to some distributions, such as uniform random, Gaussian, and Gama, according to the applications. This can be implemented in our technique without much effort. Even though the estimation of the task length has deviation, our technique can tolerate it with the dynamic work stealing technique, along with the *FLDS* policy (will explain later). In addition, we have evaluated MATRIX using highly heterogeneous MTC workload traces that have 34.8M tasks with the minimum runtime of 0 seconds, maximum runtime of 1469.62 seconds, medium runtime of 30 seconds, average runtime of 95.20 seconds, and standard deviation of 188.08 [31], and MATRIX shows great scalability (results were presented in [24]).The executor forks configurable number (usually equals to number of cores of the node) of threads to execute ready tasks. Each thread first pops tasks from the *LReadyQ*, and then from the *SReadyQ* if the *LReadyQ* is empty. Both ready queues are implemented as max priority queue based on the data size. When executing a task, the thread first queries the metadata to find the size and location of the required data, and then collects the data either from local or remote nodes. If neither queue has tasks, the scheduler does work stealing, and puts the stolen tasks in the *SReadyQ*.

4) Different Scheduling Policies

We define four scheduling policies for the DAWS technique, specified as follows:

a) MLB: maximized load balancing

MLB considers only the load balancing, and all the ready tasks are put in the *SReadyQ* allowing to be migrated. We achieve the *MLB* policy by tuning the threshold *t* in Algorithm 1 to be the maximum possible value (i.e. LONG_MAX).

b) MDL: maximized data-locality

MDL only considers data-locality, and all the ready tasks that require input data would be put in the *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: rigid load balancing and data-locality segregation

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: flexible load balancing and data-locality segregation

The *RLDS* policy may lead to poor load balancing when a task that produces large volumes of data has many children. To avoid this problem, we relax the *RLDS* to the flexible *FLDS* policy that allows tasks to be moved from the *LReadyQ* to the *SReadyQ* under certain circumstance. We set a time threshold *tt* and use a monitoring thread to check the *LReadyQ* periodically. If the estimated running time (*est_run_time*) of the *LReadyQ* is above *tt*, the thread then moves some tasks to guarantee that the *est_run_time* is below *tt*. The *est_run_time* equals to the *LReadyQ* length divided by the *throughput* of the scheduler. Assuming 1000 tasks are finished in 10sec, the *LReadyQ* has 5000 tasks, and *tt*=30sec. We calculate the number of moving tasks: *throughput*=1000/10=100tasks/sec, *est_run_time*=5000/100=50sec, 20sec longer than *tt*. 20sec takes 20/50=40% ratio, therefore, 40%*5000=2000 tasks will be moved.

5) Write Locality and Read Locality

The DAWS technique ensured the best write locality, and at the meanwhile, optimized the read locality. For write locality, every task writes the produced data locally. We had considered using ZHT as both a data and a task metadata service. However, it led to extreme difficulty in optimizing the data-locality as ZHT relies on consistent hashing to determine where to store the data. We also considered leveraging distributed file system (e.g. HDFS [27], FusionFS [28, 79, 80]) to manage the data, especially as FusionFS is optimized for write operations. We argue that the scheduling 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 was optimized by migrating a task to where the majority of data resides for large data volumes. For small data volumes, tasks are run on wherever there are available compute resources to maximize utilization.

6) Caching

As in scientific computing, the normal pattern of data flows is write-once/ready-many (according to the assumption HDFS made in the Hadoop system [29]), we have implemented a caching mechanism to reduce the data movement overheads. In some cases, moving data from one node to another is inevitable. For example, if a task requires two pieces of data that are at different nodes, at least one piece of data needs to be moved. In a data movement, we cached the moved data locally at the receiver side for the future use by other tasks. This would significantly expedite the task

execution progress. As data is written once, all the copies of the same data would have the same view, and no further consistency management would be needed.

7) List of Short Terms

To summarize and make it clear about the important short terms we use in this article, we list and explain them in Table 1 as follows.

Table 1: List of important short terms

Term	Description
KVS	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 defines the upper bound of the ratio of the data-transferring overhead to the estimated task execution length
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 the *LReadyQ* to the *SReadyQ* for the *FLDS* policy
DAWS	Data aware work stealing

2.3. Fault Tolerance

Fault tolerance refers to the ability of handing failures (e.g. nodes are down) of a system. The goal of designing fault tolerance mechanisms is at least twofold: one is that the system should still be operable under failures, the other one is that the system should handle the failures without drawing much attention of the users. Fault tolerance is an important design concern of efficient systems software, especially for exascale machines that have high failure rates. Our distributed scheduling architecture has the ability to tolerate failures with a minimum effort because of the distributed nature, and the fact that the schedulers are stateless with the ZHT key-value store managing the task metadata. When a compute node is down due to hardware failures, only the tasks in the scheduler's queues, data files in the memory and persistent storage, and metadata in the ZHT server, of that particular node are affected, which can be resolved as follows. (1) First of all, a monitoring system software (MSS) could be applied, which detects the node failures by issuing periodic "heart-beat" messages to the nodes; (2) The affected tasks can be acknowledged and resubmitted to other schedulers by the clients. In the future, we will implement mechanism that can resubmit the affected tasks automatically without any user interaction. For example, the MSS will keep a copy of all the submitted tasks, and each scheduler can write the list of task ids of local unfinished tasks to ZHT. Whenever the MSS detects a failed scheduler, the MSS looks up the task ids of the unfinished tasks of the failed scheduler from ZHT, and then resubmit the corresponding affected tasks; (3) A part of the data files were copied and cached in other compute nodes when they were transmitted for executing some tasks. In the future, we will rely on the underneath file system to handle the affected files; (4) As ZHT is used to store the metadata, and ZHT has implemented failure/recovery, replication and consistency mechanisms, MATRIX needs to worry little about the affected metadata.

2.4. Implementation Details

We had implemented a MTC task execution framework, MATRIX, for scheduling data-intensive applications. MATRIX had the fully distributed scheduling architecture shown in Figure 1, and implemented the DAWS technique. MATRIX simply used ZHT as a black box through ZHT client APIs, ensuring easier maintainability and extensibility. MATRIX codebase was made open source on Github: https://github.com/kwangiit/matrix_v2. It had about 3K lines of C++ code implementing the MATRIX client, scheduler, the executor, and the DAWS logic, along with 8K lines of ZHT codebase, plus 1K lines of auto-generated code from Google Protocol Buffer [30]. MATRIX had dependencies on ZHT [19] and Google Protocol Buffer.

3. THEORETICAL ANALYSIS OF THE DAWS TECHNIQUE

To understand the proposed DAWS technique in depth from a theoretical perspective, we give a theoretical analysis about the upper bound of the performance of the technique in terms of the overall timespan of executing a given data-intensive MTC workload. The theoretical analysis is a centralized algorithm that has a global knowledge of the system states (e.g. resource and task metadata), and aims to find the shortest overall timespan of executing a workload.

The problem is modeled as follows. The workload is represented as an direct acyclic graph (DAG), $G = (V, E)$, along with several cost functions. Each vertex v ($v \in V$) is a task, which takes $t_{exec}(v)$ unit of execution time and generates an output of data with size $d(v)$. Assume that each ready task v gets queued to wait $t_{qwait}(v)$ unit of time on average before being executed. The value of $t_{qwait}(v)$ is directly related to the compute node that runs the task v and the individual task execution time. This is because a compute node has certain amount of processing capacity that can execute a limited number of tasks in parallel. The processing capacity is usually measured as the number of idle cores. We evaluate the $t_{qwait}(v)$ of task v as the average task waiting time of all the tasks on one node to release some time constraints, with the following estimation.

Assuming on average, every core of a compute node gets k tasks that have an average execution time of l. Therefore, the λth task needs to wait $(\lambda - 1)\times l$ time before being executed. Thus, the $t_{qwait}(v)$ on average is:

$$t_{qwait}(v) = \frac{\sum_{\lambda=1}^{k}\big((\lambda - 1)\times l\big)}{k} = \frac{(k-1)\times l}{2} \quad (1)$$

Define the time taken to move $d(v)$ size of data to another compute node running a task w that requires the data is $t_{mvdata}(v, w)$ unit of time. $t_{mvdata}(v, w) = d(v)/B(v, w)$, in which $B(v, w)$ is the data transfer rate between the two compute nodes that run tasks v and w, respectively. For any arc $uv \in E$, it represents that task u is the parent of task v, meaning that task v requires the data output of task u. The parents of a task v is notated as $P(v) = \{u | uv \in E\}$.

Generally, there are two categories of locations where a task could be scheduled. One is on the compute node that is different from all the compute nodes that ran the task's parents (**case 1**). In this case, the data items that are generated by all the parents need to be transmitted, and we assume that the task itself is not moved. The other one is on one of the compute nodes that ran at least one of the task's parents (**case 2**). In this case, the data items that are generated by all the other parents that were run on

different compute nodes need to be transmitted, and we assume that the task itself is also moved.

We define the problem of getting the minimum timespan of executing a given data-intensive MTC workload as follows:

- Define $t_{ef}(v)$ as the earliest finishing time of task v.
- The problem is to find out the largest $t_{ef}(v), v \in V$, which is the overall timespan of finishing a given workload.
- Define the time taken to move a task v from one node to another that runs task w (a parent of task v) is $t_{mvtask}(v,w)$, and $t_{mvtask}(v,w) = size(v)/B(v,w)$, in which, $size(v)$ is the size of the task specifications.

We summarize the mathematical notations of the system in Table 2. Then, we devise the following recursive formulas to compute $t_{ef}(v)$:

$$t_{ef}(v) = \min\left(t_{ef}{}'(v), t_{ef}{}''(v)\right) + t_{exec}(v)$$

$$t_{ef}{}'(v) = \max_{u \in P(v)}\left(t_{ef}(u) + t_{mvdata}(u,v)\right) + t_{qwait}(v)$$

$$t_{ef}{}''(v) = \min_{u \in P(v)}\left(\max_{w \in P(v)}\left(t_{ef}(w) + t_{mvdata}(w,u)\right) + t_{mvtask}(v,w)\right) + t_{qwait}(v)$$

In the formulas, $t_{ef}{}'(v)$ is for **case 1**, and $t_{ef}{}''(v)$ is for **case 2**. Given an application workload that is represented as $G = (V,E)$ and the cost functions, as $t_{exec}(v), d(v), B(u,v), size(v), u \in V, v \in V, uv \in E$ are given, and $t_{qwait}(v)$ of each task is computed through equation (1), we could use dynamic programming to calculate $t_{ef}(v)$ for all the tasks starting with the tasks that have no parents. The biggest $t_{ef}(v)$ is the earliest time to finish the whole workload.

Table 2: Mathematical notations of the system

Notations	Descriptions
$\boldsymbol{G = (V,E)}$	A workload DAG, V represents the tasks, and E represents the data dependencies
$\boldsymbol{u, v, w}$	Arbitrary task, $u, v, w \in V$
$\boldsymbol{t_{exec}(v)}$	The execution time of task v
$\boldsymbol{d(v)}$	The data output size of task v
$\boldsymbol{size(v)}$	The size of the specification of task v in bytes
$\boldsymbol{t_{qwait}(v)}$	The waiting time of task v in the ready queue before execution
$\boldsymbol{k}$	The average number of task per core
$\boldsymbol{l}$	The average task execution time
$\boldsymbol{t_{mvdata}(v,w)}$	The time taken to move $d(v)$ size of data generated by task v to task w
$\boldsymbol{t_{mvtask}(v,w)}$	The time taken to move task v to the node that runs task w
$\boldsymbol{B(v,w)}$	The network bandwidth between two nodes that run task v and task w, respectively
$\boldsymbol{P(v)}$	The parents of task v
$\boldsymbol{t_{ef}(v)}$	The earliest finishing time of task v

Assuming that the number of tasks is n, and every task has p parents on average, the time complexity of the algorithm is $\Theta(np^2)$. The p^2 comes from the computation of $t_{ef}{}''(v)$, during which for all the p possible locations of running the task, we need to wait until the data of the last finished parent arrives (the other p). In reality, p is always much smaller than n. The memory complexity is $\Theta(n)$, as we need to memorize the values of $t_{ef}(v)$ and $d(v)$ of all the tasks in tables for looking up.

This analysis gives a sub-optimal lower bound of the overall timespan (performance upper bound) of executing a workload DAG. We call it sub-optimal, because for a task, the solution above just considers one step backwards (parents).

This doesn't consider the situation of scheduling a task to where the grand-parents or grand-grand-parents were scheduled, which will eliminate unnecessary data movements. However, finding an optimal solution is an NP-hard problem.

We will show how close our DAWS technique can achieve in performance comparing to the theoretical sub-optimal upper bound in the evaluation section.

4. EVALUATION

In this section, we present the performance evaluation of the DAWS technique. We include and shorten a part of the evaluation results from the previous work [10], including comparing MATRIX with the Falkon [32] centralized data-aware scheduler using two scientific applications; comparing different scheduling polices; and running MATRIX with benchmarking workloads. We add many more new evaluation results that enable us better understanding of the efficiency and performance of the DAWS technique, including some new visualization results of running MATRIX with the two applications; theoretical analysis results up to extreme scales; and comparison results of MATRIX with Sparrow [23] and CloudKon [24].

MATRIX was run on the Kodiak cluster from the Parallel Reconfigurable Observational Environment (PROBE) [33] of Los Alamos National Laboratory. Kodiak has 500 nodes and each node has two AMD Opteron (tm) processers 252 (2.6GHZ) and 8GB memory. We run experiments up to 100 nodes (200 cores), similar to the Falkon data-diffusion work [34, 35] we compared with.

The performance metrics used in our evaluations include *Average Time per stack per CPU*, *Efficiency*, *Utilization*, and *Throughput*, which are defined as follows:

Average Time Per Task Per CPU defines the average time of executing a task from one CPU's perspective. Ideally, each CPU would process tasks sequentially without waiting for ready 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. For example, assuming it takes 10sec to run 2000 tasks on 100 cores, this means 2 tasks-per-sec-per-cpu (2000/10/100) on average. The "Average Time Per Task Per CPU" is then 1/2=0.5sec. This metric is from the Falkon Data-Diffusion paper [30] for fair comparison.

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

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.

Throughput measures how fast the scheduling framework is able to execute tasks. It is calculated as the average number of tasks finished during a time period (e.g. sec, min, hours, and day). The *throughput* metric is useful for evaluation of the performance of short-duration (e.g. sub-second) tasks.

4.1. Evaluations of Scientific Applications

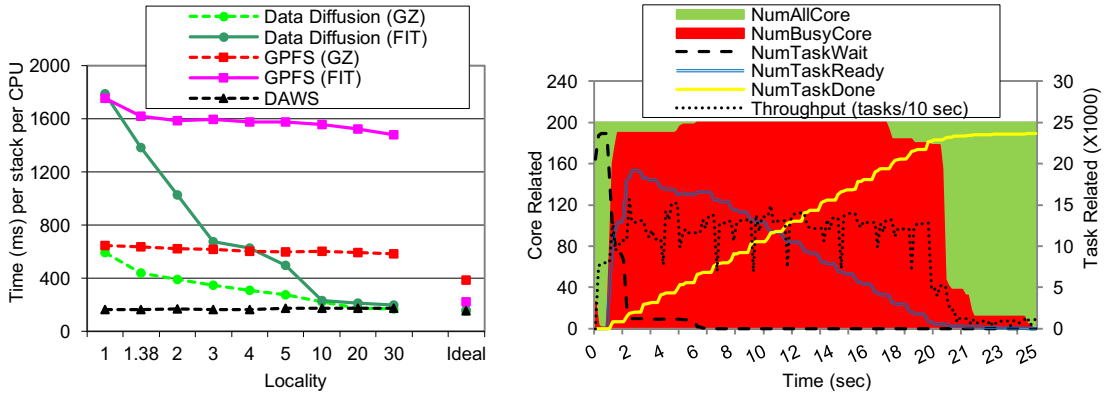
We compare MATRIX with the Falkon centralized scheduler using two scientific applications: image stacking in astronomy [34] and all-pairs in biometrics [35].

4.1.1. 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, and then co-adding many images to obtain a detectable signal that can measure their average brightness/shape.

The workload includes many parallel tasks with each one fetching an individual ROI object in a set of image files, and an aggregation task that collects all the outputs to obtain a detectable signal. In our experiments, each task required a 2MB file (after compression), executed 158ms and generated 10KB output data. The **locality number** referred to the ratio of the number of tasks to the number of files. The numbers of tasks and files of each locality number were given in [34].

We ran experiments up to 200 cores for all locality values. The *MDL* policy was applied, as 2MB of data was large. We compared the DAWS technique implemented in MATRIX with Falkon data diffusion at 128 cores (the largest scale the data diffusion ran). The results are shown in Figure 5 (a). "GZ" meant the files were compressed while "FIT" indicated the files were uncompressed. In the case of GPFS, each task read its required data from the remote GPFS parallel file system. Data diffusion first read all the data from GPFS, and then cached the data in the memory for centralized data-aware scheduling of all the tasks. MATRIX read all the data from the GPFS parallel file system, and then randomly distributed the data files to the memory of all the compute nodes. The MATRIX client submitted tasks to all the schedulers in the best case scenario. The schedulers applied the DAWS for distributed data-aware scheduling of all the tasks. In all cases, the time taken to finish the workload was clocked before the data was copied into the system, continued while the tasks were loaded into the schedulers, and then kept going until all tasks were finished.



(a) Comparison between DAWS with Data Diffusion (b) Utilization graph for locality 30

Figure 5: Evaluations using the Image Stacking application

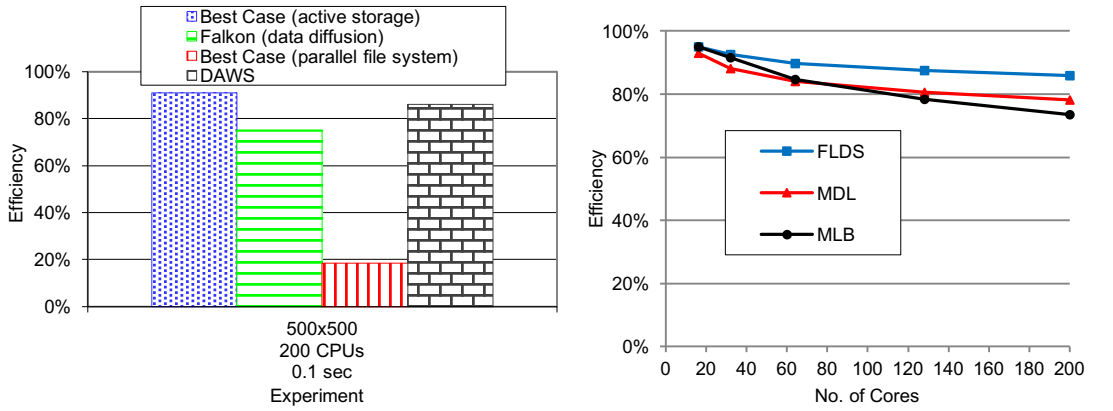
We see that at 128-core scale, the average task running time of our DAWS technique kept almost constant as locality increases, and was close to the ideal task running time (158ms). This is because the files were uniformly distributed over all compute nodes. The only overhead came from the schedulers making decisions to migrate tasks in the right spots. When locality was small, the data diffusion (GZ) experienced large average time, which decreased to be close to the ideal time when locality increased to 30, because data was initially kept in a slower parallel file system that needed to be copied to local disks. When locality was low, more amount of data accesses from the file system was required. The average times of GPFS (GZ) and GPFS (FIT) almost remained at a high constant value regardless of locality, due to the fact that data was copied from the remote GPFS upon every data access.

We show a visualization graph of MATRIX at 200 cores for locality 30 in Figure 5 (b). The utilization is calculated as the ratio of the area of the red region to that of the green region. We see there is a quick ramp-up period, after which the load is balanced across all the compute nodes. There is also a relatively long tail ramp-down period in the end when there are few tasks remaining, which is because with the *MDL* policy, no work stealing happened leading to poor load balancing in the end. But MATRIX can still achieve 75% utilization at 200 cores with fine-grained tasks (158ms).

4.1.2. All-Pairs in Biometrics

All-Pairs [35] describes a category of data-intensive applications that conduct a new function on two sets, A and B. For example, in Biometrics, it is important to find the covariance of two gene sequences by comparing piece with piece of the gene codes.

This workload included all independent tasks and every task executed for 100 ms to compare two 12MB files with one from each set. There were 500 files in each set, and we ran strong-scaling experiments up to 200 cores with a total of 500*500=250K tasks. As a task needed two files that may locate at different nodes at the worst case, one file may need to be transmitted. We used the *FLDS* policy. In the end (80% of the workload is done), we set *tt*=20 sec, which was doubled when successfully moving ready tasks from the *LReadyQ* to the *SReadyQ*.



(a) Comparison between DAWS with Data Diffusion (b) Comparison among different scheduling policies

Figure 6: Evaluations using the all-pairs application

We compared MATRIX DAWS technique with Falkon data diffusion [35] at 200 cores, and Figure 6 (a) shows the results. The "active storage" term [36] meant all the files were stored in memory. For 100-ms tasks, our DAWS technique improved data diffusion by 10.9% (85.9% vs 75%), and was close to the best case using active storage (85.9% vs 91%). This is because data diffusion applied a centralized index-server for data-aware scheduling, while our DAWS technique utilized distributed KVS, which was much more scalable. It was also worthy to point out that without harnessing data-locality (Best Case parallel file system), the efficiency was less than 20%, because all the files needed to be transmitted from the remote file system.

Although it is quite obvious that caching the data in the receiver's memory will usually be helpful to applications that have the *write-once/ready-many* pattern, we show the effect of caching of the *FLDS* policy for the all-pairs application in Figure 7. We see that without caching, the *FLDS* policy is only able to achieve less than 50% efficiency at 200-core scales. This is because all the files are uniformly distributed and each task requires two files, therefore, from the probability's perspective, about half of the tasks need to move data. With caching turned on, we record the cache-hit rate, which shows as high as above 80%. This helps significantly and contributes to

85%+ efficiency. However, we shouldn't conclude to always caching everything, because memory size is limited. Besides, depending on the access patterns, the cached data may never be reused. In the future, we will explore cache eviction mechanisms.

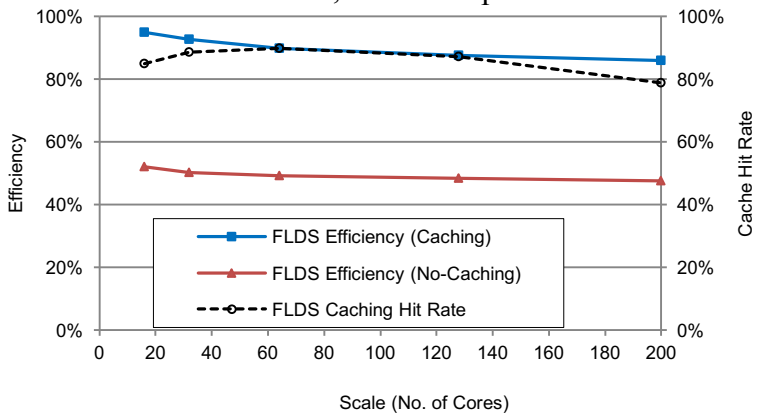


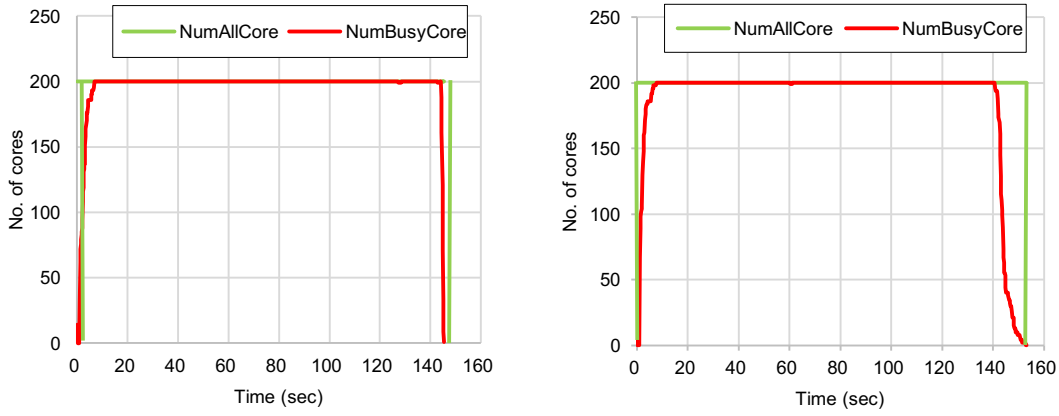
Figure 7: Comparison between Caching and No-Caching

4.1.3. Comparisons of Different Scheduling Policies

We compared three scheduling polices using the all-pairs workloads, *MLB*, *MDL*, and *FLDS*, up to 200 cores with the results shown in Figure 6 (b).

As we expected, the *MLB* policy performed the worst, because it considered only load balancing and the required data was so large that transmitting it took significant amount of time. The *MDL* policy performed moderately. From the load balancing's perspective, *MDL* did quite well except for the ending period. Because it did not allow work stealing and loads might be imbalanced at the final stage leading to a long-tail problem. The *FLDS* policy was the best, because it allowed the tasks being moved from the *LReadyQ* to the *SReadyQ* as needed. This was helpful at the final stage when many nodes were idle while a few others were busy.

To justify our explanation, we show the utilization figures of the *FLDS* and *MDL* policies in Figure 8. The utilization is the percentage of the area of red line to that of the green line. We see that both utilizations are quite high. The *MDL* policy has some long tail end where the utilization drops, while the FLDS policy does not exhibit this. Both policies have an initial ramp-up stage, during which one file (12MB) required by a task may be transferred. The transferred files are cached locally for future use. After that, because the number of files is relatively small (1000 in total), each compute node is able to cache enough files that could satisfy most of future tasks locally.



(a) Utilization of *FLDS* policy

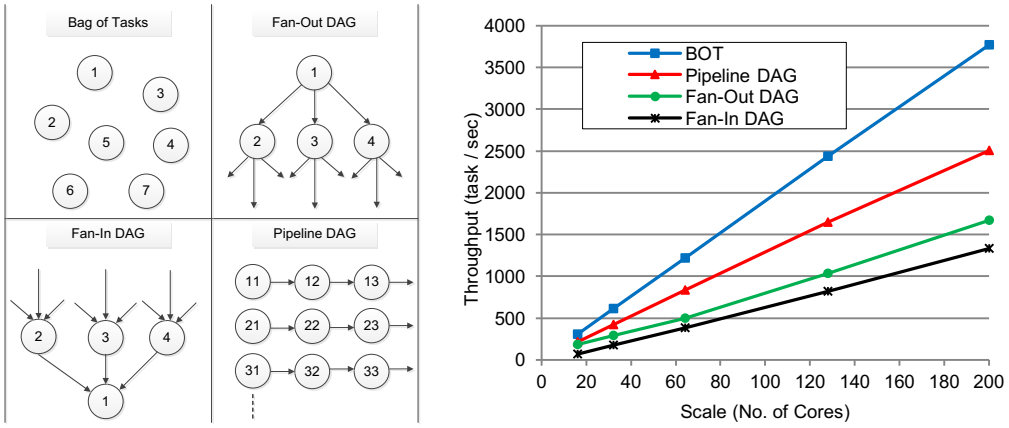
(b) Utilization of *MDL* policy

Figure 8: Utilization graph of the FLDS and MDL policies at 200 cores

The conclusions are: *for applications in which each task requires large amount of data (e.g. several Megabytes), the FLDS policy should always be the first choice; Unless the tasks require extremely large data that can easily saturate the networking, the MDL policy should not be considered; The MLB policy should only be used when tasks require small data pieces; The RLDS policy is preferable when the required data pieces have a wide distribution of size (from few bytes to several Megabytes).* Besides, MATRIX is able to change policies at runtime as explained later.

4.2. Evaluations of Benchmarking Workload DAGs

This section aimed to evaluate MATRIX with the DAWS technique using more complex benchmarking workload DAGs, namely Bag of Task (BOT), Fan-In, Fan-Out and Pipeline, which are represented in Figure 9 (a). BOT included independent tasks without data dependencies and was used as a baseline; Fan-In and Fan-Out were similar but with reverse tree-based shapes, and the parameters were the in-degree of Fan-In and the out-degree of Fan-Out; Pipeline was a collection of "pipes" and in each pipe, a task was dependent on the previous one. The parameter was the pipe size, referring to the number of tasks in a pipe.



(a) Benchmarking workload DAGs (b) Evaluation results

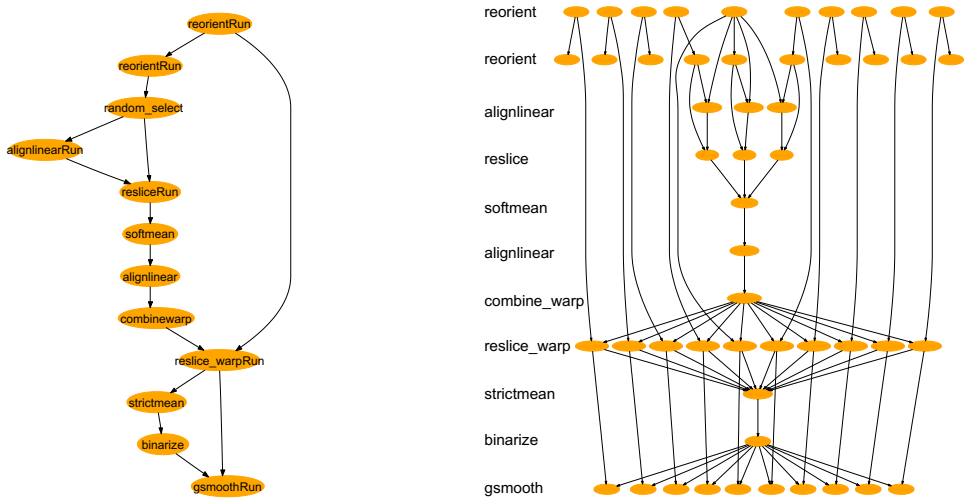
Figure 9: Evaluations using benchmarking workload DAGs

4.2.1. Representative Applications of the Benchmarking Workload DAGs

After studying the workload patterns of a category of MTC data-intensive applications using the Swift workflow system [44], we summarize the four benchmarking workload DAGs that are representative and cater to the data-flow patterns of different applications. Some applications display a single type of workload DAG pattern, while others have a combination of several workload DAG types, out of the four DAGs.

For example, the All-Pairs application in Biometrics is an example of the BOT workload DAGs, in which, the tasks are independent and every task requires two input files with each one coming from individual set. The Image Stacking application in Astronomy has a two-layer Fan-In DAG data-flow pattern. The top layer includes many parallel tasks with each one fetching an individual ROI object in a set of image files, and the bottom layer has an aggregation task that collects all the outputs to obtain a detectable signal. The workload DAGs of both applications were shown in [10]. The molecular dynamics (MolDyn) application in Chemistry domain aims to optimize and automate the computational workflow that can be used to generate the necessary parameters and other input files for calculating the solvation free energy of

ligands, and can also be extended to protein-ligand binding energy. Solvation free energy is an important quantity in Computational Chemistry with a variety of applications, especially in drug discovery and design. The MolDyn application is an 8-stage workflow. At each stage, the workload data flow pattern is either a Fan-Out or Fan-In DAG. The workload DAG was shown in [46]. The functional magnetic resonance imaging (fMRI) application in the medical imaging domain is a functional neuroimaging procedure that uses MRI technology to measure the brain activities by detecting changes in blood flow through the blood-oxygen-level dependent (BOLD) contrast [47]. In Swift, an fMRI study is physically represented in a nested directory structure, with metadata coded in directory and file names, and a volume is represented by two files located in the same directory, distinguished only by file name suffix [48]. The workload to process each data volume has a 2-pipe pipeline pattern, and each pipe consumes one file. The dominant pipeline consists of 12 sequential tasks. Figure 10 shows the workload DAGs of both one volume and ten volumes.



(a) Workflow for one volume

(b) Expanded (10 volume) workflow

Figure 10: Workload DAGs of the fMRI applications

4.2.2. Evaluation Results

MATRIX client can generate a specific workload DAG, given the input parameters such as DAG type (BOT, Fan-In, Fan-Out, and Pipeline), DAG parameter (in-degree, out-degree, and pipe size). Table 3 gives the experiment setups, which were the same for all the DAGs: the executor had 2 executing threads with each one executing 1000 tasks on average (a weak-scaling configuration); the tasks had an average running time of 50ms (0 to 100ms) and outputted an average data size of 5MB (0 to 10MB), both were generated with uniform random distribution; we set the threshold t to 0.5, a ratio of 0.5 of the data-transferring time to the task running time, the time threshold tt of the *FLDS* policy to 10 sec, and the polling interval upper bound to 50 sec; the DAG parameters (in-degree, out-degree and pipe size) were set to 10.

Table 3: Experiment Setup

Workload Parameters			**DAG Parameters**			**FLDS Policy Parameters**		
No. task per core	*Task average length (ms)*	*Task average data output size (MB)*	*Fan-Out degree*	*Fan-In degree*	*Pipeline pipe size*	*T*	*tt (sec)*	*Polling interval upper bound (sec)*
1000	50 ([0, 100])	5 ([0, 10])	10	10	10	0.5	10	50

Figure 9 (b) shows the results of scheduling all the DAGs in MATRIX using the *FLDS* policy up to 200 executing threads. BOT achieved nearly optimal performance with the throughput numbers implying a 90%+ efficiency at all scales. This is because

tasks were run locally without requiring any data. The other three DAGs showed great scalability, as the throughput was increasing linearly with the scale. Comparing the three DAGs with data dependencies, Pipeline showed the highest throughput, because each task needed at most one data from the parent. Fan-Out experienced a long ramp-up period, as at the beginning, only the root task was ready to run. As time increased, more tasks were ready resulting in better utilization. Fan-In DAG was the opposite. At the beginning, tasks were run fast, but it got slower and slower due to the lack of tasks, leading to a long tail that had worse effect than the slow ramp-up period of Fan-Out.

MATRIX showed great scalability running the benchmarking workload DAGs. In addition, MATRIX is able to run any arbitrary DAG, in addition to the four examples.

4.2.3. Explore Scalability through Theoretical Sub-optimal Solution

We have shown that the DAWS technique has great performance for the MTC data-intensive workloads at moderate scales. The questions to ask are what the quality of the results achieved with the DAWS technique is, and how scalable the technique is. We measure the quality by comparing the experimental results with those achieved through the analytical sub-optimal solution that we have devised in section 3. The task running time $t_{exec}(v)$, waiting time $t_{qwait}(v)$, task size $size(v)$, and the size of the output data $d(v)$ for each individual task are set the same as the workloads used in MATRIX. We also scale the sub-optimal solution up to 128K cores, showing the potential scalability of the DAWS at extreme scales.

We apply dynamic programming to calculate the earliest time ($t_{ef}(v)$) to finish a task. The earliest finishing time of the last task is the overall timespan to finish all the tasks, and we compute the throughput based on this. We first compare the experimental results of MATRIX with those of the analytical sub-optimal solution for the four benchmarking workloads up to 200 cores, shown in Figure 11.

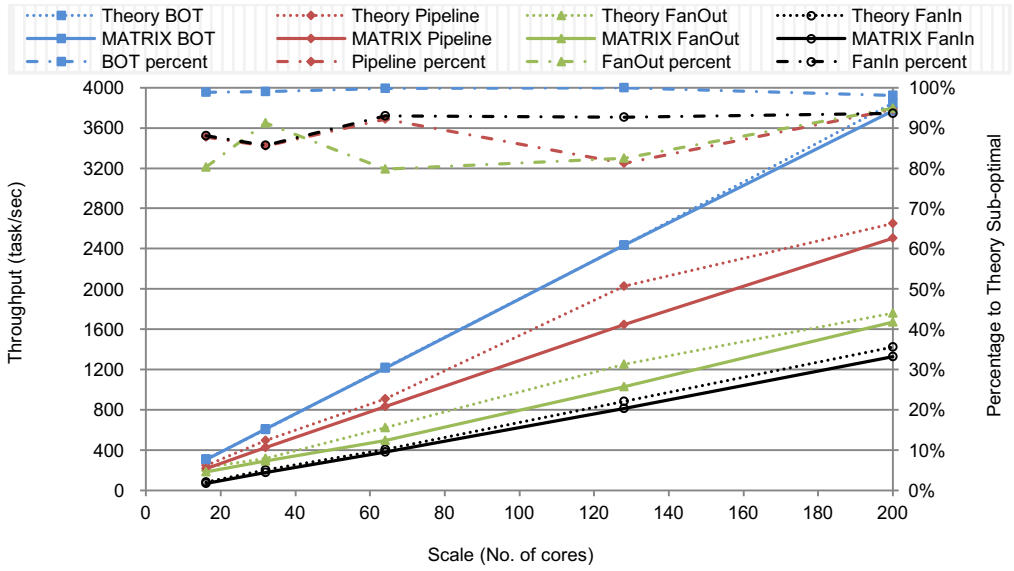


Figure 11: Comparison between MATRIX and the analytical sub-optimal solution

The solid lines are the results of MATRIX, the round dotted lines represent the results achieved from the sub-optimal solution (Theory), and the dash dotted lines show the ratio of the experimental results to those of sub-optimal solutions. The results of BOT, Pipeline, Fan-Out and Fan-In are interpreted with the colors of blue, red, green, and black, respectively. We see that for all the workloads at different scales, our DAWS technique could achieve performance within 15% compared with

the analytical sub-optimal solution. This relatively small percentage of performance loss indicates that the DAWS technique has the ability to achieve performance with high quality (85%+).

In order to highlight the numbers, we list the average percentage of performance that can be achieved through the DAWS technique for all the workloads in Table 4. We see that for the BOT workload, our technique works almost as good as the sub-optimal solution with a quality percentage of 99%, due to no data movement. For the other three workloads, the DAWS technique achieves about 85.7% of the sub-optimal for Fan-Out, 88.2% of the sub-optimal for Pipeline, and 90.6% of the sub-optimal for Fan-In, on average. The reason that the Pipeline and Fan-Out DAGs show bigger performance loss when comparing with the Fan-In DAG is because the former two DAGs have worse load balancing. For Fan-Out, every task (except for the leaf tasks) produces data of 5MB on average for 10 children, which may end up be run on the same node as their parent for minimizing the data movement overhead, leading to poor load balancing. Our *FLDS* policy mitigates this issue significantly. The same situation happens for the Pipeline DAG, but is less severe, because every task has only one child. However, the DAWS technique, configured with the *FLDS* policy, is still able to achieve 85.7% and 88.2% of the sub-optimal solution for the Fan-Out and Pipeline DAG, respectively, which demonstrate the high-quality performance.

Table 4: Average percentage of achieved performance comparing to the sub-optimal solution

Workloads	BOT	Pipeline	FanOut	FanIn
Percentage of sub-optimal	99.139636%	88.238118%	85.690323%	90.605218%

To explore the scalability of the fully distributed scheduling architecture and the DAWS technique, we compute the throughputs of the four benchmarking workloads with the theoretical solution, up to 128K cores. The configurations of all the workloads are as the same as shown in Table 3. The results are illustrated in Figure 12.

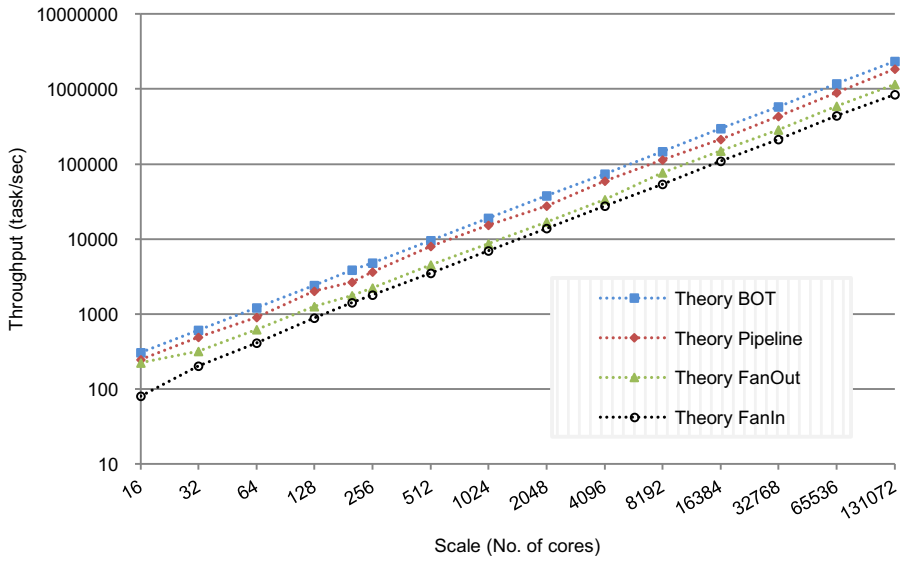


Figure 12: Computing with the theoretical sub-optimal solution up to 128K-core scale

For the four benchmarking workloads, the theoretical solution shows the same relative throughput results as MATRIX: BOT performs the best, and the second best is Pipeline, followed by Fan-Out, with Fan-In performing the worst. These results further justify our explanations of the MATRIX results in section 4.2. The throughput trends with respect to the scale indicate that the theoretical solution has great

scalability for all the workloads. At 128K-core (128M tasks) scale, the solution achieves extremely high throughput of 2.33M tasks/sec for BOT, 1.85M tasks/sec for Pipeline, 1.16M tasks/sec for Fan-Out, and 852.15K tasks/sec for Fan-In, respectively. These throughput numbers satisfy the scheduling needs of MTC data-intensive applications towards extreme scales. The trend is likely to hold towards million-core scales, and we will validate in the future. Since we have shown that the DAWS technique achieves performance within only 15% of the theoretical sub-optimal solution, we believe that the technique has the potential of the same scalability towards extreme scales.

The conclusion we can draw from this section is that the DAWS technique is not only able to achieve high quality performance results that are close to the bounded sub-optimal results within at most 15% performance loss, but has the potential to scale up to extreme scales for the data-intensive MTC workloads.

4.3. MATRIX vs. Sparrow and CloudKon

Our last sets of experiments are comparing MATRIX with the other state-of-the-art fine grained task scheduling systems that are targeting loosely-coupled parallel applications. Examples are Sparrow [23] and CloudKon [24]. We show the preliminary results of comparing MATRIX with both of them using BOT workloads. These comparisons aim at giving hints of how better MATRIX can achieve than others, and showing the potential broader impact of our technique on the Cloud data centers. We will compare using more complex workload DAGs in the future.

Sparrow [23] is a distributed scheduling system that employs multiple schedulers pushing tasks to workers (run on compute nodes). Each scheduler has a global view of all the workers. When dispatching tasks, a scheduler probes multiple workers (based on the number of tasks) and pushes tasks to the least overloaded ones. Once the tasks are scheduled to a worker, they cannot be migrated. CloudKon [24] is another scheduling system specific to the cloud environment. CloudKon has the same architecture as MATRIX, except that it leverages the Amazon SQS [37] to achieve distributed load balancing and DynamoDB [38] for task metadata management.

We compare MATRIX with Sparrow and CloudKon in the Amazon cloud up to 64 “m1.medium” instances. The instance has 1 virtual CPU, 2 compute units, 3.7GB memory, and 410GB hard disk. Since each instance has 2 compute units, we set the number of executing threads of the executor to 2 for all of the systems. At present, we compare the raw speed of executing tasks of the three scheduling systems, which is measured as the throughput of executing the “sleep 0” NOOP tasks. We conduct weak-scaling experiments, and in our workloads, each instance runs 16K “sleep 0” tasks on average. The results are shown in Figure 13. Sparrow handles the scheduler failures in the same way as MATRIX. Note that Sparrow does not persist scheduling state to disk, therefore, it does not handle worker failures. On the other hand, both CloudKon and MATRIX have persistent storage of all the metadata, as both SQS and ZHT implemented persistent storage. In our experiments, we turn off the persistent storage of both CloudKon and MATRIX to ensure a fair comparison with Sparrow.

From Figure 13, we see that all of the three scheduling systems can achieve increased throughput trend with respect to the system scale. However, MATRIX is able to achieve much higher throughput than CloudKon and Sparrow at all scales. At 64 instances, MATRIX shows throughput speedup of more than 5X (67K vs 13K) comparing with CloudKon, and speedup of more than 9X comparing with Sparrow (67K vs 7.3K). Comparing with MATRIX, CloudKon has a similar scheduling architecture (fully distributed). However, the workers of CloudKon need to pull every

task from SQS leading to significant overheads for NOOP tasks, while MATRIX migrates tasks in batches through the work stealing technique that introduces much less communication overheads. Besides, CloudKon is implemented in Java that introduces JVM overhead, while MATRIX is implemented in C++, which contributes a portion to the 5X speedup. To eliminate the effects of different programming languages, we compare CloudKon and Sparrow, both were implemented in Java, but they have different scheduling architectures and load balancing techniques. CloudKon outperforms Sparrow by 1.78X (13K vs 7.3K) at 64-instances scale, because the schedulers of Sparrow need to send probing messages to push tasks and once tasks are submitted, they cannot be migrated, leading to poor load balancing, while CloudKon relies on SQS to achieve distributed load balancing.

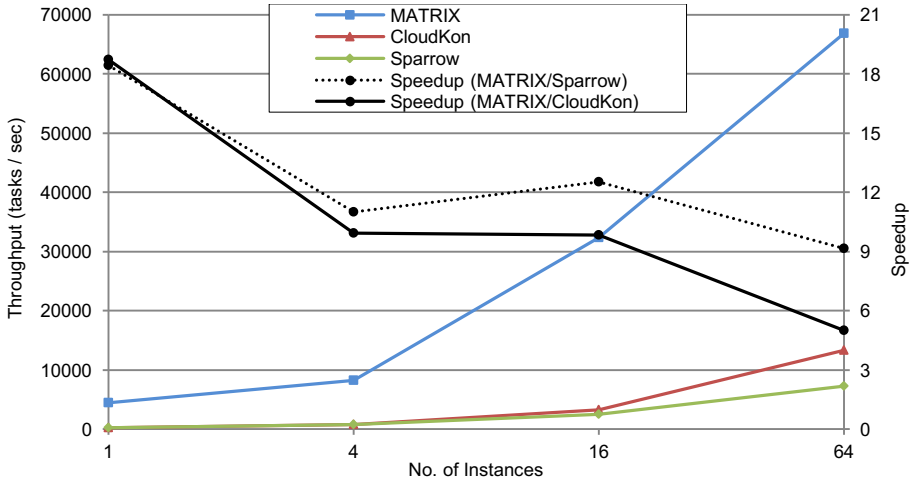


Figure 13: Comparing MATRIX with Sparrow and CloudKon

The conclusions are that MATRIX can execute loosely coupled light weight tasks much faster than the other two scheduling systems that target fine-grained workloads in the Cloud environment. MATRIX, along with the DAWS scheduling technique, has the potential to benefit the scheduling of the large-scale data processing workloads (e.g. Hadoop workloads), which we will investigate in the future.

4.4. Analysis of the DAWS Technique

We have shown that the DAWS technique is not only scalable, but is able to achieve high quality performance within 15% of the sub-optimal solution on average for different benchmarking workload DAGs. We justify that the technique is applicable to a general class of MTC data-intensive workloads. This is due to the adaptive property of the DAWS technique. The adaptive property refers to the ability of adjusting the parameters for the best performance during the runtime according to system states. We have enunciated how the adaptive work stealing works, how to choose the best scheduling policies, individually. These should be considered together, along with the other parameters, such as the ratio threshold (t), and the upper bound of the execution time of the *LReadyQ* (tt) in the *FLDS* policy. The DAWS technique could always start with the *FLDS* policy. Based on the data volumes transferred during runtime, it is able to switch to other policies. If too much data is transferred, the technique would switch to the *MDL* policy; on the other hand, the technique would switch to the *MLB* policy if few data is transmitted. In addition, we can set the initial tt value in the *FLDS* policy, double the value when moving ready tasks from the *LReadyQ* to the *SReadyQ*, and reduce the value by half when work stealing fails. In order to cooperate with the

FLDS policy, after the polling interval of work stealing hits the upper bound (no work stealing anymore), we set the polling interval back to the initial small value only if the threshold *tt* becomes small enough. This would allow to do work stealing again.

However, things can become way more complicated when running large-scale data-intensive applications that have very complex DAGs. There is still possibility that even with the adaptive properties, the DAWS technique may not perform well. The difficulties attribute to the constraint local views of each scheduler for the system states. In the future, we will explore some monitoring mechanisms to further improve the DAWS technique.

5. RELATED WORK

This section presents the work that is related to ours by identifying the similarities and differences comparing with our work, including the related work of load balancing, work stealing, and data-aware scheduling.

Load balancing is an important design goal of most large-scale parallel programming and runtime systems for the purpose of optimizing resource utilization, data movement, power consumption, or any combination of these. There are two broad categories of load balancing strategies based on applications – dynamic load balancing for those applications that create and schedule new tasks during runtime, and static load balancing for iterative applications that have persistent load patterns [49]. Besides, load balancing could be achieved in a centralized, or distributed way. Centralized load balancing has been studied extensively, such as JSQ [50], least-work-left [51], and SITA [52]. However, they all suffered from poor scalability and resilience. Distributed Load balancing employs multiple schedulers to spread out computational and communication loads evenly across processors of a shared-memory parallel machine, or across nodes of a distributed system (e.g. clusters, supercomputers, grids, and clouds), so that no single processor or node is overloaded. Clients are able to submit workload to any scheduler, and each scheduler has the choice of executing the tasks locally, or forwarding the tasks to another scheduler based on some function it is optimizing. Although distributed load balancing is likely a more scalable and resilient solution towards extreme scales, there are many challenges that must be addressed (e.g. utilization, partitioning). Fully distributed strategies have been proposed, including neighborhood averaging scheme (ACWN) [53, 56, 57]. In [57], several distributed load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model (GM) and a Hierarchical Balancing Method (HBM). Other hierarchical strategies are explored in [56] and [58].

Work stealing is an efficient distributed load balancing technique that has been used at small scales successfully in parallel languages such as Cilk [59], X10 [77], Intel TBB [78] and OpenMP, to balance workloads across threads on shared memory parallel machines [71, 72]. Theoretical work has proved that a work stealing scheduler can achieve execution space, time, and communication bounds all within a constant factor of optimal [72]. But the scalability of work stealing has not been well explored on modern large-scale systems. In particular, concerns exist that the randomized work stealing can lead to long idle times and poor scalability on large-scale clusters [71]. The work done by Diana et. al in [71] scaled work stealing to 8K processors using the PGAS programming model and the RDMA technique. A hierarchical technique that improved Diana's work described work stealing as retentive work stealing. This technique scaled work stealing to over 150K cores by utilizing the persistence

principle iteratively to achieve the load balancing of task-based applications [73]. However, these techniques considered only load balancing, not data-locality. On the contrary, our work optimized both work stealing and data-locality. SLAW [40] is a scalable task scheduler that applied the adaptive locality-aware work stealing technique and supported both work-first and help-first policies [41] at runtime on a per-task basis. The other work about data-aware work stealing technique improved data locality across different phases of fork/join programs [74]. This work relied on constructing a sample pre-schedule of work stealing tree, and the workload execution followed the pre-schedule. This involved overheads of creating the sample and was not suitable for irregular applications. Furthermore, both this work and SLAW focused on the single shared-memory environment. Another work [42] of data-aware work stealing is similar to ours in using both the dedicated and shared queues. However, before scheduling the tasks, it relied on the X10 programming model [43] to statically expose and to categorize the data-locality information. This is not adaptive to various data-intensive workloads.

Charm++ [49] supports centralized, hierarchical and distributed load balancing. It has demonstrated that centralized strategies work at scales of thousands of processors for NAMD. In [49], the authors present an automatic dynamic hierarchical load balancing method for Charm++, which scales up to 16K-cores on a Sun Constellation supercomputer for a synthetic benchmark. This paper [75] describes a fully distributed algorithm for load balancing that uses partial information about the global state of the system to perform load balancing. This algorithm, referred to as GrapevineLB, first conducts global information propagation using a lightweight algorithm inspired by epidemic [76] algorithms, and then transfers work units using a randomized algorithm. It has scaled the GrapevineLB algorithm up to 131,072 cores of Blue Gene/Q supercomputer in the Charm++ framework. However, this algorithm doesn't work well for irregular applications that require dynamic load balancing techniques; it neither considered data-aware scheduling.

Falkon [32] is a centralized task scheduling system that implemented a data diffusion approach [34,35] in the scheduling of MTC data-intensive workloads. Data diffusion applied a centralized index server to store the metadata, as opposed to our distributed KVS, leading to poor scalability at large scales.

Sparrow [23] implemented distributed load balancing to achieve weighted fair sharing of the resources, and supported the data-aware scheduling to co-locate each task with its input data, for fine-grained sub-second tasks. However, the global knowledge of all the resources of each scheduler may lead to resource contentions when the task count is large. Furthermore, Sparrow implemented a pushing mechanism by early binding the tasks to workers, which may suffer long tail problem under heterogeneous workloads.

Dryad [39] is a distributed task execution engine for coarse grained data-parallel applications. Similar to our work, Dryad supported running applications structured as workflow DAGs. However, Dryad managed metadata in a centralized way that greedily mapped tasks to where the data resides, which is not scalable.

CloudKon [24] has the similar scheduling architecture as MATRIX. CloudKon focused on the Cloud environment by relying on the Cloud services to do distributed load balancing (applying SQS [37]) and to manage the task metadata (leveraging DynamoDB [38]). Although the Cloud services have the ability to facilitate the easier development, the side effects are the potential loss of the performance and control. Furthermore, CloudKon doesn't support data-aware scheduling at present.

6. CONCLUSIONS AND FUTURE WORK

This work aimed at exploring the scalability and performance quality of the previously proposed data-aware work stealing technique that optimized both of the load balancing and data-locality for MTC data-intensive applications. We devised an analytical sub-optimal performance upper bound of the technique, implemented and evaluated the technique in MATRIX, and compared MATRIX with other scheduling systems. We also explored the scalability through the theoretical solution at extreme scales. Results show that the technique is not only scalable, but is able to perform within 15% of the sub-optimal solution.

In the future, we will deploy MATRIX on the IBM BG/Q Mira supercomputer at ANL and strive to scale MATRIX to the full scale of the Mira machine that has 768K cores. We aim to improve MATRIX by applying the ZHT communication layer, which currently supports an inclusive communication protocols, namely, TCP, UDP, UDT, and MPI. Another direction is to integrate the Swift [44] workflow engine with MATRIX to enable MATRIX to run large-scale scientific applications. Instead of having Swift to manage and scheduling the scientific applications, Swift will decompose applications to workflow DAGs, which are then submitted to MATRIX for scheduling. Another future work will be extending MATRIX to support the scheduling of the Hadoop data-processing workloads [45, 60, 63, 69]. We will utilize distributed file systems, likely the FusionFS [28, 79, 80], to help MATRIX manage data in a transparent, scalable, and reliable way. MATRIX + FusionFS will be the combination of the next generation distributed MapReduce framework, as opposed to the Hadoop + HDFS combination of the current centralized MapReduce implementation.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy under the contract DE-FC02-06ER25750 and by the National Science Foundation (NSF) under grant OCI-1054974, and also in part by the National Science Foundation (NSF) under the award CNS-1042543 (PRObE, http://www.nmc-probe.org/).

REFERENCES

1. I. Raicu, I. Foster, Y. Zhao. “Many-Task Computing for Grids and Supercomputers”, Invited Paper, IEEE MTAGS 2008.
2. 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.
3. 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.
4. W. Gentzsch, et. al. “Sun Grid Engine: Towards Creating a Compute Power Grid,” 1st International Symposium on Cluster Computing and the Grid, 2001.
5. Cobalt: http://trac.mcs.anl.gov/projects/cobalt, 2014.
6. V. Sarkar, S. Amarasinghe, et al. “ExaScale Software Study: Software Challenges in Extreme Scale Systems”, ExaScale Computing Study, DARPA IPTO, 2009.
7. 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.
8. 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.

9. 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.
10. K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, I. Raicu. “Optimizing Load Balancing and Data-Locality with Data-aware Scheduling”, IEEE International Conference on Big Data 2014.
11. 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.
12. R. D. Blumofe and C. E. Leiserson. “Scheduling multithreaded computations by work stealing”, Symposium on Foundations of Computer Science 1994.
13. 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.
14. K. Wang, K. Brandstatter, I. Raicu. “SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascales,” ACM HPC 2013.
15. D. Zhao, D. Zhang, K. Wang, and I. Raicu, "Exploring Reliability of Exascale Systems through Simulations," in Proc. 21st High Performance Computing Symposium, Bahia Resort, San Diego, CA, 2013, pp. 1-9.
16. K. Wang, J. Munuera, I. Raicu, and H. Jin. (2011, Aug. 23). Centralized and Distributed Job Scheduling System Simulation at Exascale [online]. Available: http://datasys.cs.iit.edu/~kewang/documents/summer_report.pdf
17. J. D. Ullman. “NP-complete scheduling problems”, Journal of Computer and System Sciences, Volume 10 Issue 3, June, 1975, Pages 384-393.
18. 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.
19. T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu. “ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013.
20. T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang, I. Raicu, "A Convergence of Key-Value Storage Systems from Clouds to Supercomputers", Journal of Concurrency and Computation: Practice and Experience (CCPE), 2015.
21. T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, and I. Raicu, “Graph/z: A key-value store based scalable graph processing system,” poster session, IEEE Int. Conference on Cluster Computing, 2015.
22. K. Wang, A. Rajendran, I. Raicu. “MATRIX: MAny-Task computing execution fabRIc at eXascale,” tech report, IIT, 2013.
23. 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.
24. I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, I. Raicu. “Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing”, 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) 2014.
25. 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.
26. J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, J. Nieplocha. “Scalable work stealing”, IEEE/ACM SC 2009.
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. 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.
29. A. Bialecki, et al. “Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware”, http://lucene.apache.org/hadoop/, 2005.
30. Google. “Google Protocol Buffers,” available at http://code.google.com/apis/protocolbuffers/, 2014.
31. 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.
32. I. Raicu, Y. Zhao et al. “Falkon: A Fast and Light-weight tasK executiON Framework,” IEEE/ACM SC 2007.

33. Gibson, G., Grider, G., Jacobson, A. and Lloyd, W. "PRObE: A thousand-node experimental cluster for computer systems research." Usenix ;login: 38, 3 (2013).
34. I. Raicu, Y. Zhao, I. Foster, A. Szalay. "Accelerating Large-scale Data Exploration through Data Diffusion", International Workshop on Data-Aware Distributed Computing 2008, co-locate with ACM/IEEE HPDC 2008.
35. I. Raicu, I. Foster, Y. Zhao, P. Little, C. Moretti, A. Chaudhary, D. Thain. "The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems", ACM HPDC 2009.
36. C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain. 2010. All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids. IEEE Trans. Parallel Distrib. Syst. 21, 1.
37. Amazon Simple Queue Service. Avaible online: http://aws.amazon.com/documentation/sqs/. 2014.
38. G. DeCandia, D. Hastorun, M. Jampani, et al. "Dynamo: Amazon's highly available key-value store", ACM SIGOPS Symposium on Operating Systems Principles (SOSP) 2007.
39. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. "Dryad: Distributed data-parallel programs from sequential building blocks", In Proc. Eurosys, pp. 59–72, March 2007.
40. 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.
41. Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2009.
42. J. Paudel, O. Tardieu and J. Amaral. "On the merits of distributed work-stealing on selective locality-aware tasks", 42nd International Conference on Parallel Processing (ICPP), 2013.
43. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in ACM Conference on Object-oriented Programming Systems Languages and Applications(OOPSLA), 2005, pp. 519–538.
44. Y. Zhao, M. Hategan, B. Clifford, I. Foster et al. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," IEEE Workshop on Scientific Workflows 2007.
45. J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04, CA, December, 2004.
46. Y. Zhao, I. Raicu, et al. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments," book chapter in Grid Computing Research Progress, ISBN: 978-1-60456-404-4, Nova Publisher 2008.
47. Huettel, S. A.; Song, A. W.; McCarthy, G. (2009), Functional Magnetic Resonance Imaging (2 ed.), Massachusetts: Sinauer, ISBN 978-0-87893-286-3.
48. Horn, J.V., Dobson, J., Woodward, J., Wilde, M., Zhao, Y., Voeckler, J. and Foster, I. Grid-Based Computing and the Future of Neuroscience Computation. in Methods in Mind, MITPress, 2006.
49. G. Zhang, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW 10, pages 436-444, Washington, DC, USA, 2010. IEEE Computer Society.
50. H.C. Lin, C.S. Raghavendra. An approximate analysis of the join the shortest queue (JSQ) policy, IEEE Transaction on Parallel and Distributed Systems, Volume 7, Number 3, pages 301-307, 1996.
51. M. Harchol-Balter. Job placement with unknown duration and no preemption, ACM SIGMETRICS Performance Evaluation Review, Volume 28, Number 4, pages 3-5, 2001.
52. E. Bachmat, H. Sarfati. Analysis of size interval task assignment policies, ACM SIGMETRICS Performance Evaluation Review, Volume 36, Number 2, pages 107-109, 2008.
53. L. V. Kal´e. Comparing the performance of two dynamic load distribution methods. In Proceedings of the 1988 International Conference on Parallel Processing, pages 8–11, August 1988.
54. K. Wang, X. Zhou, H. Chen, M. Lang, and I. Raicu, "Next generation job management systems for extreme scale ensemble computing," in Proc. 23rd Int. Symp. High Perform. Distrib. Comput., 2014, pp. 111–114.
55. K. Wang, X. Zhou, K. Qiao, M. Lang, B. McClelland, and I. Raicu, "Towards Scalable Distributed Workload Manager with Monitoring-Based Weakly Consistent Resource Stealing," in Proc. of the 24th Int. symposium on High-performance parallel and distributed computing, 2015.
56. A. Sinha and L.V. Kal´e. A load balancing strategy for prioritized execution of tasks. In International Parallel Processing Symposium, pages 230–237, April 1993.

57. 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.
58. M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1990.
59. Matteo Frigo, Charles E. Leiserson and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. PLDI'98 Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, 212-223.
60. K. Wang, N. Liu, I. Sadooghi, X. Yang, X. Zhou, T. Li, M. Lang, Xian-He Sun, and I. Raicu, "Overcoming Hadoop Scaling Limitations through Distributed Task Execution," in Proc. of the IEEE Int. Conference on Cluster Computing, 2015.
61. K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Exploring the Design Tradeoffs for Extreme-Scale High-Performance Computing System Software," in Proc. of the IEEE Trans. on Parallel and Dist. Systems, vol. PP, no. 99, pp. 1, 2015.
62. T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, "A Dynamically Scalable Cloud Data Infrastructure for Sensor Networks," in Proc. of the 6th Workshop on Scientific Cloud Computing, 2015.
63. K. Wang, and I. Raicu, "Scheduling Data-intensive Many-task Computing Applications in the Cloud," in the NSFCloud Workshop on Experimental Support for Cloud Computing, 2014.
64. K. Wang and Ioan Raicu. (2014, Oct. 7). "Towards Next Generation Resource Management at Extreme-Scales," Ph.D. Comprehensive Exam, Computer Science Department, Illinois Institute of Technology [online]. Available: http://datasys.cs.iit.edu/publications/2014_IIT_PhD-proposal_Ke-Wang.pdf
65. A. Kulkarni, K. Wang, and M. Lang. (2012, Aug. 17). "Exploring the Design Tradeoffs for Exascale System Services through Simulation," Summer Student Research Workshop at Los Alamos National Laboratory [online]. Available: https://newmexicoconsortium.org/component/com_jresearch/Itemid,146/id,31/task,show/view,publication/
66. X. Zhou, H. Chen, et al. (2013, Dec. 15). Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System [online]. Available: http://datasys.cs.iit.edu/reports/2013_IIT-CS554_dist-slurm.pdf
67. K. Ramamurthy, K. Wang, and I. Raicu. (2013, Dec. 15). Exploring Distributed HPC Scheduling in MATRIX [online]. Available: http://www.cs.iit.edu/~iraicu/teaching/CS554-F13/best-reports/2013_IIT-CS554_MATRIX-HPC.pdf
68. K. Wang and I. Raicu. (2014, May 23). "Achieving Data-Aware Load Balancing through Distributed Queues and Key/Value Stores," in the 3rd Greater Chicago Area System Research Workshop [online]. Available: http://datasys.cs.iit.edu/reports/2014_GCASR14_paper-data-aware-scheduling.pdf
69. T. Forlini, T. Dubucq, et al. "Benchmarking State-of-the-art Many-Task Computing Runtime Systems", poster session, ACM HPDC, 2015.
70. K. Wang. "Scalable Resource Management System Software for Extreme-Scale Distributed Systems", Computer Science Department, Illinois Institute of Technology, Doctorate Dissertation, July 2015.
71. J. Dinan, D.B. Larkins, P. Sadayappan, S. Krishnamoorthy, J. Nieplocha. "Scalable work stealing", In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), 2009.
72. Vipin Kumar, Ananth Y. Grama and Nageshwara Rao Vempaty. 1994. Scalable load balancing techniques for parallel computers. Journal of Parallel and Distributed Computing, Volume 22 Issue 1, July 1994, 60-79. Academic Press, Inc. Orlando, FL, USA.
73. J. Liander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In HPDC, 2012.
74. Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2014. Optimizing data locality for fork/join programs using constrained work stealing. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14).
75. Harshitha Menon and Laxmikant Kalé. A distributed dynamic load balancer for iterative applications. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13), 2013.
76. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance.

In Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC '87), Fred B. Schneider (Ed.).

77. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40, 10 (October 2005), 519-538.
78. James Reinders. 2007. Intel Threading Building Blocks (First ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.
79. Dongfang Zhao, Ioan Raicu. “Distributed File Systems for Exascale Computing”, Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012
80. Dongfang Zhao, Ning Liu, Dries Kimpe, Robert Ross, Xian-He Sun, and Ioan Raicu. "Towards Exploring Data-Intensive Scientific Applications at Extreme Scales through Systems and Simulations", IEEE Transaction on Parallel and Distributed Systems (TPDS) Journal 2015

