Towards Next Generation Resource Management at Extreme-Scales

Ke Wang
Department of Computer Science
Illinois Institute of Technology
kwang22@hawk.iit.edu
Fall 2014

Committee Members:

Ioan Raicu: Research Advisor, Department of Computer Science, Illinois Institute of Technology & Math and Computer Science Division, Argonne National Laboratory, iraicu@cs.iit.edu

Xian-He Sun: Department of Computer Science, Illinois Institute of Technology, sun@cs.iit.edu

Dong Jin: Department of Computer Science, Illinois Institute of Technology, dong.jin@iit.edu

Yu Cheng: Department of Electrical and Computing Engineering, Illinois Institute of Technology, cheng@iit.edu

Michael Lang: Ultra-Scale System Research Center, Los Alamos National Laboratory, mlang@lanl.gov

Abstract

With the exponential growth of distributed systems in both FLOPS and parallelism (number of cores/threads), scientific applications are growing more diverse with various workloads. These workloads include traditional large-scale high performance computing (HPC) MPI jobs, and HPC ensemble workloads that support the investigation of parameter sweeps using many small-scale coordinated jobs, as well as the fine-grained loosely-coupled many-task computing (MTC) workloads. Delivering high throughput and low latency for all the workloads has driven us designing and implementing the next-generation resource management systems that are magnitudes more scalable and available than today's centralized batch-scheduled ones at extreme-scales for both MTC and HPC applications. We devised a generic taxonomy to explore the design choices for extreme-scale system services, and proposed that the key-value stores could serve as a viable building block for scalable distributed system services. We designed and developed fully-distributed architectures for next-generation resource management systems, which are radically different from the traditional centralized architecture; we also proposed the distributed data-aware work stealing and resource stealing techniques to achieve distributed workload and resource balancing at extreme-scales. We studied the distributed architectures with the proposed techniques through simulations (SimMatrix and SimSLURM++) up to exascale with millions of nodes, as well as through real systems (MATRIX & SLURM++) at thousands of nodes' scales. We are currently working on pushing both MATRX (a fully-distributed resource management system for MTC applications) and SLURM++ (a partition-based fully-distributed resource management system for HPC applications) to support the running of real scientific MTC and HPC workloads at large scales.

1. Introduction

Predictions are that around the end of this decade, distributed systems will reach exascale (10^{18} FLOPS) with millions of nodes and billions of threads of execution [1]. There are many domains (e.g. weather modeling, global warming, national security, energy, drug discovery, etc.) that will achieve revolutionary advancements due to exascale computing [2]. Exascale computing will enable the unraveling of significant mysteries for a diversity of scientific applications, ranging from Biology, Earth Systems, Physical Chemistry, Astronomy, to Neuroscience, Economics, and Social Learning and so on [3]. Running all these applications on an exascale computing system efficiently poses significant scheduling challenges on resource management systems, such as efficiency, scalability and reliability. From this point, the terms of resource management system, resource manager, job scheduling system, job scheduler, and job management system (JMS) are used interchangeably. Also, the terms of job and task would be used interchangeably. These challenges would unlikely to be addressed by the traditional centralized JMS. These requirements have driven us to design and implement next generation JMS that are orders of magnitudes more efficient, scalable and reliable for the ensemble of scientific applications at extreme-scales.

1.1 Diversity of Scientific Applications towards Exascale Computing

With the extreme magnitude of component count and concurrency, one way to efficiently use exascale machines without requiring full-scale jobs is to support the running of diverse categories of applications [4]. These applications would combine scientific workloads from different application domains, such as the traditional high performance computing (HPC) MPI jobs, and the HPC ensemble workloads, as well as the Many-Task Computing (MTC) workloads. Given the significant decrease of Mean-Time-To-Failure (MTTF) [5][90] at exascale levels, the running of all the diverse applications should be more resilient by definition given that failures will affect a smaller part of the machines.

Traditional HPC workloads are large-scale tightly-coupled applications that use message-passing interface (MPI) programming model [6] for communication and synchronization. Resiliency is achieved relying on checkpointing. As the system scales up, check-pointing will not be scalable due to the increasing overheads to do check-pointing, and there will be less jobs requiring full-size allocation.

Other HPC workloads are ensemble workflows that support the investigation of parameter sweeps using many small-scale coordinated jobs. These individual jobs are coordinated in a traditional HPC fashion by using MPI. There are applications that do parameter sweeps to define areas of interest for larger scale high-resolution runs. These ensemble runs are managed in an ad-hoc fashion without any scheduling optimization. Ensemble workload is one way to efficiently use a large-scale machine without requiring full-scale jobs and as an added benefit this workload is then more resilient to node failures. If an individual ensemble task fails, it is just rescheduled on different nodes. Another area where ensemble runs are beneficial is in scheduling regression tests. Many applications use regression tests to validate changes in the code base. These tests are small-scale, sometimes individual nodes and are scheduled on a daily basis. A similar workload includes the system health jobs, which are run when job queues have small idle allocations available or when allocations are not preferred due to lack of contiguous nodes [7].

The other extreme case workloads come from the Many-Task Computing (MTC) [3][8] paradigm. In MTC, applications are decomposed of several orders of magnitude larger number (e.g. billions) of fine-grained embarrassingly-parallel tasks with data-dependencies. Tasks are fine-grained in both size (e.g. per-core, per-node) and duration (from sub-second to hours), and are represented as Direct Acyclic Graph (DAG) where vertices are discrete tasks and an edge denotes the data flow from one task to another. The tasks do not require strict coordination of processes at job launch as the traditional HPC workloads do. A decade ago or earlier, it was recognized that applications composed of large number of tasks may be used as a driver for numerical experiments that may be combined into an aggregate method [9]. In particular, the algorithm paradigms well suited for MTC are Optimization, Data Analysis, Monte Carlo and Uncertainty Quantification. Various applications that demonstrate characteristics of MTC cover a wide range of domains, including astronomy, physics, astrophysics, pharmaceuticals, bioinformatics, biometrics, neuroscience, medical imaging, chemistry, climate modeling, and economics [3][10].

1.2 Requirements of Next Generation Job Management Systems

Job management system (JMS) is a core system middleware for distributed computing systems. JMS is responsible for managing the system resources and scheduling application workloads. The key performance requirements for JMS are efficiency, scalability and reliability. Efficiency means the JMS needs to allocate resources and to schedule jobs fast enough to maintain high system utilizations; Scalability refers to the increasing of processing capacity (measured by throughput) as the workload (number of tasks) and computing resources scale; and Reliability requires the JMS is still functioning well under failures.

The next generation JMS for exascale computing will need to be versatile, efficient, scalable and reliable enough, in order to deliver the extremely high throughput and low latency required to maintain high performance. The JMS we are targeting at are for tightly coupled large-scale HPC environments (e.g. Clusters and Supercomputers), but other distributed systems such as the Cloud domains could also benefit from this work. Most of state-of-the-art schedulers (e.g. SLURM [11], Condor [12], PBS [13], SGE [14], Falkon [15]) have centralized master/slaves architecture where a controller is handling all the scheduling and management activities, such as metadata management, resource provisioning, job scheduling, and job execution. This centralized architecture is not suited for the demands of exascale computing, due to both poor scalability and single-point-of-failure. The popular JMS, SLURM, reported maximum throughput of 500 jobs/sec [16]; however, we will need many orders of magnitude higher job delivering rates (e.g. millions/sec) due to the significant increase of scheduling size (10X higher node counts, 100X higher thread counts, and much higher job counts), along with the much finer granularity of job durations (from milliseconds/minutes to hours/days).

1.3 Contributions

To address the significant challenges of JMS, we are designing and implementing next-generation JMS for exascale computing. We devise a generic taxonomy to explore the design choices for extreme-scale system services, and propose that the key-value stores (KVS) [17][18][89][92] could serve as a viable building block for scalable distributed system services. We take fully-distributed architectures that are radically different from the centralized one for next-generation JMS for both MTC and HPC applications. We also propose distributed work stealing [19] and resource stealing techniques to achieve distributed workload and resource balancing. We then improve the work stealing technique to be data-locality aware for data-intensive applications [20].

We studied the distributed architectures with the proposed techniques through simulations up to exascale with millions of nodes. The simulation results showed our proposed architectures and techniques have great efficiency, scalability and reliability towards exascale. We then implemented two fully-distributed job management systems, namely SLURM++ [4] and MATRIX [21][87][91], both of which utilize a distributed key-value store for distributed resource and workload management. MATRIX is focusing on executing fine-grained MTC applications, while SLURM++ is targeting scheduling the HPC MPI jobs and the HPC ensemble workloads. We have scaled MATRIX up to 1024 nodes, and scaled SLURM++ up to 500 nodes using different benchmarking workloads with promising performance results. We are working on pushing both systems to support the running of real scientific workloads at large scales.

The main contributions of this work are as follows:

(1) A generic system service **taxonomy** that decomposes services into their basic building blocks. The taxonomy could guide us to explore the design choices and quantify the overheads of distributed features, such as replication, failure/recovery and consistency models, for different services.
(2) A **key-value store (KVS) simulator** that explores different service architectures and distributed design choices up to millions of nodes.
(3) A Many-task computing (MTC) job management system (JMS) simulator, **SimMatrix**, which studies the scalability of the proposed work stealing technique up to 1-Million nodes, 1-Billion cores, and 100-Billions tasks.
(4) A distributed MTC task scheduler, **MATRIX**, which implements the work stealing technique for distributed scheduling of fine-grained MTC workloads.
(5) A distributed HPC resource management system, **SLURM++**, which implements different resource stealing techniques for distributed scheduling of HPC and HPC ensemble workloads.

2. Proposed Work

We propose to develop next-generation JMS towards exascale computing to support the running of all varieties of applications. Our work is divided into three steps. As JMS is an example of system services, the first step is to understand the design choices and the overheads of distributed features for all the system services. Therefore, we devised a generic taxonomy that serves as a principle to guide us to understand the services. At the second step, we propose that key-value stores (KVS) could be used as a building block for system services and simulate KVS to explore the design choices and distributed overheads guided by the taxonomy. After we have the taxonomy as the principle and the KVS as the building block, we then develop the next-generation JMS for both MTC and HPC applications.

2.1 System Service Taxonomy

To be able to reason about system services for HPC and existing distributed services, we have devised a taxonomy [17] by breaking services down into various core components that can be composed into a full service. The taxonomy services as generic principle for all the services, and helps us reason about distributed services as follows: (1) gives us a systematic way to decompose services into their basic building block components; (2) allows us to categorize services based on the features of these building block components, and (3) suggests the configuration space to consider for evaluating service designs via simulation or implementation.

Here we introduce the taxonomy by deconstructing a system service into their building block. A system service can be primarily characterized by its **service model**, **data model**, **network model**, **failure model**, and **consistency model**. These components are explained in details as follows:

(I) Service Model: The service model describes the high-level functionality of a service, its architecture, and the roles of the entities that make up the service. A service can be classified as client-server or peer-to-peer by its service architecture. Other properties demanded by the service such as atomicity, consistency, isolation, durability (ACID) [41], availability, permanence, partition-tolerance etc. are expressed as a part of the service model. These characteristics define the overall behavior of the service and the constraints it imposes on the other models. A transient data aggregation tool, a centralized job scheduler or resource manager with a single failover, a peer-to-peer distributed file system are some examples of the service model.

(II) Data Model: The distribution of data that a service operates on is defined by its data model. In a centralized model, a single central server is responsible for containing and maintaining all the data. Alternatively, the service data can be distributed among the servers with varying levels of replication such as partitioned (no replication), mirrored (full replication), or overlapped (partial replication).

(III) Network Model: The network model dictates how the components of a service are connected to each other. In a distributed service architecture, servers can form a structured overlay – rings; binomial, k-ary, radix trees; complete, binomial graphs; or unstructured overlay networks – random graphs. The services could be further differentiated based on deterministic or non-deterministic routing of information in the overlay network. While some overlay networks imply a complete membership set (e.g. fully-connected), others may assume a partial membership set (eg. Binomial graphs).

(IV) Failure Model: The failure model encompasses how these services deal with server failures. The most common and simplest method is a dedicated fail-over server. Using methods such as triple modular redundancy and erasure codes are other ways to deal with server failures and ensure data integrity. The failure model must also include a recovery mechanism, whether it is recovery via logs or communication with replicating peers.

(V) Consistency Model: The consistency model pertains to how rapidly changes in a distributed system are exchanged and kept coherent. Depending on the data model and the corresponding level of replication, a system service might employ differing levels of consistency. The level of consistency is a tradeoff between the server response time and how tolerant clients are to stale data. It can also compound the complexity of recovery under failure. Servers could employ weak, strong, or eventual consistency depending on the service model and the importance of the data being served.

By combining specific instances of these components we then can define a service architecture. Some specific instantiations of service architectures from the taxonomy are depicted shown in Figure 1 and Figure 2. For instance, c_{tree} is a service architecture with a centralized data model and a tree-based hierarchical overlay network, consistency and recovery model are not depicted, but would need to be identified to define a complete service architecture; d_{fc} has a distributed data model with a fully-connected overlay network whereas d_{chord} is a distributed data model and has a Chord overlay network [42] with partial membership, again the consistency and recovery model are not show graphically.

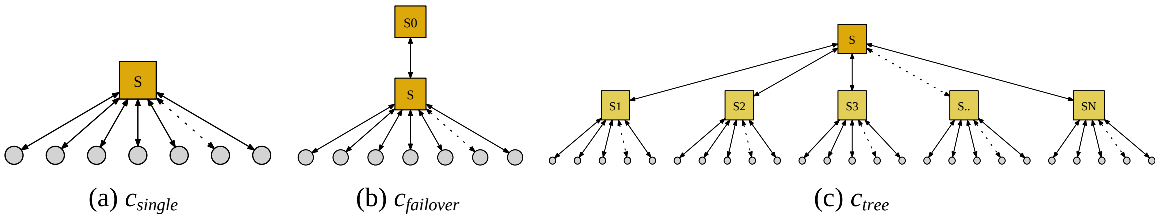


Figure 1: Centralized service architectures

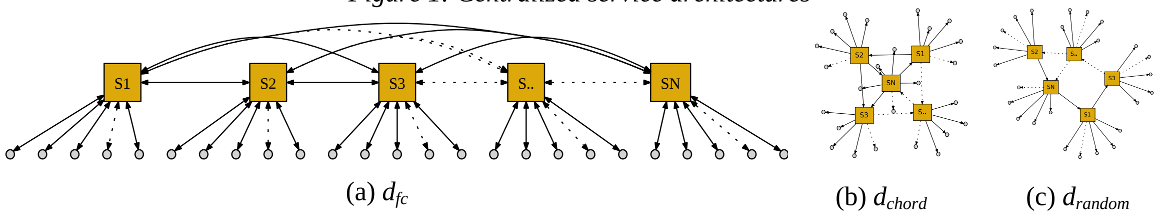


Figure 2: distributed service architectures

The taxonomy is generic and could be used as a principle for all the system services. By exploring the design choices and addressing the challenges of each component of a service, we then combine the solutions together for the whole system service. Since the job management system (JMS) is a representative of system services, this taxonomy helps us understand the architectures and the overheads of different design choices of JMS.

2.2 Key-Value Stores (KVS) serve as a Building Block

Since we have the taxonomy as a generic principle, we narrow down to one typical system service, key-value stores (KVS), and propose that distributed KVS would be a viable building block for extreme-scale system services. We generally target at services that require high performance, such as those that support system booting, system monitoring, hardware or software configuration and management, job and resource management, I/O forwarding and run-time systems for programming models and communication libraries [11][22][23][24]. For extreme-scale systems, these services all need to operate on large volumes of data in a consistent, resilient and efficient manner at extreme scales. We observe that these services commonly and naturally comprise of access patterns amenable to NoSQL abstraction, a data storage and retrieval paradigm that admits weaker consistency models than traditional relational databases. These requirements are consistent with those of large-scale distributed data centers, for example, Amazon, Facebook LinkedIn and Twitter. In these commercial enterprises, NoSQL data stores – Distributed Key-Value Stores (KVS), in particular – have been used successfully [25][26][27].

We assert that by taking the particular needs of high performance computing (HPC) into account, we can use KVS for HPC services to help resolve many of our consistency, scalability and robustness concerns. By encapsulating distributed system complexity in the KVS, we can simplify HPC service designs and implementations. For resource and job management, KVS can be used to maintain necessary job and resource status information; this is what we are doing now. For monitoring, KVS can be used to maintain system activity logs. For I/O forwarding in distributed file systems, KVS can be used to maintain file metadata, including access authority and modification sequences; this has been implemented in the FusionFS distributed file system [44]. In job start-up, KVS can be used to disseminate configuration and initialization data amongst composite tool or application processes; this is under development in MRNet [24]. Application developers from Sandia National Laboratory [28] are targeting KVS to support local check-point restart.

2.2.1 KVS Simulations

As we have motivated that KVS is a building block for extreme-scale system services, simulation can then be used up to exascale to narrow the design space for any specific KVS service application before any implementation has begun. Additionally we can eventually create modular KVS components to allow easy creation of extreme-scale services. We simulate key-value stores up to millions of nodes.

Research about real KVS is impossible at exascale with millions of nodes, because not only we lack the exascale computers, but the experimental results obtained from the real-world platforms are often irreproducible due to resource dynamics [34]. Also, with simulations, we could gain insights and conclusions much faster than we can get through real implementations. Therefore, we fall back to simulations to study the efficiency and scalability of the proposed architectures and techniques at extreme-scales. Simulations have been used extensively as an efficient method to achieve reliable results in several areas of computer science for decades, such as microprocessor design, network protocol design, and scheduling. Discrete event simulation (DES) [35] utilizes a mathematical model of a physical system to portray state changes at precise simulated time. In DES, the operations of a system are represented as a chronological sequence of events. A variation of DES is parallel DES (PDES) [36], which takes advantage of the many-core architecture to access larger amount of memory and processor capacities, and to be able to handle even more complex systems in less end-to-end time. However, PDES adds significant complexity to the simulations, adds consistency challenges, requires more expensive hardware, and often does not have linear scalability as resources are increased.

We have built a discrete event simulator of key-value stores on top of the peer to peer system simulator, PeerSim [43]. Each simulation consists of millions of clients that connect to thousands of shared servers, the number of clients and servers are configurable, and how client selects a server can be preconfigured or random, and is easily modified. The workload for the KVS simulation is a stream of PUTs and GETs. At simulation start, we model unsynchronized clients by having each simulated client stall for a random time before submitting its requests. This step is skipped when modeling synchronized clients. At this point, each client connects to a server (as described below) and sends synchronous (or blocking) GET or PUT requests as specified by a workload file. After a client receives successful responses to all its requests, the client-server connection is closed. Servers are modeled by two queues: a communication queue for sending and receiving messages and a processing queue for handling incoming requests that can be satisfied locally. Requests not handled locally are forwarded to another server. The two queues are processed concurrently, however the requests within one queue are processed sequentially. Since clients send requests synchronously, each server's average number of queued requests is equal to the number of clients that sever is responsible for.

For our distributed architectures, d_{fc} and d_{chord}, our simulator supports two mechanisms for server selection. In the first mechanism, *client selection*, each client has a membership list of all servers; a client selects a server by hashing the request key and using the hash as an index into the server list. Alternatively, a client may choose a random server to service their requests. In the second mechanism, *server selection*, each server has the membership list of part or all of the servers and clients submit requests to a dedicated server. *Client selection* has the benefit of lower latency, but leads to significant overhead in updating the membership list when servers fail. *Server selection*, on the other hand, puts a heavier burden on the servers.

Our current simulator allows us to explore the previously described KVS architectures, namely c_{single}, c_{tree}, d_{fc} and d_{chord}, here we assume a centralized data model for c_{single} and c_{tree}, and a distributed data model for d_{fc} and d_{chord}. The simulator is extendable to other network and data models. The above models can be configured with N-way replication for the recovery model and either eventual or strong for the consistency model.

2.2.2 Validation of KVS simulations

We validate our simulator against two real systems: a zero-hop KVS, ZHT [18], and an open-source implementation of Amazon Dynamo KVS, Voldemort [27]. Both systems serve as building block for system services. ZHT is used to manage distributed resource and job metadata in SLURM++ [4] and MATRIX [21][87], and to manage metadata of file systems (FusionFS [44]), while Voldemort is used to store data for the LinkedIn professional network.

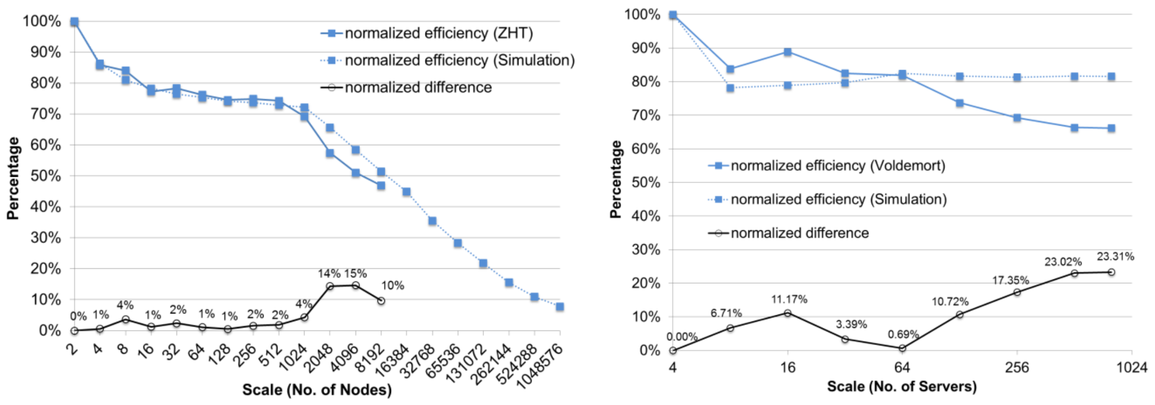


Figure 3: Validation of the key-value store simulations against ZHT and Voldemort

In the case of validating against ZHT, the simulator was configured to match the client selection that was implemented in ZHT. ZHT was run on the IBM Blue Gene/P machine (BG/P) in Argonne National Laboratory with up to 8K nodes and 32K cores. We used the published network parameters of BG/P in our simulator. We used the same workload as that used to in ZHT: each node has a client and a server, each client submits 10K requests with URD, the length of the key is 128 bits, and the message size is 134 bytes. The result in *Figure 3* (left) shows that our simulator matches ZHT with up to the largest scale (8K nodes with 32K cores) that ZHT was run. The biggest difference was only 15% at large scales. The ZHT curve depicts decreasing efficiency after 1024 nodes, because each rack of BG/P has 1024 nodes. Within 1024 nodes (one rack), the communication overhead is small and relatively constant, leading to constant efficiency (75%). After 1024 nodes, the communication spans multiple racks, leading to more overhead and decreasing efficiency.

In the case of Voldemort, we focused on validating the eventual consistency model of the simulator. The simulator was configured to match the server selection d_{fc} model, with each server backed by 2 replicas and responsible for 1024 clients, and an associated eventual consistency protocol with versioning and read-repair. We ran Voldemort on the Kodiak cluster from PROBE [45] with up to 800 servers, and 800k clients. Each client submitted 10 random requests. As shown in *Figure 3* (right part), our simulation results match the results from the actual run of Voldemort within 10% up to 256 nodes. At higher scales, due to resource over-subscription, an acute degradation in Voldemort's efficiency was observed. Resource over-subscription means that we ran way too many client processes (up to 1k) on one physical node. At the largest scale (800 servers and 800 nodes), there will be 1k client processes running on each node, leading to serious resource over-subscription.

Given the good validation results, we believe that the simulator can offer convincible performance results of the various architectural features we are interested in. This allows us to weigh the service architectures and the overheads that are induced by the various features.

2.2.3 Simulation with Real Service Workloads

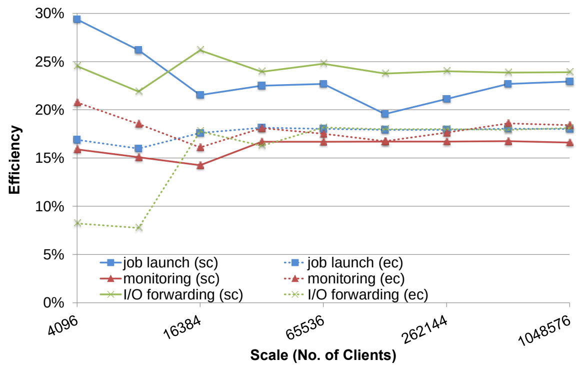
We run simulations with three workloads obtained from typical HPC services: job launch, monitoring, and I/O forwarding. The specification of each workload is listed below:

(1) **Job Launch:** The job launch workload is obtained by monitoring the messages between the server and client during a MPI job launch. Though the service is not implemented in a distributed fashion the messages to and from the clients should be representative regardless of server structure and this in turn drives the communication between the distributed servers. Job launch is characterized by control messages from the distributed servers (Get) and the results from the compute nodes back to the servers (Put).

(2) **Monitoring:** The monitoring workload is obtained from a 1600 node cluster's syslog data. This data was then categorized by message-type (denoting the key-space) and count (denoting the probability of each message). This distribution was then used to generate the workload that is completely Put dominated.

(3) **I/O Forwarding:** The I/O forwarding workload is generated by running FusionFS [44] distributed file system. The client first creates 100 files, and then operates (reads or writes with 50% probability) each file once. We collect the log of the ZHT metadata server.

We extend and feed these real workloads to our simulator in order to investigate the applicability of our KVS simulator for HPC system services. The workloads obtained are not big enough for an extreme-scale system. For job launch and I/O forwarding workloads, we repeat the workloads several times until reaching 10M requests, and the "key" of each request is generated with uniform random distribution (URD) within our 64-bit key space. For the monitoring workload, there are 77 kinds of message types with each one having a different probability. We generate 10M Put requests; the "key" is generated based on the probability distribution of the message types and is mapped to our 64-bit key space. We point out that these extensions are not enough to reflect every detail of these workloads. Nevertheless, they do reflect some important properties; the job launch and I/O forwarding workloads reflect the time serialization property and the monitoring workload reflects the probability distribution of all obtained messages.



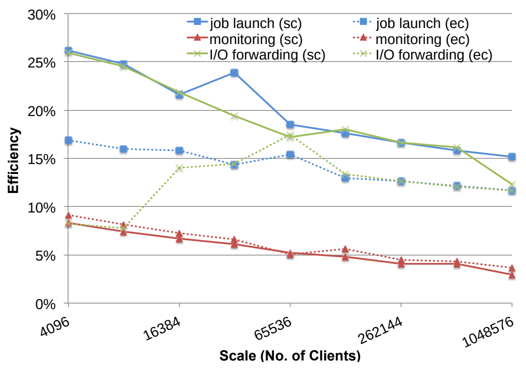


Figure 4: d_{fc} (left) and d_{chord} (right) with real workloads

Figure 4 shows the efficiency of these workloads with both strong and eventual consistency for d_{fc} (left) and d_{chord} (right). We see that for job launch and I/O forwarding workloads, eventual consistency performs worse than strong consistency. This is because these two workloads have al- most URD for both request type and the key. The ratio of the number of Get to Put requests is 50.9% to 49.1% for job launch, and 57.6% to 42.4% for I/O forwarding. For the monitoring workload, eventual consistency outperforms strong consistency because all the requests are Put type, which requires all N-1 acknowledgment messages from the other N-1 replicas in strong consistency whereas just R-1 or W-1 acknowledgment messages from the other N-1 replicas in eventual consistency. Another fact is that the efficiency of the monitoring workload is the lowest because the key space is not uniformly generated, which leads to poor load balancing.

The above results demonstrate that our simulator is capable of simulating different kinds of system services as long as the workloads of these services could be transformed to Put or Get requests, which is true for the HPC services we have investigated.

The conclusion we draw is that the centralized server doesn't scale, and distributed system architectures are necessary to reach exascale. When there are a huge amount of client requests (up to billions at exascale), fully connected topology (d_{fc}) actually scales very well under moderate failure frequency, with different replication and consistency models, though it is relatively expensive to do a broadcast to update everyone's membership list when churn happens; while partial-knowledge topology (d_{chord}) scales moderately with less expensive overhead under churn. When the communication is dominated by server messages, (due to failure/recovery, replication or consistency) rather than client request messages, then d_{chord} would have an advantage. Different consistency models have different application domains; strong consistency is more suitable for running read-intensive applications, while eventual consistency is preferable for applications that require high availability and fast response times.

2.3 Distributed Job Scheduling for Many-Task Computing Workloads

The Many-task computing (MTC) [83] paradigm comes from the data-flow driven model, and aims to define and address the challenges of scheduling fine-grained data-intensive workloads [8]. 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 have fine granularity in both size (e.g. per-core) and durations (e.g. sub-seconds to a few seconds), and do not require strict coordination of processes at job launch as the traditional HPC workloads.

The distributed scheduling architecture for fine-grained MTC applications is shown in *Figure 5*. We justify that the task scheduling system for MTC will need to be fully distributed to achieve high efficiency, scalability, and availability. In this design, each compute node runs one scheduler and one or more executors/workers. As at exascale, each compute node would have up to thousands of cores and the MTC data-intensive workloads have extremely large amount of fine-grained jobs/tasks with each task requiring one core/thread for a short amount of time (e.g. millisecond), there would need a scheduler on one "fat" compute node forming 1:1 mapping. The clients would submit workloads to any random scheduler. All the schedulers are fully connected, and receive workloads to schedule tasks to local executors. Therefore, ideally, the throughput would gain near-optimal linear speedup as the system scales. We abandon the centralized architecture due to the limited processing capacity and the single-point-of-failure issue. We bypass the hierarchical architecture as it is difficult to maintain a tree under failures, and a task may experience longer latency because it needs to go through multiple hops as opposed to only one in a fully distributed architecture.

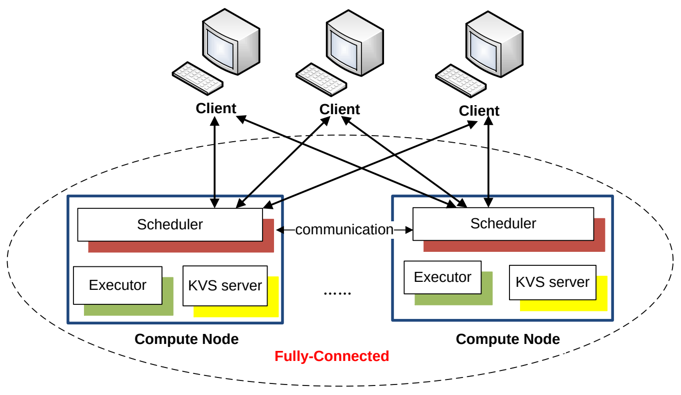


Figure 5: Distributed scheduling architectures for MTC applications

In addition, as we have already motivated there is a distributed key-value store [17][18] that manages the entire job and resource metadata, as well as the state information in a transparent, scalable and fault tolerant way. The distributed KVS should also be fully-connected, and one typical configuration is to co-locate a distributed KVS server with a controller/scheduler on the same compute node forming one-to-one mapping, such as shown in *Figure 5*.

2.3.1 Work Stealing

For the fully-distributed architecture of JMS for the fine-grained MTC applications, load balancing [29] is challenging as a scheduler only has knowledge of its own state, and therefore must be done with limited partial state. Load balancing refers to distributing workloads as evenly as possible across all the schedulers/workers, and it is important given that a single heavily-loaded scheduler would lower the system utilization significantly. This work adopts the work stealing technique [19] at the node/scheduler (instead of core/thread) level. In work stealing, the idle schedulers communicate with neighbors to balance their loads. Work stealing has been proven as an efficient load balancing technique at thread/core level in multi-core shared memory machine [19]. It is a pull-based method in which the idle processors try to steal tasks from the overloaded ones that are randomly selected. This work adopts work stealing at the node level in distributed environment, and modifies it to be adaptive according to the system states at runtime.

Work stealing is proceeded by the idle schedulers stealing workloads from neighbors. As in fully-distributed architecture, every scheduler has a global membership list, therefore, the selection of candidate neighbors (victims) from which an idle scheduler (thief) [30] would steal tasks could be static or dynamic/random. In static mechanism, the neighbors are pre-determined and will not change. This mechanism has limitation that every scheduler is confined to communicate with part of other schedulers. In dynamic case, whenever a scheduler is idle, it randomly chooses some candidate neighbors from the membership list. The traditional work stealing randomly selects one neighbor to steal tasks [30] yielding poor performance at extreme scales, because the chance that one neighbor would have queued ready task is low at large scales. In the end, we choose to have a multiple-random neighbor selection strategy that randomly selects several neighbors instead of only one, and chooses the most heavily loaded one to steal tasks. This dynamic mechanism introduces some overhead when selecting neighbors; however, we show that this overhead is minimal (the time complexity is $\Theta(n)$, where n is the number of neighbors) in Algorithm 1.

ALGORITHM 1. *Dynamic Multi-Random Neighbor Selection (DYN-MUL-SEL)*

```
Input: Node id (node_id), number of neighbors (num_neigh), and number of nodes (num_node), and the node array (nodes).
Output: A collection of neighbors (neigh).
bool selected[num_node];
for each i in 0 to num_node do
    selected[i] = FALSE;
end
selected[node_id] = TRUE; Node neigh[num_neigh]; index = -1;
for each i in 0 to num_neigh−1 do
    repeat
        index = Random( ) % num_node;
    until !selected[index];
    selected[index] = TRUE; neigh[i] = nodes[index];
end
return neigh;
```

We prove the time complexity of Algorithm 1 is $\Theta(n)$, where n is the number of neighbors (*num_neigh*). Let k be the kth neighbor to be chosen, m be the number of nodes in the system. The possibility that one neighbor that has already been chosen is chosen again is:

$$p_r = \frac{k-1}{m}$$

Let i be the number of times to select the kth neighbor, the actual value is:

$$i \times (p_r)^{i-1} \times (1-p_r)$$

So, the expected number of number of times for selecting the kth neighbor:

$$E_c = \sum_{i=1}^{\infty} i \times (p_r)^{i-1} \times (1-p_r)$$

Let's say for an exascale system, $m = 10^6$, as we could see later, the maximum dynamic number of neighbors $k_{\max} = \sqrt{m} = 10^3$, so the maximum

$$p_r = \frac{k_{\max}-1}{m} \approx \frac{k_{\max}}{m} = \frac{1}{1000}$$

So, for E_c, after $i = 3$,

$$i \times (p_r)^{i-1} \times (1-p_r) = 3 \times \left(\frac{1}{1000}\right)^2 \times \left(1-\frac{1}{1000}\right) \approx 3 \times 10^{-6} \approx 0$$

Therefore, we just need to consider $i = 1$ and $i = 2$,

$$E_c \approx 1 \times \left(\frac{1}{1000}\right)^0 \times \left(1 - \frac{1}{1000}\right) + 2 \times \left(\frac{1}{1000}\right)^1 \times \left(1 - \frac{1}{1000}\right) \approx 1$$

So, the time complexity to choose n neighbors is: $\Theta(n)$.

After an idle scheduler selects candidate neighbors, it goes through each neighbor in sequential to ask for "load information" (number of ready tasks), and tries to steal tasks from the most heavily-overloaded one. When a scheduler fails to steal tasks from the selected neighbors, because either all selected neighbors have no more tasks, or the reported extra ready tasks have been executed when the stealing happens, the scheduler waits for a period of time and then does work stealing again. We call this wait time the polling interval. Algorithm 2 gives the overall work stealing algorithm.

ALGORITHM 2. *Adaptive Work Stealing Algorithm (ADA-WORK-STEALING)*

Input: Node id (*node_id*), number of neighbors (*num_neigh*), number of nodes (*num_node*), the node array (*nodes*), initial poll interval (*poll_interval*), and the poll interval upper bound (*ub*).
Output: NULL

```
while (poll_interval < ub) do
    neigh = DYN-MUL-SEL(node_id, num_neigh, num_node, nodes); most_load_node = neigh[0];
    for each i in 1 to num_node−1 do
        if (most_load_node.load < neigh[i].load) then
            most_load_node = neigh[i];
        end
    end
    if (most_load_node.load == 0) then
        sleep(poll_interval); poll_interval = poll_interval $\times$ 2;
    else
        num_task_steal = number of tasks stolen from most_load_node;
        if (num_task_steal == 0) then
            sleep(poll_interval); poll_interval = poll_interval $\times$ 2;
        else
            poll_interval = 1;
            break;
        end
    end
end
```

If the polling interval is fixed, there would be difficulties to set the value with right granularity. If the polling interval is set to be too small, at the final stage when there are not many tasks left, many schedulers would poll neighbors to do work stealing, which would ultimately fail and lead to more work stealing communications. If the polling interval was set large enough to limit the number of work stealing events, work stealing would not respond quickly to change conditions, and lead to poor load balancing. Therefore, we implement an adaptive polling interval strategy, in which, the polling interval of a scheduler is changed dynamically similar to the exponential back-off approach in the TCP networking protocol [31]. The default polling interval is set to be a small value (e.g. 1ms). Once a scheduler successfully steals tasks, the polling interval of that scheduler is set back to the initial small value. If a scheduler fails to steal tasks, it waits the time of polling interval and doubles the polling interval and tries to do work stealing again. In addition, we set an upper bound for the polling interval. Whenever the polling interval hits the upper bound, a scheduler would not do work stealing anymore. This would reduce the amount of failing work stealing at the final stage.

The parameters of work stealing are: the number of tasks to steal, the number of dynamic neighbors, and the polling interval. We will explore these parameters though simulations up to exascale with millions of nodes, billions of cores and hundreds of billions of tasks, in order to find the optimal parameter configurations for work stealing.

2.3.2 Data-Aware Work Stealing

As more fine-grained MTC applications are becoming data-intensive and experiencing data explosion [32] such that tasks are dependent and task execution involves processing large amount of data, data-aware scheduling and load balancing are two indispensable yet orthogonal needs. Migrating tasks randomly through work stealing would compromise the data-locality and incur significant data-transferring overhead. On the other hand, struggling to map each task to the location where the data resides is not feasible due to the complexity of the computation (this mapping is a NP-complete problem [33]). Furthermore, this mapping could lead to poor load balancing due to the potential unbalanced data distribution. Therefore, we investigate scheduling methods that satisfies both needs and still achieves good performance.

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

Figure 6: Task metadata as value stored in the KVS

We propose a data-aware work stealing technique [20][88] that is able to achieve good load balancing and yet still tries to best exploit data-locality. The distributed KVS (e.g. ZHT) could store all the important meta-data information related to a specific task in a data-intensive workload, such as data dependency conditions, and data locality information of tasks. The "key" is task id, and the "value" is the important meta-data of the task that is defined in *Figure 6* including the information such as number of waiting parents, the parent list, the data object and size of each parent, and the children tasks that are dependent on the current task.

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

(a) Task waiting queue (WaitQ)

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

(b) Dedicated local task ready queue (LReadyQ), shared work stealing task ready queue (SReadyQ)

When a task is ready to run, the scheduler makes decision to put it in either the *LReadyQ* or the *SReadyQ*, according to the size and location of the data required by the task. The *LReadyQ* stores the ready tasks that require large volume of data and the majority of the required data is located at the current node; these tasks could only be scheduled and executed locally unless special policy is used. The *SReadyQ* stores the tasks that could be migrated to any scheduler for load balancing's purpose; these tasks either don't need any input data or the demanded data volume is so small that the transferring overhead is negligible. The "***load information***" queried by work stealing is the number of tasks in *SReadyQ*. The pseudo-code for decision making is given in Algorithm 3. TMD means task meta-data structure defined in *Figure 6*.

ALGORITHM 3. *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.

```
if (tm.all_data_size / est_task_length <= t) then
    SReadyQ.push(task);
else
    long max_data_size = tm.data_size.at(0);
    int max_data_scheduler_idx = 0;
    for each i in 1 to tm.data_size.size() - 1; do
        if tm.data_size.at(i) > max_data_size; then
            max_data_size = tm.data_size.at(i);
            max_data_scheduler_idx = i;
        end
    end
    if (max_data_size / est_task_length <= t); then
        SReadyQ.push(task);
    else if tm.parent_list.at(max_data_scheduler_idx) == id; then
        LReadyQ.push(task);
    else
        send task to: tm.parent_list.at(max_data_scheduler_idx)
    end
end
return;
```

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

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

In the executor, there are several executing threads that keep pulling ready tasks to execute. Each thread would first pop tasks from *LReadyQ*, and then pop tasks from *SReadyQ* if the *LReadyQ* is empty. Both *LReadyQ* and *SReadyQ* are implemented as descending priority queue based on the required data size. The larger the required data size is, the higher priority the task would be, as a task that requires larger data size usually runs longer. When executing a task, the executing thread first queries the meta-data for the size and location of the required data produced by each parent, and then gets the data either from local or remote nodes. After collecting all the data, the task will be executed. The number of executing threads is configurable; in practice it is usually configured to be the number of physical cores (a similar strategy was used in Falkon [15] on the Blue Gene/P supercomputer). As long as both queues are empty, the scheduler would start doing work stealing, and the stolen tasks would be put in the *SReadyQ*.

(c) Task complete queue (CompleteQ)

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

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

(a) maximized load balancing (MLB)

MLB considers only the load balancing, and all the ready tasks are put in the SreadyQ to be allowed for migration. We achieve the MLB policy by tuning the threshold (*t*) in Algorithm 3 to be the maximum possible value (i.e. LONG_MAX). This is the baseline work stealing strategy without taking into consideration the data locality.

(b) maximized data-locality (MDL)

MDL only considers data-locality, and all the ready tasks that require input data would be put in LReadyQ, no matter how big the data is. This is achieved by tuning the threshold (*t*) in Algorithm 3 to be 0.

(c) rigid load balancing and data-locality segregation (RLDS)

RLDS sets the threshold (t) in Algorithm 3 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) flexible load balancing and data-locality segregation (FLDS)

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

2.3.3 SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascale

To achieve the goal of developing next-generation job scheduling system for MTC applications, we divide the research work into three steps, namely simulations, real implementations and production systems. These three steps are intertwined with each other. We first use simulations to explore the feasibility and efficiency of the proposed architecture and work stealing techniques up to extreme-scales with millions of nodes.

SimMatrix [37] is a light-weight simulator that simulates job management system comprising of millions of nodes and billions of cores/tasks. We have built SimMatrix from scratch, and SimMatrix simulates both centralized scheduling and fine-grained fully distributed scheduling architecture. Careful consideration was given to the SimMatrix design, to ensure that it would scale to exascale levels on modest resources in a single-node shared memory system. We validate SimMatrix against two job management systems, Falkon [15] and MATRIX [21]. Falkon is a centralized task execution framework for MTC workloads, while MATRIX is a task scheduling system that has implemented the fine-grained fully-distributed architecture and work stealing technique. We have studied the distributed architecture and work stealing technique through SimMatrix up to 1-million nodes, 1-billion cores, and 100-billion tasks.

(a) Validation of SimMatrix against Falkon and MATRIX

SimMatrix is validated against the state-of-the-art MTC execution fabrics, Falkon (for centralized scheduling) and MATRIX (for distributed scheduling with adaptive work stealing technique). We set the number of cores per node to be 4, and the network bandwidth and latency the same as the case of IBM Blue Gene / P machine. The number of tasks is 10 times and 100 times of the number of all cores for Falkon and MATRIX respectively.

The validation results are shown in *Figure 7*. We measured SimMatrix (dotted lines) has an average 2.8% higher efficiency than Falkon (solid lines) for several sleep tasks (sleep 1, 2 and 4) in *Figure 7* (left part). SimMatrix and MATRIX are compared for raw throughput on a "sleep 0". For sleep 0 workload (*Figure 7*, right part), the simulation matched the real performance data with average 5.85% normalized difference (abs(SimMatrix - MATRIX) / SimMatrix). The reasons for these differences are twofold. Falkon and MATRIX are real complex systems deployed on a real supercomputer. Our simulator makes simplifying assumptions; for example, we increase the network communication overhead linearly with the system scale. It is also difficult to model communication congestion, resource sharing and the effects on performance, and the variability that comes with real systems. We believe the relatively small differences (2.8% and 5.85%) demonstrate that SimMatrix is accurate enough to produce convincible results (at least at modest scales).

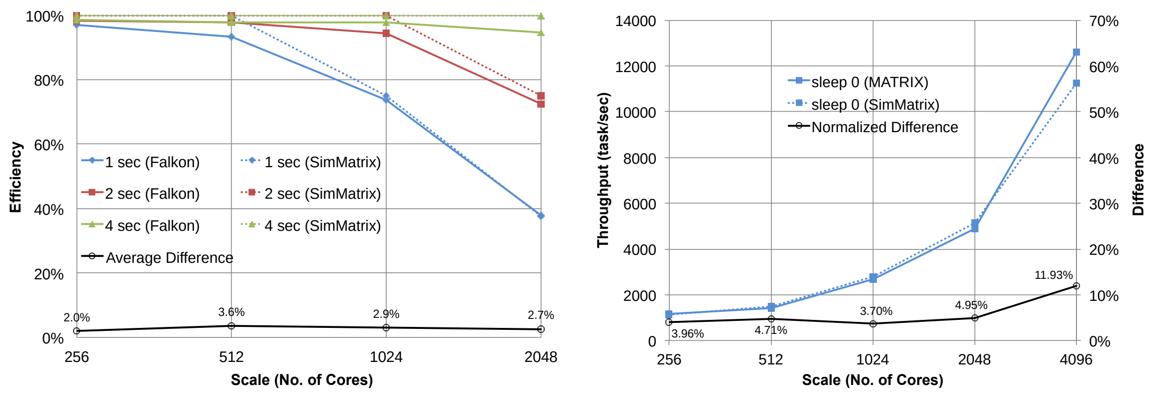


Figure 7: Validation of SimMatrix against Falkon and MATRIX

(b) Exploration of Work Stealing Parameter Space

The parameters that affects the performance of work stealing are number of tasks to steal, number of neighbors, static neighbor vs. dynamic random neighbor selection. We explore them in great detail through

SimMatrix. Each node is configured to have 1024 cores, and the number of tasks is 100 times of the number of all cores. The workloads were collected over a 17-month period comprising of 173M tasks [38][39][40]. We filtered out the logs to isolate only the 160K-core Blue Gene / P machine, which netted about 34.8M tasks with the minimum runtime of 0 seconds, maximum runtime of 1469.62 seconds, average runtime of 95.20 seconds, and standard deviation of 188.08.

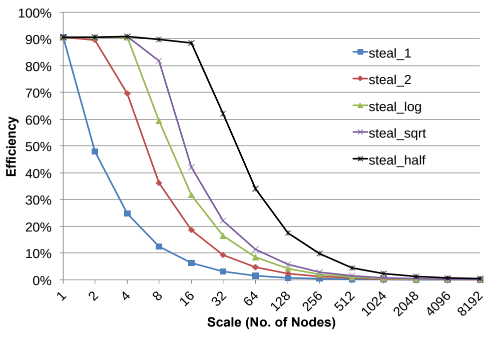


Figure 8: numbers of tasks to steal

Number of Tasks to Steal. In our five groups of experiments, steal_1, steal_2, steal_log, steal_sqrt, steal_half means steal 1, 2, logarithm base-2, square root, and half number of tasks respectively. We set the number of neighbors of a node to be 2, and uses static neighbors. The changes of the efficiency of each group with respect to the scale are shown in *Figure 8*. It shows that as the number of nodes increases, the efficiencies of steal_1, steal_2, steal_log, steal_sqrt decrease. The efficiency of steal_half keeps at the value of about 90% up to 16 nodes, and decreases after that. And the decrease speed of steal_half is the slowest. These results show that stealing half number of tasks is optimal, which confirms both our intuition and the results from prior work on work stealing [30]. The reason that steal_half is not perfect (efficiency is very low at large scale) for these experiments is that 2 neighbors of a node is not enough, and starvation can occur for some nodes that are too far in the id namespace from the original compute node who is receiving all the task submissions. The conclusion of this group of experiments is that stealing half number of tasks is optimal and having a small number of static neighbors is not sufficient to achieve high efficiency even at modest scales. We also can generalize that stealing more tasks (less than half) generally produces higher efficiencies.

Number of Neighbors of a Node. There are two ways by which the neighbors of a node are selected: static neighbors mean the neighbors (consecutive ids) are determined at first and never change; dynamic random neighbors mean that each time when does work stealing, a node randomly selects some neighbors. The results of both neighbor selection strategies are shown in *Figure 9* with the left for static and the right for dynamic. In our experiments, nb_1, nb_2, nb_log, nb_sqrt, nb_eighth, nb_quar, nb_half means 1, 2, logarithm base-2, square root, eighth, a quarter, half neighbors of all nodes, respectively.

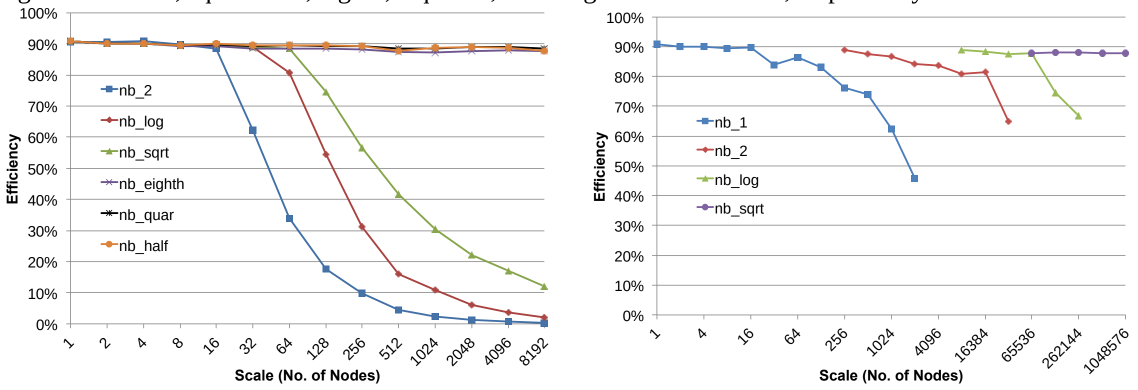


Figure 9: Number of neighbors of a scheduler; the left is for static neighbor selection, and the right is for dynamic neighbor selection

In static neighbor selection (*Figure 9*, left part), when the number of neighbors is no less than eighth of all nodes, the efficiency will keep at the value of higher than 87% within 8192 nodes' scale. For other numbers of static neighbors, the efficiencies could not remain, and will drop down to very small values. We conclude that the optimal number of static neighbors is eighth of all nodes, as more neighbors do not improve performance significantly. However, in reality, an eighth of neighbors will likely lead to too many neighbors to be practical, especially for an exascale system with millions of nodes (meaning 128K neighbors). In the search for a lower number of needed neighbors, we explore the dynamic multiple random neighbor selection technique.

In dynamic neighbor selection (*Figure 9*, right part), we first do nb_1 experiment until starting to saturate (the efficiency is less than 80%), then at which point, start to do nb_2, then nb_log, and nb_sqrt at last. The results show that nb_1 scales up to 128 nodes, nb_2 scales up to 16K-nodes, nb_log scales up to 64K-nodes, and nb_sqrt scales up to 1M-nodes, remaining the efficiency at the value about 87%. Even with 1M-nodes in an exascale system, the square root implies having 1K neighbors, a reasonable number of nodes for which each node to keep track of with modest amounts of resources.

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

(c) Scalability of the Data-aware Work Stealing

We implemented the DAWS technique in SimMatrix. We configured SimMatrix as follows: each node is configured to have 2 cores, each core executes 1000 tasks on average, and each task runs an average time of 50ms (0 to 100ms) and generates an average data size of 5MB (0 to 10MB). The parameters of work stealing are configured as the suggested optimal values. We scaled the DAWS technique through SimMatrix up to 128K cores (128M tasks). Each experiment is run at least 3 times, and the scalability results are shown in Figure 10.

We scale SimMatrix up to 128K cores running all the DAGs, the results are shown in Figure 10. We explore four different DAGs, namely Bag of Task (BOT), Fan-In, Fan-Out and Pipeline, which are represented in Figure 10, the left part. BOT workload is used as a baseline, it includes independent tasks; Fan-In and Fan-Out DAGs are similar except that they are reverse. The performance of these workloads depends on the in-degree, out-degree and the dependent data sizes. Pipeline DAG is a collection of “pipes” where each task within a pipe is dependent on the previous task.

The throughput trends indicate that the DAWS technique has nearly-linear scalability with respect to the scales for all workloads. At 128K-core scale, the DAWS achieves high throughput of 2.3M tasks/sec for BOT, 1.7M tasks/sec for Pipeline, 897.8K tasks/sec for Fan-Out, and 804.5K tasks/sec for Fan-In, respectively. The trend is likely to hold towards million-core scales, and we will validate in the future.

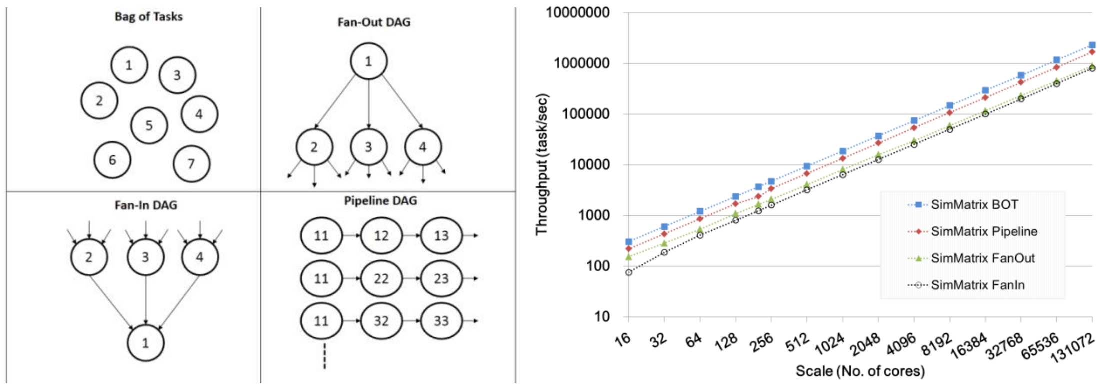


Figure 10: Left are different workload DAGs; Right are the results of running the DAGs through SimMatrix

Through simulations, we have shown that the proposed DAWS technique has the potential to scale up to the level of exascale computing.

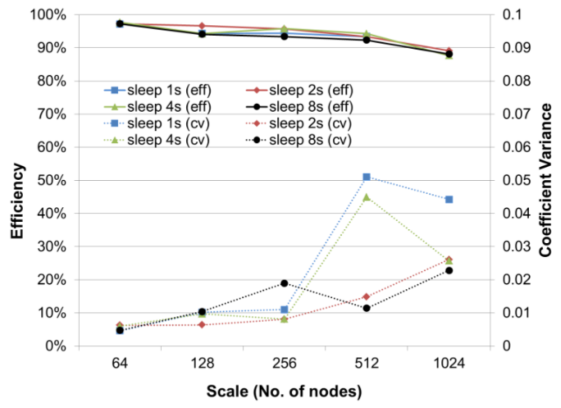
2.3.4 MATRIX: MAny-Task computing execution fabRIc at eXascale

The simulations have gained us valuable insights that the distributed scheduling architecture for MTC applications is scalable up to extreme-scales. Also, the work stealing technique and the random property has the potential to scale up to extreme-scales with optimal configuration parameters. With the help of the conclusions and insights gained from the simulations, we designed and implemented the job scheduling system targeting MTC applications, namely MATRIX [21].

MATRIX applies work stealing to achieve distributed load balancing, and supports the data-aware work stealing with the help of four distributed local queues (task waiting queue, task ready queue, task running queue, and task completion queue) and the ZHT distributed KVS. ZHT stores the task metadata including task dependency information, data flow information, data-locality information, and task execution progress. The specification of the metadata is shown in *Figure 6*.

(a) MATRIX scalability with fine-grained tasks

We have evaluated MATRIX using micro-benchmarks on an IBM Blue Gene/P supercomputer up to 4K-core scale. Figure 11 left shows the efficiency (eff) and coefficient variance (CV) (of the number of tasks finished by each scheduler) results for coarser grained tasks (sleep 1, 2, 4, 8 sec), while Figure 11 right shows these results for fine grained tasks (sleep 64, 128, 256 ms).



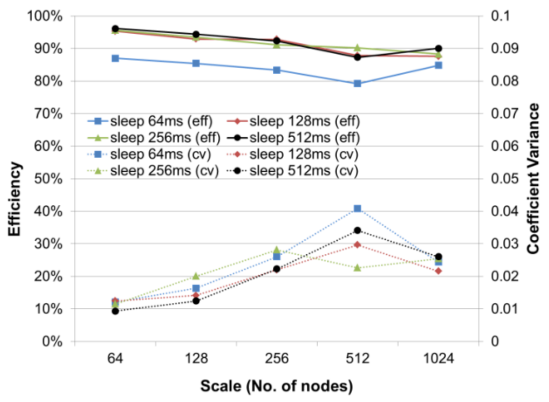
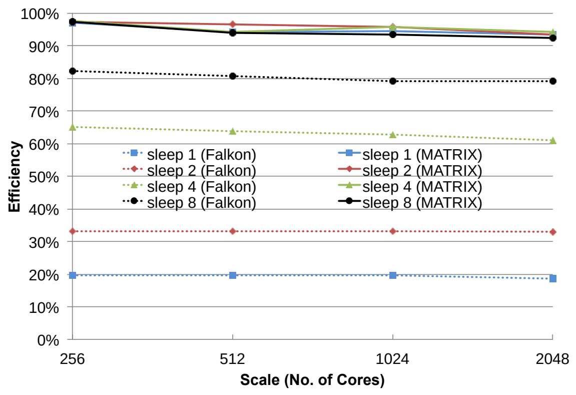


Figure 11: Scalability of work stealing algorithm for different granularities of tasks

We see that up to 1K-nodes, MATRIX actually works quite well even for fine grained sub-second tasks. With 85% efficiency for sleep-64ms workload at 1024 nodes, MATRIX can achieve throughput of about 13K task/sec. The reason that efficiency decreases with the scale is because the run time of 1000 seconds is still not enough to amortize the slow start and long trailing tasks at the end of experiment. We believe that the more tasks per core we set, the higher the efficiency will be, with an upper bound because there are inevitable communication overheads, such as the submission time. The fact that all coefficient values are small indicates the excellent load balancing ability of the adaptive work stealing algorithm. CV value of 0.05 means that the standard deviation of the number of executed tasks of each node is 500-tasks when on average each node completed 10K tasks. We also want to point out that efficiency does not drop much from coarser grained tasks to fine grained tasks, which concludes that fine grained tasks are quite efficiently executed with the work stealing algorithm.

(b) Comparison among MATRIX, Falkon, Sparrow and CloudKon

We compare MATRIX with Falkon on the Blue Gene / P machine at up to 2K-core scales. We also evaluate MATRIX in the Amazon cloud on up to 64 EC2 instances with both homogenous BOT workload and heterogeneous BOT workloads based on real traces by comparing it with Sparrow and CloudKon.



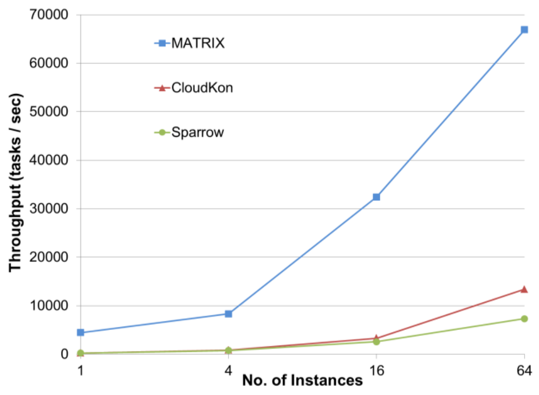


Figure 12: Comparisons of MATRIX against Falkon (left) and CloudKon and Sparrow (right)

MATRIX vs Falkon on BG/P: Falkon is a centralized task execution framework with a hierarchical task dispatching implementation. Figure 12 left shows the comparison between MATRIX and Falkon running on BG/P machine up to 512 nodes (2K cores) with sleep 1, 2, 4, and 8 sec workloads. As Falkon is a centralized task scheduler with the support of hierarchical scheduling, we see MATRIX outperforms Falkon across the board for all workloads with different durations. MATRIX achieves efficiencies starting at 92% (2K cores) up to 97% (256 cores), while Falkon only achieved efficiencies from 18% (2K cores) to 82% (256 cores). While the hierarchical architecture has moderate scalability within petascale, distributed architecture with the work stealing technique seems to be a better approach towards fine grained tasks and extreme scales.

MATRIX vs Sparrow & CloudKon on the Amazon Cloud: We also compare MATRIX to Sparrow [46] and CloudKon [47] in the Amazon cloud up to 64 instances with Bag-of-Task (BOT) workloads. We configured one scheduler and one worker running on one instance. Sparrow is a distributed task scheduler with multiple schedulers pushing tasks to workers. Each scheduler knows all the workers. When dispatching a batch of tasks, a scheduler probes multiple workers (based on the number of tasks), and pushes tasks to the least overloaded ones. Once the tasks are submitted to a worker, they cannot be migrated in case of load imbalance. CloudKon is a distributed task scheduler specific to the cloud environment. CloudKon leverages the Amazon SQS [48] to achieve distributed load balancing and DynamoDB [25] for task metadata management.

For throughput comparison, we use “sleep 0” tasks, and each instance runs 16K tasks on average. For Sparrow, it is difficult to control the exact number of tasks to be executed, as tasks are submitted with a given submission rate. We set the submission rate as high as possible (e.g. 1us) to avoid the submission bottleneck, and set a total submission time to control the number of tasks that is approximately 16K tasks on average. The result is shown in Figure 12 right. We see that even though all of them have throughput increasing trends with respect to the system scale, MATRIX has much higher throughput than CloudKon and Sparrow. Up to 64 instances, MATRIX shows throughput speedup of more than 5X (67K vs 13K) comparing to CloudKon, and speedup of more than 9 comparing to Sparrow (67K vs 7.3K). This is because MATRIX has near perfect load balancing when submitting tasks; while the schedulers of Sparrow needs to send probe messages to push tasks, and the clients of CloudKon need to push and pull tasks from SQS.

(c) MATRIX with data-aware scheduling

MATRIX has implemented the data-aware work stealing for scheduling data-intensive applications. We have evaluated MATRIX with both image stacking application in astronomy and all-pairs application in biometrics, and compared with Falkon data-diffusion technique [38][49] up to 200-core scale. The results are shown in Figure 13, the left part is for image stacking and the right is for all-pairs. DAWS stands for our data-aware work stealing technique.

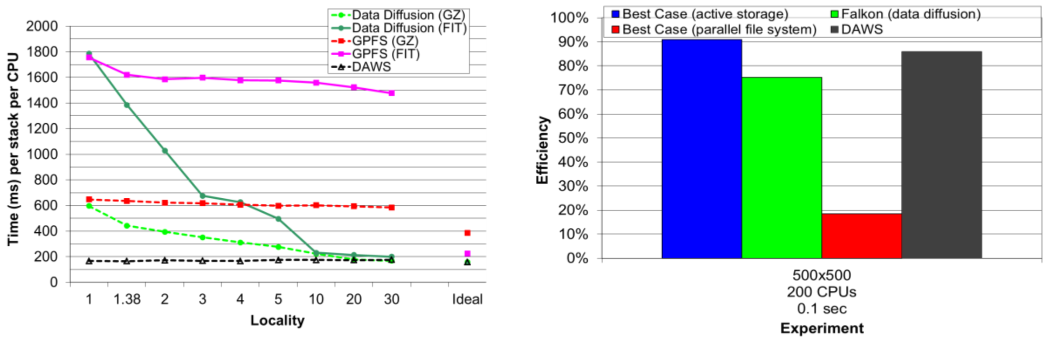


Figure 13: Comparing data-aware work stealing (DAWS) with Falkon data-diffusion

The stacking application conducts the “stacking” of image cutouts from different parts of the sky. Followed the workload characterization in [49], each task requires a 2MB file, and generates 10KB data of output. The ratio of the number of tasks to the number of files refers to locality number. The number of tasks and the number of files for each locality are given in [49], and each task would run for an average of 158 ms. Figure 13 (left) shows that the time per task of our DAWS technique keeps almost constant as locality increases at 128 cores, and very close to the ideal running time (158ms). Data Diffusion (GZ) experienced very large average time when locality is small, and decreases to be close to the ideal time when locality is 30. The reason is that in DAWS, data is uniformly distributed over compute nodes. The only overhead comes from the schedulers making decisions to push tasks in the right spots. While in Data Diffusion (GZ), as data is initially kept in a slower shared file system, the data would be copied to local disks when needed. When locality is small, the chances that the same piece of data will be reused are low therefore involving more amount of data access from the shared file system.

All-Pairs describes the behavior of a new function on sets A and sets B. For example, in Biometrics, it is very important to find out the covariance of two sequences of gene codes. In this workload, all the tasks are independent, and each task execute for 100 ms to compare two 12MB files with one from each set. We run strong-scaling experiments up to 100 nodes with a 500*500 workload size. Therefore, there would be 250K tasks in total. As a task needs two files that may locate at different nodes at the worst case, file may need to be transmitted. In Figure 13 right part, we see that DAWS improved Data Diffusion by 14.21% (85.9% vs 75%), and it is quite close to the best case using active storage (85.9% vs 91%). Data diffusion applies a centralized index-server for data-aware scheduling, while our DAWS technique utilizes distributed KVS that is much more scalable.

As we have shown that our proposed data-aware work stealing technique can perform well for data-intensive applications structured as simple DAGs, we evaluated more complex synthetic DAGs, such as those shown in Figure 10 left part.

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

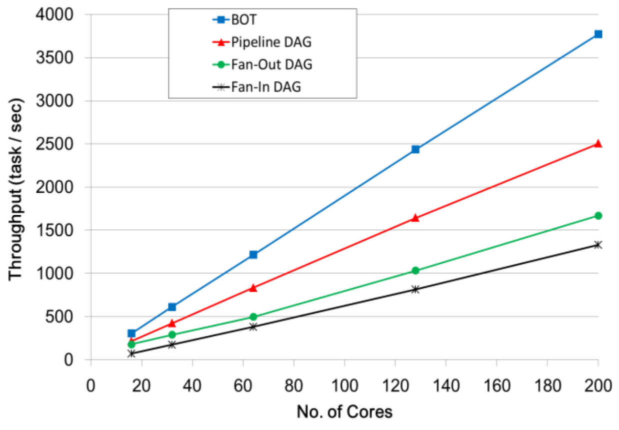


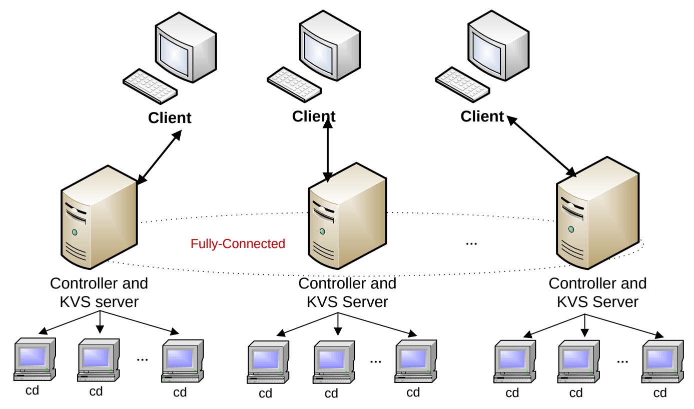
Figure 14: MATRIX runs different Benchmark DAGs

Figure 14 shows the throughput results of all the DAGs up to 200 cores and 200K tasks. We see that for BOT workloads, we can achieve nearly-perfect performance, the throughput numbers imply a 90%+ efficiency for BOT workloads at all scales. This is because tasks are all run locally without requiring any piece of data. For the other three DAGs, our technique shows great scalability, as the throughput doubles when the scale and workload double. The throughput numbers are good, considering the data size and DAG complexities. Out of the three DAGs, Pipeline workloads show the highest throughput, as each task has at most one child and one parent. The data dependency condition is easy to be satisfied. For Fan-Out DAG, our experiments experienced a relatively long ramp-up period, as at the beginning, the number of ready tasks is small. Initially, only the root task is ready to run. As time increases, there would be more and more tasks that are ready, and we had better utilization. For Fan-In DAG, it is quite the opposite. At the beginning, tasks were running very fast. But it would get slower and slower, leading to a very long tail. This is not caused by load imbalance. In the end, it gets more and more difficult for a task to be ready given the Fan-In DAG shape and properties. This very long-tail has worse effect than that is caused by the slow ramp-up period for the Fan-Out DAG.

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

2.4 Distributed Job Scheduling for High-Performance Computing Workloads

Figure 15 shows a partition-based fully-distributed architecture of JMS for large scale HPC MPI and HPC ensemble workloads. Instead of using one centralized scheduler/controller to manage all the compute daemons (cd) as centralized architecture does, there are multiple distributed controllers with each one managing a partition of cd. The controllers are fully-connected meaning that each controller is aware of all the others. The **partition size** (the ratio of the number of controllers to the number of compute daemons) is configurable, and can vary according to the different applications. For example, for a large-scale HPC workload where jobs usually require a large number of nodes to run, we can have each controller manage thousands of cd, so that the jobs are likely to be managed and launched locally (within the partition); for HPC ensemble workloads where jobs have a wide distribution of sizes, we can have heterogeneous partition sizes for each controller to support different HPC ensemble workloads.


Client
Client
Client
Fully-Connected
...
Controller and KVS server
Controller and KVS Server
Controller and KVS Server
cd
cd
cd
...
Figure 15: Distributed scheduling architecture for HPC applications

We show how the distributed KVS can be used in the distributed scheduling architecture for the HPC applications. The KVS could store the following (key, value) pairs, as listed in Table 1. For example, a controller needs to know the free/available resources of all the controllers in order to allocate resources to jobs; for a specific job, a controller needs to know the involved controllers and the compute nodes of each controller that will launch the job, and to release the resources after the job is done. The "controller id" is the identity (e.g. IP address, node name) of each controller; the "list" is a list of node identities; the "job id" is a unique identifier for a specific job. To ensure uniqueness of the "Keys" in the DKVS, we use combinations of "Keys" to generate new "Keys", such as the "job id + original controller id", "job id + original controller id + involved controller id".

Table 1: Job and Resource Data Stored in DKVS

Key	Value	Description
controller id	free node list	Free nodes in a partition
job id	original controller id	The original controller that is responsible for a submitted job
job id + original controller id	involved controller list	The controllers that participate in launching a job
job id + original controller id + involved controller id	participated node list	The nodes in each partition that are involved in launching a job

2.4.1 Resource Stealing

In the partition-based distributed scheduling architecture (*Figure 15*) of JMS, one big challenge is how to allocate the most suitable resource for a specific job and how to dynamically balance the needs of resources (free compute nodes) among all the partitions in order to make the best use of all the resources. The challenge comes from the limitation that each controller just has a local view of its own partition. We propose a resource stealing technique that can adaptively balance the resources at runtime. Resource stealing refers to a set of techniques of stealing resources from other partitions if the local partition cannot satisfy a specific job in terms of job size.

When a controller allocates nodes for a job, it first checks the local free nodes by querying the distributed KVS. If there are enough free nodes, then the controller directly allocates the nodes; otherwise, it allocates whatever resources the partition has, and queries for other partitions to steal resources from them. The simplest and most straightforward resource stealing technique does random resource stealing. The random resource stealing is given in algorithm 4. As long as a job has not been allocated enough nodes, the random resource stealing technique randomly selects a controller and tries to steal free nodes from it. Every time when the selected controller has no available nodes, the launching controller sleeps some time (*sleep_length*) and retries. If the launching controller experiences several failures (*num_retry*) in a row because the selected controller has no free nodes, it will release the resources it has already allocated to avoid the resource deadlock that will otherwise happen in the case where two controllers hold part of the resource for each individual job, but neither can be satisfied. The number of retries and the sleep length after stealing failure are critical to the performance of the algorithm, especially for many big jobs, where all the launching controllers try to allocate nodes and steal resources from each other.

ALGORITHM 4. RANDOM RESOURCE STEALING

```
Input: number of nodes required (num_node_req), number of nodes allocated (*num_node_alloc), number of controllers (num_ctl),
controller membership list (ctl_id[num_ctl]), sleep length (sleep_length), number of reties (num_retry).
Output: list of involved controller ids (list_ctl_id_inv), participated nodes of each involved controller (part_node[]).
num_try = 0; num_ctl_inv = 0; default_sleep_length = sleep_length;
while *num_node_alloc < num_node_req do
    sel_ctl_idx = ctl_id[Random(num_ctl)]; sel_ctl_node = distributed KVS_lookup(sel_ctl_id);
    if (sel_ctl_node != NULL) then
        num_more_node = num_node_req – *num_node_alloc;
again:  num_free_node = sel_ctl_node.num_free_node;
        if (num_free_node > 0) then
            num_try = 0; sleep_length = default_sleep_length;
            num_node_try = num_free_node > num_more_node ? num_more_node : num_free_node;
            list_node = allocate(sel_ctl_node, num_node_try);
            if (list_node != NULL) then
                *num_node_alloc += num_node_try; part_node[num_ctl_inv++] = list_node; list_ctl_id_inv.add(remote_ctl_id);
            else
                goto again;
            end
        else
            if (++num_try > num_retry) do
                release(list_ctl_id_inv, part_node); *num_node_alloc= 0; sleep_length = default_sleep_length;
            else
                sleep(sleep_length); sleep_length *= 2;
            end
        end
    end
end
return list_ctl_id_inv, part_node;
```

The random resource stealing algorithm works fine for jobs whose sizes are less or about the partition size. Besides, the free resources could be balanced among all the partitions quickly and dynamically during runtime due to the distributed and random features of the resource stealing technique. However, for big jobs (e.g. full-scale jobs), or under the case of high system utilization where the system has few free nodes, it may take forever to allocate enough free nodes. We will devise a monitoring-based weakly consistent resource stealing technique in the future work to resolve these big-job allocation and high system utilization issues.

2.4.2 SLURM++: a distributed JMS for HPC MPI and HPC ensemble Workloads

SLURM++ is a partition-based fully-distributed (*Figure 15*) JMS targeting the large-scale HPC MPI applications, and the HPC ensemble workloads. SLURM++ extends the SLURM resource manager by integrating the ZHT KVS for distributed state management in a scalable and resilient way. SLURM++ is comprised of multiple controllers with each one participating in job resource allocation and job launching, while ZHT is used to store all the SLURM daemons' state that is accessible for all controllers. Resource contention is solved by relying on the atomic compare and swap operation of ZHT. SLURM++ has implemented the random resource stealing technique to achieve distributed resource sharing and balancing. We compared SLURM++ with the SLURM resource manager with a variety of micro-benchmarks of different job sizes (small, medium, and large) at modest scales (500-nodes) on the Kodiak cluster at LANL [45]. SLURM++ outperforms

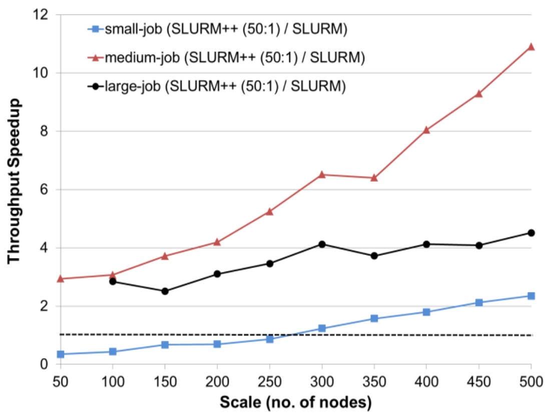


Figure 16: SLURM++ vs. SLURM

SLURM by 10X in terms of throughput. The results are shown in *Figure 16*.

The micro-benchmark contains independent “sleep 0” HPC jobs that require different number of compute nodes per job. The partition size is configured as 50; at the largest scale (500 nodes), the number of controllers is 10. We will use SLURM++ (M:N) to specify the ratio of the number of controller to the number of slurmds, where M is the number of slurmds and N is the number of controllers, such as SLURM++ (50:1), SLURM++ (1:1). We conducted experiments with three workloads: small-job workloads (50 jobs per controller, and job size is 1 node), medium-job workloads (50 jobs per controller, and job size is 1-50 nodes), and big-job workloads (20 jobs per controller, and job size is 25-75 nodes). Figure 16 shows that not only does SLURM++ outperform SLURM in nearly all cases, but the performance slowdown due to increasingly larger jobs at large scale is better for SLURM++ by 2X to 11X depending on the job size. The reason that large jobs perform worse than medium jobs is because the larger the jobs are, the more extensively they compete resources. For small jobs, SLURM++'s performance is bounded by SLURM job launching procedure leading to the smallest improvement. Another fact is that as the scale increases, the throughput speedup is also increasing. This indicates that at larger scales, SLURM++ would outperform SLURM even more.

2.5 Production Systems

The ultimate goal of our work is to run real scientific applications through the job management systems we have implemented, namely MATRIX and SLURM++. We are currently on the way to push both MATRIX and SLURM++ to be production systems. MATRIX is expected to be deployed on the machines in Argonne National Laboratory (ANL) and the Cloud environments, and SLURM++ is going to be used in the production systems at LANL. We should be able to finish the production systems by the time of my Ph.D graduation.

MATRIX will be integrated with the Swift project [7] in ANL and University of Chicago. Swift is a parallel programming system that will help MATRIX interpret large-scale scientific applications as loosely-coupled parallel tasks with the DAG abstractions. MATRIX then schedules the DAGs in a distributed way to achieve high performance. In the Cloud, with the help of data-aware scheduling, we are working on extend MATRIX to support the distributed scheduling of Map/Reduce styled data-intensive workloads, as opposed to the current centralized Hadoop scheduling system [50]. We will utilize distributed file systems, such as FusionS [44] and HDFS [51], to help MATRIX manage data in a distributed, transparent, scalable, and reliable way.

We are also now working on enabling SLURM++ to run real HPC MPI applications. By the time of writing this document, we have been able to run basic MPI applications with SLURM++. We preserved the whole communication layer of SLURM [11] (which supports the scheduling of MPI applications), and modified the original controller (i.e. slurmctld) to be distributed and to use ZHT for distributed resource management. We are testing SLURM++ with more MPI applications at large scales.

3. Related Work

This section introduces the related work of our proposal, which covers a wide range of research topics and areas. The related work could be divided into several aspects, namely traditional batch-scheduling systems, light-weight task scheduling systems, load balancing, data-aware scheduling, and distributed systems simulations.

3.1 Traditional batch-scheduling systems

Traditional batch-scheduling systems are batch-sampled specific for large-scale HPC applications. Examples of these systems are SLURM [11], Condor [12], PBS [13], and Cobalt [52]. SLURM is one of the most popular traditional batch schedulers, which uses a centralized controller (slurmctld) to manage compute nodes that run daemons (slurmd). SLURM does have scalable job launch via a tree based overlay network rooted at rank-0, but as we have shown in our evaluation, the performance of SLURM remains relatively constant as more nodes are added. This implies that as scales grow, the scheduling cost per node increases, requiring coarser granularity workloads to maintain efficiency. SLURM also has the ability to configure a “fail-over” server for resilience, but this doesn’t participate unless the main server fails. Condor was developed as one of the earliest JMSs, to harness the unused CPU cycles on workstations for long-running batch jobs. Portable Batch System (PBS) was originally developed at NASA Ames to address the needs of HPC, which is a highly configurable product that manages batch and inter-active jobs, and adds the ability to signal, rerun and alter jobs. LSF Batch [53] is the load-sharing and batch-queuing component of a set of workload-management tools. All of these systems are designed for either HPC or HTC workloads, and generally have high scheduling overheads. Other JMSs, such as Cobalt [52], typically used on supercomputers, lack the granularity of scheduling jobs at node/core level. All of them have centralized architecture with a single controller managing all the compute daemons. They take scheduling algorithms as priority over supporting fine-grained tasks. The centralized architecture works fine for HPC environment where the number of jobs is not large, and each job is big. As the distributed system scales to exascale, and the applications become more fine-grained, these centralized schedulers have bottleneck in both scalability and availability. The ubiquitous batch scheduler, SLURM, reports maximum throughput of 500 jobs/sec, which does not meet the need of millions tasks / sec required by exascale computing.

3.2 Light-weight task scheduling systems

Light-weight task scheduling systems have also been under development over the last several years, such as Falkon [15], Mesos [54], Omega [55], Sparrow [46], CloudKon [47] and Yarn [56]).

Falkon [15] is a centralized task scheduler with the support of hierarchical scheduling for MTC applications, which can scale and perform magnitude orders better than centralized batch schedulers. However, it has problems to scale to even a petascale system, and the hierarchical implementation of Falkon suffered from poor load balancing under failures or unpredictable task execution times.

Mesos [54] is platform for sharing resource between multiple diverse cluster computing frameworks to schedule tasks. Mesos allows frameworks to achieve data-locality by taking turns reading data stored on each machine. Mesos uses delay scheduling policy, and frameworks wait for a limited time to acquire nodes storing their data. However, this approach causes significant waiting time before a task could be scheduled, especially when the required data is large.

Sparrow [46] is similar to our work in that it implemented distributed load balancing for weighted fair shares, and supported the constraint that each task needs to be co-resident with input data, for fin-grained sub-second tasks. However, in Sparrow, each scheduler is aware of all the compute daemons, this design can cause a lot of resource contentions when the number of tasks are large. What’s more, Sparrow implements pushing mechanism with early binding of tasks to workers. Each scheduler probes multiple compute nodes and assigns tasks to the least overloaded one. This mechanism suffers long-tail problem under heterogeneous workloads [40] due to early binding of tasks to worker resources. We have compared Sparrow and the basic MATRIX without data-aware scheduling technique using heterogeneous workloads in [47], and MATRIX outperforms Sparrow by 9X. Furthermore, there is an implementation barrier with Sparrow as it is developed in Java, which has little support in high-end computing systems.

CloudKon [47] has similar architecture as MATRIX, except that CloudKon focuses on the Cloud environment, and relies on the Cloud services, SQS [48] to do distributed load balancing, and DynamoDB [25] as the DKVS to keep task metadata. Relying on the Cloud services could facilitate the easier development, at the cost of potential performance and control. Furthermore, CloudKon has dependencies on both Java and Cloud services, which makes its adoption in high-end computing impractical.

3.3 Load balancing

Most parallel programming systems require load balancing, which can be used to optimize resource utilization, data movement, power consumption, or any combination of these. Load balancing strategies can be divided into two broad categories – those for applications where new tasks are created and scheduled during execution (i.e. task scheduling) and those for iterative applications with persistent load patterns [57]. Centralized load balancing has been extensive studied in the past (JSQ [58], least-work-left [59], SITA [60]), but they all suffered from poor scalability and resilience.

Distributed Load balancing is the technique of distributing computational and communication loads evenly across processors of a parallel machine, or across nodes of a supercomputer, so that no single processor or computing node is overloaded. Clients would be able to submit work to any queue, and each queue would have the choice of executing the work locally, or forwarding the work to another queue based on some function it is optimizing. Although distributed load balancing at extreme scales of millions of nodes and billions of threads of execution is likely a more scalable and resilient solution, there are many challenges that must be addressed (e.g. utilization, partitioning). Fully distributed strategies have been proposed, including neighborhood averaging scheme (ACWN) [61][62][63][64] In [64], several distributed and hierarchical 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 [63] and [65].

Charm++ [57] supports centralized, hierarchical and distributed load balancing. It has demonstrated that centralized strategies work at scales of thousands of processors for NAMD. In [57], 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 [66] 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 [67] 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.

Work stealing refers to a distributed load balancing approach in which processors needing work steal computational tasks from other processors. Work stealing has been used at small scales successfully in parallel languages such as Cilk [68], to load balance threads on shared memory parallel machines [29][30][69]. Theoretical work has proved that a work-stealing scheduler can achieve execution space, time, and communication bounds all within a constant factor of optimal [29][69]. However, 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 [30].

The work done by Diana et. al in [30] has 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 has scaled work stealing to over 150K cores by utilizing the persistence principle iteratively to achieve the load balancing of task-based applications [70]. However, through simulations, our work shows that work stealing with optimal parameters works well at exascale levels with 1-billion cores.

3.4 Data-aware scheduling

Falkon implemented a data diffusion approach [38] to schedule data-intensive workloads. Data diffusion acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. However, Falkon used a centralized index server to store the metadata, as opposed to our distributed key-value store, which leads to poor scalability.

Quincy [71] is a flexible framework for scheduling concurrent distributed jobs with fine-grain resource sharing. Quincy tries to find optimal solutions of scheduling jobs under data-locality and load balancing constraints by mapping the problem to a graph data structure. Even though the data-aware motivation of Quincy is similar to our work, it takes significant amount of time to find the optimal solution of the graph that combines both load balancing and data-aware scheduling.

Dryad [72] is a general-purpose distributed execution engine for coarse-grained data-parallel applications. Dryad is similar with our work in that it supports running of applications structured as workflow DAGs. However, like the Hadoop scheduler [50], Dryad does centralized scheduling with a centralized metadata management that greedily maps tasks to the where the data resides, which is neither fair nor scalable.

SLAW [73] is a scalable locality-aware adaptive work stealing scheduler that supports both work-first and help-first policies [74] adaptively at runtime on a per-task basis. Though SLAW aimed to address issues (e.g. locality-obliviousness, fixed task scheduling policy) that limit the scalability of work stealing, it focuses on the core/thread level. The technique would unlikely to hold for large-scale distributed systems.

Another work [75] that did data-aware work stealing is similar to us in that it uses both dedicated and share queues. However, it relies on the X10 global address space programming model [76] to statically expose the data-locality information and distinguish between location-sensitive and location-flexible tasks at beginning. Once the data-locality information of a task is defined, it remains unchanged. This is not adaptive to various data-intensive workloads.

3.5 Distributed systems simulations

There are a vast number of distributed and peer-to-peer system simulation frameworks, which are typically performed at two different levels of abstraction: application level, which treats the network as just a series of nodes and edges, such as GridSim [77], SimGrid [34] and PeerSim [43]; and packet level, which models the details of the underlying network, such as OMNET++ [78], and OverSim [79].

SimGrid [34] provides functionalities for the simulation of distributed applications in heterogeneous distributed environments. SimGrid now uses PDES and claims to have 2M nodes' scalability. However, it has consistency challenge and is unpredictable. It is neither suitable to run exascale MTC applications, due to the complex parallelism. GridSim [77] is developed based on SimJava [80] and allows simulation of entities in parallel and distributed computing systems, such as users, resources, and resource brokers (schedulers). A resource can be a single processor or multi-processor with shared or distributed memory and managed by time or space shared schedulers. However, GridSim and SimJava use multi-threading with one thread per simulated element (cluster), this heavy-weight threading property makes them impossible to reach extreme scales of millions nodes or billions of cores on a single shared-memory system. Our previous work [37] showed that GridSim could simulate no more than 256 nodes, and SimGrid simulated only 64K nodes while consuming 256GB memory.

PeerSim [43] is a peer to peer system simulator that supports for extreme scalability and dynamicity. Among the two simulation engines it provides, we use the discrete-event simulation (DES) engine because it is more scalable and realistic compared to the cycle-based one for large-scale simulations. OMNeT++ [78] is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators. Built on top of OMNeT++, OverSim [79] uses DES to simulate exchange and processing of network messages. All of PeerSim, OMNet++ and OverSim have standard Chord [42] protocol implementation, we compared them and found that PeerSim is much faster and consumes much less memory.

4. Accomplishments and Conclusions

We highlight the current state of the proposed work towards developing next-generation job management system for extreme-scale computing. The papers that have been published are an important metric to measure the progress and to highlight the accomplishments achieved so far. We then draw the conclusions achieved so far.

4.1 Accomplishments

So far, we have achieved several accomplishments. We devised the generic taxonomy for distributed system services; we proposed that key-value store is a viable building block for extreme-scale system service; we have built a systematic discrete simulation environment which helps us develop simulators fast and efficient; we have developed SimMatrix, a key-value store simulator, and a simulator of SLURM++ (refereed as SimSLURM++). We then scaled these simulators up to exascale with millions of nodes. For the real implementations, we have prototyped two job management systems, namely MATRIX for fine-grained MTC applications, and SLURM++ for HPC MPI and HPC ensemble workloads. We have scaled MATRIX up to 1024 nodes (4096 cores), and SLURM++ up to 500 nodes using benchmarking workloads and some easy application workloads. The papers and documents that have been published based on our working progress are listed as follows:

Conference Papers:

(1) **Ke Wang**, Xiaobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, Ioan Raicu. “Optimizing Load Balancing and Data-Locality with Data-aware Scheduling”, IEEE BigData, 2014.

(2) **Ke Wang**, Xiaobing Zhou, Hao Chen, Michael Lang, Ioan Raicu. “Next Generation Job Management Systems for Extreme Scale Ensemble Computing”, ACM HPDC, 2014.

(3) **Ke Wang**, Abhishek Kulkarni, Michael Lang, Dorian Arnold, Ioan Raicu. “Using Simulation to Explore Distributed Key-Value Stores for Extreme-Scale Systems Services”, IEEE/ACM Supercomputing/SC 2013.

(4) **Ke Wang**, Kevin Brandstatter, Ioan Raicu. “SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascale”, ACM HPC 2013.

(5) **Ke Wang,** Anupam Rajendran, Kevin Brandstatter, Zhao Zhang, Ioan Raicu. “Paving the Road to Exascale with Many-Task Computing”, Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012.

(6) Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, **Ke Wang**, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. "FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems", IEEE International Conference on Big Data 2014

(7) Iman Sadooghi, Sandeep Palur, Ajay Anthony, Isha Kapur, Karthik Belagodu, Pankaj Purandare, Kiran Ramamurty, **Ke Wang**, Ioan 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.

(8) Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, **Ke Wang**, Anupam Rajendran, Zhao Zhang, Ioan Raicu. "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013.

(9) Dongfang Zhao, Da Zhang, **Ke Wang**, Ioan Raicu. "RXSim: Exploring Reliability of Exascale Systems through Simulations", ACM HPC 2013.

Workshop Papers:

(10) **Ke Wang**, Zhangjie Ma, Ioan Raicu. "Modelling Many-Task Computing Workloads on a Petaflop IBM Blue Gene/P Supercomputer", IEEE CloudFlow 2013.

Extended Abstracts and Posters:

(11) **Ke Wang**, Ioan Raicu. "Achieving Data-Aware Load Balancing through Distributed Queues and Key/Value Stores," 3rd Greater Chicago Area System Research Workshop (GCASR), 2014.

(12) **Ke Wang,** Abhishek Kulkarni, Xiaobing Zhou, Michael Lang, Ioan Raicu. "Using Simulation to Explore Distributed Key-Value Stores for Exascale System Services", 2nd Greater Chicago Area System Research Workshop (GCASR), 2013.

(13) **Ke Wang,** Kevin Brandstatter, Ioan Raicu. "SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascales", 1st Greater Chicago Area System Research Workshop (GCASR), 2012.

(14) Anupam Rajendran, **Ke Wang,** Ioan Raicu. "MATRIX: MAny-Task computing execution fabRIc for eXtreme scales", 2nd Greater Chicago Area System Research Workshop (GCASR), 2013.

(15) Abhishek Kulkarni, **Ke Wang**, Michael Lang. "Exploring the Design Tradeoffs for Exascale System Services through Simulation", LANL Summer Students Poster Session held at LANL during August, 2012.

Journal Papers under Review:

(16) **Ke Wang,** Anupam Rajendran, Xiaobing Zhou, Kiran Ramamurthy, Iman Sadooghi, Michael Lang, Ioan Raicu. "Distributed Load-Balancing with Adaptive Work Stealing for Many-Task Computing on Billion-Core Systems", under review at Journal of ACM Transactions on Parallel Computing (TOPC), 2014.

(17) **Ke Wang,** Dorian Arnold, Michael Lang, Ioan Raicu. "Design by Simulation: System Services for Exascale Systems", under review at Journal of IEEE Transactions on Parallel and Distributed Systems (TPDS), 2014.

(18) **Ke Wang,** Kan Qiao, Prasanna Balasubramani, Tonglin Li, Iman Sadooghi, Xiaobing Zhou, Michael Lang, Ioan Raicu. "Load-balanced and Locality-aware Scheduling for Data-Intensive Workloads at Extreme-Scales", Journal of Cluster Computing, SI on Big Data, 2014.

Also, there are some **technical reports** ([21][84][85][86]) that we have published online.

4.2 Conclusions

According to the progress we have achieved so far, we are able to make the following conclusions.

Fully distributed architectures are Scalable: We showed that the fully-distributed architectures are potential to scale towards extreme-scales through simulations. One architecture is the 1:1 mapping fine-grained fully-distributed architecture for MTC applications. We simulated this architecture through SimMatrix, and scaled it up to millions of nodes, billions of cores and hundreds of billions of tasks. The other architecture is the partition-based fully-distributed architecture of JMS for HPC MPI and HPC ensemble workloads. We simulated this architecture through the key-value store simulations up to millions of nodes, and concluded that this is scalable with moderate amount of messages of distributed features. These two architectures have been validated through the MATRIX job scheduling system up to 1204 nodes , and the SLURM++ resource management system up to 500 nodes, respectively.

Key-value stores are a building block for extreme-scale system services: We motivated that key-value stores (KVS) are a viable building block for extreme-scale HPC system services. This statement lays the foundations for developing distributed system services that are highly available, scalable and reliable at extreme-scales. We decomposed the services into their basic building blocks and devised a general taxonomy for classifying HPC system services. The taxonomy gives us a systematic way to decompose services into their basic building block components, allows us to categorize services based on the features of these building block components, and suggests the configuration space to consider for evaluating service designs via simulation or implementation. We defined four major service architectures based on the service taxonomy, and narrowed this taxonomy down for key-value stores. We explored different design choices through the KVS simulations. We simulated KVS up to millions of nodes on a shared-memory machine, and studied different architectures (i.e. centralized, hierarchical and distributed) and distributed features (i.e. failure, replication, consistency models). The simulator was validated against existing distributed KVS-based services. Via simulation, we demonstrated how distributed features affected performance at scales. We also emphasized the general use of KVS to HPC services by feeding real service workloads into the simulator. According to this conclusion, we then implemented MATRIX and SLURM++, both of which are using the distributed key-value store, ZHT, to store the important resource and job/task metadata information. Anther system that demonstrated the general use of distributed KVS towards the development of highly scalable services is the FusionFS distributed file systems. FusionFS applied ZHT to store all the metadata of files and the provenance of the file system.

Work stealing is a scalable distributed load balancing technique: We proved that the proposed adaptive work stealing technique is scalable up to exascale with 1-million nodes, 1-billion cores, and 100-billion tasks through SimMatrix simulations. We identified the important parameters that are important to the performance of work stealing. The parameters are number of tasks to steal, number of neighbors, static neighbor vs. dynamic random neighbors, and the polling interval. We then conducted simulations to find out the optimal parameter configurations that push work stealing to exascale. The suggested optimal parameters for the adaptive work stealing are to steal half the number of tasks from their neighbors, and to use the square root number of dynamic random neighbors. Even though the optimal parameter space may not hold for all the application workloads, our work stealing technique has the ability to change these parameters dynamically at runtime in a decentralized way. Each scheduler observes the system state at runtime according to whether the current work stealing succeeds or not. If a scheduler steals tasks successfully, then it keeps the optimal parameter configuration; otherwise, if a scheduler experiences several failures in a row, it can change the suggested parameters. The ability of self-adjustments would not only help work stealing achieve much higher scalability, but expand the application spaces that work stealing could apply. We implemented the work stealing technique in the MATRIX scheduler, and the evaluation of MATRIX has validated our conclusion. Also, with the help of distributed KVS, work stealing would achieve distributed data-aware scheduling for data-intensive applications.

Exascale applications are diverse: Even though we have not run many applications in our simulations and real implementations, we conclude that the scientific application for exascale computing are becoming more diverse covering a wide range of categories, namely the large-scale HPC MPI application, the HPC ensemble workloads, and the fine-grained loosely-coupled MTC applications. We investigated the largest available traces of real MTC workloads, collected over a 17-month period comprising of 173M tasks [38][39][40]. We filtered out the logs to isolate only the 160K-core BG/P machine, which netted about 34.8M tasks with the minimum runtime of 0 seconds, maximum runtime of 1469.62 seconds, average runtime of 95.20 seconds, and standard deviation of 188.08. We plotted the Cumulative Distribution Function (CDF) of the 34.8M tasks in Figure 17. Most of the tasks have the lengths ranging from several seconds to hundreds of seconds, and the medium task length is about 30 seconds (just one third of the average: 95.2 seconds). We have also evaluated the data-aware work stealing technique using the image stacking application in astronomy and all-pairs application in biometrics. We have started to look into the HPC MPI applications and the MPI ensemble workloads on the previous IBM Blue Gene/P supercomputers in ANL and through the PETSc tool [81] (Portable, Extensible Toolkit for Scientific Computation that encapsulate MPI underneath) developed by ANL. There are also applications in Los Alamos National Laboratory for the HPC ensemble workloads.

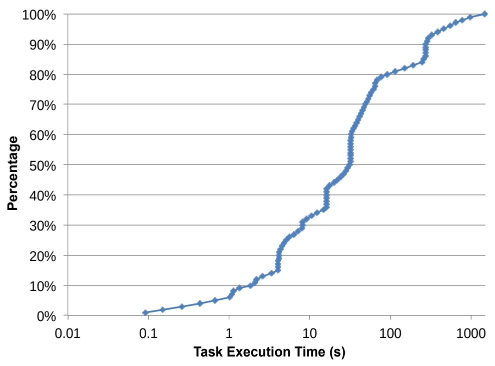


Figure 17: Cumulative Distribution Function of the MTC workloads

5. Future Work

The achievements we have accomplished to data have laid the foundations for a broad yet clear sort of future work. All the future work aims to push our job management systems to scalable production systems that could run a variety of applications at extreme-scales efficiently. We first list all the future directions of our work. Then we present our plans for the future publications, which are followed by a rough timeline of our work towards the final dissertation.

5.1 Future Directions

The future directions of our work are listed as follows:

Theoretical Analysis: We have proposed several techniques for addressing the significant challenges of distributed scheduling of next generation job management systems for extreme-scale computing. Those techniques include the random resource stealing algorithm that achieves the distributed resource balancing for distributed HPC scheduling, the work stealing technique to achieve the distributed MTC scheduling, and the data-aware work stealing technique for the data-intensive applications. Even though we have evaluated and implemented these techniques through either simulations and real systems using specific workloads, we still do not know to what extend these technique could work well for general workloads. We have no clue about the complexity (time and space complexities) of these techniques. We are not sure yet whether there exists lower or upper bounds of the complexity of our techniques for the general workloads. We plan to conduct theoretical analysis from the mathematics' perspective, and focus on the time complexities of these techniques, such as the theoretical analysis we have done for the Algorithm 1 (multiple random neighbor selection algorithm). We hope to understand whether there are bottlenecks in these techniques, which would drive us to devise more scalable and efficient techniques.

Efficient Resource Stealing Techniques for SLURM++: The random resource stealing algorithm that achieves the distributed resource balancing for the partition-based distributed HPC scheduling works fine for jobs whose sizes are less or about the partition size. Besides, the free resources could be balanced among all the partitions quickly and dynamically during runtime due to the distributed and random features of the resource stealing technique. However, for big jobs (e.g. full-scale jobs), or under the circumstance that the system is under high system utilization, we have experienced extremely long time to allocate enough free nodes. This is because the totally random property has no knowledge of the free resources in the whole system. This would not be a problem for the random work stealing algorithm as we expect that the total number of tasks is magnitudes more than the number of free resources. Besides conducting a theoretical analysis about this technique, we are seeking other more efficient resource stealing techniques in SLURM++. One alternative is using monitoring of global state combined with weakly consistent distributed states of each controllers. The ideal is as follows. There would be a monitoring service that keeps polling the free resources of all the partitions periodically, and puts the global resource state in the ZHT as one (key, value) pair. Each controller will then query this (key, value) pair and have a global view of the whole system resource state stored periodically. Based on this (key, value) pair, each controller stores all the number of free resources of all the partitions as a binary-search tree (BST) data structure. Then each controller would find the most suitable resources that satisfy a job, and update the BST data structure. Each controller would have a weakly consistent view of the global system resource state compared to that of the monitoring service. This solution would improve the performance of the resource stealing as it has weakly consistent global view of the system state, as opposed to the non-knowledge of the random technique. We are implementing this technique in SLURM++ and hope to resolve the problem of the random technique.

Extreme-Scales: Our goal is to develop the next-generation job management systems for extreme-scales. Currently, SLURM++ has scaled up to 500 nodes, while MATRIX has scaled up to 1024 nodes. These scales are not even close to the extreme-scales with millions of nodes, or the largest scale of today's supercomputers. Even though we have scaled the architectures and the proposed techniques up to millions of nodes through simulations, it would be arbitrarily to conclude that SLURM++ and MATRIX could have the same scalability. Actually, neither systems even have the same scalability as the ZHT KVS (scaled up to 8K nodes) even though both systems are using ZHT. The reason is that SLURM++ and MATRIX use their own network communication layers which are not robust as that of ZHT. The ideal to scale MATRIX and SLURM++ up to at least the same scale of ZHT is to decompose the ZHT communication layer as a standalone network library that could be used by any system service, at least by MATRIX. As SLURM++ uses the same network communication patterns as the SLURM resource manager, we will expect SLURM++ to have the same scalability as SLURM that has been deployed on several of today's largest supercomputers. We will continue to scale both MATRIX and SLURM++ on several large distributed systems, such as the

IBM BG/Q machine at ANL that has 768K cores with 3M hardware threads, and the Blue Water machine at UIUC that has 49K CPUs.

Applications: Our ultimate goal is to eventually apply the distributed job management systems to run all the ensemble of scientific applications for optimized performance.

The plan to enable MATRIX to support large-scale MTC applications is to integrate MATRIX with the Swift. Swift is a parallel programming system and workflow engine for MTC applications that cover a variety of disciplines, ranging from Biology, Earth Systems, Physical Chemistry, Astronomy, to Neuroscience, Economics, and Social Learning and so on. Swift will serve as the high-level data-flow parallel programming language between the applications and MATRIX. Swift would essentially output many parallel and/or loosely coupled distributed jobs/tasks with the necessary task dependency information, and submit them to MATRIX. Instead of having Swift manage the DAG, the DAG would be managed in a distributed way by MATRIX through the ZHT distributed KVS. Swift has been scaled to distribute up to 612 million dynamically load balanced tasks per second at scales of up to 262,144 cores [82]. This extreme scalability would absolutely advance the progress of making MATRIX supporting large-scale scientific application at extreme-scales.

Another direction is to push MATRIX in the elastic Internet and Cloud environment to run the MapRedcue-style workloads. The current MapReduce framework has a centralized task dispatcher to dispatch the data-intensive workloads to mappers that have data for specific tasks. We will extend MATRIX to support the MapReduce framework to enable distributed scheduling for the MapReduce data-intensive workloads. We will utilize distributed file systems, such as FusionFS [44], to help MATRIX implement data locality and the data-aware scheduling with the help of ZHT KVS. MATRIX + FusionFS will be the combination of distributed version of MapReduce framework, as opposed to the current centralized Hadoop + HDFS combination. The applications will come from our collaborator in the Hortonworks Inc that implemented the YARN [56] task scheduler for the data-intensive Hadoop workloads.

For SLURM++, we are going to focus on the large-scale HPC MPI jobs and the HPC ensemble applications. We are going to look into the applications running on the previous IBM Blue Gene/P supercomputer and the current Blue Gene/Q supercomputer in ANL, as well as the large-scale numeric applications programmed with MPI through the PETSc tool [81] (Portable, Extensible Toolkit for Scientific Computation that encapsulate MPI underneath) developed by ANL. There are also applications in Los Alamos National Laboratory for the HPC ensemble workloads.

5.2 Plans for future publications

We have clear plans to submit for publications during the period of the next year. We list each planned application with the research topic as follows.

(1) Theoretical Analysis and Efficient resource stealing technique for SLURM++; IEEE TPDS (November, 2015)

(2) MATRIX with Hadoop integration for the Cloud environment; ACM HPDC'15 (January, 2015)

(3) SLURM++ running the HPC MPI workloads at scales; IEEE/ACM SC'15 (April, 2015)

(4) Integrate all work into a journal about scheduling and scientific applications; (July, 2015)

5.3 Timeline

The timeline of my future work of the next year is roughly based on the publication plans. I will take my Ph.D comprehensive exam in October, 2014. A year after that, I plan to do my Ph.D final dissertation defense. The steps are listed in Table 2.

Table 2: Tasks toward the final Ph.D dissertation defense

Data	Description
9/19/2014	Comprehensive proposal sent to committee
10/07/2014	Comprehensive exam
08/07/2015	Dissertation Draft sent to committee
10/14/2015	Dissertation Defense

6. References

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

[2] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. “Exascale computing study: Technology challenges in achieving exascale systems”, 2008.

[3] I. Raicu, Y. Zhao, I. Foster. “Many-Task Computing for Grids and Supercomputers”, 1st IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) 2008.

[4] Ke Wang, Xiaobing Zhou, Hao Chen, Michael Lang, Ioan Raicu. “Next Generation Job Management Systems for Extreme Scale Ensemble Computing”, ACM HPDC 2014.

[5] I. Raicu, P. Beckman, I. Foster, “Making a Case for Distributed File Systems at Exascale,” ACM Workshop on Large-scale System and Application Performance (LSAP), 2011.

[6] M. Snir, S.W. Otto, S.H. Lederman, D.W. Walker, J. Dongarra, “MPI: The Complete Reference,” MIT Press, 1995.

[7] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde, “Swift: Fast, Reliable, Loosely Coupled Parallel Computation,” IEEE Workshop on Scientific Workflows, 2007.

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

[9] D. Abramson, J. Giddy, L. Kotler, “High performance parametric modeling with Nimrod/G: Killer application for the global grid,” In Proc. International Parallel and Distributed Processing Symposium, 2000.

[10] I. Raicu, I. Foster, M. Wilde, Z. Zhang, A. Szalay, K. Iskra, P. Beckman, Y. Zhao, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, D. Thain, “Middleware Support for Many-Task Computing,” Cluster Computing Journal, 2010.

[11] M.A. Jette, A.B. Yoo, M. Grondona, “SLURM: Simple Linux utility for resource management,” 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003), pp. 44–60, Seattle, Washington, USA, June 24, 2003.

[12] 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.

[13] B. Bode, D.M. Halstead, et. al, “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters,” Usenix, 4th Annual Linux Showcase & Conference, 2000.

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

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

[16] M. Jette and Danny Auble, “SLURM: Resource Management from the Simple to the Sophisticated,” Lawrence Livermore National Laboratory, SLURM User Group Meeting, October 2010.

[17] 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.

[18] 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.

[19] R. D. Blumofe and C. Leiserson. “Scheduling multithreaded computations by work stealing”, In Proc. 35th Symposium on Foundations of Computer Science (FOCS), pages 356–368, Nov. 1994.

[20] Ke Wang, Xiaobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, Ioan Raicu. "Optimizing Load Balancing and Data-Locality with Data-aware Scheduling", IEEE BigData 2014.

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

[22] N. Ali, P. Carns, K. Iskra, et al. Scalable I/O Forwarding Framework for High-Performance Computing Systems. In CLUSTER, pages 1-10, 2009.

[23] A. Vishnu, A. R. Mamidala, H. Jin, and D. K. Panda. Performance Modeling of Subnet Management on Fat Tree InfiniBand Networks using OpenSM. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18 - Volume 19, IPDPS '05, pages 296.2–, Washington, DC, USA, 2005.

[24] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In in: Proc. IEEE/ACM Supercomputing '03, 2003.

[25] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, SOSP '07, 2007.

[26] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev., 44:35–40, April 2010. ISSN: 0163-5980.

[27] A. Feinberg. Project Voldemort: Reliable Distributed Storage. ICDE, 2011.

[28] M. A. Heroux. Toward Resilient Algorithms and Applications, April 2013. Available from http://www.sandia.gov/~maherou/docs/HerouxTowardResilientAlgsAndApps.pdf

[29] 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.

[30] 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.

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

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

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

[34] H. Casanova, A. Legrand, and M. Quinson. "SimGrid: a Generic Framework for Large-Scale Distributed Experiments." In 10th IEEE International Conference on UKSIM/EUROSIM'08.

[35] J. Banks, J. Carson, B. Nelson and D. Nicol. Discrete-event system simulation - fourth edition. Pearson 2005.

[36] J. Liu. Wiley Encyclopedia of Operations Research and Management Science, chapter Parallel discrete-event simulation, 2009.

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

[38] Ioan Raicu, Ian T. Foster, Yong Zhao, Philip Little, Christopher M. Moretti, Amitabh Chaudhary and Douglas Thain. The quest for scalable support of data-intensive workloads in distributed systems. HPDC'09 Proceedings of the 18th ACM international symposium on High performance distributed computing, 207-216. ACM New York, NY, USA. DOI:http://dx.doi.org/10.1145/1551609.1551642.

[39] Ioan Raicu, Ian Foster, Mike Wilde, Zhao Zhang, Kamil Iskra, Peter Beckman, Yong Zhao, Alex Szalay, Alok Choudhary, Philip Little, Christopher Moretti, Amitabh Chaudhary and Douglas Thain. Middleware support for many-task computing. Journal of Cluster Computing, Volume 13 Issue 3, September 2010, 291-314. Kluwer Academic Publishers Hingham, MA, USA. DOI:http://dx.doi.org/10.1007/s10586-010-0132-9.

[40] Ke Wang, Zhangjie Ma and Ioan Raicu. Modelling Many-Task Computing Workloads on a Petaflop IBM Blue Gene/P Supercomputer. 2nd International Workshop on Workflow Models, Systems, Services and Applications in the Cloud (CloudFlow 2013) of IPDPS, Massachusetts USA, May 20-24, 2013.

[41] T. Haerder and A. Reuter. 1983. Principles of transaction-oriented database recovery.ACM Comput. Surv. 15, 4, December 1983, 287-317.

[42] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev., 31(4):149–160, August 2001.

[43] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09), 2009.

[44] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Phil Carns, Robert Ross, and Ioan Raicu. FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems. IEEE BigData, 2014.

[45] G. Grider. "Parallel Reconfigurable Observational Environment (PRObE)," available from http://www.nmc-probe.org, October 2012.

[46] 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.

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

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

[49] Ioan Raicu, Yong Zhao, Ian T. Foster, and Alex Szalay. Accelerating large-scale data exploration through data diffusion. In Proceedings of the 2008 international workshop on Data-aware distributed computing (DADC '08).

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

[51] 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.

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

[53] LSF: http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/index.html, 2014.

[54] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. NSDI'11 Proceedings of the 8th USENIX conference on Networked systems design and implementation. USENIX Association Berkeley, CA, USA. 2011.

[55] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. EuroSys '13 Proceedings of the 8th ACM European Conference on Computer Systems. Pages 351-364. ACM New York, NY, USA, 2013. DOI:http://dx.doi.org /10.1145/ 2465351.2465386.

[56] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13). ACM, New York, NY, USA.

[57] 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.

[58] 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

[59] M. Harchol-Balter. Job placement with unknown duration and no preemption, ACM SIGMETRICS Performance Evaluation Review, Volume 28, Number 4, pages 3-5, 2001

[60] E. Bachmat, H. Sarfati. Analysis of size interval task assignment policies, ACM SIGMETRICS Performance Evaluation Review, Volume 36, Number 2, pages 107-109, 2008.

[61] 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

[62] W. W. Shu and L. V. Kal ́e. A dynamic load balancing strategy for the Chare Kernel system. In Proceedings of Supercomputing ’89, pages 389–398, November 1989

[63] 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

[64] 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

[65] 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.

[66] 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.

[67] 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.).

[68] 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.

[69] 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.

[70] J. Liander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In HPDC, 2012.

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

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

[73] 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.

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

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

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

[77] R. Buyya and M. Murshed. “GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing,” The Journal of Concurrency and Computation: Practice and Experience (CCPE), Volume 14, Issue 13-15, Wiley Press, Nov.-Dec., 2002.

[78] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops, Simutools ’08, pages 60:1– 60:10, Marseille, France, 2008.

[79] I. Baumgart, B. Heep, and S. Krause. Oversim: A flexible overlay network simulation framework. In IEEE Global Internet Symposium, 2007.

[80] Kreutzer, W., Hopkins, J. and Mierlo, M.v. “SimJAVA - A Framework for Modeling Queueing Networks in Java.” Winter Simulation Conference, Atlanta, GA, 7-10 December 1997. pp. 483-488.

[81] Argonne National Laboratory, PetSc, http://www.mcs.anl.gov/petsc/, 2014.

[82] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, Ian T. Foster Compiler techniques for massively scalable implicit task parallelism Proc SC 2014.

[83] 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.

[84] Ke Wang, Juan Carlos Hernández Munuera, Ioan Raicu, Hui Jin. "Centralized and Distributed Job Scheduling System Simulation at Exascale", Illinois Institute of Technology, Department of Computer Science, Technical Report, 2011.

[85] Kiran Ramamurthy, Ke Wang, Ioan Raicu. "Exploring Distributed HPC Scheduling in MATRIX", Illinois Institute of Technology, Department of Computer Science, Technical Report, 2013.

[86] Xiaobing Zhou, Hao Chen, Ke Wang, Michael Lang, Ioan Raicu. "Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System", Illinois Institute of Technology, Department of Computer Science, Technical Report, 2013.

[87] Anupam Rajendran, Ke Wang, Ioan Raicu. "MATRIX: MAny-Task computing execution fabRIc for eXtreme scales", 2nd Greater Chicago Area System Research Workshop (GCASR), 2013.

[88] Ke Wang, Ioan Raicu. "Achieving Data-Aware Load Balancing through Distributed Queues and Key/Value Stores," 3rd Greater Chicago Area System Research Workshop (GCASR), 2014.

[89] Abhishek Kulkarni, Ke Wang, Michael Lang. "Exploring the Design Tradeoffs for Exascale System Services through Simulation", LANL Summer Students Poster Session held at LANL during August, 2012.

[90] Dongfang Zhao, Da Zhang, Ke Wang, Ioan Raicu. "RXSim: Exploring Reliability of Exascale Systems through Simulations", ACM HPC 2013.

[91] Ke Wang, Anupam Rajendran, Kevin Brandstatter, Zhao Zhang, Ioan Raicu. "Paving the Road to Exascale with Many-Task Computing", Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012.

[92] Ke Wang, Abhishek Kulkarni, Xiaobing Zhou, Michael Lang, Ioan Raicu. "Using Simulation to Explore Distributed Key-Value Stores for Exascale System Services", 2nd Greater Chicago Area System Research Workshop (GCASR), 2013.