SCALABLE RESOURCE MANAGEMENT SYSTEM SOFTWARE FOR EXTREME-SCALE DISTRIBUTED SYSTEMS

BY

KE WANG

DEPARTMENT OF COMPUTER SCIENCE

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved ___________________________
Adviser

Chicago, Illinois
July 2015

ACKNOWLEDGEMENT

I would like to express the deepest appreciation to my research advisor Dr. Ioan Raicu for his consistent support, patience, and guidance to my research work, which lead to the success of this dissertation. I still remember the very first day when I came to talk to Dr. Raicu to express my interest in doing research, his enthusiasm in explaining the research topics and impacts had inspired me and driven me to devote years to making long term research contributions. Dr. Raicu has taught me a lot, including technical knowledge, the ways to find solutions to research problems, how to write research paper, even how to efficiently use basic tools, such as Word and Excel. I hope that the research contributions we have made together will eventually be beneficial to the whole community of distributed systems.

This dissertation would not be possible without the support of Michael Lang, a scientist of Los Alamos National Laboratory (LANL), who was my supervisor during my one-year internship in LANL. We used to brainstorming on research topics and solutions, and Lang was always energetic and encouraging. After the internship, we have been keeping in touch and collaborating on research, and Lang offers continuous support and always gives good advices. I highly express my gratitude to Lang for his support.

I would also like to thank all the other committee members, namely Dr. Xian-He Sun, Dr. Dong (Kevin) Jin, and Dr. Yu Cheng, for their generous helps in coordinating the proposal and final defense dates, and lightening suggestions in writing this dissertation.

In addition, I would like to thank all my colleagues and collaborators for their time spent and efforts made in countless meetings, brainstorms, and running experiments.

My colleagues are Tonglin Li, Dongfang Zhao, Iman Sadooghi, Xiaobing Zhou, Anupam Rajendran, Kiran Ramamurthy, Zhangjie Ma, Juan Carlos Hernández Munuera, Kevin Brandstatter, Thomas Dubucq, Tony Forlini, Kan Qiao, Ning Liu, and Xi Yang. My collaborators are Abhishek Kulkarni, Dorian Arnold, Zhao Zhang, Benjamin Mcclelland, and Ralph Castain.

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

This dissertation is dedicated to my wife, Jingya Chi, for her unconditional support, caring, loving, and understanding, standing beside me.

Last but not least, I would like to thank my parents and my sister for their endless supports that make this dissertation possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Definition
c_{single}	Centralized architecture with a data server
c_{tree}	Centralized architecture with aggregated data servers
d_{fc}	Fully distributed architecture
d_{chord}	Distributed architecture with a chord overlay
p_r	The possibility that one neighbor is chosen again
E_c	The expected number of times of selection
KVS	Key-value stores
WaitQ	Task waiting queue
LReadyQ	Dedicated local task ready queue
SReadyQ	Shared work stealing task ready queue
CompleteQ	Task complete queue
MLB	Maximized load balancing policy
MDL	Maximized data locality policy
RLDS	Rigid load balancing and data-locality segregation
FLDS	Flexible load balancing and data-locality segregation
DAWS	Data aware work stealing
$G=(V,E)$	A workload DAG
u,v,w	Arbitrary task
$t_{exec}(v)$	The execution time of task v
$d(v)$	The data output size of task v
$t_{qwait}(v)$	The waiting time of task v in the ready queue
$t_{mvdata}(v,w)$	The time taken to move $d(v)$ size of data
$P(v)$	The parents of a task v
$t_{ef}(v)$	The earliest finishing time of task v

ABSTRACT

Distributed systems are growing exponentially in the computing capacity. On the high-performance computing (HPC) side, supercomputers are predicted to reach exascale with billion-way parallelism around the end of this decade. Scientific applications running on supercomputers are becoming more diverse, including traditional large-scale HPC jobs, small-scale HPC ensemble runs, and fine-grained many-task computing (MTC) workloads. Similar challenges are cropping up in cloud computing as data-centers host ever growing larger number of servers exceeding many top HPC systems in production today. The applications commonly found in the cloud are ushering in the era of big data, resulting in billions of tasks that involve processing increasingly large amount of data.

However, the resource management system (RMS) software of distributed systems is still designed around the decades-old centralized paradigm, which is far from satisfying the ever-growing needs of performance and scalability towards extreme scales, due to the limited capacity of a centralized server. This huge gap between the processing capacity and the performance needs has driven us to develop next-generation RMSs that are magnitudes more scalable.

In this dissertation, we first devise a general system software taxonomy to explore the design choices of system software, and propose that key-value stores could serve as a building block. We then design distributed RMS on top of key-value stores. We propose a fully distributed architecture and a data-aware work stealing technique for the MTC resource management, and develop the SimMatrix simulator to explore the distributed designs, which informs the real implementation of the MATRIX task execution framework. We also propose a partition-based architecture and resource sharing techniques for the HPC resource management, and implement them by building the Slurm++ real workload manager and the SimSlurm++ simulator. We study the distributed designs through real systems up to thousands of nodes, and through simulations up to millions of nodes. Results show that the distributed paradigm has significant advantages over centralized one. We envision that the contributions of this dissertation will be both evolutionary and revolutionary to the extreme-scale computing community, and will lead to a plethora of following research work and innovations towards tomorrow's extreme-scale systems.

CHAPTER 1

INTRODUCTION

In this chapter, we introduce the extreme-scale distributed systems, summarize the workload diversity of extreme-scale computing, motivate the work of delivering scalable resource management system software towards extreme scales, highlight the research contributions, and describe the organizations of this dissertation.

1.1 Extreme-scale Distributed Systems

Technology developing trends indicate that the distributed systems are approaching the era of extreme-scale computing. *On the high-performance computing side,* predicts are that around the end of this decade, supercomputers will reach exascale (10^{18} Flop/s) comprising of thousands to millions of nodes, up to one billion threads of execution of parallelism [1] [2], and tens of petabytes of memory. Exascale computers will enable the unraveling of significant mysteries for a diversity of scientific applications, ranging from Astronomy, Biology, Chemistry, Earth Systems, Economics, to Neuroscience, Physics, and Social Learning and so on [3]. The US President made reaching exascale a top national priority, claiming it will "dramatically increase our ability to understand the world around us through simulation" [197]. 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.

Exascale computing will bring new fundamental challenges in how we build computing systems and their hardware, how we manage them and program them through scalable system software. The techniques that have been designed decades ago will have to be dramatically changed to support the coming wave of extreme-scale general purpose parallel computing. The five most significant challenges of exascale computing are: *Concurrency and Locality*, *Resilience*, *Memory and Storage*, *Energy and Power* [3] [4]. Any one of these challenges, if left unaddressed, could halt progress towards exascale computing.

Concurrency and Locality refers to how we will harness the many magnitude orders of increased parallelism. The largest supercomputers have increased in parallelism at an alarming rate. In 1993, the largest supercomputer had 1K cores with 0.00006 PFlop/s (10^{15}), in 2004 8K cores with 0.035 PFlop/s, in 2011 688K cores with 10.5 PFlop/s, in 2015 3.12M cores with 34 PFlop/s, and in about 5 years, supercomputers will likely reach billions of threads/cores with 1000 PFlp/s [2]. The many-core computing era will likely increase the intra-node parallelism by two or three orders of magnitude, which cannot be efficiently utilized through the current tightly coupled programming models like Message Passing Interfaces (MPI) [64] that focus on optimizing the inter-node communications. Efficient shared-memory programming languages with automatic parallelism and localiy optimizations will be needed to cooperate with MPI, such as OpenMP [195], Cuda [194], and OpenCL [193].

Resilience refers to the capability of making all the infrastructure (hardware), system software and applications fault tolerant in face of a decreasing mean-time-to-failure (MTTF). Simulation studies [25] showed that at exascale level, the MTTF of a compute node will likely fall down to hours, making the current *checkpointing* method useless. More efficient fault tolerant mechanisms are demanded on the hardware level, such as chipkill correct memories [196]. In addition, the centralized design paradigm of current HPC system software needs to be changed to avoid single-point-of failures. On the application side, MPI is the main communication library used for synchronization of applications, requiring restarting the applications when node failure happens. Therefore, application programmers need to develop transparent task migration techniques.

Memory and Storage refers to optimizing and minimizing data movement through the memory hierarchy (e.g. persistent storage, burst buffer [187], solid state memory [198], volatile memory, caches, and registers). Exascale will bring unique challenges to the memory hierarchy never seen before in supercomputing, such as a significant increase in concurrency at both the node level (number of cores is increasing at a faster rate than the memory subsystem performance), and at the infrastructure level (number of cores is increasing at a faster rate than persistent storage performance). The memory hierarchy will change with new technologies (e.g. non-volatile memory [199]), implying that programming models and optimizations must adapt. Optimizing exascale systems for data locality will be critical to the realization of future extreme scale systems.

Energy and Power refers to the ability to keep the power consumption of a supercomputer at a reasonable level, so that the cost to power of a system does not dominate the cost of ownership. The DARPA Exascale report [2] defined probably the single most important metric, namely the *energy per flop*, and capped the power consumption of a single supercomputer to be 20MW. Given the energy consumption of current top 1 supercomputer which uses 17.8MW of power [200] and the increase in performance by 30X (to reach exascale), we can conclude that we need to reduce the energy per flop by 30X to make exascale computing viable. In addition, efficient data-aware scheduling techniques are needed to minimize the data movement, resulting in less power consumption.

On the cloud computing side, similar challenges will exist, as the data centers host ever-growing larger number of servers that exceed many top HPC systems in production today. Scalable solutions at all the IaaS [201] (e.g. light-weight virtualization techniques that allow scaling both vertically and horizontally), PaaS [202] (e.g. distributed resource management and scheduling techniques that support low latency and high utilization), and SaaS [203] (e.g. standard web delivery interfaces that allow custumers to export applications transparently) levels will be needed to address these challenges.

This dissertation aims to lay the foundations of addressing the challenges of extreme scale computing through building scalable resource management system software.

1.2 Workload Diversity towards Extreme-scale Computing

As distributed systems are approaching extreme scales, the supported applications are becoming diverse, resulting in workloads with high heterogeneity. *On the high performance computing side*, with the extreme magnitudes of component count and parallelism at exascale, one way to efficiently use the machines without requiring full-scale jobs is to support the running of diverse categories of applications [22]. These applications would combine scientific workloads covering various domains, such as traditional large-scale high performance computing (HPC), HPC ensemble runs, and loosely coupled many-task computing (MTC). The support of running of all the diverse applications improves the resilience by definition given that failures will affect a smaller part of the machines.

Traditional HPC workloads are typically large-scale long-duration tightly coupled applications that require many computing processors (e.g. half or full-size of the whole machine) for a long time (e.g. days, weeks) and use MPI [64] to communication and synchronization among all the processors. Resiliency is achieved relying on the *checkpointing* mechanism. As the system scales up, *check-pointing* will not be scalable due to the increasing overheads to do *checkpointing*. Besides, there will be less and less HPC applications that can scale up to exascale, 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. These 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. 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.

There is a growing set of large-scale scientific applications which are loosely coupled in nature, containing many small jobs/tasks (e.g. per-core) with shorter durations (e.g. seconds), along with large volumes of data – these applications are referred to as many-task computing (MTC) applications, including those from astronomy, bioinformatics, data analytics, data mining, and so on. MTC applications are typically structured 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. The algorithm paradigms well suited for MTC are Optimization, Data Analysis, Monte Carlo and Uncertainty Quantification.

Similar to the MTC scientific applications, *the data analytics applications in the Internet industry* are ushering in the era of big data, which involves archiving, processing, and transferring increasingly large amount of data. Take the Facebook data warehouse as an example, the data size had been growing from 15TB in 2007 to 700TB in 2010 (46X) [188], which, based on the increasing trend, can reach up to tens of petabytes by today. The big data analytics workloads range from ad-hoc queries, simple reporting applications, to business intelligence applications and more advanced machine learning algorithms, generating billions of loosely coupled tasks with data dependencies.

1.3 Motivations

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

The RMS for extreme-scale computing needs to be efficient, scalable and reliable enough, in order to deliver high performance (e.g. high throughput, high utilization, and low latency) for all types of workloads. However, the state-of-the-art RMS of both supercomputers and Internet are still designed around the decades-old centralized paradigm, which are far from being able to satisfy the ever-growing needs of performance of the RMS towards extreme scales, due to the single-point-of failure and limited capacity of a centralized server in handling all the resource management activities, such as system state management, resource allocation, job scheduling and job launching. Examples of RMS of supercomputers are SLURM [30], Condor [83], PBS [84], SGE [85], and Falkon [78]. They all have a centralized architecture. SLURM has been widely used as a production system on quite a few top supercomputers in the world. SLURM reported a maximum throughput of 500 jobs/sec [79] for simple batch jobs. While this is significantly better than others, however, we will need many orders of magnitude higher job delivering rates 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). Furthermore, our experimental results in SLURM showed that SLURM crashed and was unable to handle more than 10K concurrent job submissions with average job length of 429ms. Falkon is a pioneering MTC task execution framework that can deliver several thousands of fine-grained tasks per second. However, we have already seen the saturation point of Falkon [66] [78] on the full scale of an IBM Blue Gene / P supercomputer (160K cores) [204] in Argonne National Laboratory, not even to mention the billion-core system scale and billions-task MTC applications at exascale.

In the Internet domain, the challenges of RMS at extreme scales are only severer. Statistics show that Google now processes 40K+ search queries per second (3.5 billion searches per day and 1.2 trillion searches per year) [205], and the number of searches per year is increasing exponentially; Alibaba experienced a peak workload of 70K orders being processed per second during the 24-hour period of Singles Day of 2014 [206]. Google recently published its cluster manager, named Borg [154], which offers a general interface for various workloads, such as the low-latency production jobs and long-running batch jobs. Borg applies a centralized resource manager (BorgMaster) to monitor and manage resources, and a separate centralized scheduler process to allocate resources and schedule jobs. Although Borg managed to improve the scalability of the centralized architecture through a few techniques such as score caching, equivalence classes, and relaxed randomization, we believe that the continued growths of system size and applications will eventually hit the ultimate scalability limit, if the centralized architecture remains. YARN [112] and Mesos [106] are the RMS of the Hadoop [118], and Spark [160] ecosystems, respectively. Even though the scheduling framework is decoupled from the resource manager in both systems, the resource manager and the per-application scheduler still have a centralized architecture, leading to capped scalability.

The huge gap between the processing capacity and the performance needs has driven us to design and implement the next-generation resource management systems that are magnitudes more scalable than today's centralized architectures. *The goal of this dissertation is to deliver scalable resource management system software that can manage the numerous computing resources of extreme-scale systems and efficiently schedule the workloads onto these resources for execution, in order to maintain high performance*.

1.4 Dissertation Roadmap

To achieve the goal of building scalable resource management systems, in this dissertation, we first devise a general system software taxonomy to explore the design choices (e.g. architectures, and protocols of maintaining scalable and distributed services) of extreme-scale system software, and propose that key-value stores [27] [51] could serve as a building block. We then design distributed architectures and techniques on top of key-value stores, including real resource management systems, as well as simulators. Specifically, we propose a fully distributed architecture and a distributed data-aware work stealing technique for the resource management of MTC applications, and develop the SimMatrix simulator [50] to explore the distributed architecture and technique, which informs the design and actual implementation of the MATRIX task execution framework [23] [24] [231]. We also propose a partition-based distributed architecture and distributed resource sharing techniques (e.g. random resource stealing [22] and monitoring-based weakly consistent resource stealing [140]) for the resource management of HPC applications, and implement them by building the Slurm++ workload manager [140] and the SimSlurm++ simulator. We study the proposed architectures and techniques through real systems (i.e. MATRIX and Slurm++) up to thousands of nodes, and through simulations (i.e. SimMatrix and SimSlurm++) up to millions of nodes. Results show that the distributed designs have significant advantages over centralized ones in scalable resource management. We envision that the research contributions of this dissertation will be both evolutionary and revolutionary to the community of extreme-scale system software, and will lead to a plethora of following research work and innovations towards tomorrow’s extreme-scale systems.

The remainder of this dissertation is organized as follows. Chapter 2 introduces background knowledge of distributed systems, the three computing paradigms and the resource management system software. Chapter 3 devises a system software taxonomy for general HPC system software, and proposes that distributed key-value stores are a fundamental building block for extreme-scale HPC system software. Chapter 4 presents the MATRIX many-task computing task execution framework. Chapter 5 proposes a data-aware scheduling technique for scheduling data-intensive applications. Chapter 6 presents the Slurm++ workload manager for HPC applications. The related work is listed in Chapter 7, followed by Chapter 8, which concludes the dissertation, highlights the research contributions, proposes future work, and analyzes the long-term impacts of this dissertation.

CHAPTER 2

BACKGROUND

This chapter introduces the background information of distributed systems, the three distributed computing paradigms, and the functionalities of the resource management system software that this dissertation is targeting.

2.1 Distributed Systems

A distributed system is a collection of autonomous computers that are connected through networks and system software, with the goal to offer the users a single, integrated view of sharing the resources of the system [207]. There are various types of distributed systems, including cluster, supercomputer, grid, and cloud.

2.1.1 Cluster. A cluster is a relatively small-size distributed system, ranging from a few nodes to a few thousand nodes. The compute nodes of a cluster are usually made of commodity processors, and with each one running an operating system. In most cases, the compute nodes are homogeneous, configured with the same hardware and operating system [208]. Comparing with a single computer, clusters can deliver better performance with more computing capacity, and higher availability due to the configuration of redundant failover nodes, thus a single-node failure will not cause the whole system down. Furthermore, clusters are usually more cost-effective than single computers that offer comparable computing power and availability. The compute nodes are allocated to perform computational tasks that are controlled and scheduled by the resource management and job scheduling system software. They are connected with each other through fast local area networks (LAN), and use libraries such as MPI for network communication to exchange data and synchronize, enabling the clusters to offer high performance computing with over low-cost commodity hardware.

2.1.2 Supercomputer. A supercomputer is an extreme case of computer cluster that is highly-tuned with high-bandwidth network interconnections and customized operating system, with the developing trend of consisting of many compute nodes with high intra-node parallelism. Supercomputing is also referred to as high performance computing, because supercomputers are typically deployed as capacity computing facilities that are governed by national laboratories and large-scale computing centers. They are used to run large-scale scientific applications, including both simulations and real experiments, which require much large parallel computing power. Examples of supercomputers are the IBM Blue Gene serials - Blue Gene/L/P/Q [204], the Japanese K-computer [209], and the Chinese Tianhe supercomputers [200]. Figure 1 shows the developing trends of computing power of the Top500 [200] supercomputers, measured in Flop/s for the past twenty years, and projecting out for the next five years. For the top one supercomputer, we have surpassed the petaflop/s era (the top one supercomputer today has a peak speed of 33.9 petaflop/s), and it is predicted that we will reach the exascale (1000 petaflop/s or exaflop/s) around the end of this decade [62].

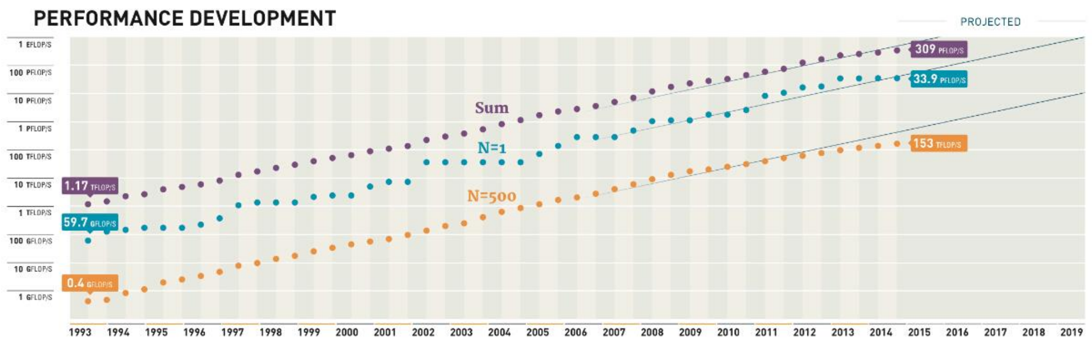


Figure 1. Developing trend of the Top 500 supercomputers

2.1.3 Grid. Grids are composed of multiple clusters that are geographically dispersed in several institutions. These multi-site clusters are typically heterogeneous with each one having their own hardware, system software, and networks, and they are connected through loosely coupled wide area networks (WAN). The grand vision of "Grid Computing" is an analogy to the power grids where users have control over electricity through the wall sockets and switches without any consideration of where and how it is generated [210]. Dr. Foster et al. defines the Grid Computing as deploying distributed computing infrastructure to offer large scale resource sharing, innovative applications, and high performance computing to the dynamic collections of individuals and institutions (referred to as virtual organizations) through standard, secure and coordinated resource sharing protocols, so that they can accomplish their advanced scientific and engineering goals [211]. Examples of grids include the TeraGrid [212], the Enabling Grids for E-sciencE (EGEE) [213], and the ChinaGrid [214]. Some of the major grid middleware are the GridFTP [215] (a file transfer protocol), the Globus [216] (a file transfer service with web interfaces), and the Unicore [217]. Grids are well suited for running high throughput computing applications that are loosely coupled in nature, while are more challenging for high performance computing ones due to the fact that resources can be geographically distributed, resulting in high latency between nodes.

2.1.4 Cloud. Clouds are distributed systems deployed over the Internet, aiming to deliver the hosted services to all the Internet users, through web interfaces. Clouds apply a "pay-as-you-go" utility model to share resources and provide services to users dynamically on their demands through virtualization techniques, in order to achieve coherence, high system utilization, and economies of scale of the Internet. Clouds can be categorized as private clouds, public clouds and hybrid clouds, according to the ownership. The services are divided into three levels, namely Infrastructure-as-a-Service (IaaS) [201], Platform-as-a-Service (PaaS) [202], and Software-as-a-Service (SaaS) [203], from bottom up.

IaaS provides services of accessing, monitoring and managing the hardware infrastructures, such as compute, storage, networking, so that the users don't have to purchase the hardware. Examples of IaaS are Amazon Web Services (AWS) [218], Google Compute Engine (GCE) [219], and Microsoft Windows Azure [220]. PaaS is the set of services designed to provide a platform to customers, so that they can develop their code and deploy their applications without worrying about the complexities of building and maintaining the infrastructures. Amazon Elastic Compute Cloud (EC2) [221] and Google App Engine (GAE) [222] are exmaples of PaaS. SaaS offers services that deliver applications through web browsers to users without the needs of installing and running applications on local computers. Exmamples of SaaS are Gmail, Salesforce [223], and Workday [224].

2.1.5 Summary of Distributed Systems. We summarize the different categories of distributed systems in Figure 2 [40].

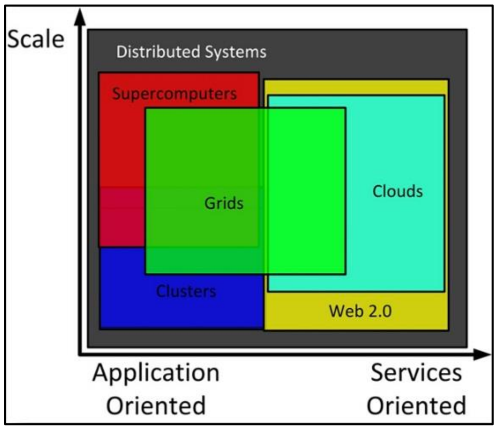


Figure 2. Categories of distributed systems

As seen, clusters and supercomputers are application oriented, comprising of many compute nodes connected through interconnects to run large-scale scientific applications, with the goals of offering high performance and low latency. Supercomputers are extreme cases of clusters with large scales and highly tuned hardware and software. Clouds and web servers are services oriented, delivering elastic services over the Internet with an on-demand utility model. Grids can be used to either run scientific applications or deliver services among virtualized organizations through standard resource sharing protocols. Grids overlap with all of clusters, supercomputers, and clouds.

2.1.6 Testbeds. This section describes the testbeds of distributed systems that are used in conducting experiments in this dissertation. The testbeds range from single shared-memory parallel machine, cluster, supercomputer, to cloud.

- Fusion: The fusion machine (fusion.cs.iit.edu) is a shared-memory parallel compute node in the datasys laboratory of IIT. The machine has an x86_64 achitecture, 48 CPUs of the AMD Opteron (tm) model at 800MHz with 4 sockets, 256GB memory, and a linux distributition openSUSE 11.2 (x86_64). This machine has been used extensively for simulation studies.
- Kodiak: Kodiak is a cluster from the Parallel Reconfigurable Observational Environment (PROBE) at Los Alamos National Laboratory [120]. Kodiak has 500 nodes, and each node has an x86_64 architecture, 2 AMD tm CPUs at 2.6GHz, 8GB memory, and uses an OS of the Ubuntu 12.04 distribution. Kodiak has a shared network file system (NFS). The network supports both 1Gb Ethernet and 8Gb InfiniBand. Our experiments use the default interconnect of 1Gb Ethernet.

- BG/P: The Blue Gene / P (BG/P) is an IBM supercomputer from Argonne National Laboratory (ANL) [204]. The machine is over 500 TFlops, and has 40 40960 nodes with 40 racks. Each one has four IBM Power PC450 processors at 850MHz, 2GB memory, and uses the ZeptOS [9]. The BG/P has both GPFS [225] and PVFS [226] file systems available. A 3D torus network is used for point-to-point communications.
- Amazon EC2 Medium: One of the Cloud testbeds we used in our experiments is the Amazon EC2 Medium environment that had 64 "m1.medium" instances. Each instance had one x86_64 CPU, 2 ECUs, 3.7GB memory, 410GB hard drive, and used an OS of the Ubuntu 12.10LTS distribution.
- Amazon EC2 Large: The other Cloud testbeds we used is the Amazon EC2 Large environment that had 128 "m3.large" instances. Each instance had 2 x86_64 CPUs, 6.5ECUs, 7.5GB memory and 16GB storage of SSD, and used an OS of the Ubuntu 14.04LTS distribution.

2.2 Three Distributed Computing Paradigms

The three distributed computing paradigms are high-performance computing (HPC), high-throughput computing (HTC), and many-task computing (MTC). Each of them has its own application domains and users.

2.2.1 High-performance computing. High-performance computing (HPC), sometimes referred to as supercomputing, is the use of the powerful computing capacities of clusters and supercomputers for running advanced scientific application programs with the goal speeding up the programs as much as possible. HPC environments deliver a tremendous amount of computing power on applications, emphasizing the performance measurement of Flop/s. A large-scale HPC machine is usually governed by the national laboratories and the national computing centers in a central place. The most common users of HPC systems are scientific researchers, engineers, institutions, and government agencies.

HPC applications usually require many computing processors (e.g. half, or full-size of the whole supercomputer) to discover the scientific mysteries through simulations and experiments. Examples of these applications are weather forecasting, probabilistic analysis, brute force code breaking, exploring the structures of biological gene sequences, chemical molecules, the human brain, as well as simulations of the origin of the universe, spacecraft aerodynamics, and nuclear fusion [141]. The application programs generate jobs that are tightly coupled, typically use the MPI programming model [64] to communicate and synchronize among all the processors, and rely on the *checkpointing* mechanism for reliability.

2.2.2 High-throughput computing. High-throughput computing (HTC) describes a category of applications that compute on the many loosely coupled tasks with different data pieces. Scientists and engineers engaged in this sort of work are concerned with the number of Flop per month or per year achieved through the computing environment, as opposed to the measurement of Flop per second of the HPC environment.

HTC emphasizes on efficiently harnessing all the available distributed computing resources to finish as many tasks as possible during a long period (e.g. weeks, months, even years). This was driven by the facts that many individuals and groups cannot afford an HPC machine and the HPC environment was inconvenient to them because they have to wait for a long time to get their allocations for a limited amount of time. As computer hardware has been becoming cheaper and more powerful, these users tend to moving away from the HPC environment to desktop workstations and PCs, which they could afford and are available whenever they need those. Though each personal computer has significantly lower computing power than that of a HPC machine, there may be thousands or even millions of them, forming an environment with distributed ownership that is the major obstacle the HTC environment has to overcome for better expanding the available resource pools to individuals. This is also where the high efficiency is not playing a major role in an HTC environment.

Workflow systems [68] [94] are an important middleware of HTC environments. Example jobs of the HTC applications include different patient data in large-scale biomedical trials, different parts of a genome or protein sequence in bioinformatics applications, different random numbers in a simulations based on Monte Carlo methods, and different model parameters in ensemble simulations or explorations of parameter spaces [83].

2.2.3 Many-task computing. Many-Task Computing (MTC) was introduced by Dr. Raicu et al. [39] [40] in 2008 to bridge the gap between HPC and HTC, and to describe a class of applications that do not fit easily into the categories of HPC or HTC. We depict the relationships of the three paradigms in Figure 3 [40].

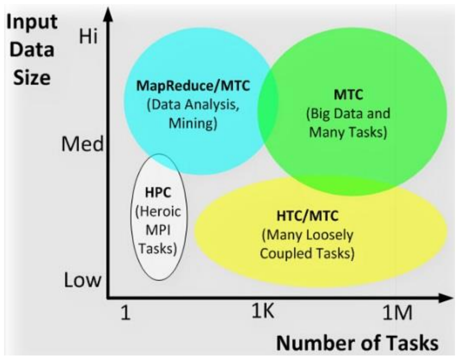

Input Data Size
Hi
Med
Low
MapReduce/MTC (Data Analysis, Mining)
MTC (Big Data and Many Tasks)
HPC (Heroic MPI Tasks)
HTC/MTC (Many Loosely Coupled Tasks)
1
1K
1M
Number of Tasks

Figure 3. Relationships of the three computing paradigms

MTC applications are dramatically different from typical HPC ones. HPC applications require many computing cores to execute job in a cooperative way, while MTC ones are decomposed as many fine-grained tasks that are loosely coupled and do not require strict coordination of processes at job launch. On the other hand, MTC has features that distinguish it from HTC. HTC applications emphasize on long-term solutions to applications run on platforms such as clusters, grids and clouds, through either workflow systems or parallel programming systems. On the contrary, MTC applications demand a short time to solution, and may be decomposed into a large number of fine-grained tasks (small and short duration) that are both communication intensive and data intensive. The aggregate number of tasks and the volumes of involved data may be extremely large.

MTC is driven by the data-flow programming model [165]. Many MTC applications are structured as direct acyclic graphs (DAG) [166], where the vertices are discrete tasks and the edges represent the data flows from one task to another. In the majority of cases, the data dependencies of the tasks of MTC applications will be files in a file system deployed across all the compute resources, such as the FusionFS file system [42]. For many applications, DAGs are a natural way to decompose the computations, which bring many flexibilities, such as allowing tasks to be executed on multiple machines simultaneously; enabling applications to continue under failures, if no data dependency exists among tasks, or tasks write their results to persistent storage as they finish; and permitting applications to be run on varying numbers of nodes without any modification. Examples of the MTC applications cover wide disciplines, ranging from astronomy, physics, astrophysics, pharmaceuticals, bioinformatics, biometrics, neuroscience, medical imaging, chemistry, climate modeling, to economics, and data analytics [53].

2.3 Resource Management System Software

Resource management system (RMS) is a vital system software deployed on distributed systems. The RMS is in charge of many activities, such as managing and monitoring all the resources (e.g. compute, storage, and network), state management, data management, resource partitioning, resource allocation, job scheduling, and job launching. This section defines the terms of entities of RMS, explains the activities of RMS, and gives three examples of RMS for HPC, MTC, and Hadoop applications.

2.3.1 Terms of Entities of RMS. We define the terms of the important entities of a RMS as follows.

Resource management system: A resource management system (RMS) typically comprises multiple components, such as resource manager, job queues, scheduler, and compute daemon. There are many alternatives to the term of resource management system (RMS), such as workload manager, cluster manager, job management system, and job scheduling system. For MTC environment, the terms of task scheduling framework, task execution framework, task execution fabric all mean RMS. In some cases, for simplicities, all the terms of resource manager, job manager, job scheduler, and task scheduler also mean RMS. They should not be confused with the resource manager and scheduler components of a RMS.

Resource: Resource could mean any hardware piece of a compute node. The important resources to a RMS are hardware threads, CPU cores, memory, and networks.

Job/Task: A job is a program to be executed, which is submitted by an end-user. A job can have resource requirements attached, such as number of nodes and cores. Task is finer grained, and a job may include many tasks with each one executing different computations. An application may be comprised of multiple jobs. RMS of HPC environment usually works at the job level (coarse granularity), RMS of HTC environment works at both the job level and task level, and RMS of MTC environment typically works at the task level (fine granularity).

Resource manager: A resource manager (RM) is a component of a RMS. RM manages and monitors all the resources, performs activities of resource partitioning, resource allocation, and maintains job metadata information. The RM usually accepts job submissions. Other terms such as controller, master are occasionally used in specific RMS.

Job Queue: Job queues are used to hold jobs that are waiting to be scheduled. Sometimes when the system runs out resources, or has no enough resources to satisfy a job’s resource requirement, the job is then kept in the queue. Job queues are also useful when there are too many current job submissions, so that the RM and scheduler cannot process all of them at the same time. Another important reason to have job queues is to support different scheduling policies, such as preemptive scheduling, priority-based scheduling, and gang scheduling. These policies are used very common for HPC jobs.

Scheduler: A scheduler is another important component of RMS, which schedules and launches jobs to pre-allocated compute nodes for execution. The scheduler could be implemented either as a separate thread of the RM, or as a standalone process. The former has better performance, while the latter is more scalable. One alternative to the term of scheduler is dispatcher.

Compute daemon: A compute daemon (CD) is a standalone process running on all the compute nodes. A CD executes the actual job/task program, communicates with other CDs to exchange and synchronize data, returns the results or writes them in persistent storage systems. A compute node can have multiple CDs running in parallel. Sometimes, the CDs are referred to as slaves, corresponding to the master term. A CD is also named executor.

2.3.2 Activities of RMS. We explain the major activities of a RMS in brief as follows.

- *Resource management:* manage and organize the resources in an efficient way (e.g. hash tables, database systems), in order to offer fast accessing (e.g. lookup, insert, update) speed.
- *Resource monitoring:* monitor the system states of the compute nodes, including the health states (e.g. down or up, slow or normal), temperature, power consumption, etc.
- *Resource partitioning:* partition the compute nodes into different groups for better utilization and specific resource requirements (e.g. affinity, GPUs, SSDs).
- *Resource allocation:* allocate resources to a job, according to the job's resource requirement.
- *Job scheduling:* schedule jobs onto the allocated compute nodes. Different scheduling policies may be applied.
- *Job launching:* launch a job and transform the job program to the compute nodes with one or more job steps. Scalable job launching techniques, like hierarchical tree-based job launching, are required for large-scale HPC jobs.

2.3.3 Examples of Resource Management System. We give three examples of RMS that all have a centralized design, namely the SLURM [30] workload manager for the HPC applications, the Falkon [78] task execution framework for the MTC applications, and the

YARN [112] resource manager for the Hadoop applications. These examples help us understand the architecture and components of a RMS, and the procedure of running a job from job submission, to resource allocation, to job scheduling, until the job is finished.

2.3.3.1 SLURM Workload Manager

SLURM [30] is a workload manager designed for HPC applications. SLURM has been widely deployed on many HPC machines, ranging from clusters to many of the most powerful supercomputers in the worlds. The architecture and components of SLURM are shown in Figure 4 [30].

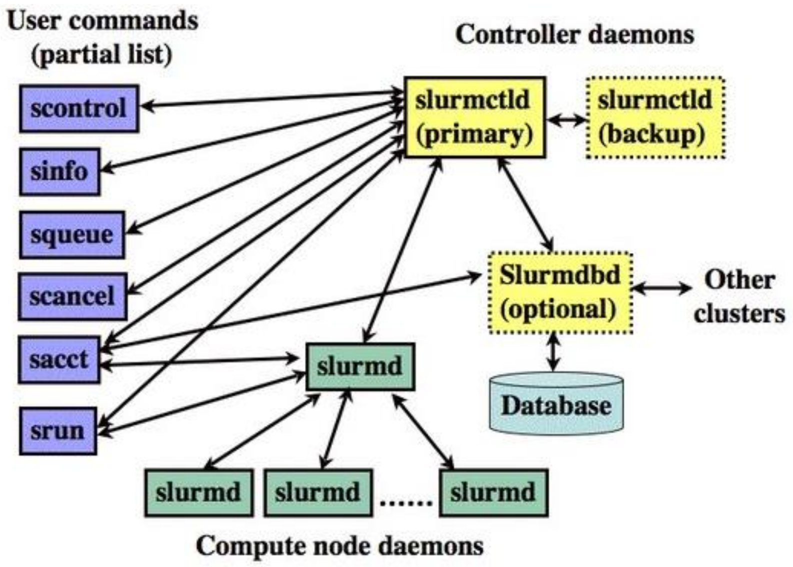


Figure 4. SLURM architecture and components

SLURM uses a centralized controller (slurmctld) to manage all the compute daemons (slurmd) that are run on the compute nodes. The slurmctld is made of three subcomponents: the Node Manager monitors the state of nodes; the Partition Manager allocates resources to jobs for scheduling and launching; and the Job Manager schedules the jobs and manages the job states. The users submit jobs to the slurmctld through user commands, such as those shown in Figure 4. To improve scalability, SLURM delegate the job launching responsibility to the user commands. In addition, SLURM does scalable job launch via a tree based overlay network rooted at rank-0. The slurmctld is made fault tolerant through a failover, which does not participate unless the main server fails. Besides, SLURM offers the option to store the resource and job information in database. Other RMS like Cobalt [114], PBS [153], and SGE [85], have a similiar architecture as SLURM.

Figure 5 [30] illustrates an example of running the srun command that submits an interactive job. Firstly, the job is submitted to the slurmctld, which then replies the resource allocations to the command. After getting the resource allocation, srun divides the job into one or multiple job steps, and then launches the job steps to all the allocated slurmds. During execution, the slurmds send output to srun and the srun reports status to slurmctld periodically, until the job is finished. After the job is done, srun sends message to release the resources, to slurmctld, which then sends epilog information to the slurmds.

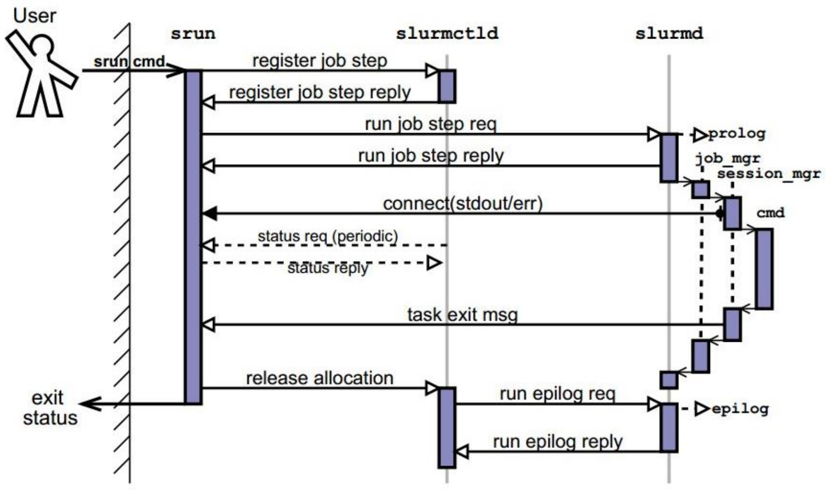
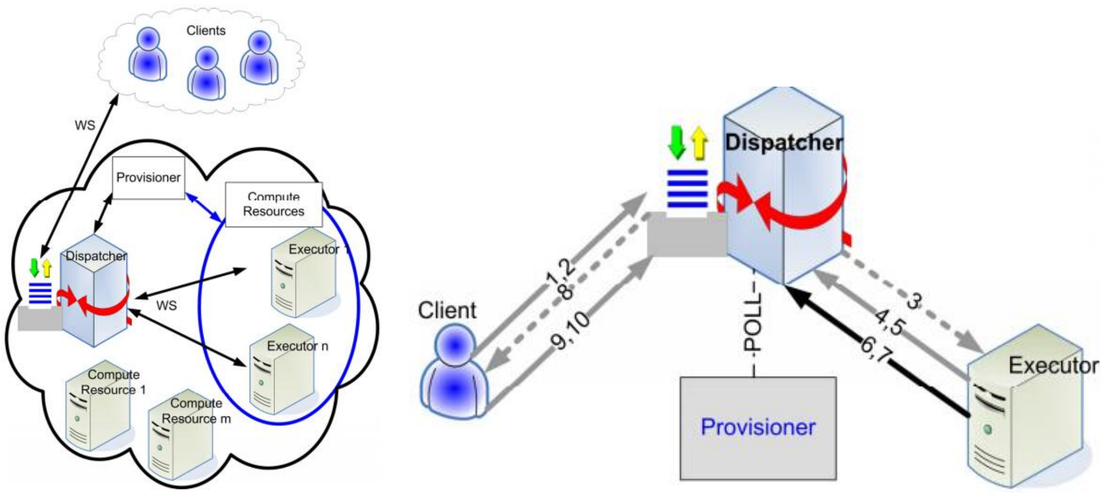


Figure 5. The srun command of SLURM

2.3.3.2 Falkon Task Execution Framework

Falkon is a lightweight task execution framework for MTC applications. Falkon has been deployed on multiple platforms, including the ANL/UC Linux cluster, the

SiCortex machine, and the BG/P supercomputer. Falkon comprises a centralized dispatcher, a provisioner, and zero or more executors, as shown in Figure 6 [78] (a). Figure 6 (b) illustrates the message exchanges among the components, labeled with numbers. Two numbers denote both *send* and *receive* via Web Services (WS) messages (solid lines); while one number denotes only a *send* notification message performed via TCP (dotted lines).



(a) Falkon components (b) Message exchanges among components

Figure 6. Falkon architecture and components

A client uses a pre-requested endpoint reference to submit tasks (messages 1, 2), monitor wait for notifications (message 8), and retrieve results (messages 9, 10). The dispatcher accepts task submissions from clients, and schedules tasks to the executors according to different scheduling policies. The provisioner manages to create and destroy executors on the compute nodes, under the guidance of the dispatcher. The provisioner monitors the dispatcher state by polling it periodically (message POLL), and determines when and how many additional executors should be created, and for how long before releasing them. The newly created executors register with the dispatcher, and execute tasks with the following procedure: the dispatcher notifies the executors for available tasks (message 3); the executor request tasks (message 4); the dispatcher schedules tasks to the executors (message 5); the executors run the supplied tasks and returns the results (message 6), and finally the dispatcher acknowledges (message 7).

2.3.3.3 YARN Resource Manager

YARN is the next generation RMS for general Hadoop clusters that run various categories of applications (e.g. Hadoop, MPI, and scientific workflows). YARN utilizes a resource manager (RM) to monitor and allocate resources, delegates per-application master (AM) to launch tasks to resource containers managed by the node manager (NM) on the computing nodes. Figure 7 [112] depicts the YARN architecture and components (shown in blue), and the running of two applications (shown in in yellow and pink).

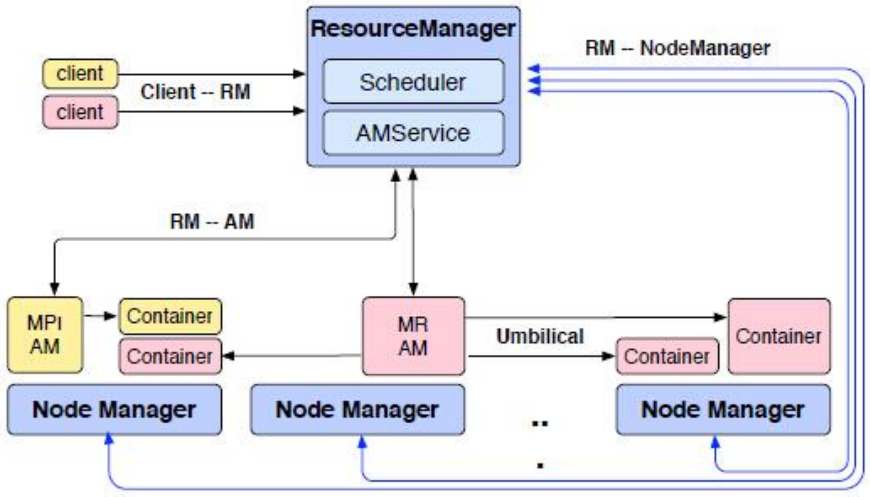


Figure 7. YARN architecture and components

The centralized RM coordinates the resource allocations for all the applications. The resource of a compute node is organized as containers (e.g. <2GB RAM, 1 CPU>) by the NM running on the mode. An NM monitors local resource availability and manages container lifecycle. The RM assembles the resource from all the NMs to gain a global view, which enforces various scheduling policies.

A client submits jobs to the RM that passes the accepted jobs to the scheduler to be executed. Once the resource requirement of a job is satisfied, the scheduler allocates a container to create an AM for the job. The AM requests containers from the RM, and schedules and launch the job to all the granted resource containers with a lease to complete all the job tasks. The AM is also responsible for managing all the lifecycle activities of a job, such as elastic resource allocations, managing the execution flow of the workload DAG (e.g., reduce tasks take the outputs of map tasks as input), load balancing, and performing other optimization techniques.

2.4 Performance Metrics

This section defines the performance metrics that will be used in our simulations and experiments to evaluate the performance of various techniques and systems. Some metrics are used across the evaluation results of all chapters, such as *throughput*, *efficiency*, while the others are only used in certain results.

- *Throughput*: measures how fast the RMS can execute a given workload, with the unit of jobs per time period (e.g. sec, hours). It is calculated as the number of jobs/tasks finished dividing by the total time of finishing them. The total time is the time difference between the earliest submission time and the latest finishing time of all the jobs. This metric is only meaningful to the jobs with short durations (e.g. a few milliseconds or less), and the higher throughput number means the less overheads and better performance. For long-duration jobs, the throughput cannot reveal the performance directly.
- *Per Client Throughput*: average client throughput from the client's perspective. For each client, the client throughput is calculated as the number of requests

(submitted by the client) finished dividing by the time to finish the requests. It measures how fast requests are processed viewed by the client.

- *Coefficient variance*: aims to show the performance of work stealing [52] towards distributed load balancing. We use the *coefficient variance* of the numbers of tasks finished by all the schedulers as the measurement – the smaller value indicates the better performance. This metric only has significance for homogeneous workloads with tasks having the same running time.
- *Efficiency:* shows the proportion of time when the system is actually executing tasks. Other than executing tasks, the system may do other work, such as auxiliary computing, moving tasks around for load balancing, and network communications, which we need to minimize. A higher efficiency (~100%) means a better performance. It is the percentage of the ideal running time (T_{ideal}) to the actual running time (T_{actual}) of a workload, which quantifies the average utilization of the system. The higher efficiency ($\left(100\times T_{ideal}/T_{actual}\right)\%$) indicates less overheads of the scheduling framework. Given a workload that has p phases and one phase cannot be started until the previous one has been finished, we can compute the ideal running time, $T_{ideal}=\sum_{\lambda=1}^{p}T_{ideal(\lambda)}$, in which $T_{ideal(\lambda)}$ is the ideal running time of the λth phase. Assuming in the λth phase, on average, each core is executing k tasks with an average length of l. Therefore, $t_{ideal(\lambda)}=k\times l$. For instance, the bag-of-task workload has only one phase, while a typical mapreduce job has two phases (e.g. map phase and reduce phase).

- *Utilization* is the instantaneous metric that measures the ratio of resources (CPU cores) that are occupied to execute tasks during some instant time. It is usually presented when doing visualization of the system state according to the logs generated by the monitoring program.
- *Average Time Per Task Per CPU:* defines the average time of executing a task from one CPU's perspective. In an ideal case, each CPU would process tasks sequentially without waiting. Therefore, the ideal average time should be equal to the average task length of all the tasks. In reality, the average time should be larger than the ideal case, and the closer they are the better. For example, assuming it takes 10sec to run 2000 tasks on 100 cores, this is 2 tasks-per-sec-per-cpu (2000/10/100) on average, meaning 0.5sec per task per CPU.
- *Average Task-Delay Ratio:* denoted as $\overline{r_{td}}$, is computed as the normalized difference between the average ideal task turnaround (*itt*) time, $\overline{T_{itt}}$, and the average actual task turnaround (*att*) time $\overline{T_{att}}$. Therefore, $\overline{r_{td}} = \left(\overline{T_{att}} - \overline{T_{itt}}\right) / \overline{T_{itt}}$. For each task n, the turnaround time (*tt*), denoted as $T_{tt(n)}$, is the time interval between the time when the task is launched and the time when the task is finished. This metric requires the RMS to record the detailed timestamps of each task, from which, we can know the turnaround time of each task. Therefore, we can compute $\overline{T_{att}}$ after running a workload that include k tasks: $\overline{T_{att}} = \frac{\sum_{n=1}^{k} T_{tt(n)}}{k}$. Assuming on average, each core in the system is executing k

tasks with an average length of l. Therefore, the nth task needs to wait $(n-1)\times l$ time before being executed, meaning that $T_{tt(n)} = (n-1)\times l + l = nl$. Therefore, we have $\overline{T_{itt}} = \frac{\sum_{n=1}^{k}(nl)}{k} = \frac{n+1}{2}\times l$. This metric measures how fast a framework can response to a workload from each task's perspective. The smaller $\overline{r_{td}}$ means faster response time and lower scheduling overheads.

- *Average scheduling latency per job*: is the average scheduling latency of all the jobs. The scheduling latency of a job is the time difference between the submission time of the job and the time when the job is granted with the required resources. This metric shows how well the resource allocation technique employed by a RMS can perform. The smaller number indicates a less overhead and better performance.
- *Average ZHT message count per job*: figures out the average number of messages interchanged between the controllers and the ZHT servers (a distributed key-value store) of all jobs. Each ZHT operation (e.g. *insert*, *lookup*, and *compare and swap*) of the controller is counted as a message. The smaller message count means the less overheads of using the ZHT key-value store.
- *Speedup:* is a measurement of relative performance improvement. It is often used when comparing two results.

CHAPTER 3

SYSTEM SOFTWARE TAXONOMY

Owing to the extreme parallelism and the high component failure rates of tomorrow’s exascale, high-performance computing (HPC) system software will need to be scalable, failure-resistant, and adaptive for sustained system operation and full system utilizations. Many of the existing HPC system software are still designed around a centralized server paradigm and hence are susceptible to scaling issues and single points of failure. In this chapter, we explore the design tradeoffs for scalable system software at extreme scales. We devisee a general system software taxonomy by deconstructing common HPC system software into their basic components. The taxonomy helps us reason about system software as follows: (1) it gives us a systematic way to architect scalable system software by decomposing them into their basic components; (2) it allows us to categorize system software based on the features of these components, and finally (3) it suggests the configuration space to consider for design evaluation via simulations or real implementations. Further, we propose that the key-value store is a viable building block to build scalable system software at extreme scales. We then evaluate different design choices of key-value stores through simulations up to millions of nodes using both benchmarking workloads and workloads from real system software traces. We envision that the conclusions we will draw in this chapter help to lay the foundations of developing next-generation HPC system software for extreme scales.

3.1 Exascale HPC System Software

System software is a collection of important middleware services that offer to upper-lay applications integrated views and control capabilities of the underlying hardware components. Generally, system software allows applications full and efficient hardware utilization. A typical system software stack includes (from the bottom up) operating systems (OS), runtime systems, compilers, and libraries [1]. Technological trends indicate that exascale high-performance computing (HPC) systems will have billion-way parallelism [2], and each node will have about three orders of magnitude more intra-node parallelism than that of the node of today's peta-scale systems [4]. Exascale systems will pose fundamental challenges of managing parallelism, locality, power, resilience, and scalability [3] [5] [6].

Current HPC system software designs focus on optimizing the inter-node parallelism by maximizing the bandwidth and minimizing the latency of the interconnection networks but suffer from the lack of scalable solutions to expose the intra-node parallelism. New loosely coupled programming models (e.g. many-task computing [39], over-decomposition [7], and MPI + OpenMP [8]) are helping to address intra-node parallel-ism for exascale systems. These programming models place a high demand on system software for scalability, fault-tolerance and adaptivity. However, many of the existing HPC system software are still designed around a centralized server paradigm and, hence, are susceptible to scaling issues and single points of failure. Such concerns suggest a move towards fundamentally scalable distributed system software designs – a move further motivated by the growing amount of data (and metadata) that servers need to maintain in a scalable, reliable, and consistent manner.

The exascale community has been exploring research directions that address exascale system software challenges, such as lightweight OS and kernels (e.g. ZeptoOS [9], Kitten [10]); asynchronous and loosely coupled runtime systems (e.g. Charm++ [11], Legion [12], HPX [13], STAPL [14], and Swift [15] [91]); load balanced and locality-aware execution and scheduling models (e.g. MATRIX [23] [26], ParallelX [16], and ARMI [17]); automatic and auto-tuning compilers (e.g. ROSE [18], SLEEC [19]). The general collections of HPC system software are those that support system booting, system monitoring, hardware or software configuration and management, job and resource management, I/O forwarding, and various runtime systems for programming models and communication libraries. As HPC systems approach exascale, the basic design principles of scalable and fault-tolerant system architectures need to be investigated for HPC system software implementations. Instead of exploring the design choices of each system software at every stack level individually and in an ad hoc fashion, this work aims to develop a general framework that allows for systematic explorations of the design space of HPC system software and to evaluate the impacts of different design choices.

In this chapter, the questions we intend to answer are: what are the scalabilities of different system soft-ware architectures (centralized, hierarchical, distributed); and at what scales and levels of reliability and consistency does distributed design outweigh the extra complexity and overhead of centralized and hierarchical designs.

To answer the questions, we devise a general taxonomy [228] that classifies system software based on basic components, to identify their performance and scaling limits. By identifying the common basic components and focusing on designing these core components, we will enable faster prototyping and development of new system software.

We then motivate key-value stores (KVS) as a building block for HPC system software at extreme scales, and then using KVS as a case study, we explore design tradeoffs of system software. Via simulation, we explore the scalability of each system architecture and quantify the overheads in supporting reliability at extreme scales. We believe the work presented in this article lays the foundations for the development of the next-generation, extreme-scale HPC system software. *This chapter makes the following contributions:*

- Devise a comprehensive taxonomy by deconstructing system software into their core components.
- Propose that key-value stores are a viable building block of extreme-scale HPC system software.
- Develop a simulation tool to explore key-value store design choices for large-scale HPC system software.
- Conduct an inclusive evaluation of different system architectures (centralized, hierarchical, and distributed) under various design choices, such as different replication, recovery, and consistency models, using both synthetic and real workload traces.

3.2 Key-value Stores in HPC

3.2.1 Building Blocks for HPC. We motivate that KVS is a building block for HPC system software at extreme scales. The HPC system software, which we generally target, are those that support system booting, system monitoring, hardware or software configuration and management, job and resource management, I/O forwarding, and various runtime systems for programming models and communication libraries [30] [31] [32] [33]. For extreme-scale HPC systems, these system software all need to operate on large volumes of data in a scalable, resilient and consistent manner. We observe that such system software commonly and naturally comprise of data-access patterns amenable to the NoSQL abstraction, a lightweight 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, such as, Amazon, Facebook, LinkedIn and Twitter. In these commercial enterprises, NoSQL data stores – Distributed Key-Value Stores (KVS) in particular – have been used successfully [34] [35] [36] in deploying software as a service (SaaS). We assert that by taking the particular needs of HPC system into account, we can use KVS for HPC system software to help resolve many scalability, robustness, and consistency issues.

By encapsulating distributed system complexities in the KVS, we can simplify HPC system software designs and implementations. Giving some examples as follows: For resource management, KVS can be used to maintain necessary job and node status information. For monitoring, KVS can be used to maintain system activity logs. For I/O forwarding in file systems, KVS can be used to maintain file metadata, including access authority and modification sequences. In job start-up, KVS can be used to disseminate configuration and initialization data amongst composite tool or application processes (an example of this is under development in the MRNet project [33]). Application developers from Sandia National Laboratory [37] are targeting KVS to support local checkpoint/restart protocols. Additionally, we have used KVS to implement several system software, such as a many-task computing (MTC) task execution [38] [39] [40] [41] framework – MATRIX [23] [24] [26], where KVS is used to store the task metadata information, and a fuse-based distributed file system, FusionFS [42], where the KVS is used to track file metadata.

3.2.2 Centralized Architecture's Bottleneck. HPC system software designed around the centralized architecture suffer from limited scalability, high likelihood of non-recoverable failures and other inefficiencies. To validate this, we assess the performance and resilience of a centralized file-backed KVS.

We implement a KVS prototype. Each request (put, get, and delete) was turned into a corresponding file sys-tem operation (write, read and remove, respectively) by the server. A request with a 16-byte payload is consist of a (key, value) pair. We run the prototype on a 128-node machine with AMD 2GHz Dual-Core Opteron and 4 GB of memory per node. Compute nodes are connected with Gigabit Ethernet. At every second boundary, the throughput attained by the server is measured to deter-mine the maximum throughput during operation.

Figure 8 shows the peak throughput is achieved at 64 clients with the configuration of one client per node. As multiple clients per node are spawned, the throughput decreases due to network contention. At relatively modest scales, centralized server shows significant performance degradation due to contention.

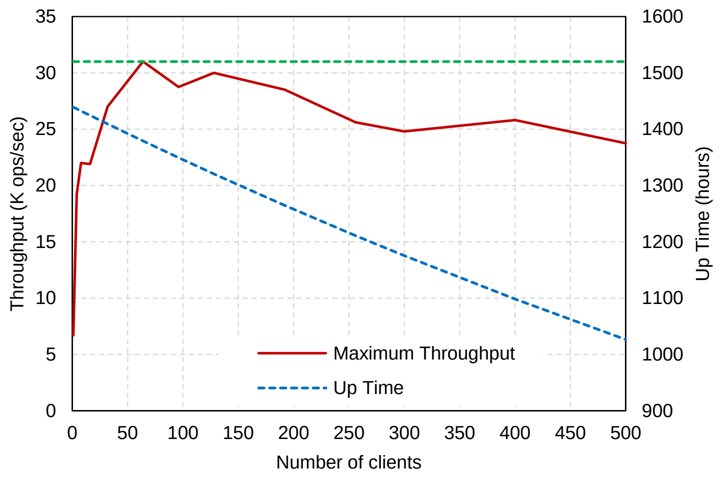


Figure 8. Performance and Resilience of Centralized KVS

To measure the reliability of the centralized server, we set the failure rate of the server to be dependent on the number of clients it is serving due to the increasing loads on the server. Considering an exponential distribution of server failures, the relationship between the server's up time and the number of clients is: $up\ time = Ae^{\lambda n}$, where A is the up time with zero client, λ is the failure rate, and n is the number of clients. Assuming a 2-month up time with zero client ($A = 1440$ hours), and a 1-month (i.e. 720 hours) up time with 1024 clients of a single server, we show the trend of the server up time with respect to the number of clients (dotted blue line) in Figure 8. The reliability decreases as the scale of the system increases.

At exascale, the above results would be amplified to pose serious operability concerns. *While not surprising results, these results motivate alternative distributed architectures that support scalability, reliability and consistency in a holistic manner.* These issues can be addressed by identifying the core components required by system software, such as a global naming system, an abstract KVS, a decentralized architecture, and a scalable, resilient overlay network.

3.3 HPC System Software Taxonomy

In contrast to the traditional HPC system software that are tightly coupled for synchronized workloads, SaaS developed for the Cloud domain is designed for loosely asynchronous embarrassingly parallel workloads in distributed systems with wide area networks. As HPC systems are approaching exascale, the HPC system software will need to be more asynchronous and loosely coupled to expose the ever-growing intra-node parallel-ism and hide latency. To be able to reason about general HPC system software at exascale, we devise a taxonomy by breaking system software down into various core components that can be composed into a full system software. We introduce the taxonomy, through which, we then categorize a set of system software.

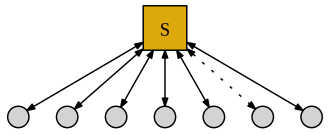
A system software can be primarily characterized by its *service model*, *data layout model*, *network model*, *recovery model*, and *consistency model*. These components are explained in detail as follows:

- *Service Model* describes system software functionality, architecture, and the roles of the software's composite entities. Other properties such as atomicity, consistency, isolation, durability (ACID) [28], availability, partition-tolerance etc. also are expressed as parts of the service model. These characteristics define the overall behavior of the system software and the constraints it imposes on the other models. A transient data aggregation tool, a centralized job scheduler, a resource manager with a single failover, a parallel file system are some examples of the service model.
- *Data Layout Model* defines the system software data distribution. In a centralized model, a single server is responsible for maintaining all the data. Alternatively, the data can be partitioned among distributed servers with varying levels of replication, such as partitioned (no replication), mirrored (full replication), and overlapped (partial replication).
- *Network Model* dictates how system software components are connected. In a distributed network, servers can form structured overlays – rings, binomial, k-ary, n-cube, radix trees; complete, binomial graphs; or unstructured overlay – random graphs. The system software could be further differentiated based on deterministic or non-deterministic information routing in the overlay network.

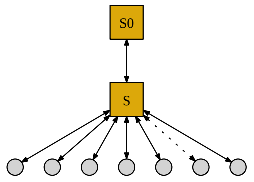
While some overlay networks imply a complete membership set (e.g. fully-connected), others assume a partial membership set (e.g. binomial graphs).

- *Recovery Model* describes how system software deals with server failures with minimum manual intervention. The common methods include failover, checkpoint-restart, and roll-forward. Triple modular redundancy and erasure coding [20] are additional ways to deal with failures and ensure data integrity. The recovery model can either be self-contained, such as recovery via logs from persistent storage, or require communication with replicas to retrieve.
- *Consistency Model* pertains to how rapidly data changes in a distributed system are propagated and kept coherent. Depending on the data layout model and the corresponding level of replication, system software may employ different levels of consistency. The level of consistency is a tradeoff between the server's response time and how tolerant clients are to stale data. It can also compound the complexity of recovery under failures. Servers could employ weak, strong, or eventual consistency depending on the importance of the data.

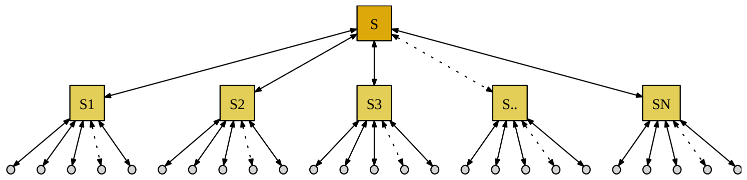
By combining specific instances of these components, we can define a system architecture of system software. Figure 9 and Figure 10 depict some specific system architectures derived from the taxonomy. For instance, c_{tree} is a system architecture with a centralized data layout model and a tree-based hierarchical overlay network; d_{fc} architecture has a distributed data layout model with a fully-connected overlay network, whereas d_{chord} architecture has a distributed data layout model and a Chord overlay network [29] with partial membership. Recovery and consistency models are not depicted, but would need to be identified to define a complete service architecture.



(a) c_{single}

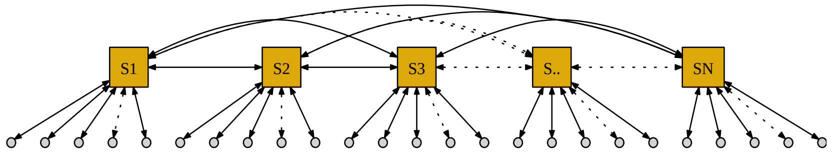


(b) c_{single} fail-over

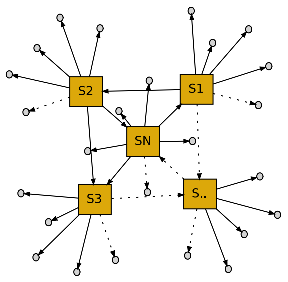


(c) c_{tree}

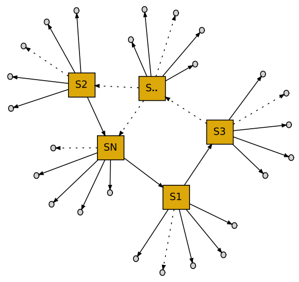
Figure 9. Centralized system architecture



(a) d_{fc}



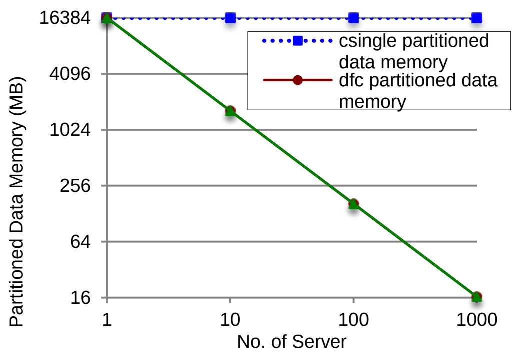
(b) d_{chord}



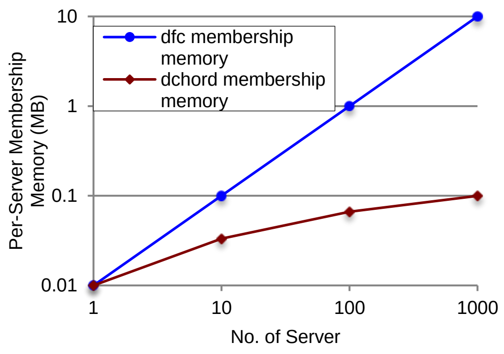
(c) d_{random}

Figure 10. Distributed system architecture

Looking into the memory requirements of these architectures allows deriving observations analytically. Figure 11 (a) shows the per-server memory requirement of the client data for different architectures, assuming 16GB client data. A single server must have the memory capacity to hold all the data, where the d_{fc} and d_{chord} architectures partition the data evenly across many servers. Figure 11 (b) illustrates the per-server memory requirements to store the server membership information, assuming each server identification is 10KB. This is trivial for a single server. For d_{fc}, it grows linearly with the number of severs, while for d_{chord}, the relationship is logarithm.



(a) Data memory per server



(b) Membership memory per server

Figure 11. Memory requirements for different architectures

To demonstrate how common HPC system software would fit into the taxonomy, we have classified, at a high-level, some representative system software in Table 1.

3.4 Key-value Stores Simulation

Having motivated KVS as a building block for extreme-scale HPC system software, using the taxonomy we narrow the parameter space and focus on the major KVS components. Then we can use simulation to evaluate the design spaces for any specific KVS applications before any implementation. Additionally, we can create modular KVS components that allow the easy creation of extreme-scale system software. This section presents the design and implementation details of a KVS simulator [232]. The simulator allows us to explore all the system architectures, namely c_{single}, c_{tree}, d_{fc} and d_{chord}. Here we assume a centralized data layout model for c_{single} and c_{tree}, and a distributed data layout model for d_{fc} and d_{chord}. The simulator is extendable to other network and data layout models. The architectures can be configured with N-way replication for the recovery model and either eventual or strong consistency for the consistency model. The conclusions that we will draw from KVS simulations can be generalized to other system software, such as job schedulers, resource managers, I/O forwarding, monitoring, and file systems.

Table 1. Representative system services categorized through the taxonomy

System Software	*Service Model*	*Data Layout Model*	*Network Model*	*Recovery Model*	*Consistency Model*
Voldemort	Key-value store	Distributed	Fully-connected	N-way Replication	Eventual
Cassandra	Key-Value Store	Distributed	Fully-connected	N-way Replication	Strong and Eventual
D1HT	Key-Value Store	Distributed	Hierarchical	N-way Replication	Strong
Pastry	Key-Value Store	Distributed	Partially-connected	N-way Replication	Strong
ZHT	Key-Value Store	Distributed	Fully-connected	N-Way Replication	Strong and Eventual
Riak	Key-value store	Distributed	Partially-connected	N-way Replication	Strong and Eventual
Charm++	Runtime System	Distributed	Hierarchical	N-way Replication	Strong
Legion	Runtime System	Distributed	Hierarchical	None	Strong
STAPL	Runtime System	Distributed	Nested/Hierarchical	N-Way Replication	Strong
HPX	Runtime System	Distributed	Fully-connected	N-Way Replication	Strong
SLURM	Resource Manager	Replicated	Centralized	Fail-Over	Strong
Slurm++	Resource Manager	Distributed	Fully-connected	Fail-Over	Strong
MATRIX	Task Scheduler	Distributed	Fully-connected	None	Strong
OpenSM	Fabric Manager	Replicated	Centralized	Fail-Over	Strong
MRNet	Data Aggregation	Centralized	Hierarchical	None	None
Lilith	Data Distribution	Replicated	Hierarchical	Fail-Over	Strong
Yggdrasil	Data Aggregation	Replicated	Hierarchical	Fail-Over	Strong
IOFSL	I/O Forwarding	Centralized	Hierarchical	None	None
FusionFS	File System	Distributed	Fully-connected	N-way Replication	Strong and Eventual

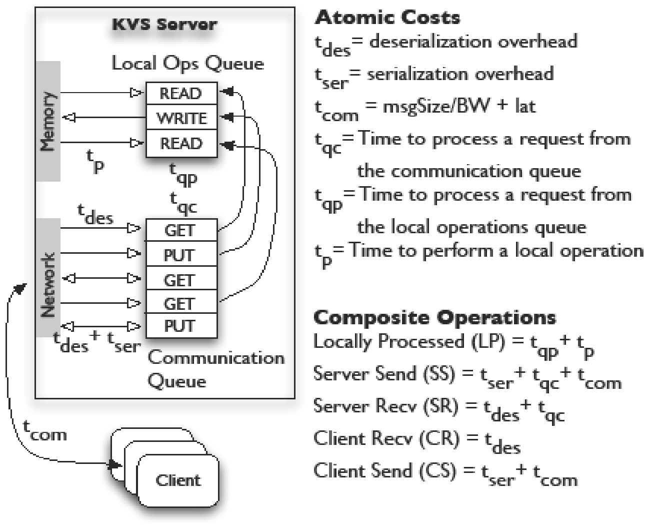
3.4.1 Simulator Overview. Simulations are conducted up to exascale levels with tens of thousands of nodes (each one has tens to hundreds of thousands threads of execution) running millions of clients and thousands of servers. The clients are simulated as compute daemons that communicate with the servers to store data and system states. The millions of clients are spread out as millions of compute daemon processes over all the highly parallel compute nodes. Furthermore, the number of clients and servers are configurable, and how a server is selected by client 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. The data records stored in the servers are (*key*, *value*) pairs; the *key* uniquely identifies the record and the *value* is the actual data object. By hashing the *key* through some hashing function (e.g. modular) over all the servers, the client knows the exact server that is storing the data.

Servers are modeled to maintain two queues: a *communication queue* for sending and receiving messages and a *processing queue* for handling incoming requests that operate on the local data. Requests regarding other servers' data cannot be handled locally and are forwarded to the corresponding servers. 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 server is responsible for.

For the 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.

3.4.2 Cost Parameters. The simulation results are dependent on the attribution of the communication and processing costs due to the system software architecture, we explain and justify the cost parameters in this subsection. Figure 12 shows the scenario of server selection with five enqueued operations, three of which are resolved locally and forwarded to the local operations queue, and two are resolved remotely by forwarding to other servers. All of the architectures under study are derived from this basic design using communication and processing costs.



Atomic Costs

t_{des} = deserialization overhead

t_{ser} = serialization overhead

t_{com} = msgSize/BW + lat

t_{qc} = Time to process a request from the communication queue

t_{qp} = Time to process a request from the local operations queue

t_p = Time to perform a local operation

Composite Operations

Locally Processed (LP) = $t_{qp} + t_p$

Server Send (SS) = $t_{ser} + t_{qc} + t_{com}$

Server Recv (SR) = $t_{des} + t_{qc}$

Client Recv (CR) = t_{des}

Client Send (CS) = $t_{ser} + t_{com}$

Figure 12. KVS client/server simulator design

The composite overheads include client send, (CS), client receive, (CR), server send (SS), server receive (SR) and local request processing, (LP). CS is comprised of message serialization time, t_{ser}, and message transmission time, t_{com}, calculated as $msgSize/BW + lat$, where $msgSize$ is the message size in bytes, BW is the peak network bandwidth and lat is half of the round-trip time (RTT). CR is the message deserialization overhead, t_{des}. SS includes the time when the server finishes the last queued task in the communication queue t_{qc}, the overhead of packing a message by the server t_{ss}, and t_{com}. SR is the summation of t_{qc}, the overhead of unpacking a message by the server t_{sr}. LP includes the time when the server finish processing the last queued request t_{qp}, and the request processing time t_p. For a locally resolved query, the time to finish it consists of client send overhead CS, server receive overhead SR, locally processed time LP, server send to client overhead SS, and client receive overhead CR. This is applicable to all the architectures. For a remotely resolved request in d_{fc}, the time to finish it includes client send overhead CS, server receive overhead SR, server forwarding request overhead $SS+SR$, locally processed time LP, server returning processing result overhead $SS+SR$, server send to client overhead SS, and client receive overhead CR. For a remotely resolved request of d_{chord} involving k hops to find the predecessor of the responsible server, the time to finish the request includes client send overhead CS, server receive overhead SR, overhead of finding the predecessor $2 \times k \times (SS+SR)$, server forwarding request overhead to the responsible server $SS+SR$, locally process LP, server returning processing result overhead $SS + SR$, server send to client overhead SS, and client receive overhead CR. The time to resolve a query locally (t_{LR}) and the time to resolve a remote query (t_{RR}) is given by

$$t_{LR} = CS + SR + LP + SS + CR$$

For d_{fc}: $$t_{RR} = t_{LR} + 2 \times (SS + SR)$$

For d_{chord}: $$t_{RR} = t_{LR} + 2 \times k \times (SS + SR)$$

where k is the number of hops to find the predecessor.

3.4.3 Data Layout and Network Models. The data layout and network models supported are: centralized data server (c_{single}), centralized data server with aggregation servers in a tree overlay (c_{tree}), distributed data servers with fully connected overlay (d_{fc}), distributed data servers with partial connected overlay (d_{chord}).

For c_{single} and c_{tree}, all data is stored in a single server. The main difference is that c_{tree} has a layer of aggregation servers to whom the client submits requests. The aggregation size (number of requests being packed before sending) of each individual aggregation server is dynamically changing according to the loads. It is equal to the number of clients, which havemore requests left to be processed, among all the clients that the aggregation server is responsible for. The upper bound is the number of clients the aggregation server serves. Currently, they only do request aggregation. In the future, it is possible to simulate PUT caches in these servers for read intensive workloads.

For d_{fc} and d_{chord}, the *key* space along with the associated data *value* is evenly partitioned among all the servers ensuring a perfect load balancing. In d_{fc}, data is hashed to the server in an interleaved way (*key* modular the server id), while in d_{chord}, consistent hashing [44] is the method for distributing data. The servers in d_{fc} have global knowledge of all servers, while in d_{chord}, each server has only partial knowledge of the other servers; specifically this is logarithm of the total number of servers with base 2 and is kept in a table referred to as the *finger table* in each server.

3.4.4 Recovery Model. The recovery model defines how a server recovers its state and how it rejoins the system after a failure. This includes how a recovered server recovers its data and how to update the replica information of other servers that are affected due to the recovery. The first replica of a failed server is notified by an external mechanism (EM) [29] (e.g. a monitoring system software that knows the status of all servers) when the primary server recovers. Then the first replica sends all the replicated data (including the data of the recovering server and of other servers for which the recovering server acts as a replica) to the recovered server. The recovery is done once the server acknowledges that it has received all data.

3.4.4.1 Failure/Recovery

Fail/recover events are generated because of servers failing and rejoining the dynamic overlay network. The servers can fail the system very frequently in an extreme-scale system, in which the mean-time-to-failure (MTTF) could be in the order of hours [25] [65]. For simplicity, in our simulator, we assume that fail/recover events happen at fixed periods and there is only one server failing or recovering in the system at a given time. At the very beginning, all servers are up in the system. When fail/recover event occurs, the simulator randomly picks one server and flips its status (up to down, down to up). In order to notify other servers about a node's failure, we implement both the eager and lazy methods. In the eager method, we assume an EM sending a failure message to notify other online servers. This would be done either with a broadcast message (in d_{fc}) or with chaining in which every node notifies the left node in their finger tables (for d_{chord}). In the lazy method, servers are not notified but realize the failure of a node when they try to communicatewith it. When a node recovers, it gets the membership list from the EM and then does a broadcast (d_{fc}), or it receives its finger table from the EM and then notify all the servers that should have the joined node in their finger tables (d_{chord}) [29]. In our simulations, the failure event without a replication model impies that when a server fails, all the messages (requests coming from the clients, or forwarding messages from other servers) would fail. In addition, the clients would not try the failed requests again, even if the failed server recovers. The purpose of this is to isolate the effect of failures so that it can be measured separately.

3.4.4.2 Failure/Recovery with Server Replication

We implement a replication model in the simulator for handling failures. In c_{single} and c_{tree}, one or more failovers are added; while in d_{fc} and d_{chord}, each server replicates its data in the consecutive servers (servers have consecutive id numbers from 0 to server count - 1). Failure events complicate server replication model. When a server fails, the first replica sends the failed server's data to an additional server to ensure that there are enough replicas. In addition, all the servers that replicate data on the failed server would also send their data to one more server. The clients can tolerate server failures by identifying the replicas of a server as consecutive servers.

Our simulator implements different policies for the clients to handle server failures, such as *timeouts*, *multi-trial*, and *round robin*. For example, in the *timeouts* policy, a client would wait a certain time for the server to respond. If the server does not respond after the timeout, the client then turns to the next replica. In addition, our simulator has the ability to handle communication failures by relying on the EM. The EM monitors the status of all the servers by issuing periodic heartbeat messages. When a link failure of a server happens, the EM detects it according to the failed heartbeat message and then notifies the affected clients, which then direct requests to the next replica.

3.4.5 Consistency Model. Our simulator implements two consistency models: strong consistency and eventual consistency [21].

3.4.5.1 Strong Consistency

In strong consistency, updates are made with atomicity guarantee so that no two replicas may store different values for the same *key* at any given time. A client sends requests to a dedicated server (*primary replica*). The *get* requests are processed and returned back immediately. The *put* requests are first processed locally and then sent to the replicas; the *primary replica* waits for an acknowledgement from each other replica before it responds back to the client. When a server recovers from failure, before getting back all its data, it caches all the requests directed to it. In addition, the first replica (notified by the *EM*) of the newly recovered server migrates all pending *put* requests, which should have been served by the recovered server, to the recovered server. This ensures that only the *primary replica* processes *put* requests at any time while there may be more than one replicas processing *get* requests.

3.4.5.2 Eventual Consistency

In eventual consistency, given a sufficiently long period of time over which no further updates are sent, all updates will propagate and all the replicas will be consistent eventually, although different replicas may have different versions of data of the same *key* at a given time. After a client finds the correct server, it sends requests to a random replica (called the *coordinator***)**. This is to model inconsistent updates of the same *key* and to achieve load balancing, among all the replicas. There are three key parameters to the consistency mechanism: the number of replicas–*N*, the number of replicas that must participate in a quorum for a successful *get* request–*R*, and the number of replicas that must participate in a quorum for a successful *put* request-*W*. We satisfy $R+W>N$ to guarantee "read our writes" [21]. Similar to Dynamo [34] and Voldemort [36], we use *vector clock* to track different data versions and detect conflicts. A *vector clock* is a <serverId, counter> pair for each *key* in each server. It specifies how many updates have been processed by the server for a *key*. If all counters in a *vector clock* V1 are no larger than all corresponding ones in a *vector clock* V2, then V1 precedes V2, and can be replaced by V2. If V1 overlaps with V2, then there is a conflict.

For a *get* request, the *coordinator* reads the value locally, sends the request to other replicas, and waits for <value, vector clock> responses. When a replica receives a *get* request, it first checks the corresponding *vector clock*. If it precedes the *coordinator's vector clock*, then the replica responds with success. Otherwise, the replica responds failure, along with its <value, vector clock> pair. The *coordinator* waits for $R-1$ successful responses, and returns all the versions of data to the client who is responsible for reconciliation (according to an application-specific rule such as "largest value wins") and writing back the reconciled version.

For a *put* request, the *coordinator* generates a new *vector clock* by incrementing the counter of the current one by 1, and writes the new version locally. Then the *coordinator* sends the request, along with the new *vector clock* to other replicas for quorum. If the new *vector clock* is preceded by a replica's, the replica accepts the update and responds success; otherwise, responds failure. If at least *W*−1 replicas respond success, the *put* request is considered successful.

3.4.6 KVS Simulator Implementation Details. After evaluating several simulation frameworks such as OMNET++ [46], OverSim [47], SimPy [48], PeerSim [49], we chose to develop the simulator on top of PeerSim because of its support for extreme scalability and dy-namicity. We use the discrete-event simulation (DES) [50] engine of PeerSim. In DES, every behavior in the system is converted to an event and tagged with an occurrence time. All the events are inserted in a global event queue that is sorted based on the event occurrence time. In every iteration, the simulation engine fetches the first event and executes the corresponding actions, which may result in following events. The simulation terminates when the queue is exhausted. The simulation procedure is depicted in Figure 13.

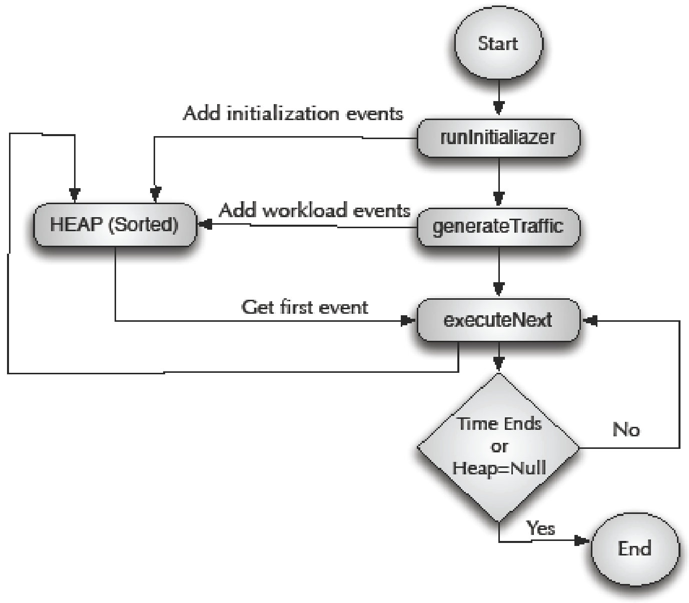


Figure 13. Simulation procedure flowchart

A variation of DES is parallel DES (PDES) [43], which takes advantage of the many-core architecture to access larger amount of memory and processor capacities, and 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.

Table 2. Parameter names and descriptions

Name	***Description***
BW	Network bandwidth
lat	Network congestion latency
$msgSize$	Message size
$idLength$	Key length (in bits)
t_{ss}	Server message packing overhead
t_{sr}	Server message unpacking overhead
t_{cs}	Client message packing overhead
t_{cr}	Client message unpacking overhead
t_p	Time to process a request locally
numReqPerClient	Number of request per client
numClientPerServ	Number of clients per server
FailureRate	Failure frequency
numReplica	Number of replicas
R	Number of succesful Get responses
W	Number of succesful Put responses
numTry	Number of retries before a client talks to the replica

Several parameters define the simulator environment and operation. The simulation is built on top of PeerSim, which is developed in Java, and has about 10,000 lines of code.

The input to the simulation is a configuration file, which specifies the system architecture, the values of the parameters, etc. The names and descriptions of the parameters are shown in Table 2. There are no other dependencies. The simulator is made open source on Github: https://github.com/kwangiit/KVSSim.

3.5 Evaluation

Our evaluation aims to give insights into the design spaces of HPC system software through KVS simulations, and to show the capabilities of our simulator in exposing costs inherent in design choices. We evaluate the overheads of different architectures as we vary the major components defined in section 3.3. We present results by incrementally adding complex features such as replication, failure/recovery, and consistency, so that we can measure the individual contributions to the overheads due to supporting these distributed features. This overhead is reflected in the communication intensity of the specific architectures and the additional communication because of the additional features.

3.5.1 Experimental Setup. All the experiments are run on the Fusion machine. The software versions used are: Sun 64-bit JDK version 1.6.0_22; PeerSim jar package version 1.0.5. The largest amount of memory required for any of the simulations is 25GB and the longest running time is 40 minutes (millions of clients, thousands of servers, and tens of millions of requests). Given our lightweight simulator, we can explore an extremely large-scale range.

3.5.1.1 Parameters

The parameters presented in Table 2 are displayed in Table 3 with their values. The network parameters are chosen to reflect large-scale systems such as the BG/P, and the

Kodiak cluster. The base request-processing time is taken from samples of processing time from services such as memcached [45] and ZHT [51].

Table 3. Simulation parameters

Name	***BG/P***	***Kodiak***
BW	6.8Gbps	1.0Gbps
lat	100us	12us
$msgSize$	10KB	10KB
$idLength$	128bits	128bits
t_{ss}	50us	40us
t_{sr}	50us	40us
t_{cs}	50us	40us
t_{cr}	50us	40us
t_p	500us	1500us
numReqPerClient	10	10
numClientPerServ	1024	1024
numTry	3	3

3.5.1.2 Workloads

The simulations are performed with up to 1 million clients each submitting 10 Get or Put requests. We did experiments to verify that higher numbers (e.g. 100, 1k, 10k) of get/put requests gave the same results. These values are configurable by changing the parameters, *numReqPerClient* and *numClientPerServ*, from Table 3. For d_{fc} and d_{chord}, we increment the number of clients by 1024 and the number of servers by 1 as we scale.

In exploring the overhead of different distributed system service features (section 3.5.2 through section 3.5.6), we use the synthetic workload, in which, 10M tuples of <*type*, *key, value*> are generated with a uniform random distribution (URD) (50% Gets and 50% Puts) and placed in a workload file. Each client would then read 10 requests in turn and execute their workloads. Realistic workloads are also employed. They are described in more details and applied in section 3.5.7, where we feed the KVS simulator with these distributed HPC service traces to show the generality of KVS.

3.5.1.3 Validation

We validate our simulator against two real systems: a zero-hop KVS, ZHT [51], and an open-source implementation of Amazon Dynamo KVS, Voldemort [36]. Both systems serve as building block for system services. ZHT is used to manage metadata of file systems (FusionFS), monitor task execution information of job scheduling systems (MATRIX), and to store the resource and job information for our distributed job launch prototype which is under improvement, while Voldemort is used to store data for the LinkedIn professional network.

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 BG/P machine 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 14 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 larger overheads.

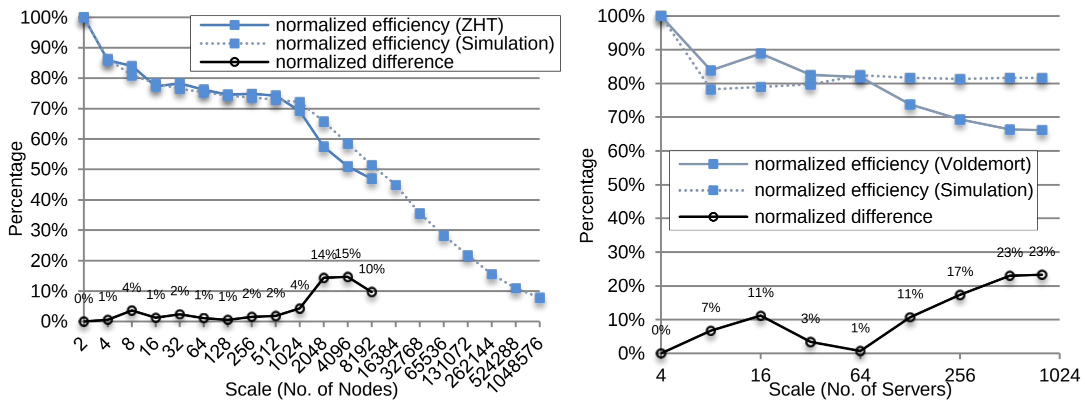


Figure 14. Validation of the simulator against ZHT and Voldemort

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 with up to 800 servers, and 800k clients. Each client submitted 10 random requests. As shown in Figure 14, 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 excessively 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 on each node, leading to serious resource over-subscription.

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

3.5.2 Architecture Comparisons. We compare different architectures (*c_{single}* vs *c_{tree}*, and *d_{fc}* vs *d_{chord}*) for the basic scenario (no replication, failure/recovery or consistency models) with synthetic workloads to investigate the tradeoffs between these architectures at increasingly scales.

3.5.2.1 c_{single} vs. c_{tree}

Figure 15 shows the comparison between c_{single} and c_{tree}. We see that before 16 clients, c_{tree} performs worse than c_{single} due to that the small gather size (at most 16) is insufficient to make up the additional latency of the extra communication hop. Between 32 (1 aggregation server) to 16K clients (16 aggregation servers with each one managing 1K clients), c_{tree} performs better than c_{single} because of the larger gather sizes (32 to 1K). After 32K clients, the individual performance is degrading; the relative performance gap is decreasing and finally disappearing. This is because the per-request processing time is getting larger when the number of clients increases due to contentions, which renders that the communication overhead is negligible.

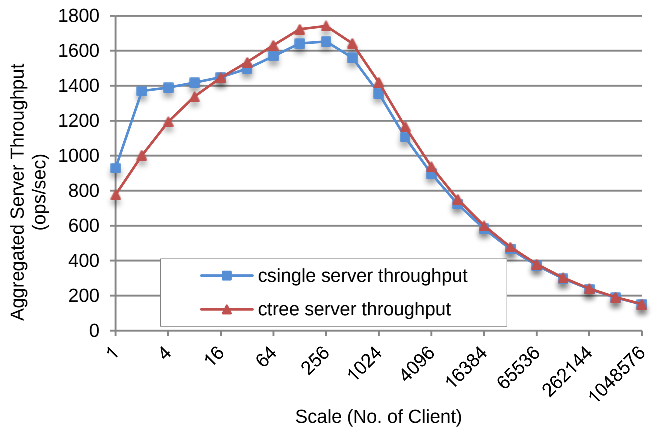


Figure 15. Throughput of csingle vs ctree

To model the server contention due to the increasing number of clients, we run a prototype of a centralized KVS (c_{single}) implementation up to 1K nodes and apply a polynomial regression on the processing time with respect to the number of clients with the base of 500ms. Within 1K nodes, the changing of processing time is shown in Table 4. The 1K client-processing time (637 ms) is used in d_{fc} and d_{chord} as each server manages 1K clients. For c_{single} and c_{tree}, Beyond 1K nodes, we increase the processing time linearly with respect to the client count.

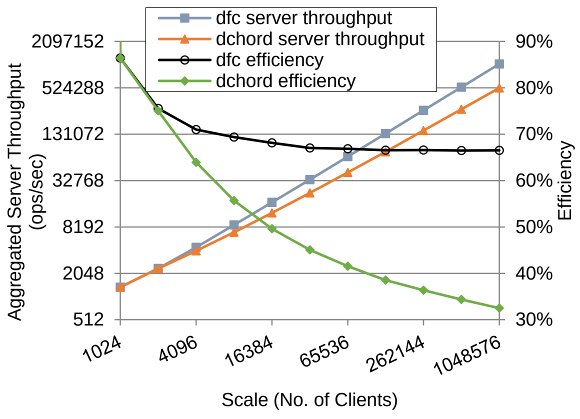
In Figure 15, the values after 1K clients are linear models. There could be other models (e.g. logarithm, polynomial, exponential) between processing time and the number of clients depending on the server implementation (e.g. multi-threading, event-driven, etc). We only use the calibrated values up to 1K clients. We show the results after 1K clients merely to point out that *there is a severe server contention in a single server at large scales, leading to poor scalability of the centralized architecture.*

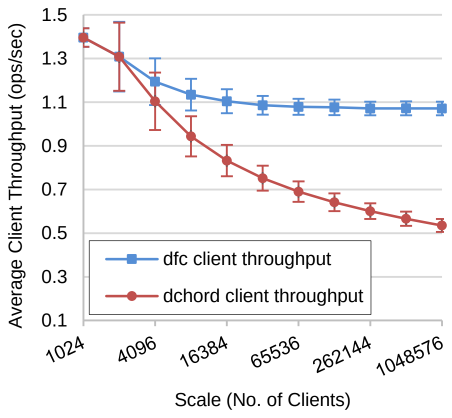
Table 4. Processing time as a function of number of clients

Number of Clients	1	2	4	8	16	32	64	128	256	512	1K
Processing Time (ms)	613	611	608	601	588	567	537	509	505	541	637

3.5.2.2 dfc vs. dchord

The comparison between d_{fc} and d_{chord} is shown in Figure 16. Each server is configured to manage 1K clients. With d_{fc}, we observe that the server throughput almost scales linearly with respect to the server count, and the efficiency has a fairly constant value (67%) at extreme scales, meaning that d_{fc} has great scalability. With d_{chord}, we see slightly less throughput as we scale up, and the efficiency decreases smoothly (Figure 16(a)). This is due to the additional routing required by d_{chord} to satisfy requests: one-hop maximum for d_{fc} and *logN* hops for d_{chord}. We show the average per-client throughput for both d_{fc} and *d_{chord}* in Figure 16(b). Up to 1M clients, d_{fc} is about twice as fast as d_{chord} from the client's perspective. From the error bars, we see that d_{chord} has higher deviation of the per-client throughput than that of d_{fc}. This is again due to the extra hops required to find the correct server in d_{chord}.



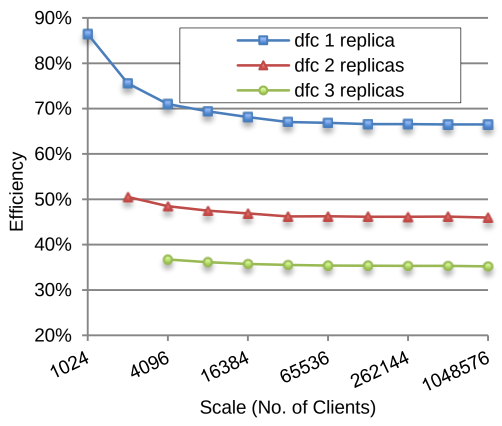


(a) Server throughput and efficiency (b) Average throughput per client

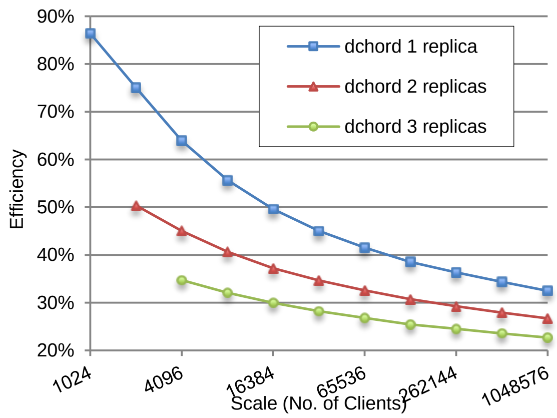
Figure 16. d_{fc} and d_{chord} performance comparison

The conclusion is that at the base case, the partial connectivity of d_{chord} results in latency as high as twice as that of the full connectivity of d_{fc}, due to the extra routing.

3.5.3 Replication Overhead. This section investigates the replication overhead associated with d_{fc} and d_{chord}. The data is collected for comparison of 1 to 3 replicas and the results are shown in Figure 17. It shows that there is always additional cost for additional replicas due to the added communication and processing overhead involved in propagating the Put requests to the extra replicas. Comparing d_{fc} and d_{chord}, we see that d_{fc} has more overhead than dchord when adding extra replicas. This is due to the low efficiency of d_{chord}, since d_{chord} has higher overhead for routing, the additional fixed overhead for the replicas is relatively small when comparing with the routing overhead. d_{fc} the relatively low routing overhead results in a larger impact on efficiency. In d_{fc}, the first added replica adds over 20% overhead (67% to 46%) and decreases the efficiency by 29.9% (20% / 67%); the second added replica introduces over 10% overhead (46% to 35%) and decreases the efficiency by 23.9%. While in d_{chord}, overheads of the first and second added replicas are 6% (33% to 27%) and 4% (27% to 23%), and efficiency decreased by 18.2% and 14.8%, respectively at the largest scale.



(a) d_{fc}

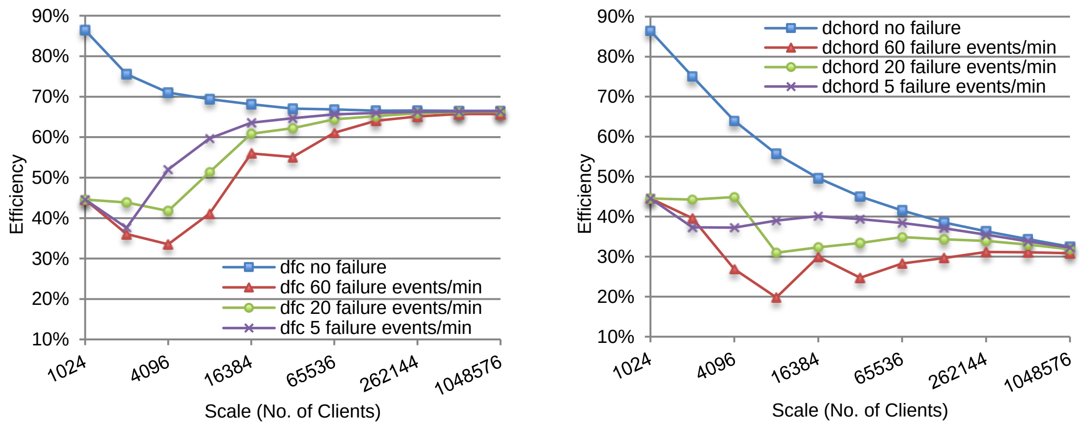


(b) d_{chord}

Figure 17. Replication overheads

3.5.4 Server Failure Effects. We add failure events (servers fail and possibly recover) to the simulator to emulate the failure rates of extreme-scale class systems. Here we do not use extra replicas, a request to a "failed" server would be dropped and the client would not retry. This is to measure the overhead of dynamicity of the service architectures. A failure event has to be forwarded to every other server in the d_{fc} network, whereas in the d_{chord} network, it is sent to $logN$ nodes resulting $(logN)^2$ messages. Different failure frequencies (high 60/min, medium 20/min, and low 5/min) are studied with the results shown in Figure 18, for both d_{fc} and d_{chord}.

As seen in Figure 18, the higher the failure frequency is, the more overhead introduced (lower efficiency curve for higher failure frequency). However, the dominating factor is the client messages. These client request-processing messages dwarf the number of communication messages of the failure events, which is a secondary factor even at the frequency of 60 events/min. Furthermore, this effect is getting dominating as the system scales up; the efficiency gaps are getting smaller and smaller until they disappear at the largest scales. For example: given 1M clients each sending 10 requests, 1K servers, and 5 failure events, for d_{fc}, we have at most 10M client-forwarding messages and failure events only require 51K messages (small compared to 10M), while for d_{chord}, we have 1M*log(1K)=10M forwarding message, and $5(\log(1K))^2=500$ failure messages. This illustrates how client-request messages dominate even with the added messages required to deal with server failures and recovery. Figure 18 shows that d_{fc} is more efficient than d_{chord} at the studied failure rates.



(a) d_{fc} (b) d_{chord}

Figure 18. Server failure effects

In order to validate the correctness of failure events represented in our simulator, we show the number of communication messages of one failure event in Figure 19. In this experiment, we expect to see d_{fc} increases linearly while d_{chord} increases logarithmically, we turn off the client workload messages and configure the simulator to process merely 10 failure events, and collect the average number of messages. The regression models in Figure 19 validate the linear and logarithmic relationships of the number of messages with respect to the number of servers for *d_{fc}* and *d_{chord}*, respectively. The R-Square values of the models are 0.865 and 0.998, which demonstrate that our models are representative.

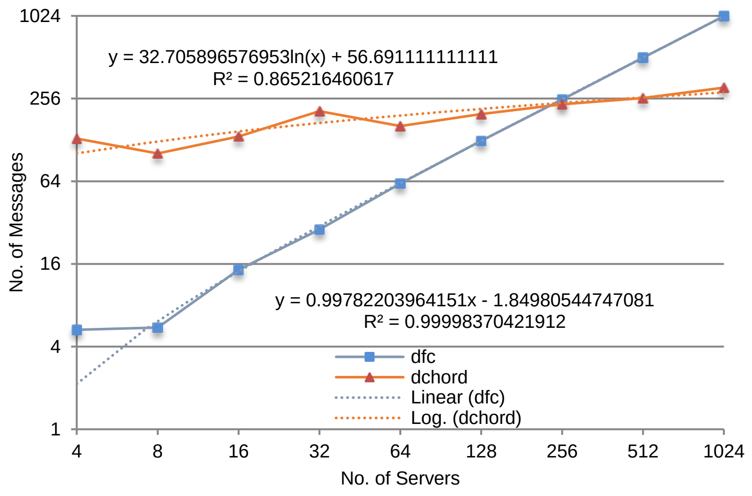


Figure 19. Regression of the number of messages of failure events

3.5.5 Server Failures with Replication. This section explores the overhead of failure events when a server is configured to keep updated replicas for resilience. We choose the *multi-trial* policy: the clients resend the failed requests to the primary server several times (an input parameter *numTry*, which is set to a default value of 3) before turning to the next replica.

Figure 20 displays the *efficiency* comparison between the base d_{fc} and d_{fc} configured with failure events and replication, and between the base d_{chord} and d_{chord} configured with failure events and replication, respectively. We use 3 replicas, set the failure rate to be 5 failure events per minute, and apply a strong consistency model. As seen in Figure 20, both d_{fc} and d_{chord} have significant *efficiency* degradation when failures and replication are enabled (blue solid line vs blue dotted line, red solid line vs red dotted line). The performance degradation of d_{fc} is more severe than that of d_{chord} - 44% (67% to 23%) for d_{fc} vs. 17% (32% to 15%) for d_{chord}. We explain the reasons with the help of Table 5.

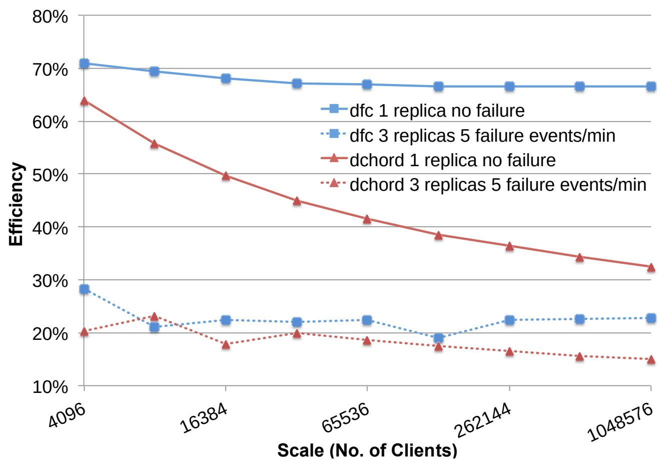


Figure 20. Server failure effect with replication

Table 5. Message count (#msg) for d_{fc}, d_{chord} with and without failure and replica (F&R)

	Request-process #msg				***failure #msg***		***strong consistency #msg***	
# Clients	d_{fc}	d_{chord}	d_{fc} (F&R)	d_{chord} (F&R)	d_{fc} (F&R)	d_{chord} (F&R)	d_{fc} (F&R)	d_{chord} (F&R)
4096	143.4K	185.0K	312.4K	246.2K	33	4.5K	217.7K	87.1K
8192	307.3K	491.1K	404.1K	596.2K	42	445	175.4K	170.9K
16384	634.8K	1.2M	726.1K	1.5M	66	28.6K	336.5K	377.1K
32768	1.3M	2.8M	1.4M	3.0M	114	712	665.4K	662.0K
65536	2.6M	6.4M	2.7M	6.6M	210	590	1.3M	1.3M
131072	5.2M	14.3M	5.3M	14.5M	402	888	2.6M	2.6M
262144	10.5M	31.3M	10.6M	31.7M	786	996	5.3M	5.2M
524288	21.0M	67.9M	21.1M	68.4M	1.6K	1.1K	10.5M	10.5M
1048576	41.9M	146.6M	42.0M	147.1M	3.1K	1.3K	21.0M	21.0M

Table 5 lists the message count of each property (process request, failure, strong consistency) for both d_{fc} and d_{chord}. We see that at extreme scales, the request-process message count (dominant factor) does not increase much when turning on failures and replicas for both d_{fc} and d_{chord}. The message count of failure event is negligible, and of strong consistency increases significantly at the same rate for both d_{fc} and d_{chord}. However, these added messages account for 1/3(20M/60M) for d_{fc}, while less than 1/8 (20M/170M) for d_{chord}. Due to the high request-process message count in d_{chord}, the overhead of d_{fc} seems more severe. The replication overhead is costly, which indicates that tuning a system software to the appropriate number of replicas will have a large impact on performance.

3.5.6 Strong and Eventual Consistency. We compare the consistency models. We enable failures with 5 failure events per minute and use 3 replicas. Like Dynamo [34], we configure (N, R, W) to be (3, 2, 2). Figure 21 shows the *efficiency* results of both d_{fc} and d_{chord}.

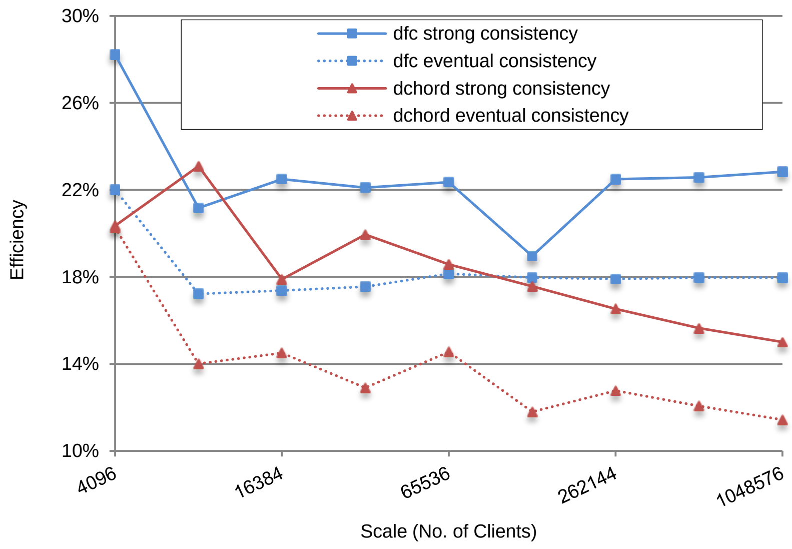


Figure 21. Strong consistency and eventual consistency

We see that eventual consistency has larger overhead than strong consistency. From strong to eventual consistency, *efficiency* reduces by 4.5% for d_{fc} and 3% for d_{chord} at extreme scales. We also list the number of messages for request-process, failure events, and consistency models in Table 6. We observe that the request-process message count doesn't vary much for both d_{fc} and d_{chord}. However, for consistency messages, eventual consistency introduces about as twice (41M/21M) the number of messages as that of strong consistency. This is because in eventual consistency, each request would be forwarded to all N=3 replicas and the server waits for R=2 and W=2 successful acknowledgments. With strong consistency, just the *put* requests would be forwarded to all other replicas. Eventual consistency gives faster response times to the clients but with larger cost of communication overhead.

Table 6. Message count (#msg) of strong consistency (*sc*) and eventual consistency (*ec*)

	process #msg				***failure #msg***				***consistency #msg***			
	sc		***ec***		***sc***		***ec***		***sc***		***ec***	
#client	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}
4K	312K	246.2K	141.5K	211K	30	4.6K	30	360	217.7K	87K	167.2K	164.5K
8K	404K	596.2K	391.9K	683K	40	450	50	590	175.4K	171K	340.2K	328.2K
16K	726K	1.5M	733.1K	1.5M	67	28.6K	90	23.7K	336.5K	377K	668.2K	655.4K
32K	1.4M	3.0M	1.4M	3.1M	110	710	150	830	665.4K	662K	1.3M	1.3M
66K	2.7M	6.7M	2.7M	6.6M	210	590	210	770	1.3M	1.3M	2.6M	2.6M
128K	5.3M	14.5M	5.3M	14.8M	400	890	530	1.1K	2.6M	2.6M	5.3M	5.3M
256K	21M	68.4M	21.0M	68.7M	1.6K	1.1K	2.1K	1.4K	10.5M	10.5M	21.0M	21.0M
1M	42M	147.1M	42.0M	148M	3.1K	1.3K	4.1K	1.6K	21.0M	21.0M	42.0M	42.0M

3.5.7 KVS Applicability to HPC System Software. In this section, we show that KVS can be used as building block for developing HPC system software. First, we conduct simulations with three workloads, which were obtained from real traces of three system software: job launch using SLURM [30], monitoring by Linux Syslog, and I/O forwarding using the FusionFS [42] distributed file system. In the next two chapters, we will see two

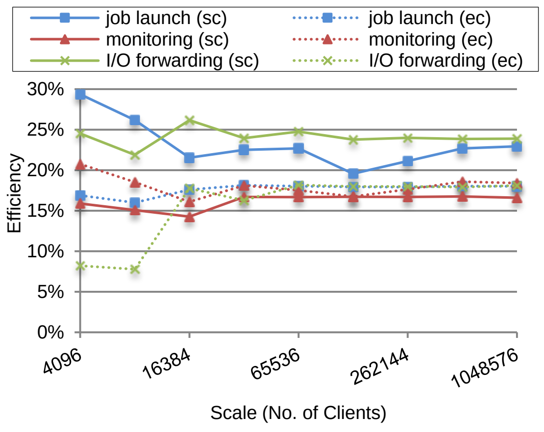
RMS that use a KVS (i.e. ZHT [51]) for distributed state management, namely the SLURM++ workload manager [22], and a MTC task scheduler, MATRIX [23].

3.5.7.1 Simulation with Real Workload Traces

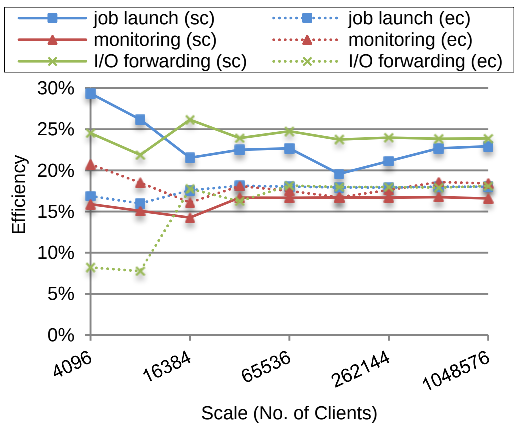
We run simulations with three workloads obtained from typical HPC system software, listed as follows:

- *Job Launch:* this workload is obtained from monitoring the messages between the server and client during an MPI job launch in SLURM resource manager. Though the job launch is not implemented in a distributed fashion, the messages should be representative regardless of the server structure, and in turn drive the communications between the distributed servers. The workload is characterized with the controlling messages of the slurmctld (*get*) and the results returning from the slurmds (*put*).
- *Monitoring:* we get this workload from a 1600-node cluster's syslog data. The data is categorized by message-type (denoting the *key* space) and count (denoting the frequency of each message). This distribution is used to generate the workload that is completely *put* dominated.
- *I/O Forwarding:* We generate this workload by running the FusionFS distributed file system. The client creates 100 files and operates (*reads* or *writes* with 50% probability) on each file once. We collect the logs of the ZHT metadata servers that are integrated in FusionFS.

We extend these workloads to make them large enough for exascale systems. For job launch and I/O forwarding, we repeat the workloads several times until reaching 10M requests, and the *key* of each request is generated with uniform random distribution (URD) within 64-bit *key* space. The monitoring workload has 77 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 64-bit *key* space. We point out that these extensions reflect some important properties of each workload, even though cannot reflect every details: 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. We run these workloads in our simulator, and present the *efficiency* results for d_{fc} and d_{chord} with both strong and eventual consistency, in Figure 22. We see that for job launch and I/O forwarding, eventual consistency performs worse than strong consistency. This is because both workloads have almost URD for request type and the *key*. For monitoring workload, eventual consistency does better because all requests are *put* type. The strong consistency requires acknowledgments from all the other *N*-1 replicas, while the eventual consistency just requires *W*-1 acknowledgments. Another fact is that the monitoring workload has the lowest *efficiency* because the *key* space is not uniformly generated, resulting in poor load balancing.



(a) d_{fc}



(b) d_{chord}

Figure 22. dfc and dchord with real workloads

The above results demonstrate that our KVS framework can simulate various system software as long as the workloads could be mirrored to *put* or *get* requests, which is true for the HPC system software we have investigated.

3.6 Conclusions and Impact

The goal of this work was to propose a general system software taxonomy for exascale HPC system software, and to ascertain that a specific HPC system software should be implemented at certain scales with certain levels of replication and consistency as distributed systems. We devised a *system software taxonomy*. Four classes of system architectures were studied through a *key-value store simulator*. We conducted extreme-scale experiments to quantify the overheads of different recovery, replication and consistency models for these architectures. We also showed how KVS could be used as a building block for general system software. The motivation was that a centralized server architecture does not scale and is a single point of failure. Distributed system architectures are necessary to expose the extreme parallelism, to hide latency, to maximize locality, and to build scalable and reliable system software at exascale.

The conclusions of this work are: (1) KVS is a viable building block; (2) when there are a huge amount of client requests, d_{fc} scales well under moderate failure frequency, with different replication and consistency models, while d_{chord} scales moderately with less expensive overhead; (3) when the communication is dominated by server messages (due to failure/recovery, replication and consistency), d_{chord} will have an advantage; (4) 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 (shown in Figure 23).

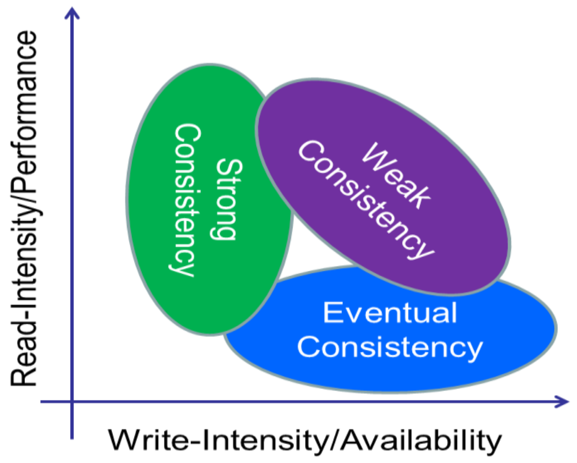


Figure 23. Guide to choose different consistency models

We have high hope that the proposed HPC system software taxonomy, along with the idea of using key-value store as a building block for system state management, will lay the foundations of developing and deploying next generation system software for exascale supercomputers. The design philosophies of the HPC system software need to be dramatically transformed to better support the tremendous growths of both intra-node and inter-node parallelism of the upcoming exascale machines, from centralized architectures to more distributed ones for better scalability, bringing new challenges of maintaining high available, consistent services. As there are numerous system software needed on exascale machines, from the top application level to the bottom OS level, the taxonomy will be very handy in categorize them and characterize the system software of each category. In this way, we only need to solve the challenges of one system software, and the solutions will automatically apply to others in the same category. This will open doors to new research directions of developing more taxonomies that are inclusive.

The proposal of using key-value stores as a building block is meaningful and will have long-term impacts on the exascale HPC system software community. Many system software will be beneficial from using key-value stores as a system state management service, including those that support system booting, system monitoring, hardware or software configuration and management, job and resource management, I/O forwarding, and various runtime systems for programming models and communication libraries. We will see examples of resource management and job/task scheduling system software that are built on top of key-value store in the following chapters of this dissertation. This will drive new research directions of developing scalable key-value stores for HPC environments. Furthermore, as key-value stores have been widely used in the Internet domains, this work will be able to bridge the gap between HPC and cloud computing, by running suitable HPC applications in the cloud environment, and vice versa, in order to improve the accessibility, system utilization, performance of the distributed systems as an ensemble.

CHAPTER 4

MATRIX TASK EXECUTION FRAMEWORK

Large-scale scientific applications of many-task computing (MTC) are well suited to be run on exascale machines that will be comprised of hundreds of thousands to millions of nodes with up to a billion threads of execution. Exascale MTC applications will likely employ over-decomposition to generate many more tasks (e.g. billions) than available parallelism, which are loosely coupled and fine-grained in both job/task sizes (e.g. per-core) and durations (e.g. from millisecond to hours). Over-decomposition has been shown to improve utilization at large scales as well to make fault tolerance more efficient, but poses significant challenges on the job schedulers to deliver orders of magnitude higher throughput (e.g. millions/sec) than that can be achieved through the state-of-the-art centralized schedulers. This chapter aims to address the scheduling challenges and throughput needs of exascale MTC applications through a distributed scheduling architecture and load-balancing technique (i.e. work stealing). We explore the scalability of work stealing in balancing workloads through a lightweight discrete event simulator, SimMatrix, which simulates both centralized and distributed MTC task execution frameworks up to 1-million nodes, 1-billion cores, and 100 billion tasks, on both shared-memory many-core machine and distributed exascale billion-core machine. With the insights gained from the simulations, we develop a real distributed task execution framework, MATRIX, which implements the work stealing technique to achieve distributed load balancing, and employs a distributed key-value store (i.e. ZHT) to manage task metadata. MATRIX is deployed on an IBM Blue Gene/P (BG/P) supercomputer and

Amazon cloud running both independent tasks and tasks with dependencies specified as Direct Acyclic Graphs (DAG). We compare MATRIX with Falkon (a centralized MTC task scheduler) on BG/P up to 4K cores. MATRIX maintains throughputs as high as 13K tasks/sec and 90%+ efficiency for 1-sec tasks while Falkon only achieves 20% efficiency. We also compare MATRIX with Sparrow and CloudKon in cloud up to 64 EC2 instances, and MATRIX outperforms Sparrow and CloudKon by 9X and 5X respectively with 67K tasks/sec and 80%+ efficiency. We expect the improvements to grow at larger scales due to the distributed nature of work stealing. We believe that work stealing is generalizable to other domains, such as scheduling on many-core computing, grids, data centers, and clouds.

4.1 Many-task Computing Task Execution Framework

Applications that fit into the many-task computing (MTC) category cover a wide range of scientific domains, ranging from astronomy, astrophysics, bioinformatics, biometrics, chemistry, climate modeling, to data analytics, economics, medical imaging, neuroscience, pharmaceuticals, and physics [38] [40]. MTC uses over-decomposition [7] to decompose applications into embarrassingly parallel tasks that are loosely coupled and fine grained in both task sizes (e.g. per-core) and durations (e.g. millisecond to hours). Applications of MTC are structured as Direct Acyclic Graphs (DAGs), where the vertices are discrete tasks and the edges are data dependencies. MTC applications are well suited to be run on supercomputers, as the machines have tremendous computing capacity for the task executions, and high bandwidth networks for data movement among dependent tasks.

To better illustrate the fact that the tasks of MTC are fine-grained in task durations, we investigated the largest available trace of real MTC workloads, collected over a 17-

month period comprising of 173M tasks [38] [41]. We filtered out the logs to isolate only the 160K-core BG/P supercomputer from Argonne National Laboratory, 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 of the 34.8M tasks, shown in Figure 24.

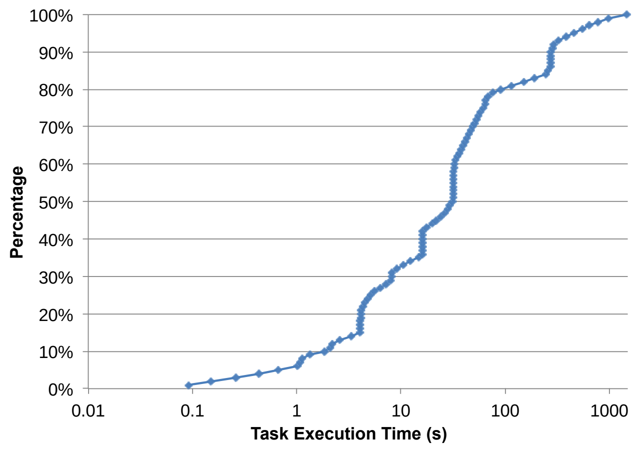


Figure 24. Cumulative Distribution Function of the MTC workloads

We see that about 10% of the tasks have lengths less than 1 second, most of the tasks have lengths ranging from several seconds to hundreds of seconds, and the medium task length is about 30 seconds, which is just one third of the average (95.2 seconds). These 34.8M tasks are partitioned into 1395 BOTs (where BOTs are clearly marked in the logs).

With the supercomputers approaching exascale with billions of threads of execution, we expect that the task lengths will be scaled down accordingly because the machines will be more powerful, resulting in many more short tasks. In addition, the MTC applications are also developing and scaling up with an extremely fast speed, resulting in exposing orders of magnitudes of fine-grained tasks (e.g. billions) that need to be scheduled and executed within a short amount of time, in order to achieve high system utilization.

However, the current job schedulers and execution frameworks of supercomputers have been designed around the centralized paradigm for tightly coupled long-running HPC applications, which are not able to fulfill the scheduling needs of MTC applications, due to the lack of scheduling granularity at the core/thread level and the inherent scalability limitation of the centralized architecture. To bridge the gap between the needs of task scheduling and execution, and the poor performance of the current schedulers of supercomputers for MTC applications, we make efforts to develop scalable task scheduling and execution framework towards exascale computing.

We propose that the task scheduling framework of MTC in an exascale computing environment will need to be fully distributed in order to achieve high performance, meaning high throughput (e.g. millions/sec), utilization, availability, as well as low latency, as opposed to the centralized and/or hierarchical designs. There should be as many schedulers as the compute nodes, forming a 1:1 mapping. This is demanded, as at exascale, each compute node will have high intra-node parallelism (e.g. thousands to tens of thousands cores), which requires one dedicated scheduler to manage resource and schedule the fine-grained tasks in local, preserving the data locality and hiding the latency. 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. The fully distributed architecture is scalable, as all the schedulers could receive workloads and participate in scheduling tasks. Therefore, ideally, the throughput would gain linear speedup as the system scales up. In addition, the architecture will be fault tolerant, leading to high availability, because a node failure only affects tasks running on that node. Furthermore, the simulation work in chapter 3 showed that fully distributed architecture has advantages over others when the client-processing events dominate, which is true for MTC applications because the number of tasks needed to be scheduled and executed are huge.

Load balancing [102] is an important goal of fully distributed task execution framework of MTC. Load balancing refers to distributing workloads as evenly as possible across all the schedulers, and it is critical given that a single heavily loaded scheduler would lower the system utilization significantly. However, it is challenging to achieve load balancing in fully distributed architecture because not only a scheduler has limited partial knowledge of its own state, but the task executions of applications may cause load imbalance during runtime. Therefore, load balancing must be achieved through the corporations of all the schedulers dynamically.

In this chapter, we propose an adaptive work stealing technique [87] to achieve distributed load balancing at extreme scales. We explore the scalability of the work stealing technique through a lightweight discrete event simulator, SimMatrix [50], which simulates task scheduling framework comprising of millions of nodes and billions of cores/tasks. With the insights gained from the simulations, we develop a real distributed task execution framework, MATRIX, which implements the work stealing technique to achieve distributed load balancing, and employs a distributed key-value store (i.e. ZHT [51]) to manage task metadata. This chapter focuses on evaluating the distributed architecture and scheduling techniques with compute-intensive MTC tasks that are either bag of tasks without dependencies, or have task dependencies on the task itself. We leave the topic of scheduling data-intensive MTC applications in Chapter 4. *The main contributions of this chapter can be summarized as follows:*

- Propose an adaptive work stealing technique, which applies dynamic multiple random neighbor selection, and adaptive polling interval strategies.
- Design and implement the SimMatrix simulator of MTC task scheduling system that simulates different scheduling architectures, such as centralized, hierarchical, and distributed, and various scheduling techniques, such as FIFO and work stealing.
- Implement a real distributed task execution framework, MATRIX, which adopts the adaptive work stealing for distributed load balancing and a distributed key-value store for task metadata management.
- Comparing the resource consumption and performance of SimMatrix with those of two existing distributed system simulators (i.e. SimGrid and GridSim).
- Evaluate work stealing via both SimMatrix with the simulations of various platforms and MATRIX, using a variety of workloads with task granularities ranging from milliseconds to seconds.
- Compare the performance of MATRIX with that of Falkon, Sparrow, and CloudKon on both the BG/P supercomputer and the Amazon cloud.

4.2 Fully Distributed Scheduling Architecture

We have motivated that the MTC paradigm will likely require a fully distributed task scheduling architecture for exascale machines. The architecture is shown in Figure 25. The scheduling system has four components: client, scheduler, executor, and key-value store (KVS). Each compute node runs a scheduler, an executor and a KVS server. The client issues requests to generate a set of tasks, puts task metadata into KVS servers, submits tasks to all schedulers, and monitors the execution progress. The schedulers are fully connected, and map tasks to the local executor. Whenever a scheduler has no tasks, it communicates with other schedulers to migrate ready tasks through load balancing techniques (e.g. work stealing). Each executor forks several (usually equals to number of physical cores of a machine) threads to execute ready tasks concurrently.

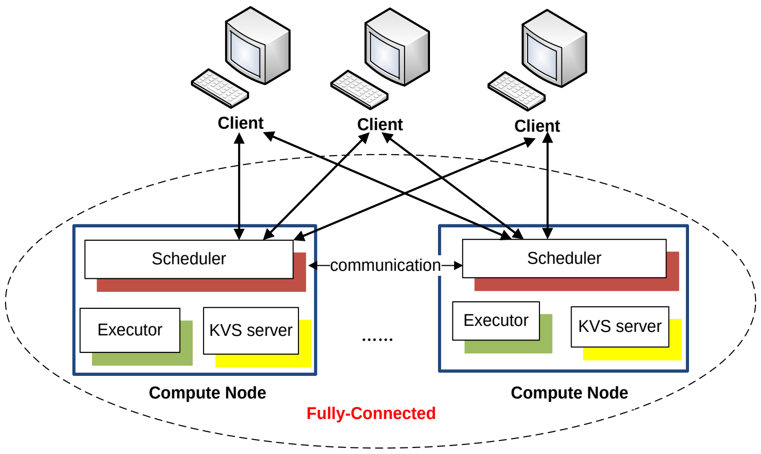


Figure 25. Fully distributed scheduling architecture

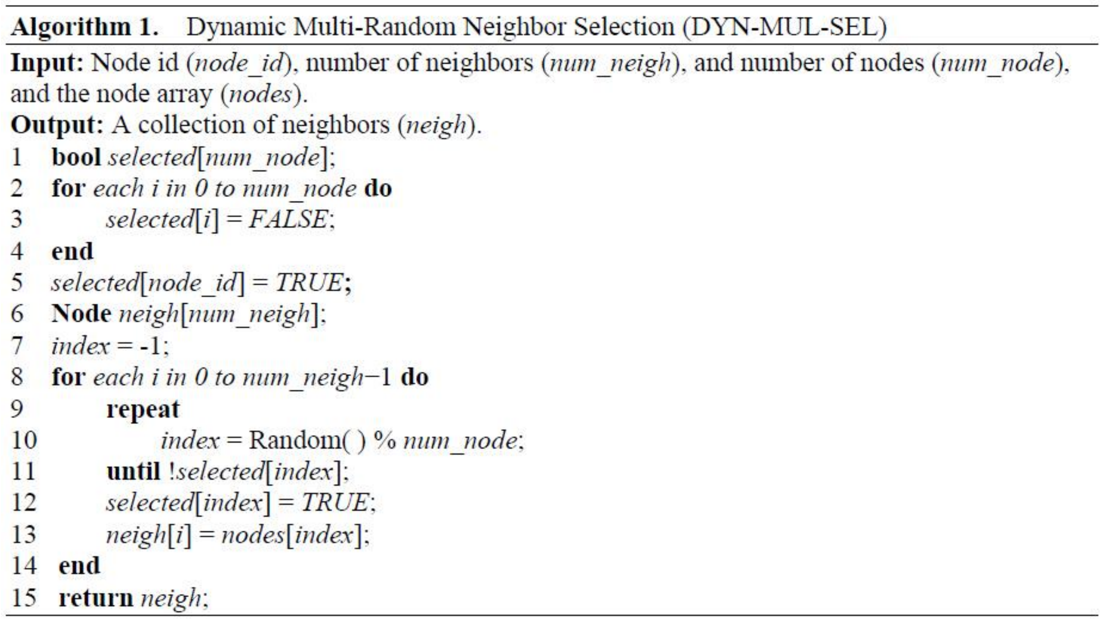
As chapter 3 proposed that KVS is a viable building block of extreme-scale system software, we integrate a distributed KVS, ZHT, in our scheduling system to monitor the execution progress and to keep the metadata of all the tasks and system states in a distributed, scalable, and fault tolerant way. ZHT is a zero-hop persistent distributed KVS with each ZHT client having a global view of all the ZHT servers. For each operation (e.g. insert, lookup, remove) of ZHT server, there is a corresponding client API. The client calls the API, which sends a request to the exact server (known to the client by hashing the key) that is responsible for the request. Upon receiving a request, the ZHT server executes the corresponding operation. ZHT serves as a data management building block for extreme scale system software, and has been tested up to 32K cores on an IBM Blue Gene /P supercomputer [51].

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

4.3 Adaptive Work Stealing Technique

Work stealing has been proven as an efficient load balancing technique at the thread/core level in shared memory environment [87] [116]. It is a pull-based method in which the idle processors randomly steal tasks from the overloaded ones. Our fully distributed scheduling system adopts work stealing at the node level in distributed environment. Work stealing is proceeded by the idle schedulers stealing workloads from others, as referred to neighbors. When a scheduler is idle, it randomly chooses some candidate neighbors. Then, it goes through all neighbors in sequential to query the "load information" (number of ready tasks), and tries to steal tasks from the most heavily loaded neighbor. In fully distributed architecture, every scheduler has a global membership list and is aware of all others. Therefore, the selection of candidate neighbors (victims) from which an idle scheduler (thief) steals tasks could be static or dynamic/random.

In the static mechanism, the neighbors of a scheduler are pre-determined as consecutive nearby identities and will not change. This mechanism has limitation that every scheduler is confined to communicate with part of other schedulers. In the dynamic case, whenever a scheduler is idle, it randomly chooses some candidate neighbors from the membership list. The traditional work stealing algorithm randomly selects one neighbor [105] per stealing operation, leading to poor performance at extreme scales. Instead, we choose to have a *multiple random neighbor selection strategy* that randomly selects several candidate neighbors instead of one, and chooses the most heavily loaded one to steal tasks upon every work stealing operation. This dynamic mechanism increases the chances of choosing the overloaded neighbors to balance loads. On the other hand, the multiple neighbor section strategy introduces some overhead when selecting neighbors; however, we show that this overhead is minimal (the time complexity is Θ(*n*), where *n* is the number of neighbors) in Algorithm 1, shown in Figure 26.



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*).

```
1   bool selected[num_node];
2   for each i in 0 to num_node do
3       selected[i] = FALSE;
4   end
5   selected[node_id] = TRUE;
6   Node neigh[num_neigh];
7   index = -1;
8   for each i in 0 to num_neigh−1 do
9       repeat
10          index = Random( ) % num_node;
11      until !selected[index];
12      selected[index] = TRUE;
13      neigh[i] = nodes[index];
14  end
15  return neigh;
```

Figure 26. Algorithm of dynamic multi-random neighbor selection

We prove the time complexity of Algorithm 1 of the dynamic multi-random neighbor section strategy is Θ(*n*), where $n = num\_neigh$, the number of neighbors to be chosen.

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 $i = 3$:

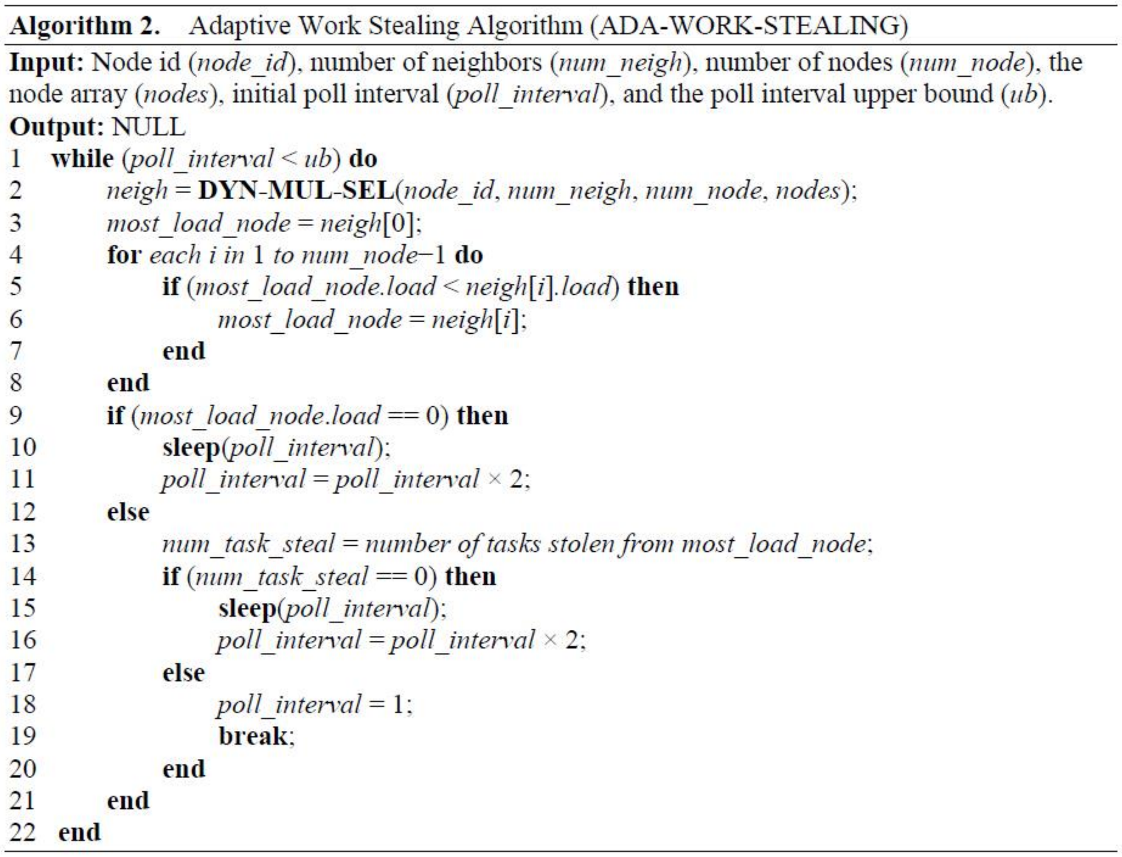
$$i \times (p_r)^{i-1} \times (1-p_r) = 3 \times \left(\frac{1}{1000}\right)^2 \times \left(1 - \frac{1}{1000}\right) \approx 3 \times 10^{-6} \approx 0$$

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

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

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

After an idle scheduler selects candidate neighbors, it queries each neighbor in sequential about the *load information*, referring to the number of tasks to be executed. Then, the idle scheduler tries to steal tasks from the most heavily overloaded one. When a scheduler fails to steal tasks from the selected neighbors, because either all neighbors have no tasks, or the most heavily loaded scheduler had already executed the reported tasks when actual stealing happens, it waits for a period of time and does work stealing again. We call this wait time the *poll interval*. Algorithm 2 shown in Figure 27 gives the overall work stealing procedure. Each scheduler checks the local task and resource information. Whenever a scheduler runs out of tasks to be executed, it signals the adaptive 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
```

Figure 27. Algorithm of adaptive work stealing

In order to improve the performance of the work stealing technique, we implement an adaptive *poll interval* strategy. If the *poll interval* is fixed, there would be difficulties to set the value to the right granularity. If the *poll interval* is too small, at the final stage when there are not many tasks left, many schedulers would poll neighbors to do work stealing.

These would eventually fail leading to more work stealing communications. If the *poll interval* is set large enough to limit the number of work stealing events, work stealing would not respond quickly to changing conditions, and lead to poor load balancing. Therefore, we propose an adaptive strategy to adjust the *poll interval* during runtime. The *poll interval* of a scheduler is changed dynamically similar to the exponential back-off approach used in the TCP networking protocol [89] to control the network congestion. The default value is set to be small (e.g. 1 ms). Once a scheduler successfully steals tasks, it sets the *poll interval* back to the default value. Otherwise, the scheduler waits the time of *poll interval* and doubles it and tries to do work stealing again. In addition, we set an upper bound, and whenever the *poll interval* hits the upper bound, a scheduler would not do work stealing anymore. This would significantly reduce the amount of failing work stealing operations at the final stage.

The parameters that can affect the performance of the adaptive work stealing technique are number of dynamic neighbors, number of tasks to steal, and the poll interval. The values of these parameters need to be set correctly according to the applications, and need to be changed dynamically at runtime according to the system state. Without the dynamic features, the work stealing technique may not scale efficiently up to exascale. We will show our thorough explorations of the work stealing parameters in simulations.

4.4 SimMatrix – Simulator of Task Execution Framework at Exascale

Research about real scheduling system is impossible at exascale, because not only we lack the exascale computers, but also the experimental results obtained from the real-world platforms are often irreproducible due to resource dynamics [88]. Therefore, we fall back to simulations to study various task scheduling architectures and algorithms. Like the key-value store simulator in Chapter 3, we develop a lightweight discrete event simulator (DES) [119], called SimMatrix [50], which simulates MTC task execution framework comprising of millions of nodes and billions of cores/tasks. Careful consideration was given to the SimMatrix architecture, to ensure that it would scale to exascale levels on modest resources of a single node. SimMatrix supports both centralized (e.g. first-in-first-out or FIFO) and distributed (e.g. work stealing) scheduling. We will use SimMatrix to study the scalability and performance of different scheduling architectures and techniques.

4.4.1 SimMatrix Architectures. The simulated scheduling architectures of SimMatrix are shown in Figure 28. For simplicity, we assign consecutive integer numbers as the node identities (node ids), ranging from 0 to the number of nodes N-1.

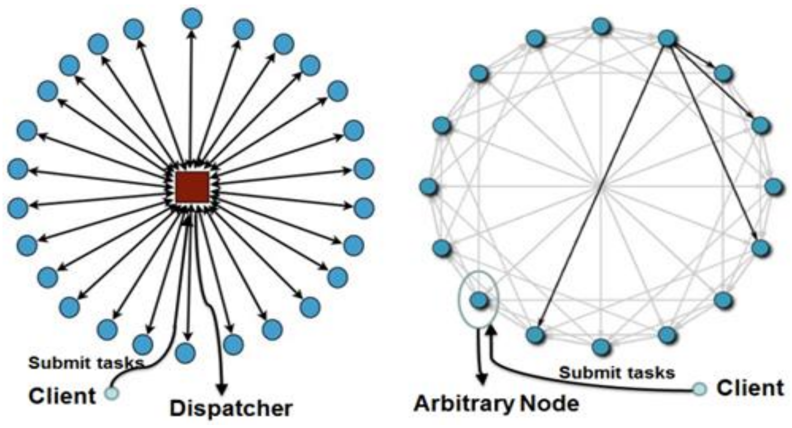


Figure 28. SimMatrix architectures: the left is centralized; the right is distributed

SimMatrix supports the granularity of scheduling at the node/core level at extreme scales. The simulated system could be centralized (Figure 28 left), where a single dispatcher (typically located on a single node called head node) maintains a task queue and manages the resources of all the compute nodes, the scheduling of tasks, and the task execution state updates. It could also be distributed (Figure 28 right), where each computing node maintains a task scheduler, and they manage local resources and cooperate with each other to achieve load balancing. The centralized approach suffers scalability and single-point-of failure. We believe that distributed scheduling with innovative load balancing techniques (e.g. work stealing) is the scheduling approach to exascale. Another one is the hierarchical architecture, where several schedulers are organized as a tree-based topology. SimMatrix could be easily extended to support hierarchical scheduling [233].

4.4.2 Centralized Scheduler. In the centralized architecture, a dispatcher maintains a task queue and manages the resources of all the compute nodes, the scheduling of tasks, and the task execution state updates. All tasks are submitted to the dispatcher by the client. The dispatcher then assigns tasks to the first node that has free executing cores using the FIFO policy [155]. None of the compute nodes has queues to hold extra tasks. If all cores are busy executing tasks, the dispatcher will wait until some tasks are finished releasing the corresponding cores. Then, it schedules tasks again to the nodes that have idle cores. This procedure continues until all the tasks are finished.

4.4.2.1 Task Description

Each task is modeled to have various attributes. The basic attributes are task length meaning the time taken to complete the task, task size meaning the number of cores required to execute the task, and task timestamps including submission time, queued time, start time, end time. For applications that have dependencies on task and data, more attributes will be added to describe a task, such as the input data size, the output data size, the parent tasks, and the children tasks.

4.4.2.2 Global Event Queue

Before settling on SimMatrix being a DES, we explored the maximum number of threads that could be supported by Java JVM, and found that on our 48-core system with

256GB of memory, 32K threads is the upper bound. Since it is not feasible for us to run 1M threads in Java (or C/C++ which we also explored), we decide on creating an object per simulated node. Any behavior is converted to an event, and all events are put in a global event queue, and sorted based on the occurrence time. We advance the simulation time to the occurrence time of the first event removed from the queue. The events are:

- TaskEnd: Signals a task completion event, leading to free a processing core. The dispatcher will advance to the next task to schedule. The compute node with the available core will wait for the dispatcher to assign the next task.
- Submission: Client submits tasks to the dispatcher, triggered when the length of the task queue in the dispatcher is below a predefined threshold.
- Log: Signals to write a record to a summary log file, including the information such as the simulation time, number of all cores, number of executing cores, the length of task queue, instant throughput, etc.

The performance of the event queue is critical. It has to be scalable to billions of events, and be subjected to frequent operations. We use the TreeSet [157] data structure in Java. It is a set of elements ordered using their natural ordering, or by a comparator provided at set creation time. In SimMatrix, it is ordered by a comparator based on the event occurrence time, along with the event Id (if events have the same occurrence time). The TreeSet is implemented based on Red-Black tree [90] [156] that guarantees $\Theta(\log n)$ time for removing and inserting, and $\Theta(1)$ time for getting the first event.

4.4.2.3 Node Load Information

Since the compute node has no task queue to hold tasks to be executed, the load of a node of the centralized scheduler is defined as the number of busy cores ranging from 0 to the number of cores of a node. The dispatcher manages the resources of all the compute nodes and has a global view of the loads of all the nodes. The dispatcher needs to access the load information continuously as long as there are tasks to be executed, to find the next nodes that have available executing cores. If we were to naively go through all the nodes to get the load information, the simulator would be inefficient when the number of nodes is large (e.g. 1 million).

We implement the load information using a nested hash map data structure. The *key* is the load (from 0 to number of cores), while the value is a hash set that contains the node ids whose loads are all equal to the *key*. This means that nodes in the simulator are grouped together in buckets that have similar loads. Every time when the dispatcher wants to schedule some tasks to a node, it goes through all the load buckets sequentially, finds the first set of nodes that have idle cores (load is less than the number of cores), and then assigns tasks to all the nodes in a FIFO pattern. As the number of cores per node is relatively small (e.g. 1000 cores), we consider this lookup operation taking $\Theta(c)$ time, where c is the number of cores of a node, and $c<=1000$. Once the right load level is identified, inserting, searching or removing an element in the nested hash map takes only $\Theta(1)$ time. This nested data-structure helps reduce the time complexity by orders of magnitude, from a $\Theta(n)$ (n is the number of nodes) to $\Theta(c*1)$ for one scheduling decision, and allows the simulator to run orders of magnitude faster at exascale.

4.4.2.4 Static and Dynamic Task Submission

SimMatrix allows the client to submit tasks in either a static or a dynamic way. In the static submission, the client would submit all tasks (either predefined in a workload file or generated by a specific workload generator) in one batch to the dispatcher as soon as possible. This is to eliminate the potential bottleneck of the task submission. The dynamic task submission applies when the number of tasks is tremendously large. SimMatrix allows task submission throttling to limit the memory footprint of the simulator to only the active tasks. Essentially, the client would divide the tasks into many batches, and submit the tasks one after another until all the tasks are submitted. The simulator set a threshold for the number of tasks in the task queue of the dispatcher. If the task queue length is below the threshold, the client will submit the next batch of tasks.

4.4.2.5 Logs

In order to help generate the evaluation results, as well as for visualization's purpose, we write some information into logs. We have two logs, one records the per-task information (can be very large for exascale simulations), while the other one records the summary over some defined unit of time (quite efficient regardless of scale of experiment). The per-task log records information such as task id, compute node id, submission time, queue wait time, execution time, and exit code that identifies whether the task was executed successfully or not. The summary log records information such as the simulation time, number of all cores, number of busy cores, task queue length, and instance throughput.

The per-task log is optional due to the potential large overheads of writing a huge file. If enabled, a record is logged whenever a 'TaskEnd' event happens. The summary log is mandatory, and is implemented by submitting 'Log' events to the global event queue. At the beginning when simulation time is 0, we insert a 'Log' event. When handling a 'Log' event, we remove it and insert the next 'Log' event that happens after a fixed amount of simulation time. This way, we can ensure that the increment of the simulation time between two consecutive records is constant, making plots of the results easier to manage.

4.4.3 Distributed Scheduler. One of the major motivations of developing SimMatrix is to study different distributed scheduling techniques at exascale, assuming that the centralized schedulers would not scale up to exascale levels. This section describes the distributed scheduler that uses the adaptive work stealing technique to achieve load balancing. With the distributed architecture, each scheduler on one compute node maintains a task queue. The distributed scheduler shares common features with the centralized one, such as task description and dynamic task submission.

Tasks can be submitted to any arbitrary node. For simplicity, we let the client submit all tasks to the first node (id = 0). This is the worst-case scenario from a load balancing perspective. SimMatrix also allows the client to submit tasks in the best-case scenario, where the tasks are submitted to all the nodes in a load balancing fashion (e.g. uniform random, modular). Every compute node has a global knowledge of all other nodes in the system (membership list). Figure 28 (right part) shows a fully connected homogeneous topology. All nodes have the same amount of neighbors and cores; in this example, the neighbors of a node are just its several left and right nodes with consecutive ids, we call this schema as the static neighbor selection. In addition, our simulator allows dynamic random neighbor selection, which means every time when doing work stealing, a node selects several neighbors randomly from the membership list.

When a node runs out of tasks to be executed, it will ask the load information of all the neighbors in turn. In the distributed scheduling system, the load of a node is the task queue length minus the number of idle cores of that node. The idle node tries to steal tasks from the heaviest loaded one. When a node receives a load information request, it will send its load information to the calling neighbor. If a node receives work stealing request, it then checks its task queue, if which is not empty, the node will send some tasks to the neighbor, or it will send information to signal a steal failure. When a node fails to steal tasks, it will wait some time (referred to as the poll interval), and then try again. The termination condition is that all the tasks submitted by the client are finished. We do this by setting a global counter that is visible for all the simulated nodes.

The simulated distributed scheduler also has a global event queue that has the same implementation as that of the centralized one. This global event queue allows the simulator to be implemented in a relatively straightforward manner, easing the implementation, tuning, and debugging. The trade-off is perhaps the limited concurrency. However, as we will show in the evaluation section, even with this design, SimMatrix is able to outperform several other simulators significantly. The types of events are defined as follows.

- TaskEnd: A task completion event. The compute node starts to execute another task if its task queue is not empty. This results in inserting another 'TaskEnd' event. Otherwise, a 'Steal' event is triggered to steal tasks. If the node is the first node that accepts all the tasks and its task queue length is below a threshold, a 'TaskReception' event will be triggered on the client's side.
- Log: The same as the centralized scheduler.
- Steal: Signals the work stealing algorithm to invoke the steal operation. First, the node asks for the load information of its neighbors in turn, and then selects the most loaded one to steal tasks by inserting a 'TaskReception' event. If all neighbors have no tasks, the node will wait for some time to 'Steal' again.
- TaskDispatch: Dispatch tasks to a neighbor. If at the current time, the node happens to have no tasks, it will inform the neighbor to steal tasks again by

inserting a 'Steal' event from the neighbor. Otherwise, the node dispatches a part (e.g. half) of its tasks in the task queue to the neighbor by inserting a 'TaskReception' event from that neighbor.

- TaskReception: Signals the receiving node to accept task by increasing the length of its task queue. The tasks received are either from the client, or from a neighbor.
- Visualization: Visualize the load information of all nodes.

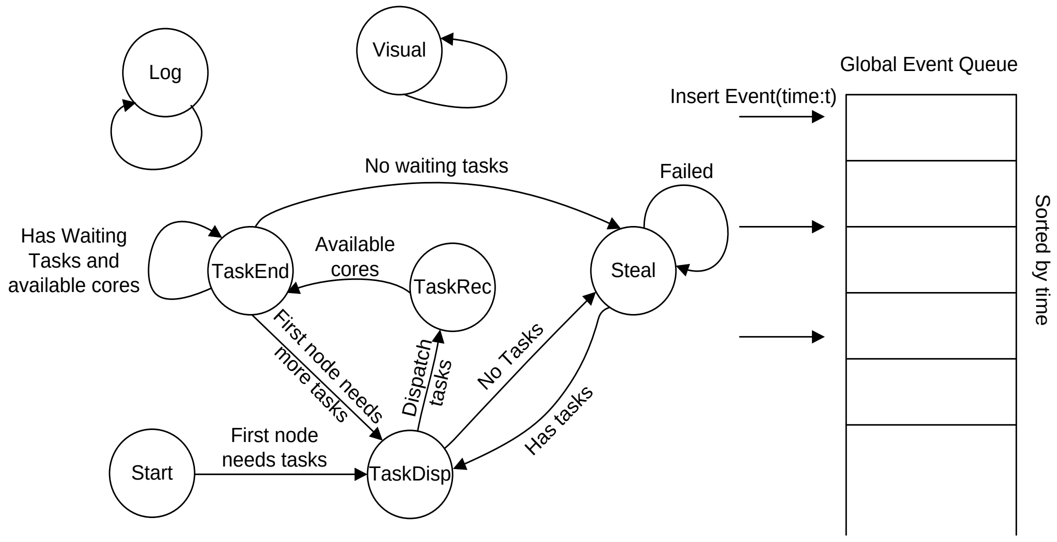


Figure 29. Event State Transition Diagram

The state transition diagram of all the events is shown in Figure 29, where each state is an event that is executed, and the next state is the event to be inserted into the event queue signaled after finishing the current event. For example, if the current event is "TaskEnd", meaning that a node finishes a task and has one more available core. If the node has more tasks to be executed, it will insert another "TaskEnd" event for the available core; otherwise, it will steal tasks from neighbors. In addition, if the current node is the first node and needs more tasks, it will ask the clients to submit the next batch of tasks.

4.4.4 SimMatrix Implementation Details. We developed SimMatrix from scratch in Java. SimMatrix has 2600+ lines of code, out of which 800 lines of code are for the centralized simulator and the rest 1800 lines of code are for the distributed scheduler. Before developing our own simulator, we explored the possibilities of leveraging other existing distributed system simulation frameworks, such as Jist [158], SimGrid [88], GridSim [109], and PeerSim [48]. We found that none of the simulators have the ability to simulate the exascale level (e.g. millions of nodes, billions of cores, and hundreds of billions of tasks) we are studying. We implemented a workload generator for SimMatrix, which can generates workload according to different distributions of task lengths, such as uniform random distribution and Gamma distribution. The source code of SimMatrix is made open source on http://datasys.cs.iit.edu/~kewang/software.html.

4.5 MATRIX – a Real MTC Task Execution Framework

In addition to the SimMatrix simulator, we also develop a real MTC task execution framework. MATRIX implements the distributed scheduling architecture (shown in Figure 25) and the adaptive work stealing technique to achieve load balancing. MATRIX also integrates a distributed key-value store (KVS), ZHT [51], to monitoring execution progress and maintain system and task metadata.

4.5.1 MATRIX Components. The basic MATRIX components, along with the communication messages among the components are shown in Figure 30.

Each compute node runs a scheduler, an executor and a ZHT server. The executor is implemented as a separate thread in the scheduler. All the schedulers are fully connected with each one knowing all of others. The client is a benchmarking tool that issues request to generate a set of tasks, and submits the tasks to any scheduler. The executor keeps executing tasks of a scheduler. Whenever a scheduler has no more waiting tasks, it initiates work stealing to steal tasks from neighbors. The tasks being stolen would be migrated from the victim to the thief. In this scenario, task migration does not involve moving data. We leave the exploration of scheduling of data-intensive MTC applications in the next chapter.

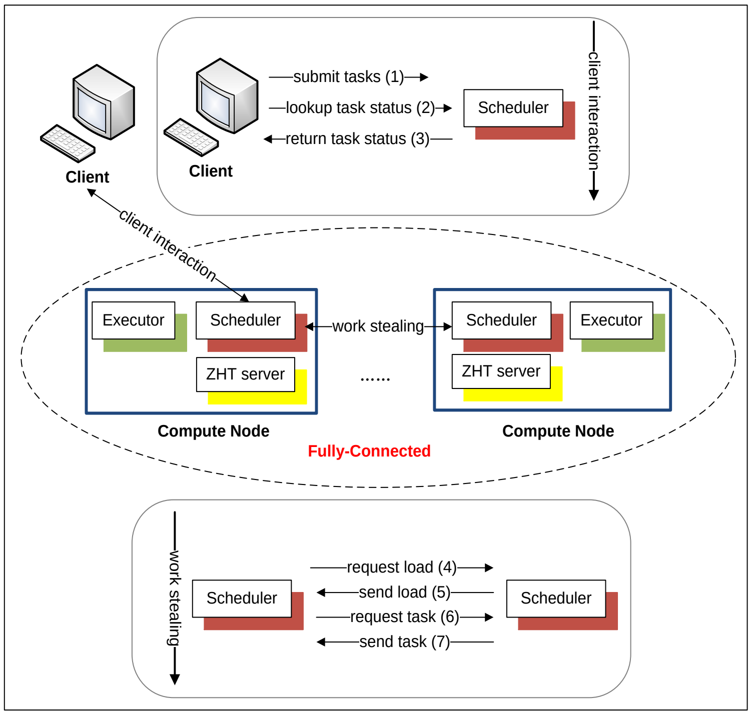


Figure 30. MATRIX components and communications among them

ZHT is used to keep the system and task metadata in a distributed, scalable, and fault tolerant way. Each scheduler is initialized as a ZHT client, and calls the ZHT client APIs (*lookup*, *insert*, *remove*, etc) to insert, query and update information when task state has been changed. The *key* is the task id, and the *value* is the important information related to a specific task, such as the time timestamps including submission time, queued time, executing time, and finish time, the task status, such as waiting, being executed, or completed, the task migrating history, and task dependency information of lists of parents and children.

Figure 30 also shows the communicating messages between client and scheduler, represented as client interaction; and those among schedulers, represented as work stealing. In client interaction, the client first submits tasks to any scheduler (message 1), and it can check task status information (message 2). The scheduler would return the task status information to client after it queries the ZHT server (message 3). In work stealing, a thief scheduler first requests the load information of each potential victim (message 4). The potential victims then send load information to the thief (message 5). The thief then requests tasks from the most heavily overloaded victim (message 6), which then sends tasks to the thief (message 7).

4.5.2 Task Submission. In MATRIX, a client can submit tasks to any arbitrary scheduler. The tasks are submitted in batches (configurable in size) to improve the per client throughput. The number of clients is typically configured in a 1:1 ratio between clients and schedulers. The way a client submits tasks can vary according to requirements.

In the best-case situation, a client would submit all the tasks to all the schedulers in a load-balanced way. Each scheduler will get a portion of tasks. The client can submit tasks by hashing each task to corresponding scheduler using some load-balanced hashing function (e.g. task id modular scheduler id). In addition, we can have as many clients as schedulers, and evenly divide the total tasks among the clients. Each client would submit the tasks to a corresponding scheduler. This is the best-case situation, because tasks are evenly distributed to all the schedulers in terms of number of tasks. If all the tasks are homogenous (tasks are all the same), then load is perfectly balanced, and there is no need to do work stealing. For heterogeneous workload where tasks have different durations, work stealing is useful when some schedulers finish their tasks earlier than others do.

In the worst-case situation, all tasks are submitted to only one arbitrary scheduler. All the other schedulers would have no tasks at the beginning. Therefore, work stealing runs from the beginning to ensure that all the tasks are quickly distributed among all compute nodes evenly to reduce the time for completing the execution of a workload. Assuming there are *m* schedulers and each one talks to *n* neighbors, so ideally within $\log_n^m$ steps, the tasks should be balanced across all the nodes.

4.5.3 Execution Unit. The executor that runs on every compute node maintains three different queues: task wait queue, task ready queue, and task complete queue. The scheduler would put all the incoming tasks from the client to the task wait queue. Figure 31 shows an example of the executor with 4 executing threads, T1 to T4. With these three queues, MATRIX can support running tasks with dependencies specified by certain direct acyclic graph (DAG) [70]. In this chapter, we only considers the dependencies of tasks themselves.

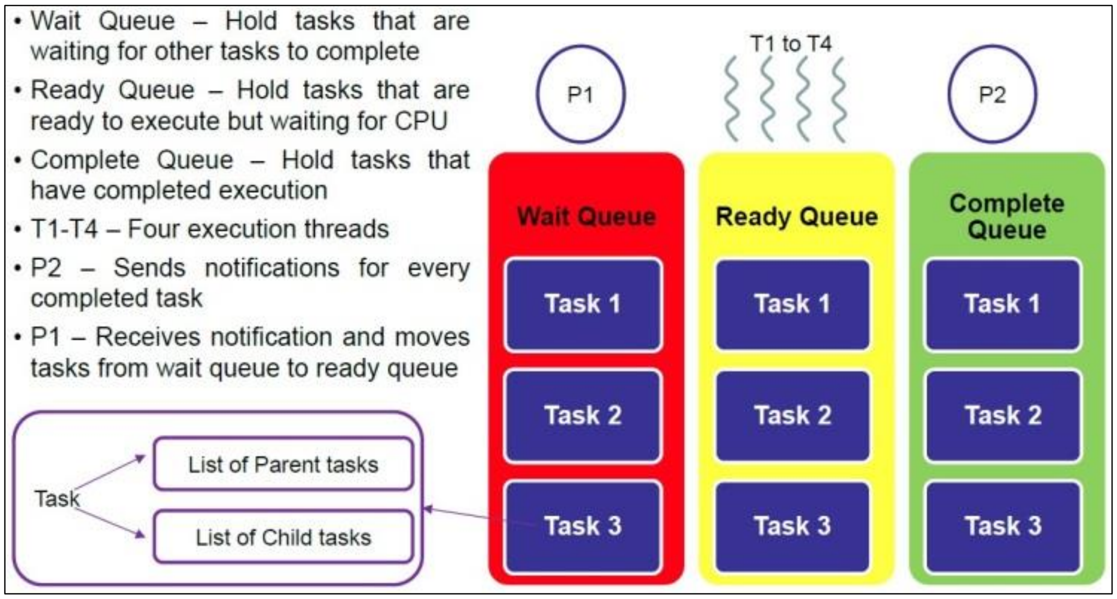

• Wait Queue – Hold tasks that are waiting for other tasks to complete
• Ready Queue – Hold tasks that are ready to execute but waiting for CPU
• Complete Queue – Hold tasks that have completed execution
• T1-T4 – Four execution threads
• P2 – Sends notifications for every completed task
• P1 – Receives notification and moves tasks from wait queue to ready queue
Task
List of Parent tasks
List of Child tasks
P1
T1 to T4
P2
Wait Queue
Ready Queue
Complete Queue
Task 1
Task 2
Task 3

Figure 31. Execution Unit in MATRIX

All the tasks will be put in the wait queue initially. The task dependency is stored in ZHT in the following way. Each task has two fields: a counter, and a list of child tasks. The *key* is the task id, the two fields are part of the *value*. The counter represents the number of parent tasks that should complete before this particular task can be executed. The list of child tasks is the list of tasks that are waiting for this particular task to complete. A program P1 would keep checking every task in the wait queue to see whether the dependency conditions for that task are satisfied or not by querying ZHT server. The dependency conditions would be satisfied only if all the parent tasks have been finished (the counter is 0). Once a task is ready to run, it would be moved from the wait queue to the ready queue by P1.

There are several executing threads in the executor, which keep pulling tasks from the ready queue to execute the tasks in the FIFO way. The number of executing threads is configurable, but in practice, it is usually configured to be the number of physical cores (a similar strategy was used in Falkon [78] on the BG/P machine).

After finishing the execution of a task, the task is then moved to the complete queue. As long as the ready queue is empty, the scheduler would do work stealing, and try to steal tasks from randomly selected neighbors. Only the tasks in the ready queue can be stolen. Tasks in the wait queue would be marked as unreachable for other schedulers.

For each task in the complete queue, another program P2 in the executor is responsible for sending a completion notification message to ZHT server for each child task. The ZHT server would then update the dependency status of each child of that particular task by decreasing the counter by 1. As long as the dependency counter of a task is 0, the task would be ready to run. Whenever the status of a task is changed, the scheduler would update the changing information by inserting the updated information to ZHT server for that specific task.

4.5.4 Client Monitoring. In MATRIX, client does not have to be alive during the whole experiment to wait notifications of the completions of tasks. Instead, MATRIX has a monitoring program on the client side that can poll the task execution progress. As the MATRIX client is also initialized as a ZHT client, the monitoring program can query the status of a specific task by sending query message to ZHT servers, which then query the task information, and return the result to client. In addition, in order to know the execution progress of a workload, the monitoring program could periodically send messages to the ZHT servers to get information about how many tasks have been finished. The termination condition is that all the tasks submitted by the client are finished. The monitoring program can also record logs about the system state, such as the number of total, busy and free execution threads, the lengths of the three queues for each scheduler. This information is helpful for debugging and for ease of visualization of the system state.

4.5.5 Implementing Details. We developed MATRIX in C++. We implemented the client code that generates tasks according to the input task requirements, inserts the task dependencies to ZHT servers, submits tasks, and monitors the execution progress. The DAG generation and management are currently implemented as an internal functionality to the MATRIX client. We also implemented the scheduler code that includes the executor program as a separate thread. The input to MATRIX is a configuration file that specifies all the parameters, such as the number of tasks, the batch size (number of tasks packed to be submitted once), task dependency type, task length, number of executing cores, work stealing initial poll interval, work stealing poll interval upper bound, and so on. These summed up to around 2.5K lines of code representing 1-year of development, in addition to the 8K lines of codebase from ZHT. We have scaled this prototype on BG/P machine up to 1K nodes (4-K cores) with promising results. MATRIX has dependencies on Linux, ZHT, NoVoHT [159], and Google Protocol Buffer [95]. The code of the MATRIX prototype is made open source, available on Github: https://github.com/kwangiit/matrix

4.6 Performance Evaluation

This section presents the performance evaluation, including all the simulation and experimental results. We start by evaluating the task scheduling architectures and techniques through SimMatrix. Furthermore, we evaluate the real MATRIX task execution framework.

4.6.1 Explorations through SimMatrix. This section presents the explorations of the task scheduling architectures and techniques through the SimMatrix simulator up to exascale, including the validation of SimMatrix against Falkon [78] (a centralized light-weight MTC task scheduler), and MATRIX [23]; the experimental results showing the resource requirement of SimMatrix with scales ; the exploration of work stealing up to millions of nodes, billions of cores, and hundreds of billions of tasks ; the comparison between centralized and distributed schedulings ; the comparisons between SimMatrix, SimGrid and GridSim ; and the application domains of SimMatrix. We have two benchmarking workloads that are primarily used in this section:

- AVE_5K: The average task length is 5000 seconds (0 - 10000), with uniform distribution. The number of tasks is 10 times of the number of all cores.
- ALL_1: All tasks have 1-second length. The number of tasks is 10 times of the number of all cores.

4.6.1.1 Simulation Validation

SimMatrix is validated against the state-of-the-art MTC execution fabrics, Falkon (for centralized scheduling) and MATRIX (for distributed scheduling with work stealing technique). We set the number of cores per node to 4, and the network bandwidth and latency the same as the case of BG/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 32 and Figure 33.

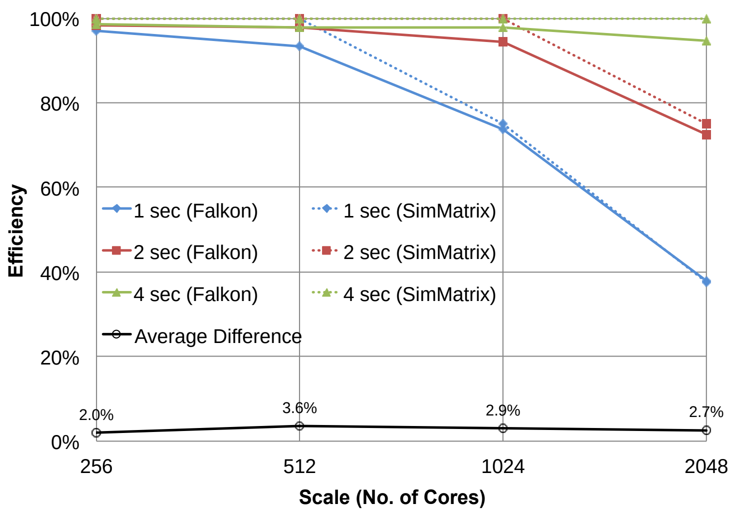
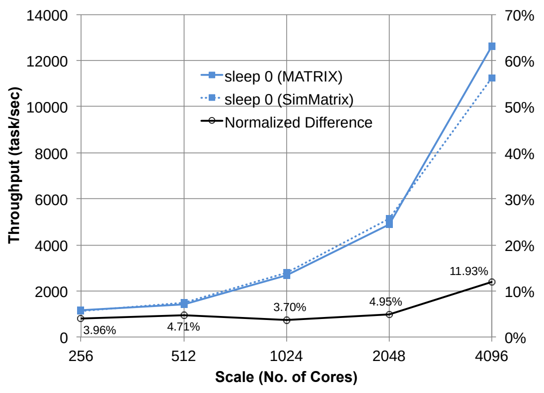
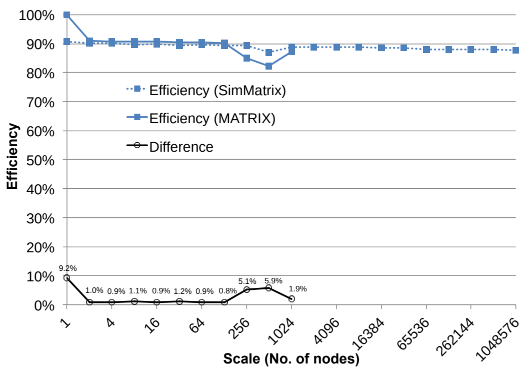


Figure 32. Validation of SimMatrix against Falkon



(a) "sleep 0" workload



(b) real workload trace

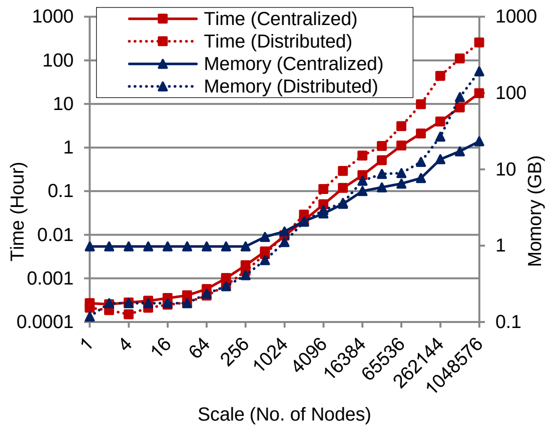
Figure 33. Validation of SimMatrix against MATRIX.

We measured the centralized scheduling (dotted lines) of SimMatrix has an average of 2.8% normalized difference in efficiency compared to Falkon (solid lines) for several sleep tasks, such as sleep 1 sec, 2 sec and 4 sec in Figure 32. The difference is calculated as abs(SimMatrix - Falkon) / SimMatrix. SimMatrix and MATRIX are compared for raw throughput using a “sleep 0” workload, and for efficiency using the real workload trace. For “sleep 0” workload (Figure 33 (a)), the simulation matched the real performance data with an average of 5.85% normalized difference (abs(SimMatrix - MATRIX) / SimMatrix), and for real workload trace (Figure 33(b)), we achieved a mere 2.6% difference.

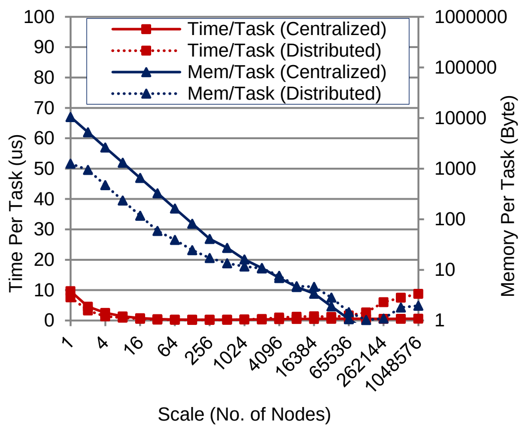
The reasons for these differences are twofold. Falkon and MATRIX are real complex systems deployed on a real supercomputer. Our simulator makes simplifying assumptions, such as the network; for example, we do not model communication congestion, resource sharing and the effects on performance, and the 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).

4.6.1.2 Resource Requirements of SimMatrix

SimMatrix is lightweight in terms of time and memory requirements, and extreme scalable with up to 1 million of node, 1 billion cores, and 100 billion tasks. We show the time and memory requirements of SimMatrix with scales in Figure 34. In these experiments, we use the workloads from the real MTC application traces, shown in Figure 24. Each node is configured to have 1000 cores, and the number of tasks is 100 times of the number of all cores. At exascale with 1M nodes, there will be 1B cores and 100B tasks. Based on the 34.8M tasks, we generated a workload for each scale, from 1 node (1000 cores) to 1M nodes (1 billion cores). If the number of tasks is less than 34.8M, we just randomly selected them from the 34.8M tasks. If the number of tasks is greater than 34.8M, we took the overall tasks several rounds, and randomly selected the rest tasks.



(a) Overall resource consumption



(b) Average resource consumption

Figure 34. Resource consumption of SimMatrix

Figure 34 (a) shows that both the time and memory consumptions increase with a slower rate than the system scale, which means that our simulations are resource efficient. At exascale with 1M nodes, 1 billion cores and 100 billion tasks, the centralized architecture consumes just 23.3GB memory, 17.6 hours, and the distributed architecture needs about 192.1GB memory, 256.4 hours (still moderate considering the extreme scale). Considering per task average resource requirement (Figure 34 (b)), SimMatrix just needs 0.6us and 0.24Byte per task for centralized scheduling, while 8.8us and 1.97Byte per task for distributed scheduling. These low costs at exascale levels will lead to innovative studies in scheduling algorithms at unprecedented scales.

4.6.1.3 Exploring Work Stealing Parameter Space

The parameters that could affect the performance of work stealing are number of tasks to steal, number of neighbors of a node, static neighbor vs. dynamic random neighbor selection. We explore them in detail through SimMatrix. The workload we used is the same as that in section 4.6.1.2.

(1) Number of Tasks to Steal

Our experiments used the worst-case task submission. 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 that each scheduler has 2 static neighbors. The resutlts are shown in Figure 35. We see 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. Moreover, the decreasing speed of steal_half is the slowest.

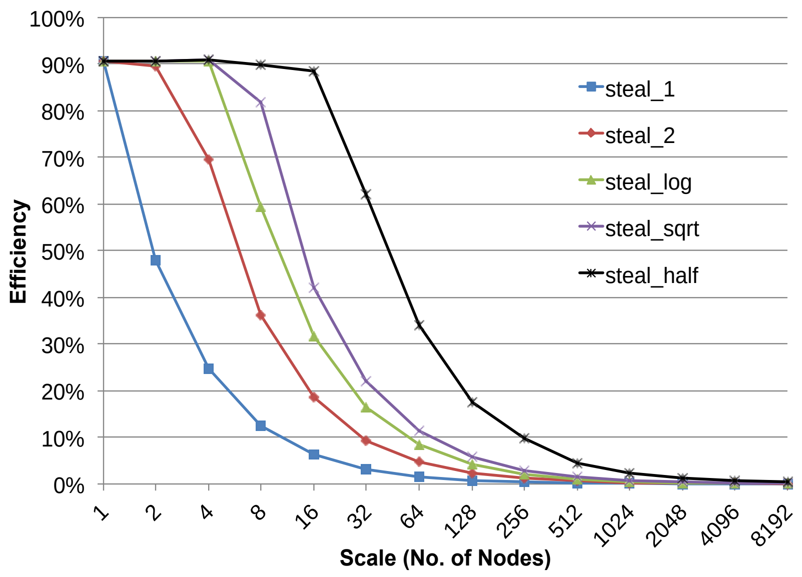
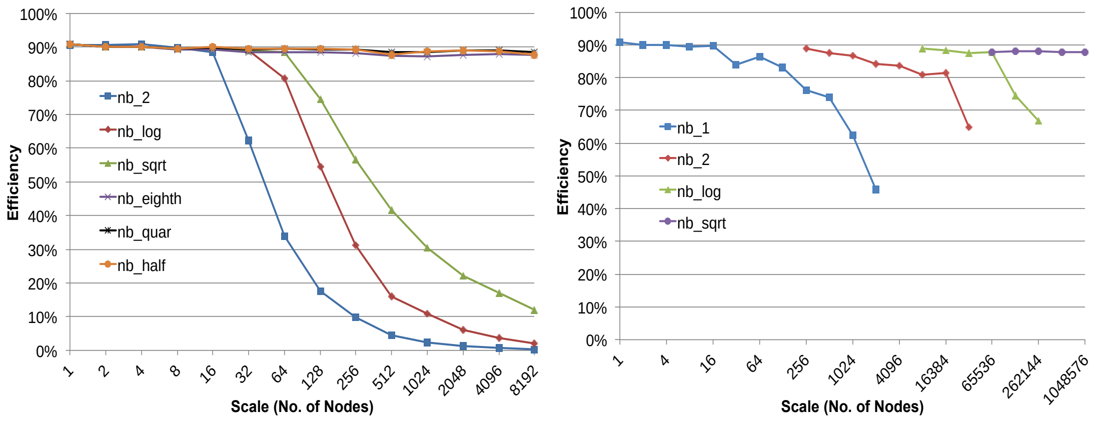


Figure 35. Different numbers of tasks to steal with respect to scale

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 [87]. The reason that steal_half is not perfect (efficiency is very low at large scale) for these experiments is that 2 static neighbors 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 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.

(2) 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 nearby schedulers) are determined at first and never change; dynamic random neighbors mean that every time when does work stealing, a scheduler randomly selects some neighbors. The results of both neighbor selection strategies are shown in Figure 36.



(a) static neighbor selection (b) dynamic neighbor selection

Figure 36. Different numbers of neighbors

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 schedulers, respectively. In static case (Figure 36 (a)), when the number of neighbors is no less than eighth of all schedulers, the efficiency will keep at 87%+ within 8192 nodes' scale. For other numbers, 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 schedulers, 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. In the search for a lower number of needed neighbors, we explore the dynamic multiple random neighbor selection technique.

In dynamic case (Figure 36 (b)), we first do nb_1 experiments until starting to saturate (efficiency < 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 87% efficiency. Even with 1M-nodes in an exascale system, the square root implies having 1K neighbors, a reasonable number that each node can keep track of.

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*.

4.6.1.4 Centralized vs. Distributed Scheduling

We compare the centralized and distributed schedulers, in terms of system efficiency and throughput. Each simulated node is configured to have 1000 cores. We do two groups of experiments. The first uses the AVE_5K workload, and the second uses ALL_1, for both schedulers. The results are shown in Figure 37 and Figure 38.

We see that for AVE_5K, before 8K nodes, both the centralized and distributed schedulers have the efficiency higher than 95%. However, after that, the centralized scheduler drops its efficiency by half until almost 0 up to 1M nodes, and saturates the throughput of about 1000 task/sec (due to 1ms process time of the dispatcher derived from Falkon) as the system scale doubles. On the other hand, the distributed scheduler has efficiency of 90%+ with nearly perfect scale-up to 1M nodes, where the throughput doubles as the system scale doubles, up to 174K tasks/sec.

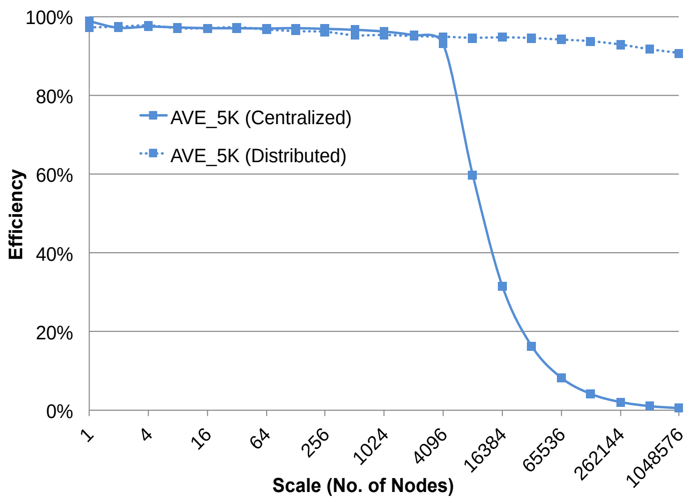


Figure 37. Efficiency of centralized and distributed scheduling (AV_5K)

For ALL_1, the centralized scheduling saturates at about 8 nodes with upper bound throughput of about 1000 tasks/sec, while the distributed one slows down the increasing speed after 128K nodes with throughput of about 60M tasks/sec; it finally reaches 1M nodes with a throughput of 75M tasks/sec. The reason that the distributed scheduler begins to saturate at 128K nodes is because at the final stage when there is not much tasks, work stealing requires too many messages (because almost all nodes are out of tasks leading to more work staling events) as the system scales up, to the point where the number of messages is saturating either the network and/or processing capacity. After 128K nodes, the number of messages per task increases exponentially. One way to address this message chocking at large scales is to set an upper bound of the poll interval. When a node reaches the upper bound, it would not do work stealing anymore. In addition, we believe that having sufficiently long tasks to amortize the cost of this many messages would be critical to achieve good efficiency at exascale. With an upper bound of 75M tasks/sec, the distributed scheduler could handle workloads that have an average length of at least 14 seconds with 90%+ efficiency. It is worth noting that the largest trace of MTC workloads [53] has shown MTC tasks having average length of 95sec.

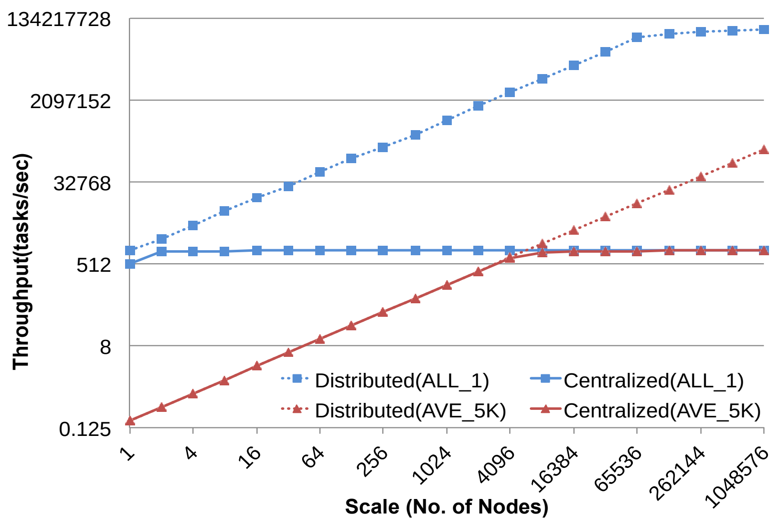


Figure 38. Throughput of centralized and distributed scheduling

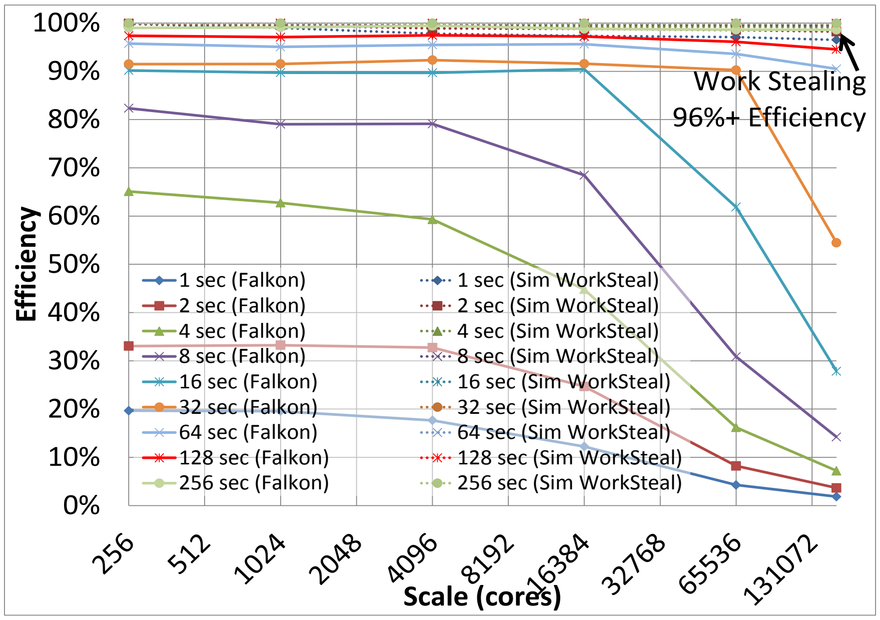


Figure 39. Comparing distributed scheduling of SimMatrix with centralized Falkon

We also compare the distributed scheduling of work stealing of SimMatrix with the Falkon centralized MTC scheduler up to 160K cores, the full scale of the BG/P machine where Falkon was run. The results are shown in Figure 39. We see that the distributed scheduling architecture, along with work stealing, is able to maintain a 96%+ efficiency even with 1sec tasks, when Falkon was only able to achieve 2% efficiency with 1sec tasks at full 160K-core scale, requiring task lengths of 256sec to achieve 90%+ efficiencies.

4.6.1.5 Visualization

In order to aid the evaluation of the work stealing technique, we developed a visualization component to graphically show the load balancing by showing the changes of the load information across all nodes and the task executing progress in SimMatrix. We do visualization of the load of each node at 20 frames per second (about the minimum frequency the human eye cannot see). We map the loads of the nodes to colors, and each node is represented as a tile in a canvas. The nodes are mapped to tiles depending just on their 'id' and are assigned in a row-wise manner. Due to our neighbors' distribution policy, the nodes that are beside each other row-wise are neighbors. This helps to visualize hot spots or starvation instances, due to poor scheduling configuration.

Figure 40 gives an example about how the load is transmitted from the first tile to its neighbors (inside the black rectangles) and so on. The first tile is the node that receives all tasks from the client, and thus appears to be the most loaded. The mapping from load to color is achieved through a transformation that considers a load rate. The rate from 0 to 1 represents how heavily loaded each node is. A fully loaded node will be mapped to the red color, and nearly idle nodes are mapped to a soft green. Completely idle nodes are mapped to white for easy differentiation. We reached a final transformation with some empirical parameters in the HSB color space that gives us good results: Hue = (1-rate)*0.36, Saturation = 1.0-(0.4*(1-rate)), and Brightness = 1.0.

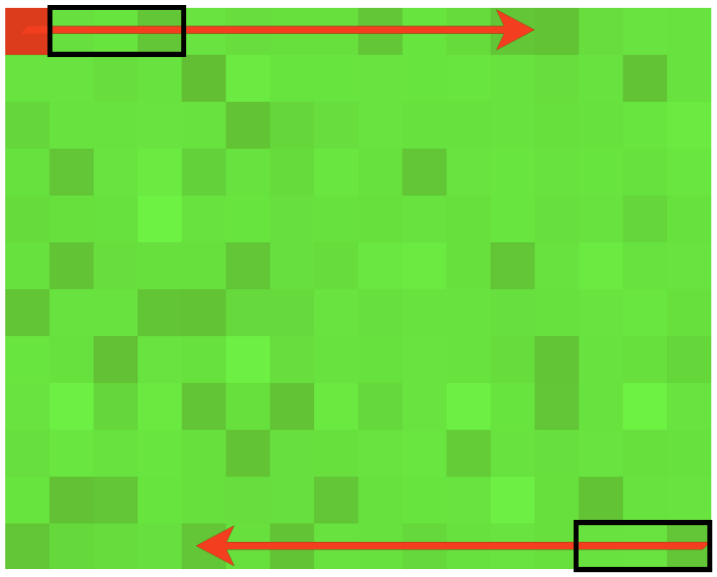
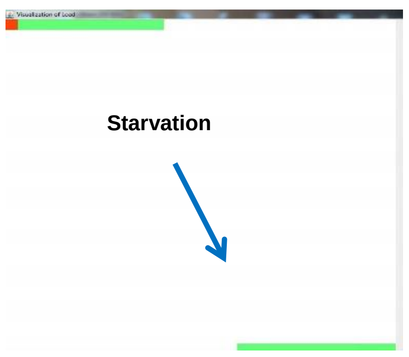


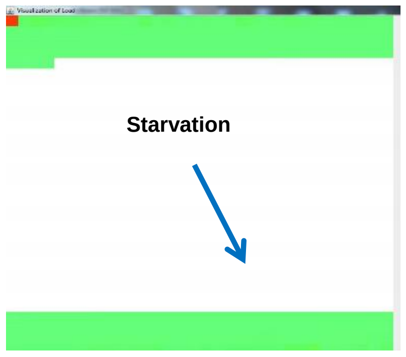
Figure 40. Load distribution flow for 256 nodes, and 6 neighbors

Figure 41 shows an example of 1024 nodes under different work stealing configurations, some lead to starvation (2 and squared root number of static neighbors), while others have relatively uniform load balancing (a quarter of static neighbors and squared root of dynamic neighbors). We present the representative graphs for different number of neighbors when the system is stable.

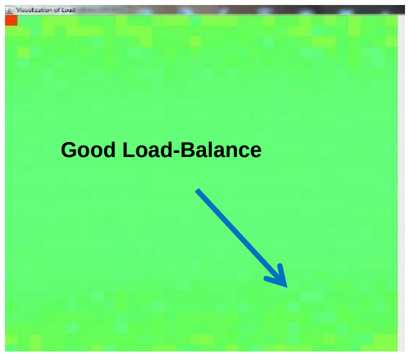
We developed a tool for SimMatrix to show the task execution progress with respect to time visually. We show the visualization graph of the execution of SimMatrix up to exascale with millions of nodes, billions of cores, and hundreds of billions of tasks, using the real workload traces in Figure 42. The utilization is calculated as the area-of-green-region / area-of-red-region. We could see that the work stealing has nearly 100% utilization after about 2K seconds elapsed; it takes that long to reach a stable load balancing where every node has tasks to execute, a total of 100 billion tasks. The overall utilization of the work stealing is about 82% after taking into account the ramp up and ramp down periods of the experiment, with an average of 8.3M tasks/sec.



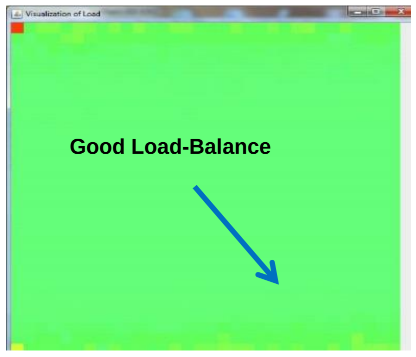
(a) 2 static neighbors



(b) a squared root static neighbors



(c) a quarter static neighbors



(d) a squared root dynamic neighbors

Figure 41. Visualization at 1024 nodes via SimMarix

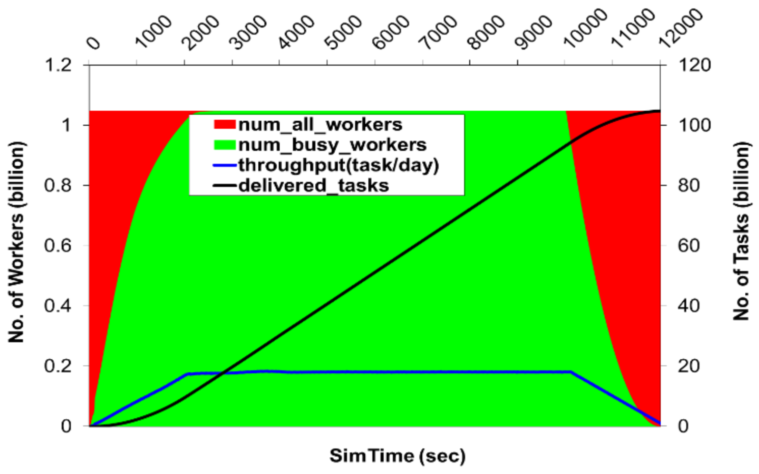


Figure 42. Visualization of at exascale via SimMatrix

4.6.1.6 SimMatrix vs. SimGrid and GridSim

We compare SimMatrix with SimGrid and GridSim, in terms of resource requirement per task with scales. As neither SimGrid nor GridSim supports explicit distributed scheduling, we compare them using centralized scheduling.

SimGrid provides functionalities for the simulation of distributed applications in heterogeneous distributed environments. It is a PDES, being claimed the scalability of 2 million nodes [88]. We examined SimGrid, went for the MSG interface, and used the basic Master/Slaves application. We used the AVE_5K workload, and converted the task length to the value of million instructions (MI), as the computing power is represented as MIPS. Each slave has 1000 cores, with each core 4000MIPS (about 1GFlops as 1 CPU cycle usually has 4 instructions), so the computing power of 1 million nodes is 1GFlops$\times$1M$\times$1K=1EFlop, achieving the exascale computing.

GridSim [109] 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. It is a multi-threaded simulator, where each entity is a thread. We developed an application on top of GridSim, which consists of one user (has tasks) and one broker (centralized scheduler) and several resources (computing nodes). Each resource is configured having just one node (Machine), which then has 1000 cores (PEs).

As the saturated throughput of SimGrid is about 2000, in order to make fair comparison, we configured SimMatrix having exactly the same throughput upper bound by setting the processing time per task to be 0.0005 sec (which is 0.001 sec before and achieved the 1000 upper bound). The comparison results are shown in Figure 43 and Figure 44.

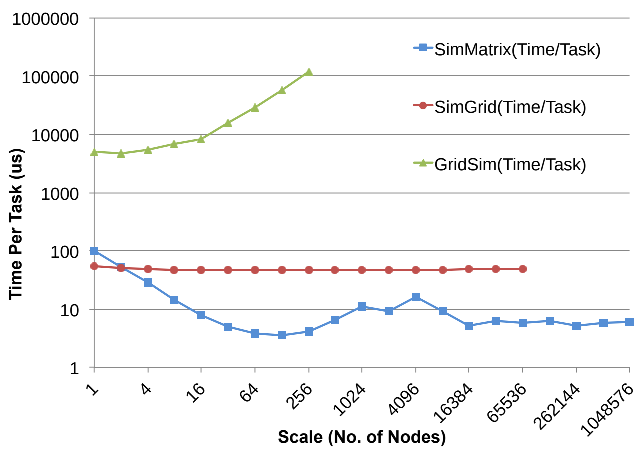


Figure 43. Comparison of time per task

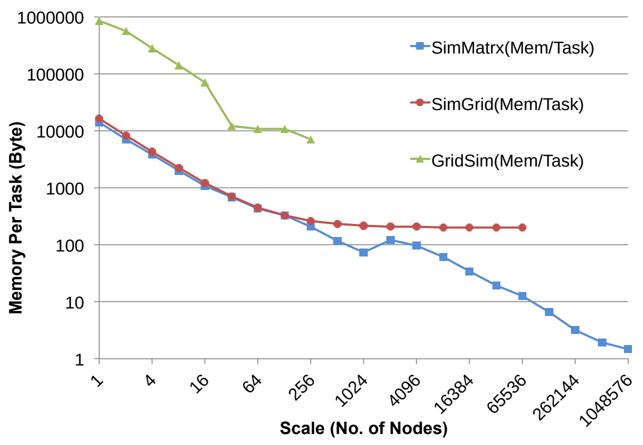


Figure 44. Comparsion of memory per task

Notice that for GridSim, we just scaled up to 256 nodes, as it took significant time to run larger scales. The time per task of GridSim is significantly worse than other two. It is increasing as the system scales up, while SimMatrix and SimGrid experienced decreasing or constant time per task. This shows the inefficiency and poor scalability of the design of one thread per entity of GridSim. SimGrid could scale up to 65K nodes, however, after which point it ran out of memory (256GB). The memory per task of SimGrid decreases two magnitudes from 1 node to 256 nodes and keeps constant after that. However, the SimMatrix scales up to 1M nodes without any problems (14.1GB memory, and 17.4 hours), and it is likely to simulate even greater scales with moderate resource requirement. What's more, SimMatrix requires almost the same amount of memory as SimGrid at the scale of less than 512 nodes, however, after that SimMatrix is more memory efficient (memory per task keeps decreasing with scales) than SimGrid. We also noticed after 1 node, SimMatrix is more time efficient than SimGrid; the time per task of SimMatrix is one magnitude smaller than that of SimGrid. The conclusion is that SimMatrix is light-weight and has less resource requirement at larger scales.

4.6.1.7 Application Domains of SimMatrix

We believe that the work presented in SimMatrix on adaptive work stealing is generalizable to other domains, such as the data centers, the grid environment, the workflow systems [71] [72] [73] [74] [93], and the many-core computing.

(1) Data Centers

Large-scale data centers are composed of thousands of (10 to 100 $\times$ in near future) servers geographically distributed around the world. Load balancing among all the servers with data-intensive workloads is very important, yet non-trivial. SimMatrix is extensible to enable the study of different network topologies connecting all the servers and data-aware scheduling, which could be applied in scheduling of data centers.

(2) Grid Enviroment

Not only could SimMatrix be configured as homogeneous scheduling system, it can also be tuned into heterogeneous one. Different Grids could configure SimMatrix and do scheduling individually without interaction with each other.

(3) Many-core Computing

The processor clock frequency has plateaued since 2002 [97], despite Moore's Law [98] being very much alive. The main reason stems from the power consumption increases with higher processor speeds, leading manufacturers to use the additional transistors towards implementing multiple simpler cores instead of making existing cores more complex with higher speed. This was the start of the multi-core era, soon to be many-core era, fueled by the development of tens to thousands of cores on a single processor (e.g. Intel MIC [99], NVIDIA GPU [100]).

Predictions are that the number of general-purpose cores/threads per processor will be in thousands by the end of this decade [2]. With this level of concurrency in one node, the core topology within a node and load balancing in the core level are very important factors to the node efficiency and power consumption. 2D-mesh topology seem to be promising as they are easy to build, cheap to manufacture, and are power efficient. 3D-mesh is also interesting as processor manufacturing is advancing to stacked cores and memory; this allows greater bandwidth interconnect between cores and memory, hence having greater scalability. We used SimMatrix to configure a 1K-core processor (instead of the 1M-node distributed system we evaluated in prior sections). SimMatrix supports multi-dimensional mesh topology interconnects; we explored both 2D-mesh and 3D-mesh at 1K-core scales. Cores have consecutive ids (from 0 to number of cores $N-1$).

Workload: We generate a finer grained workload from the real workload traces than. In order to simulate a MTC workload tuned for a single many-core node (as opposed to a large-scale distributed system), we divided the task runtime of each task from the original MTC workload by 1000. The resulting workload has the following high-level characteristics: minimum runtime of 0 seconds, maximum runtime of 1.469 seconds, average runtime of 95ms, and standard deviation of 0.188. Communication overheads were also reduced by one tenth compared to those found in a distributed system. The actual workload and communication latencies are not critical to be exact, as we aim to understand the trends in performance for the work stealing technique for many-core processors.

The specification of the core topology is as follows:

2D mesh**:** core i has a two-dimension coordinate (x, y), where $x = (\text{int})\left(i \div \sqrt{N}\right)$ and $y = (\text{int})\left(i \bmod \sqrt{N}\right)$. The hop distance between two cores (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$.

3D mesh: core i has a three-dimension coordinate (x, y, z), where $x = (\text{int})\left(i \div N^{\frac{2}{3}}\right)$, $y = (\text{int})\left(\left(i - x \cdot N^{\frac{2}{3}}\right) \div \sqrt[3]{N}\right)$, and $z = (\text{int})\left(i \bmod \sqrt[3]{N}\right)$. The hop distance between two cores (x_1, y_1, z_1) and (x_2, y_2, z_2) is $|x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$.

We used work stealing on the many-core processor, and the maximum hop count as the parameter for selecting the neighbors of each core. Limiting the number of hops aims at increasing data locality and avoiding expensive random all-to-all communication. We varied the number of maximum hop count and conducted experiments with up to 1024 cores for 2D mesh, and 1000 cores for 3D mesh. Based on our previous results, we set the number of tasks to steal to half. For consistency with prior sections, we submit all tasks to the central core (with id = N / 2) and let the work stealing algorithm load balance all tasks. The communication overhead between two cores is proportional to their hop count.

In Figure 45 (left part), we see in order to achieve 90%+ efficiency at 1024 cores with 2D mesh topology, the hop count should be no less than 13 (the maximum number of neighbors is: $4\times\sum_{i=1}^{13} i = 364$). Figure 45 (right part) shows that, in order to achieve 90%+ efficiency at 1000 cores with 3D mesh topology, the hop count should be no less than 5 (the maximum number of neighbors is: $5\times 6+4\times 4\times 3+6=84$). It was not surprising that the 3D-mesh could do better than the 2D-mesh, but what was surprising was that it reduced the communication latency to a third from 13 hops down to 5 hops. It is noteworthy that coarser granularity workloads would require less number of hops to achieve the same level of efficiency, due to higher ratio between computing time and scheduling time.

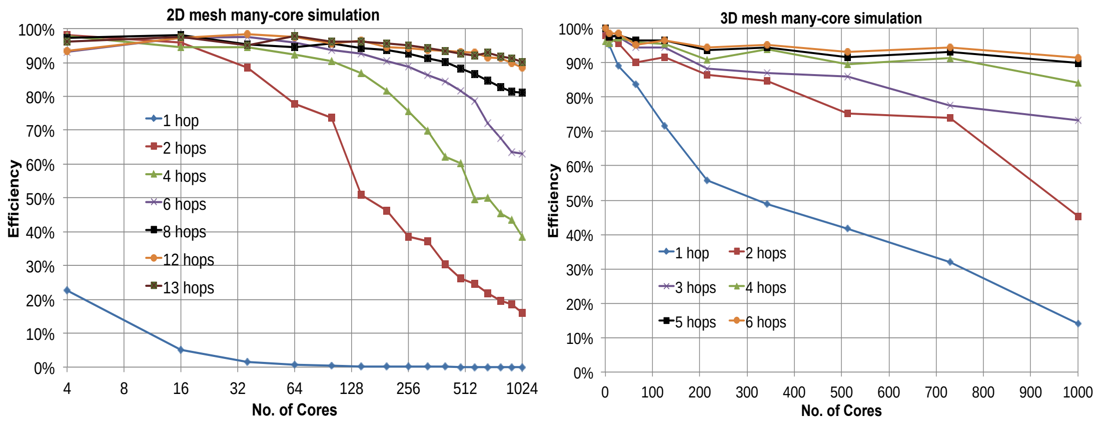


Figure 45. Simulating many-core processor

We have begun working on exploring the support of many-task computing workloads on today’s 1K-core many-core systems, such as NVIDIA GPUs. We have developed the GeMTC [111] framework that allows parallel programming systems such as Swift to efficiently execute fine-grained tasks on hundreds of thousands of CUDA-cores. We plan to investigate the use of work stealing in improving the performance of GeMTC as accelerators are achieving ever-growing number of cores.

4.6.2 MATRIX Performance and Scalability. We run MATRIX on the BG/P up to 1K nodes (4K cores) with both homogeneous Bag-of-Task (BOT) workloads that include per-core independent sleep tasks with different lengths (the workloads used in SimMatrix are all BOT workloads), and synthetic workloads with different DAG types of dependencies, such as Fan-In DAG, Fan-Out DAG, and Pipeline DAG. The workloads are represented in Figure 46.

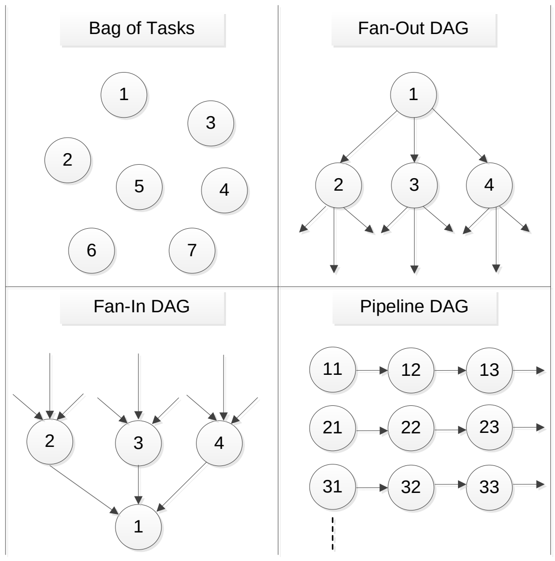
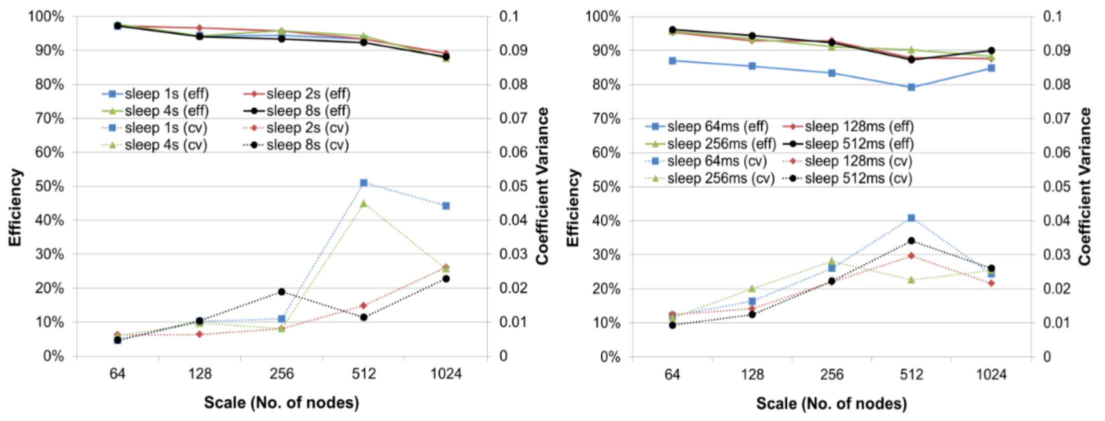


Figure 46. Different types of dependent workloads

Fan-In DAG and Fan-Out DAG are similar except that they are the inverse of each other. The performance of these workloads depends on the in-degree and out-degree of nodes in the graph. Pipeline DAG is a collection of pipes where each task within a pipe is dependent on the previous task. The system utilization depends on the number of pipelines, as at any instant of time the number of ready tasks is equal to the number of pipelines because only one task in a pipeline can be executed at any given time. To amortize the potential slow start and long trailing tasks at the end, we run 1000 seconds for all the experiments. 1000 seconds could balance well between the efficiency and the running time of a large-scale experiment. The number of task would be 1000 * number of cores * number of nodes / task length.

4.6.2.1 MATRIX with BOT Workloads

Figure 47 (a) shows the efficiency (eff) and coefficient variance (cv) results for coarser grained tasks (sleep 1, 2, 4, 8 sec), while Figure 47 (b) shows these results for fine grained tasks (sleep 64, 128, 256 ms).



(a) coarser-grained workloads (b) fine-grained workloads

Figure 47. Scalability of the work stealing technique

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 that 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. A 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.

4.6.2.2 MATRIX with Task Dependency

We evaluate MATRIX using workloads with different DAG types of dependencies, such as Fan-In DAG, Fan-Out DAG, and Pipeline DAG. We keep the BOT workloads as a baseline. The client has a workload generator that can generate different kinds of workloads with these dependencies, and then submits the tasks to schedulers. The efficiency results are shown in Figure 48. All tasks in the workloads are sleep tasks and have an execution time of 8 seconds. The BOT has highest efficiency because there is no dependency, and all tasks are in the ready queue immediately to be executed. For Fan-In and Fan-Out DAGs, completion of one task might satisfy the dependencies of many other tasks thus providing many ready tasks at any instant of time.

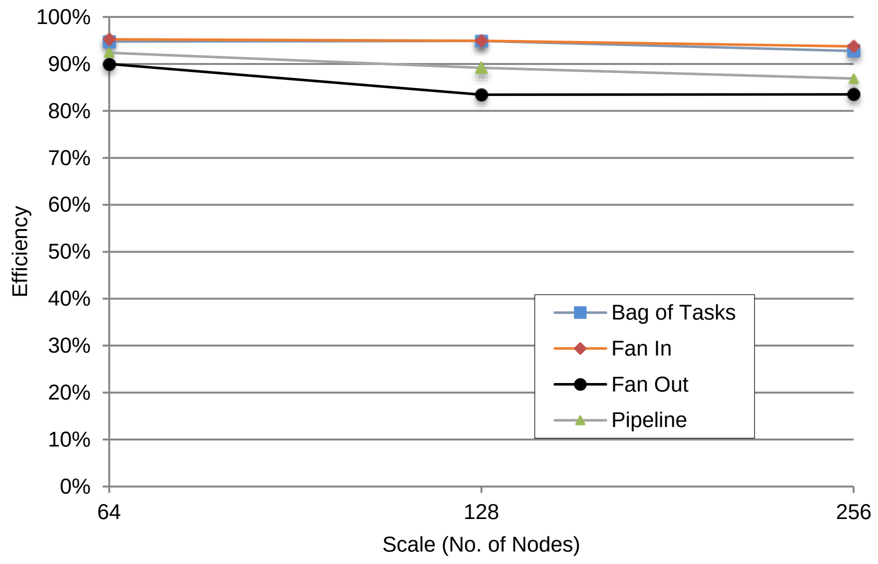


Figure 48. Efficiency results of different types of dependencies of workloads

We set the degree of Fan-In and Fan-Out to 4, equal to the number of executing cores. For Pipeline DAG, the efficiency depends on the number of pipelines. In our experiment, we set the number of pipelines to 4. We see that the work stealing technique, even under a variety of task dependencies, is able to perform well (83%~95% efficient, depending on the dependency patterns and scale) at scale.

4.6.2.3 System Utilization

We show the system utilization when running MATRIX with different types of workloads in Figure 49. For the BOT and Fan-In workloads, it seems that the load gets balanced quickly on the entire system and thus takes shorter time to finish the workload when compared to Fan-Out workload. The reason for lower utilization for Pipeline workload is that, there are not sufficient tasks available to keep the entire system busy. This can be improved by increasing the number of pipelines.

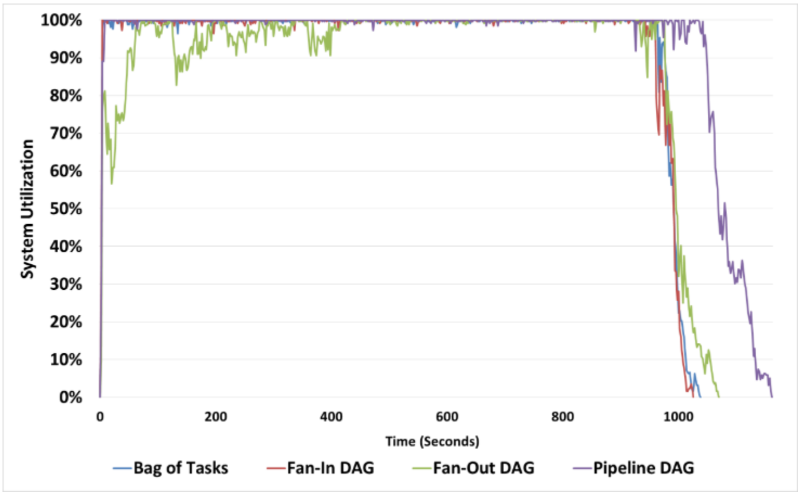


Figure 49. System utilization of MATRIX

A more detailed analysis was made for BOT workload in terms of duration of each task. As seen from Figure 50, it is generally difficult to get higher efficiency for shorter jobs since load balancing is not perfect for shorter running tasks (32ms).

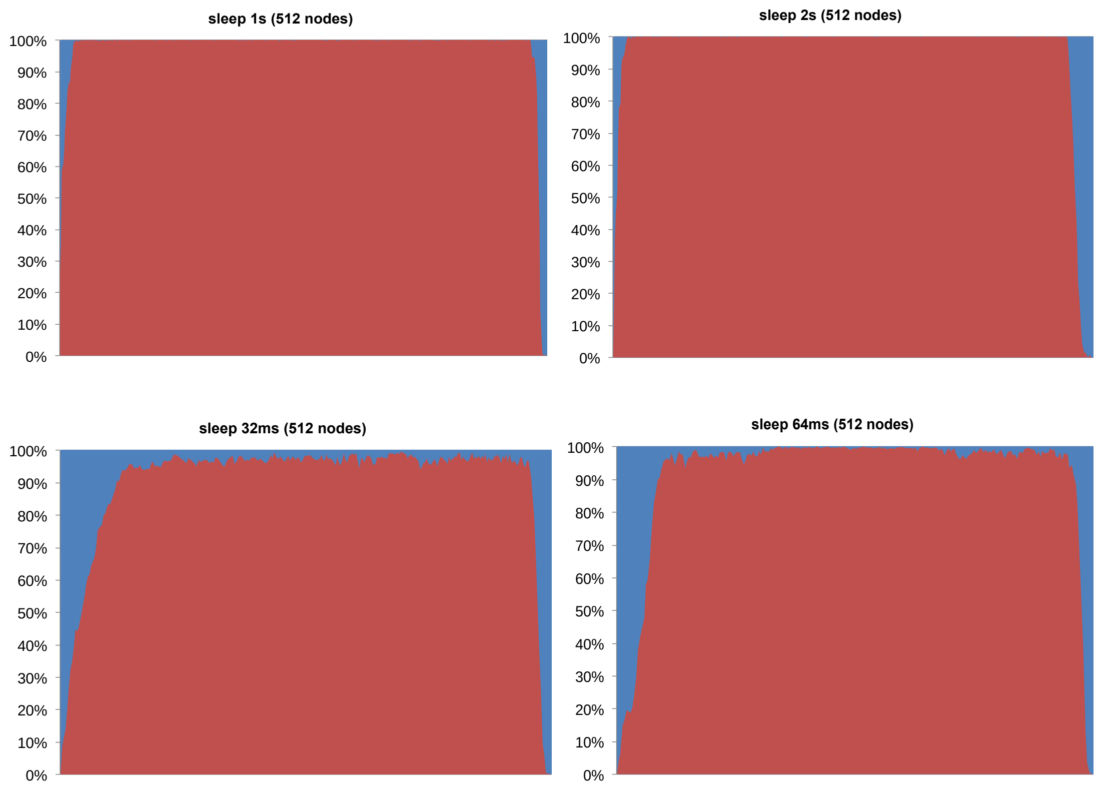


Figure 50. Visualization at 512 nodes for tasks of durations of 1s, 2s, 32ms and 64ms

Figure 51 shows a comparison of load balancing for sleep 1s and sleep 32ms tasks on 512 nodes. The start figures (Figure 51 top half) indicate the convergence of the load (i.e. how quickly the entire load gets balanced) on 512 nodes. The sleep 32ms workload had more number of tasks so that the run time of the experiment is longer to amortize the ramp up and ramp down time. As seen for the sleep 32ms workload, the time taken for the entire workload to get distributed evenly is more than double than that required for sleep 1s workload. The end figures (Figure 51 bottom half) tell us about the end of experiments when there are very few tasks left.

The sleep 32ms workload has longer tail than the sleep 1s workload. There can be two reasons for such behavior. The sleep 32ms workload had more number of tasks. Another possible explanation for this behavior is that, when the task length is short, before the tasks could be stolen for load balancing, it is executed at the node where it is present. This means the queues are changing state so fast that by the time for stealing work from a neighbor, there is nothing to steal. Tasks migration is very low as they all run fast, and any work stealing that tries to occur would likely fail.

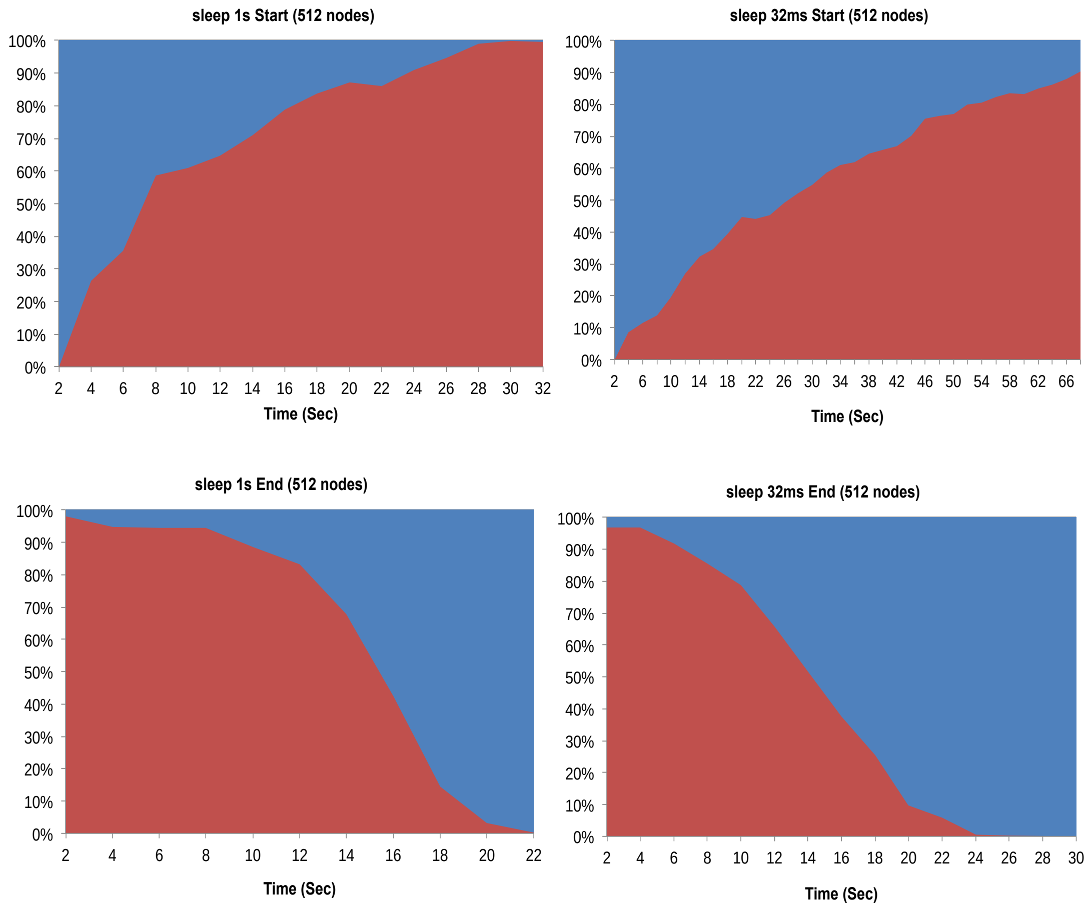


Figure 51. Comparison of work stealing performances at 512 nodes

4.6.2.4 Comparison among MATRIX, Falkon, Sparrow and CloudKon

We compare MATRIX with Falkon on the BG/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.

(1) MATRIX vs Falkon on BG/P

Falkon is a centralized task scheduler with a hierarchical task dispatching implementation. Figure 52 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.

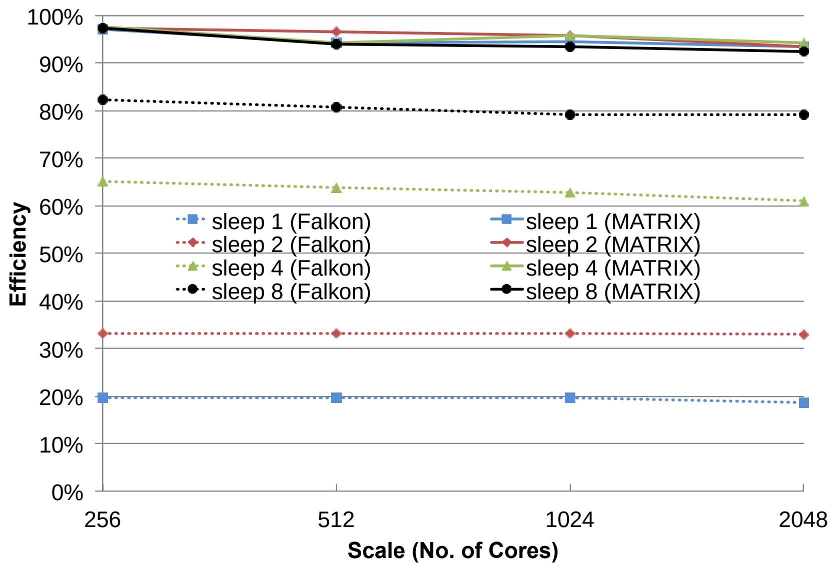


Figure 52. Comparison between MATRIX and Falkon

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.

(2) MATRIX vs Sparrow & CloudKon on Amazon Cloud

We also compare MATRIX with the other state-of-the-art fine-grained task scheduling systems that are targeting loosely coupled parallel applications. Examples are

Sparrow [81] and CloudKon [82]. We show the preliminary results of comparing MATRIX with both of them using BOT workloads. These comparisons aim at giving hints of how better MATRIX can achieve than others, and showing the potential broader impact of our technique on the Cloud data centers.

Sparrow is a distributed scheduling system that employs multiple schedulers pushing tasks to workers (run on compute nodes). Each scheduler has a global view of all the workers. When dispatching tasks, a scheduler probes multiple workers (based on the number of tasks) and pushes tasks to the least overloaded ones. Once the tasks are scheduled to a worker, they cannot be migrated. CloudKon is another scheduling system specific to the cloud environment. CloudKon has the same architecture as MATRIX, except that it leverages the Amazon SQS [108] to achieve distributed load balancing and DynamoDB [34] for task metadata management.

We compare MATRIX with Sparrow and CloudKon on the Amazon EC2 Medium testbed. We compare the raw speed of executing tasks of the three scheduling systems, which is measured as the throughput of executing the “sleep 0” NOOP tasks. We conduct weak-scaling experiments, and in our workloads, each instance runs 16K “sleep 0” tasks on average. 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. We configured MATRIX with the best-case task submission scenario. The result is shown in Figure 53. we see that all of the three scheduling systems can achieve increased throughput trend with respect to the system scale. However, MATRIX is able to achieve much higher throughput than CloudKon and Sparrow at all scales. At 64 instances, MATRIX shows throughput speedup of more than 5X (67K vs 13K) comparing with CloudKon, and speedup of more than 9X comparing with Sparrow (67K vs 7.3K). Comparing with MATRIX, CloudKon has a similar scheduling architecture (fully distributed). However, the workers of CloudKon need to pull every task from SQS leading to significant overheads for NOOP tasks, while MATRIX migrates tasks in batches through the work stealing technique that introduces much less communication overheads. Besides, CloudKon is implemented in Java that introduces JVM overhead, while MATRIX is implemented in C++, which contributes a portion to the 5X speedup. To eliminate the effects of different programming languages, we compare CloudKon and Sparrow, both were implemented in Java, but they have different scheduling architectures and load balancing techniques. CloudKon outperforms Sparrow by 1.78X (13K vs 7.3K) at 64-instances scale, because the schedulers of Sparrow need to send probing messages to push tasks and once tasks are submitted, they cannot be migrated, leading to poor load balancing, while CloudKon relies on SQS to achieve distributed load balancing.

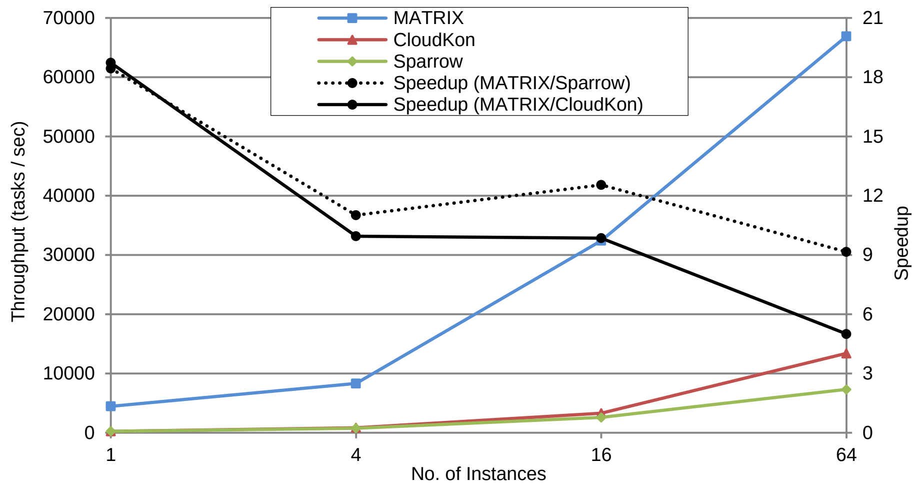


Figure 53. Throughput comparison results

For efficiency, we investigated the workloads from the real MTC application traces. Out of the 34.8M bag of tasks, we isolated only the sub-second tasks, resulting in about 2.07M tasks with the minimum runtime of 0 seconds, maximum runtime of 1 seconds, average runtime of 343.99ms, and standard deviation of 265.98. We configured each instance to run 2K heterogeneous tasks on average. For each scale (from 1 to 64 instances), we generated workloads with large standard deviation of the average run time of the tasks that each worker executes. We sorted all the 2.07M tasks, and let each worker random uniformly select 2K tasks from different percentiles of the sorted 2.07M tasks. The results are shown in Figure 54.

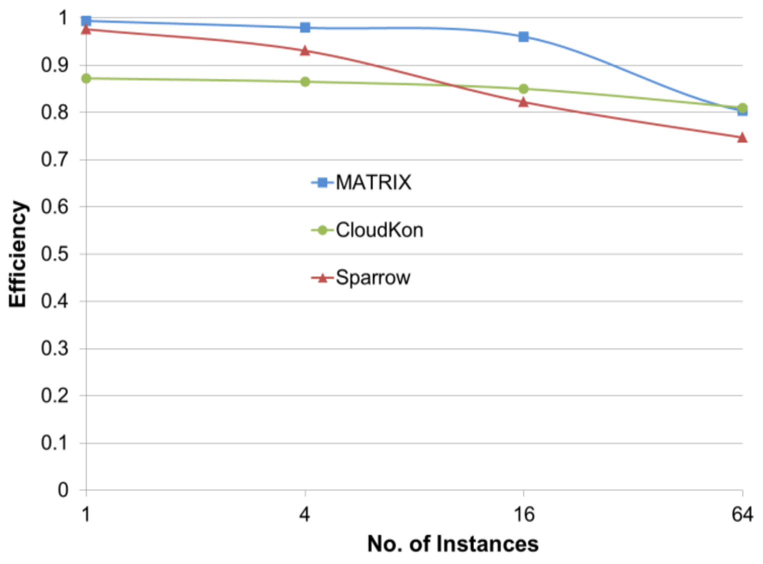


Figure 54. Efficiency comparison results

We see that MATRIX and CloudKon can keep high efficiency (95%+ and 85%+) up to 16 instances. From 16 to 64 instances, the efficiency of MATRIX degraded from 96% to 80%, while CloudKon experienced efficiency degradation from 85% to 80%. This is likely because the virtualized Cloud environment increases network communication overhead more seriously in MATRIX than in CloudKon, as CloudKon uses SQS for load balancing and SQS is designed specifically for cloud environment, while MATRIX is designed for running on supercomputers that have much advanced inter-connection with significantly higher bandwidth and lower latency (MATRIX shows great scalability on the BG/P supercomputer). In Sparrow, efficiency starts to degrade from 1 to 4 instances, and keeps decreasing at larger scales (74% efficiency at 64 instances) showing the poor scalability of the probing and performing early binding of tasks to workers. It is likely that MATRIX would have significantly outperformed CloudKon and Sparrow if the task granularities are smaller (e.g. 3ms instead of 343ms), based on the raw throughput results presented in Figure 47.

The conclusions we can draw after observing the performance of MATRIX is that *work stealing can perform load balancing better than the probing and pushing mechanism. The cloud environment (including network virtualization) has a significant cost for MATRIX that likely generates a significant amount of network communication as the work stealing algorithm performs the load balancing.*

4.7 Conclusions and Impact

Many-task computing applications are an important category of applications running on supercomputers. As supercomputers are approaching exascale and the MTC applications are growing significantly resulting in tremendously large amount of loosely couple fine-grained tasks. To achieve high system utilization for exascale computing, the MTC task scheduling system needs to be distributed, scalable, and available to deliver millions of tasks per second for fine-grained tasks, at the granularity of node/core levels. Distributed scheduling architecture, along with efficient load balancing techniques, is promising to achieve the highest job throughput and system utilization.

In this chapter, we proposed a fully distributed scheduling architecture, an adaptive work stealing technique to achieve distributed load balancing at extreme scales. In order to explore the scalability of work stealing technique, we developed a lightweight discrete event simulator, SimMatrix, which simulates task execution frameworks comprising of millions of nodes and billions of cores/tasks. Via SimMatrix, we explored a wide range of parameters of work stealing at exascale, and discovered the optimal parameter configuration of MTC: *number of tasks to steal is half*, *number of dynamic random neighbors is a square root of the number of all nodes*, and *we need a dynamic polling interval policy*. We applied work stealing to balance tasks across a variety of systems, from thousands of real-cores to billion-core simulated exascale systems. Guided by the insights gained from simulations, we implemented a real prototype of MTC task execution framework, MATRIX, which applies the proposed work stealing technique and is deployed on the BG/P machine with up to 1K-node (4K-core) scale, and was compared with Falkon, Sparrow and CloudKon with good results.

We expect this work to have transformative impacts on the many-task computing research community that has been active and successful for a decade, owing to the radical distributed scheduling architecture and technique for compute management, making extreme scale computing more tractable. It will revolutionize asynchronous programming paradigms to address the challenges of locality and low latency of future exascale supercomputers, and open the door to a broader class of MTC applications that would have normally not been tractable at such extreme scales. It will also bring new research directions to the exascale computing area to better expose the extreme-scale parallelism in an asynchronous way, hide the latency, and optimize the power consumption. These research directions include distributed architectures of resource management, task scheduling, task scheduling, distributed load balancing techniques, data-aware scheduling, topology-aware scheduling, and power-aware scheduling. The research contributions will catalyze the development of many MTC system software, such as parallel workflow systems (e.g. Swift [15] [67], Pegasus [69], Dryad [75]), asynchronous runtime systems (e.g. Charm++ [92], Legion [12], HPX [13], STAPL [14], Spark [160]), and distributed file systems (e.g. HDFS [117], FusionFS [42]).

CHAPTER 5

DATA-AWARE SCHEDULING

Data driven programming models such as many-task computing (MTC) have been prevalent for scheduling data-intensive scientific applications. We have proposed a fully distributed task scheduling architecture that employs as many schedulers as the compute nodes to make scheduling decision for MTC applications of exascale computing. Achieving distributed load balancing and best exploiting data-locality are two important goals for the best performance of distributed scheduling of data-intensive MTC applications. Chapter 4 focused on accomplishing the load balancing goal through an adaptive work stealing technique. In this chapter, we propose a data-aware work stealing technique to optimize both load balancing and data-locality by using both dedicated and shared task ready queues in each scheduler. Tasks are organized in queues based on the input data size and location. Distributed key-value store is applied to manage task metadata. We implement the technique in MATRIX, a distributed MTC task execution framework. We evaluate the technique using workloads from both real scientific applications and micro-benchmarks, structured as direct acyclic graphs. Results show that the proposed data-aware work stealing technique performs well. In addition, we devise an analytical sub-optimal upper bound of the performance of the proposed technique; explore the scalability of the technique through simulations at extreme scales. Results show that the technique is scalable, and can achieve performance within 15% of the sub-optimal solution. Furthermore, we propose to overcome the Hadoop scaling issues through MATRIX for large-scale data analytic applications in the Internet domain.

5.1 Data Driven Programming Model

Many-task computing (MTC) is one of the data-driven programming models. Large-scale scientific applications of MTC are data-intensive such that task execution involves consuming and producing large volumes of input and output data with data dependencies among tasks. The data-intensive MTC applications cover a wide range of disciplines, including data analytics, bioinformatics, data mining, and astronomy, astrophysics, and MPI ensembles [39]. The big data phenomenon has expedited the evolution of paradigm shifting from compute-centric model to data-centric one. MTC applications are usually structured as Direct Acyclic Graphs (DAG) [70], in which the vertices are small discrete tasks and the edges represent the data flows from one task to another. MTC will likely require a fully distributed task scheduling architecture (as opposed to the centralized one) that employs as many schedulers as the compute nodes to make scheduling decisions, in order to achieve high efficiency, scalability, and availability [24] for exascale machines with billion-way parallelism [2].

The two important but conflicting goals of the distributed scheduling of data-intensive MTC applications are load balancing and data-locality [24]. We focused on accomplishing load balancing [104] for compute-intensive MTC applications in Chapter 3, through an adaptive work stealing technique. On the other hand, for data-intensive applications, data-locality aware scheduling is another important goal [63] that requires mapping a task to the node that has the input data, aiming to minimize the overheads of moving large volumes of data through network [229]. Work stealing has been demonstrated as a scalable load balancing technique. However, the work stealing is data-locality oblivious, and the action of moving tasks randomly regardless of the data-locality may incur significant data-transferring overhead. To best exploit data-locality, we need to map each task to where the data resides. This is infeasible because this mapping is NP-complete [115], and leads to poor load balancing due to the potential unbalanced data distribution.

To optimize between both goals, in this chapter, we propose a data-aware work stealing (DAWS) technique that is able to achieve good load balancing and also tries to best exploit data-locality. Each scheduler maintains both dedicated and shared task ready queues that are implemented as max priority queues [116] based on the data size a task requires. Tasks in the dedicated queue are confined to be executed locally unless special policy is applied, while tasks in the shared queue may be migrated through work stealing among schedulers for balancing loads. A ready task is put in either queue based on the size and location of the required input data. A distributed key-value store (KVS) (i.e. ZHT [51]) is applied as a metadata service to keep task metadata, including data dependency and locality information, for all the tasks. We implement the technique in MATRIX [23] [230], a MTC task execution framework. In addition, we devise an analytical sub-optimal upper bound of the proposed technique; explore the scalability of the technique through simulations at extreme scales; and propose to overcome the Hadoop scaling issues through MATRIX for large-scale data analytic applications in the Internet domain. *This chapter makes the following new contributions:*

- Propose a data-aware work stealing technique that combines load balancing with data-aware scheduling.
- Devise an analytical model to analyze the DAWS technique. This model gives a sub-optimal upper bound of the performance of the technique and helps us understand it in depth from a theoretical perspective.

- Implement the technique in MATRIX and evaluate MATRIX up to hundreds of nodes showing good performance under different scheduling policies.
- Explore the scalability of the DAWS technique through the SimMatrix simulator at extreme scales.
- Propose to address scalability issues of Hadoop/YARN through decentralized scheduling with MATRIX.
- An inclusive comparison between MATRIX and YARN using both benchmarking and real application workloads, up to 256 cores on the Amazon AWS Cloud.

5.2 Data-Aware Work Stealing

This section proposes the data-aware work stealing (DAWS) technique, the implementation details of the DAWS technique in both MATRIX and SimMatrix, and the analytic model that gives the sub-optimal performance of the proposed technique.

5.2.1 Data-aware Work Stealing Technique. Chapter 3 presented the fully distributed scheduling architecture (Figure 25) and the adaptive work stealing technique (section 4.3) to achieve the distributed load balancing for compute-intensive MTC applications. The adaptive work stealing technique is data-locality oblivious. This may incur significant data movement overheads for data-intensive workloads. We present the ideals that combine work stealing with data-aware scheduling.

5.2.1.1 Distributed KVS Used as a Meta-Data Service

We applied the ZHT key value store to maintain the task metadata in MATRIX. The task metadata is stored as (*key*, *value*) records for all the tasks, the *key* is task id, and the *value* is the important information related to the task. We formally define the *value* of a task metadata as the data structure, shown in Figure 55. The *value* includes the following information: the task status (e.g. queuing, being executed, and finished); data dependency conditions (*num_wait_parent*, *parent_list*, *children*); data locality information (*data_object*, *data_size*, *all_data_size*); task timestamps that record the times of different phases (e.g. submission, queued, execution, and end) of the task; and the task migrating history from one scheduler to another in the system.

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

Figure 55. Task metadata stored in ZHT

5.2.1.2 Task Submission

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

5.2.1.3 Task Removal

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

5.2.1.4 Distributed Queues in Scheduler

Each scheduler maintained four local task queues: wait queue (*WaitQ*), dedicated local ready queue (*LReadyQ*), shared work stealing ready queue (*SReadyQ*), and complete queue (*CompleteQ*), as shown in Figure 56. These queues [236] hold tasks in different states (stored as metadata in ZHT). A task is moved from one queue to another when state changes. With these queues, the system supports scheduling tasks with data dependencies specified by an arbitrary DAG.

Figure 57 displays the flowchart of the execution procedure of a task, during which the task is moved from one queue to another. Initially, the scheduler puts all the incoming tasks from the client to the *WaitQ*. A program (P1 in Figure 56) keeps checking every task in the *WaitQ* to see whether the task is ready to run by querying the metadata from ZHT.

The task metadata has been inserted into ZHT by the client. Specifically, only if the value of the field of "*num_wait_parent*" in the *TMD* is equal to 0 would the task be ready to run. When a task is ready to run, the scheduler makes decision to put it in either the *LReadyQ* or the *SReadyQ*, or push it to another node. The decision-making procedure is shown in the rectangle that is marked with dotted-line edges in Figure 57.

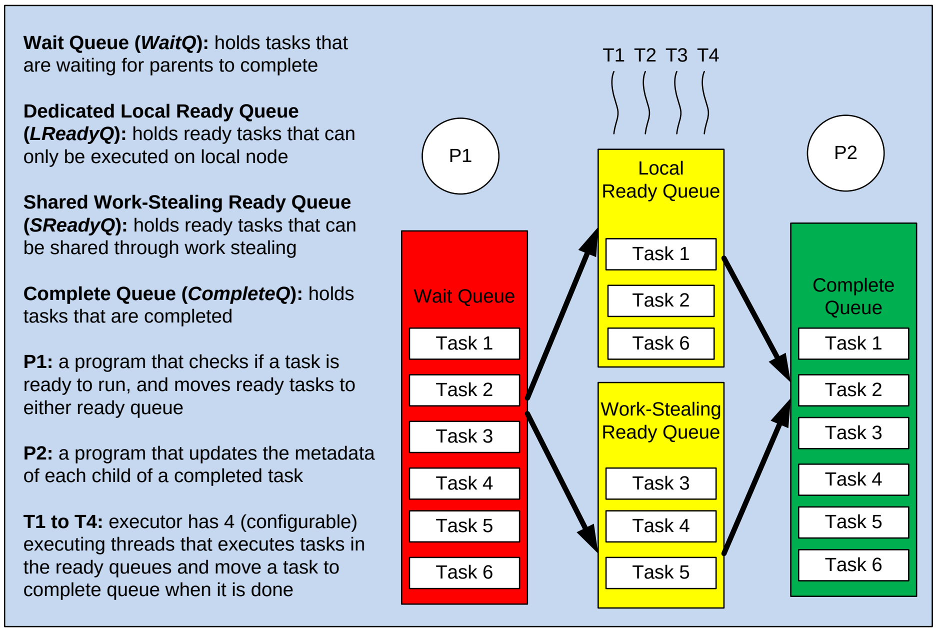

Wait Queue (*WaitQ*): holds tasks that are waiting for parents to complete
Dedicated Local Ready Queue (*LReadyQ*): holds ready tasks that can only be executed on local node
Shared Work-Stealing Ready Queue (*SReadyQ*): holds ready tasks that can be shared through work stealing
Complete Queue (*CompleteQ*): holds tasks that are completed
P1: a program that checks if a task is ready to run, and moves ready tasks to either ready queue
P2: a program that updates the metadata of each child of a completed task
T1 to T4: executor has 4 (configurable) executing threads that executes tasks in the ready queues and move a task to complete queue when it is done
T1 T2 T3 T4
P1
P2
Wait Queue
Task 1
Task 2
Task 3
Task 4
Task 5
Task 6
Local Ready Queue
Task 1
Task 2
Task 6
Work-Stealing Ready Queue
Task 3
Task 4
Task 5
Complete Queue
Task 1
Task 2
Task 3
Task 4
Task 5
Task 6

Figure 56. Specification of task queues in a MATRIX scheduler

When a task is done, it is moved to the *CompleteQ*. Another program (P2 in Figure 56) is responsible for updating the metadata for all the children of each completed task. P2 first queries the metadata of the completed task to find out the children, and then updates each child's metadata as follows: decreasing the "*num_wait_parent*" by 1; adding current scheduler id to the "*parent_list*"; adding the produced data object name to the "*data_object*"; adding the size of the produced object to the "*data_size*"; increasing the "*all_data_size*" by the size of the produced data object.

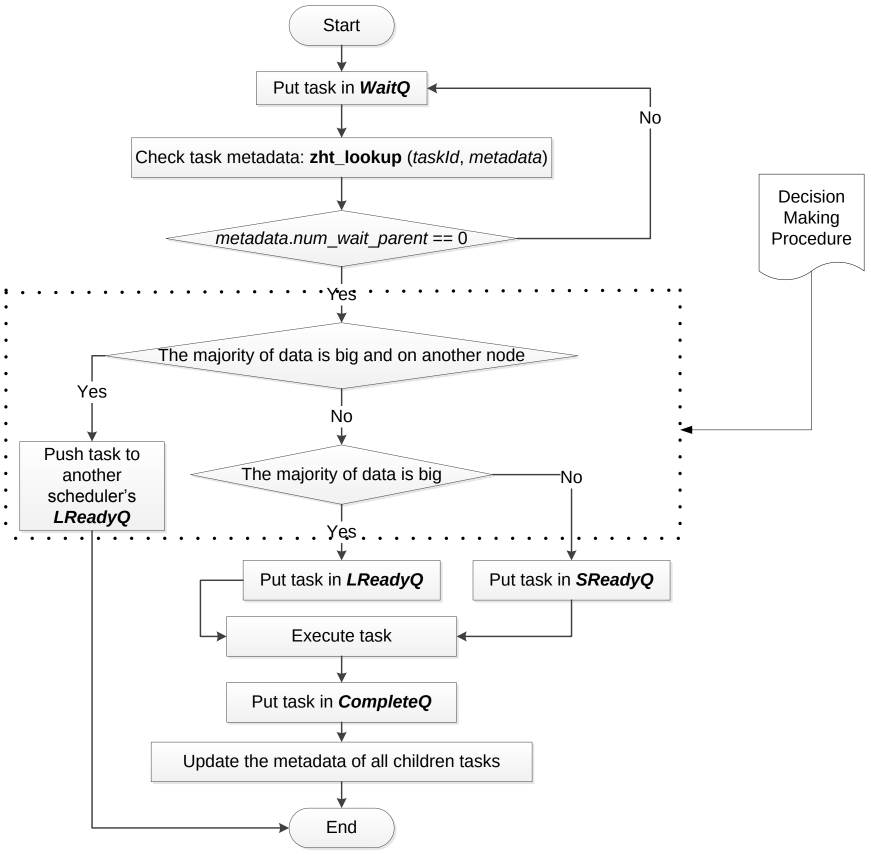
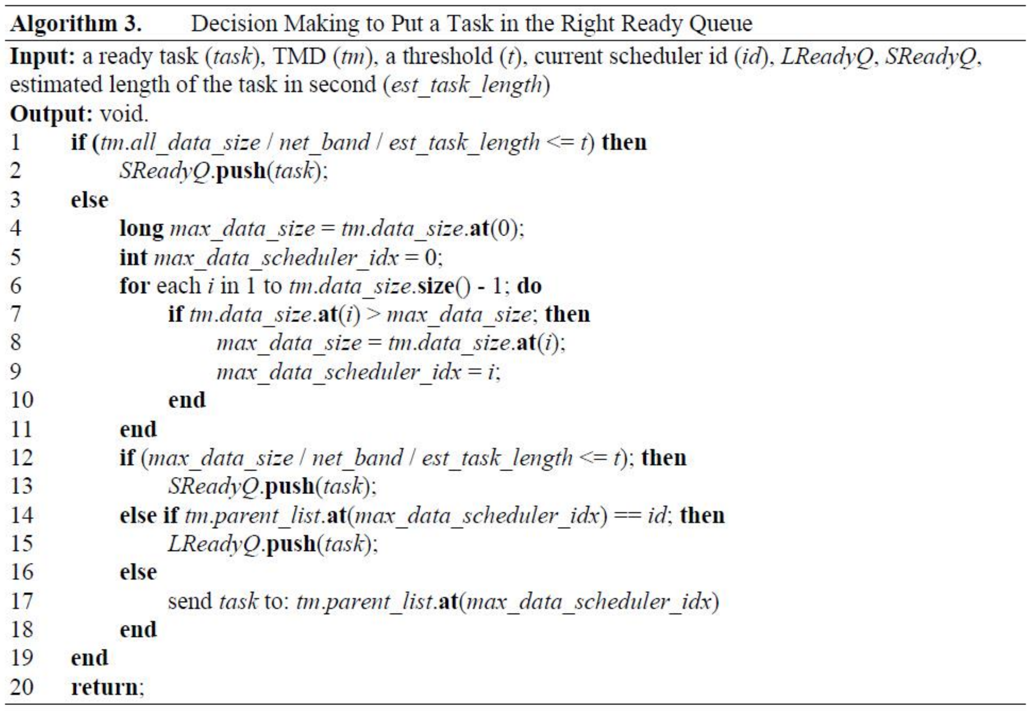

Start
Put task in ***WaitQ***
Check task metadata: **zht_lookup** (*taskId*, *metadata*)
metadata.num_wait_parent == 0
No
Yes
Decision Making Procedure
The majority of data is big and on another node
Yes
No
Push task to another scheduler's ***LReadyQ***
The majority of data is big
No
Yes
Put task in ***LReadyQ***
Put task in ***SReadyQ***
Execute task
Put task in ***CompleteQ***
Update the metadata of all children tasks
End

Figure 57. Flowchart of a task during the execution procedure

5.2.1.5 Decision Making Algorithm

We explain the decision-making procedure (the dotted-line rectangle in Figure 57) that decides to put a ready task in either the *LReadyQ* or the *SReadyQ*, or push it to another node, given in Algorithm 3 shown in Figure 58.

The *SReadyQ* stores the tasks that can be migrated to any scheduler for load balancing's purpose (lines 1 – 13), the "***load information***" queried by work stealing is the length of the *SReadyQ*; these tasks either don't need any input data or the demanded data volume is so small that the transferring overhead is negligible. The *LReadyQ* stores the tasks that require large volumes of data and the majority of the data is at the current node (lines 14 – 15); these tasks are confined to be scheduled and executed locally unless special policy is used. If the majority of the input data is large but at a different node, the scheduler then pushes the task to that node (lines 16 – 18). When a scheduler receives a pushed task, it puts the task in the *LReadyQ*. The threshold *t* defines the upper bound of the ratio of the data-transferring overhead (data size divided by network bandwidth: *net_band*) to the estimated task execution length (*est_task_length*). The smaller *t* means the less tolerance of moving data: If *t* is smaller, in order to put the task in *SReadyQ* (meaning that the task can be migrated through work stealing technique and that moving the data is tolerable), the task's all required data size (*tm.all_data_size*) needs to be smaller. This means less tolerance of moving a decent large amount of data.



Algorithm 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.

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

Figure 58. Algorithm of making decisions to put a task in the ready queue

As we do not know ahead how long a task will be running, we will need to predict the *est_task_length* in some ways. One method is to use the average execution time of the completed tasks as the *est_task_length*. This method works fine for workload of largely homogeneous tasks that have small variance of task lengths. For highly heterogeneous tasks, we can assume the task length conforms to some distributions, such as uniform random, Gaussian, and Gama, according to the applications. This can be implemented in our technique without much effort. Even though the estimation of the task length has deviation, our technique can tolerate it with the dynamic work stealing technique, along with the *FLDS* policy (will explain later). In addition, we have evaluated MATRIX using highly heterogeneous MTC workload traces (shown in Figure 24**)** that have 34.8M tasks with the minimum runtime of 0 seconds, maximum runtime of 1469.62 seconds, medium runtime of 30 seconds, average runtime of 95.20 seconds, and standard deviation of 188.08 [96], and MATRIX showed great scalability. The executor forks configurable number (usually equals to number of cores of the node) of threads to execute ready tasks. Each thread first pops tasks from the *LReadyQ*, and then from the *SReadyQ* if the *LReadyQ* is empty. Both ready queues are implemented as max priority queue based on the data size. When executing a task, the thread first queries the metadata to find the size and location of the required data, and then collects the data either from local or remote nodes. If neither queue has tasks, the scheduler does work stealing, and puts the stolen tasks in the *SReadyQ*.

5.2.1.6 Different Scheduling Policies

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

MLB (maximized load balancing): MLB considers only the load balancing, and all the ready tasks are put in the *SReadyQ* allowing to be migrated. We achieve the *MLB* policy by tuning the threshold *t* in Algorithm 3 to be the maximum possible value (i.e. LONG_MAX).

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

RLDS (rigid load balancing and data-locality segregation): RLDS sets the threshold *t* in Algorihtm 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).

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

5.2.1.7 Write Locality and Read Locality

The DAWS technique ensured the best write locality, and at the meanwhile, optimized the read locality. For write locality, every task writes the produced data locally. We had considered using ZHT as both a data and a task metadata service. However, it led to extreme difficulty in optimizing the data-locality as ZHT relies on consistent hashing to determine where to store the data. We also considered leveraging distributed file system (e.g. HDFS [117], FusionFS [42]) to manage the data, especially as FusionFS is optimized for write operations. We argue that the scheduling strategies are not affected by the actual method of data storage. We envision allowing data to be stored in a distributed file system as future work.

Read locality was optimized by migrating a task to where the majority of data resides for large data volumes. For small data volumes, tasks are run on wherever there are available compute resources to maximize utilization.

5.2.1.8 Caching

As in scientific computing, the normal pattern of data flows is write-once/ready-many (according to the assumption HDFS made in the Hadoop system [118]), we have implemented a caching mechanism to reduce the data movement overheads. In some cases, moving data from one node to another is inevitable. For example, if a task requires two pieces of data that are at different nodes, at least one piece of data needs to be moved. In a data movement, we cached the moved data locally at the receiver side for the future use by other tasks. This would significantly expedite the task execution progress. As data is written once, all the copies of the same data would have the same view, and no further consistency management would be needed.

5.2.1.9 Fault Tolerance

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

5.2.2 Implementation Details. We implement the proposed data-aware work stealing technique in the MATRIX task execution framework for scheduling data-intensive MTC applications and the big data applications in the Internet domains. In our new implementation, MATRIX simply uses ZHT as a black box through ZHT client APIs, this ensures easier maintainability and extensibility. The new version of MATRIX codebase is also made open source on Github: https://github.com/kwangiit/matrix_v2. It hs about 3K lines of C++ code implementing the MATRIX client, scheduler, the executor, and the DAWS logic, along with 8K lines of ZHT codebase, plus 1K lines of auto-generated code from Google Protocol Buffer [95]. MATRIX has dependencies on ZHT [51] and Google Protocol Buffer.

In order to study the proposed DAWS technique at extreme-scales, we also simulate it in SimMatrix. We add quite a few more event message types, such as moving data from one node to another, querying and updating the task metadata, monitoring the system status and the task execution progress, etc. The ZHT key-value store is simulated with an in-memory hash table in each scheduler, which supports the *lookup*, *insert*, and *remove* operations. The new version of SimMatrix codebase is again made open source on Github: https://github.com/kwangiit/SimMatrix. The input to SimMatrix is a configuration file that specifies the parameters, such as the simulated system environment (including number of cores per compute node, the network bandwidth, latency, and packing/unpacking overheads), the workloads, the scheduling policy, etc. Except for a java virtual machine, SimMatrix has no other dependencies.

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

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

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

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

Define the time taken to move $d(v)$ size of data to another compute node running a task w that requires the data is $t_{mvdata}(v,w)$ unit of time. $t_{mvdata}(v,w) = d(v)/B(v,w)$, in which $B(v,w)$ is the data transfer rate between the two compute nodes that run tasks v and w, respectively. For any arc $uv \in E$, it represents that task u is the parent of task v, meaning that task v requires the data output of task u. The parents of a task v is notated as $P(v) = \{u | uv \in E\}$. Generally, there are two categories of locations where a task could be scheduled. One is on the compute node that is different from all the nodes that ran the task's parents (*case 1*). In this case, the data items that are generated by all the parents need to be transmitted, and we assume that the task itself is not moved. The other one is on one of the compute nodes that ran at least one of the task's parents (*case 2*). In this case, the data items generated by all the other parents that were run on different compute nodes need to be transmitted, and we assume the task itself is also moved.

We define the problem of as follows:

- Define $t_{ef}(v)$ as the earliest finishing time of task v.
- The problem is to find out the largest $t_{ef}(v), v \in V$, which is the overall timespan of finishing a given workload.
- Assume the time taken to move a task is a small constant value, C_{mvtask}.

We devise the following recursive formulas to compute $t_{ef}(v)$:

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

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

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

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

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

This analysis gives a sub-optimal lower bound of the overall timespan (performance upper bound) of executing a workload DAG. We call it sub-optimal, because for a task, the solution above just considers one-step backwards (parents). This does not consider the situation of scheduling a task to where the grand-parents or grand-grand-parents were scheduled, which will eliminate unnecessary data movements. However, finding an optimal solution is an NP-hard problem.

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

5.3 Evaluation

In this section, we present the performance evaluation of the DAWS technique. We first conduct experiments using workloads structured as DAGs coming from two scientific applications, namely image stacking from astronomy [113] and all-pairs from biometrics [53]. We compare our results with those achieved through data-diffusion technique in Falkon [78], which employed a centralized data-aware scheduler. Next, we compare different scheduling polices using the all-pairs workload. Then, we run benchmarking workload DAGs. We further present the results achieved from the theoretical analysis, and from running the SimMatrix simulator up to extreme scales. We run MATRIX on the Kodiak cluster and SimMatrix on the Fusion machine.

5.3.1 Evaluations of Scientific Applications. We compare MATRIX with the Falkon centralized scheduler using two scientific applications: image stacking in astronomy [113] and all-pairs in biometrics [53] [121]. These two applications represent different data-intensive patterns.

5.3.1.1 Image Stacking in Astronomy

This application conducts the "stacking" of image cutouts from different parts of the sky. The procedure involves re-projecting each image to a common set of pixel planes, then co-adding many images to obtain a detectable signal that can measure their average brightness/shape. The workload DAG is represented as in Figure 59. The dotted lines represent independent tasks (t_0 to t_n) fetching ROI objects in a set of image files (f_0 to f_m) that are randomly distributed, and then generate an output individually. The last task (t_{end}) waits until collecting all the outputs, and then obtains a detectable signal.

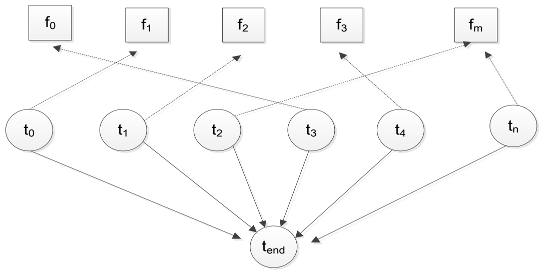
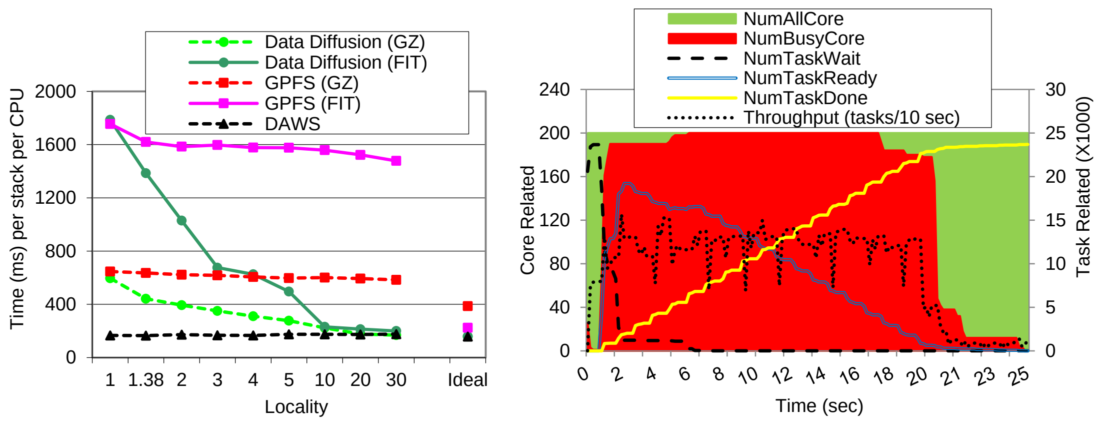


Figure 59. Image Stacking Workload DAG

Followed the workload characterization in [113], in our experiments, each task would require a file that has 2MB of data, and generates 10KB data of output. The ratio of the number of tasks to the number of files refers to *locality number*. Locality 1 means the number of tasks equals to the number of files, and each task requires a unique file. Locality n means that the number of tasks is n-times of the number of files, and each file is required by exactly n tasks. The higher the locality is, the less number of files and tasks there would be. The number of tasks and the number of files for each locality are given in [113]. We evaluate different locality values, i.e. 1, 1.38, 2, 3, 4, 5, 10, 20, 30. Each task would run for an average of 158 ms (as reported in [113]).

We ran experiments up to 200 cores for all locality values. The *MDL* policy was applied, as 2MB of data was large. We compared the DAWS technique implemented in

MATRIX with Falkon data diffusion at 128 cores (the largest scale the data diffusion ran). The results are shown in Figure 60 (a). “GZ” meant the files were compressed while “FIT” indicated the files were uncompressed. In the case of GPFS, each task read its required data from the remote GPFS parallel file system. Data diffusion first read all the data from GPFS, and then cached the data in the memory for centralized data-aware scheduling of all the tasks. MATRIX read all the data from the GPFS parallel file system, and then randomly distributed the data files to the memory of all the compute nodes. The MATRIX client submitted tasks to all the schedulers in the best-case scenario. The schedulers applied the DAWS for distributed data-aware scheduling of all the tasks. In all cases, the time taken to finish the workload was clocked before the data was copied into the system, continued while the tasks were loaded into the schedulers, and then kept going until all tasks were finished.



(a) Comparing DAWS with Data diffusion (b) Utilization graph for locality 30

Figure 60. Evaluations using the Image Stacking application

We see that at 128-core scale, the average task running time of our DAWS technique kept almost constant as locality increases, and was close to the ideal task running time (158ms). This is because the files were uniformly distributed over all compute nodes.

The only overhead came from the schedulers making decisions to migrate tasks in the right spots. When locality was small, the Data Diffusion (GZ) experienced large average time, which decreased to be close to the ideal time when locality increased to 30. Because, data was initially kept in a slower parallel file system that needed to be copied to local disks. When locality was low, more amount of data accesses from the file system was required. The average times of GPFS (GZ) and GPFS (FIT) remained at a high constant regardless of locality, due to that data was copied from the remote GPFS upon every data access. The performance increases slightly with higher locality, likely due to OS-level caching.

We show a visualization graph of MATRIX at 200 cores for locality 30 in Figure 60 (b). The utilization is calculated as the ratio of the area of the red region to that of the green region. We see there is a quick ramp-up period, after which the load is balanced across all the compute nodes. There is also a relatively long tail ramp-down period in the end when there are few tasks remaining, which is because with the *MDL* policy, no work stealing happened leading to poor load balancing in the end. But MATRIX can still achieve 75% utilization at 200 cores with fine-grained tasks (158ms).

The time per task of DAWS experienced a slight increase from Locality 1 (167ms) to 30 (176ms). We explain the reason in Figure 61, which shows the efficiencies of different localities in terms of scale. From Figure 61, we see that the efficiency decreases slightly with respect to both scale and locality (but still keeps above 75% efficiency for task length of 158ms). The reason is that the number of files per compute node is decreasing as the scale and locality increase. Therefore, more tasks on a compute node could not be run locally and need to be moved to the right nodes causing significant network traffic and load imbalance. At the extreme case where the locality is infinitely large, there would be only one file on one compute node, eventually all the tasks need to be run on that node.

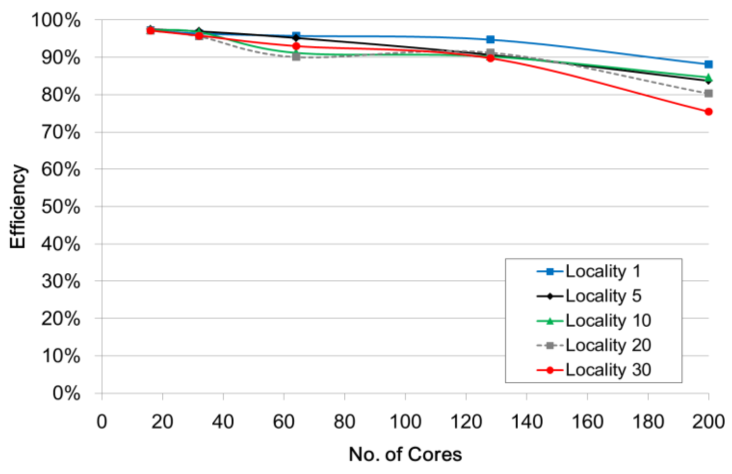


Figure 61. Efficiency of DAWS with Localities at scales

5.3.1.2 All-Pairs in Biometrics

All-Pairs [53] [121] is a common benchmark for data-intensive applications that describes the behavior of a new function on sets A and sets B. For example, in Biometrics, it is very important to find out the covariance of two sequences of gene codes. In this workload, all the tasks are independent, and each task execute for 1 second to compare two 12MB files with one from each set.

Figure 62 shows the an example of the workload DAG, in which four independent tasks operate on two sets of two files, and each task requires one file from each set.

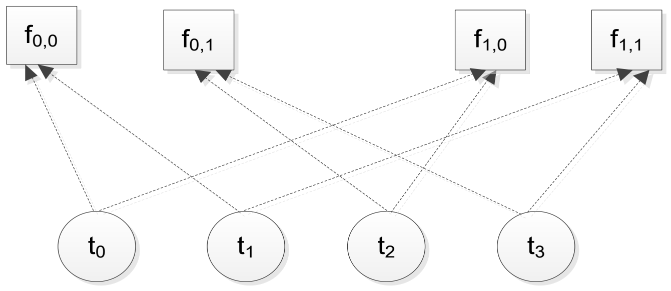
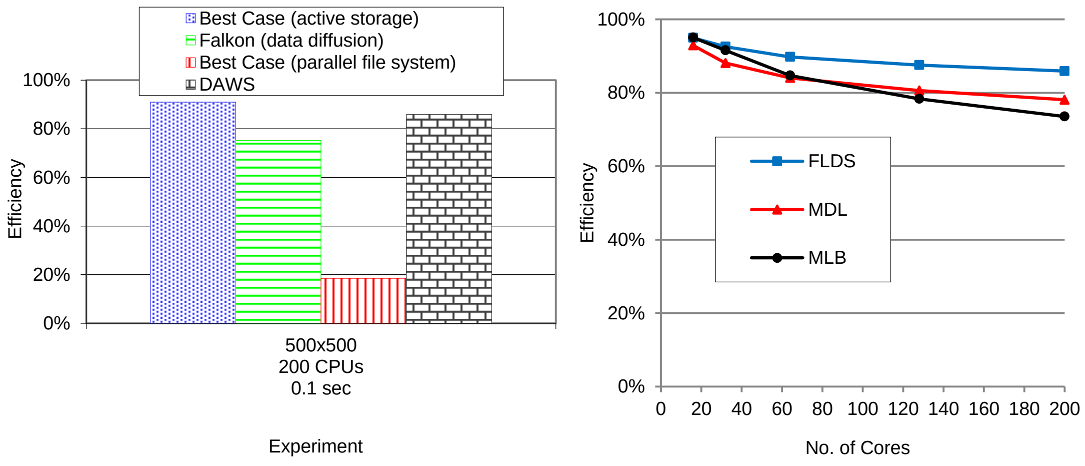


Figure 62. All-Pair Workload DAG

We run strong-scaling experiments up to 100 nodes with a 500*500 workload size. Therefore, there would be 250K tasks in total. All the 1000 files from two sets are uniformly distributed to each compute node, and all the tasks are randomly distributed. As a task needs two pieces of data files that may locate at different nodes at the worst case, one piece of data may need to be transmitted. To make the workload more data-intensive, we reduced the task running time by 10X, resulting in 100-ms running time with 24MB of data requirement. This is the same workload referenced in [53]. We use the FLDS policy, and at the end (80% of the workload is done) of the experiments, we set the time threshold tt to be 20 seconds initially, which is then decreased by half when moving ready tasks from LReadyQ to SReadyQ.



(a) Comparing DAWS with data diffusion (b) Comparing different scheduling policies

Figure 63. Evaluations using the all-pairs application

We compared MATRIX DAWS technique with Falkon data diffusion [53] at 200 cores, and Figure 63 (a) shows the results. The “active storage” term [121] meant all the files were stored in memory. For 100-ms tasks, our DAWS technique improved data diffusion by 10.9% (85.9% vs 75%), and was close to the best case using active storage

(85.9% vs 91%). This is because data diffusion applied a centralized index-server for data-aware scheduling, while our DAWS technique utilized distributed KVS, which was much more scalable. It was also worthy to point out that without harnessing data-locality (Best Case parallel file system), the efficiency was less than 20%, because all the files needed to be transmitted from the remote file system.

Although it is obvious that caching the data in the receiver's memory will usually be helpful to applications that have the *write-once/ready-many* pattern, we show the effect of caching of the *FLDS* policy for the all-pairs application in Figure 64.

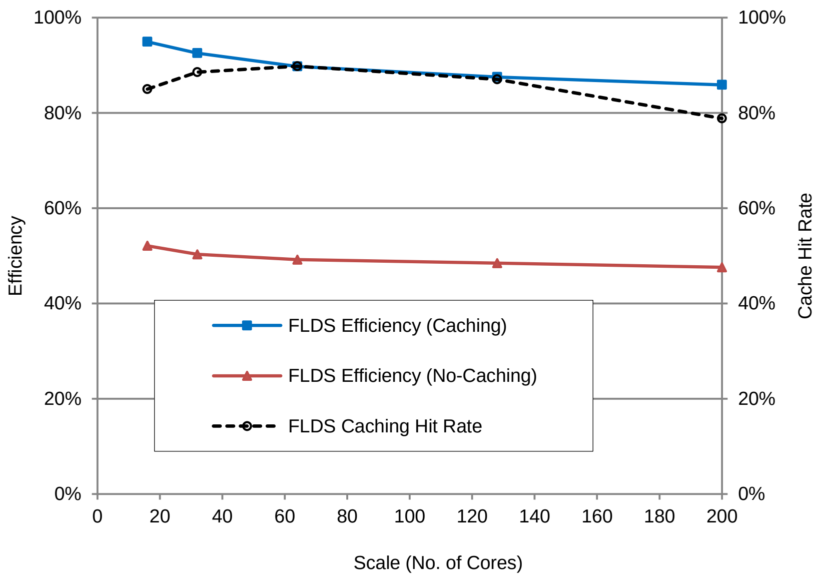


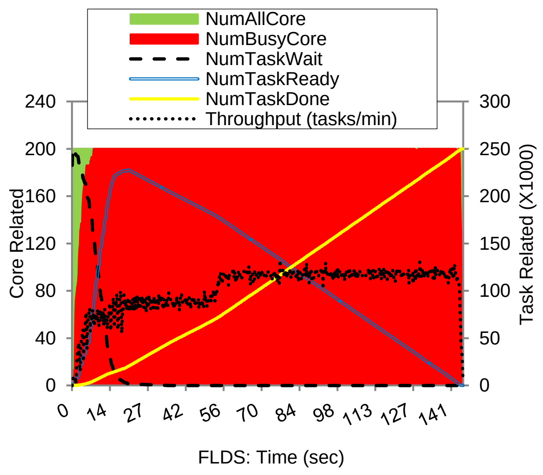
Figure 64. Comparison between Caching and No-Caching

We see that without caching, the *FLDS* policy is only able to achieve less than 50% efficiency at 200-core scales. This is because all the files are uniformly distributed and each task requires two files, therefore, from the probability's perspective, about half of the tasks need to move data. With caching turned on, we record the cache-hit rate, which shows as high as above 80%. This helps significantly and contributes to 85%+ efficiency. However, we should not conclude to always caching everything, because memory size is limited. Besides, depending on the access patterns, the cached data may never be reused. In the future, we will explore cache eviction mechanisms.

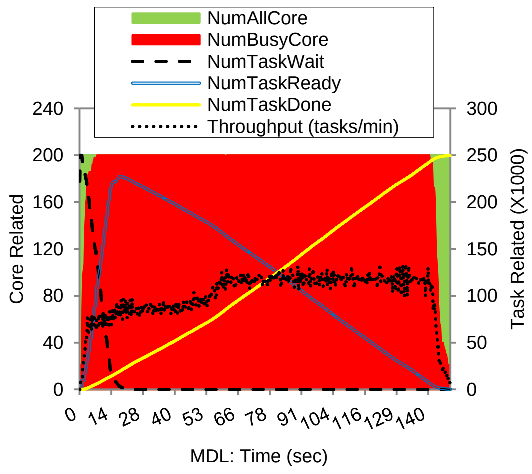
5.3.1.3 Comparisons of Different Scheduling Policies

We compared three scheduling polices using the all-pairs workloads, *MLB*, *MDL*, and *FLDS*, up to 200 cores with the results shown in Figure 63 (b).

As we expected, the *MLB* policy performed the worst, because it considered only load balancing and the required data was so large that transmitting it took significant amount of time. The *MDL* policy performed moderately. From the load balancing's perspective, *MDL* did quite well except for the ending period. Because it did not allow work stealing and loads might be imbalanced at the final stage leading to a long-tail problem. The *FLDS* policy was the best, because it allowed the tasks being moved from the *LReadyQ* to the *SReadyQ* as needed. This was helpful at the final stage when many nodes were idle while a few others were busy. To justify our explanation, we show the utilization figures of the *FLDS* and *MDL* policies in Figure 65.



(a) Utilization of FLDS policy



(b) Utilization of MDL policy

Figure 65. Utilization graph of the FLDS and MDL policies at 200 cores

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

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

5.3.2 Evaluations of Benchmarking Workload DAGs. We evaluate MATRIX using benchmarking workload DAGs, namely BOT, Fan-In, Fan-Out and Pipeline, represented in Figure 46. The difference between the workload DAGs in this section and those in section 4.6.4 is that the tasks are dependent on input and out data, while they were dependent on tasks themselves in section 4.6.4. BOT included independent tasks without data dependencies and was used as a baseline; Fan-In and Fan-Out were similar but with reverse tree-based shapes, and the parameters were the in-degree of Fan-In and the out-degree of Fan-Out; Pipeline was a collection of “pipes”. In each pipe, a task was dependent on the previous one. The parameter was the pipe size, meaning number of tasks in a pipe.

5.3.2.1 Representative Applications of the Benchmarking Workload DAGs

After studying the workload patterns of a category of MTC data-intensive applications using the Swift workflow system [15] [67], we summarize the four benchmarking workload DAGs that are representative and cater to the data-flow patterns of different applications. Some applications display a single type of workload DAG pattern, while others have a combination of several workload DAG types, out of the four DAGs.

For example, the All-Pairs application in Biometrics is an example of the BOT workload DAGs, in which, the tasks are independent and every task requires two input files with each one coming from individual set. The Image Stacking application in Astronomy has a two-layer Fan-In DAG data-flow pattern. The top layer includes many parallel tasks with each one fetching an individual ROI object in a set of image files, and the bottom layer has an aggregation task that collects all the outputs to obtain a detectable signal. The workload DAGs of both applications were shown in Figure 59 and Figure 62. The molecular dynamics (MolDyn) application in Chemistry domain aims to optimize and automate the computational workflow that can be used to generate the necessary parameters and other input files for calculating the solvation free energy of ligands, and can also be extended to protein-ligand binding energy. Solvation free energy is an important quantity in Computational Chemistry with a variety of applications, especially in drug discovery and design. The MolDyn application is an 8-stage workflow. At each stage, the workload data flow pattern is either a Fan-Out or Fan-In DAG. The workload DAG was shown in [126]. The functional magnetic resonance imaging (fMRI) application in the medical imaging domain is a functional neuroimaging procedure that uses MRI technology to measure the brain activities by detecting changes in blood flow through the blood-oxygen-level dependent (BOLD) contrast [127]. In Swift, an fMRI study is physically represented in a nested directory structure, with metadata coded in directory and file names, and a volume is represented by two files located in the same directory, distinguished only by file name suffix [128]. The workload to process each data volume has a 2-pipe pipeline pattern. The dominant pipeline consists of 12 sequential tasks. Figure 66 shows the workload DAGs of both one volume and ten volumes.

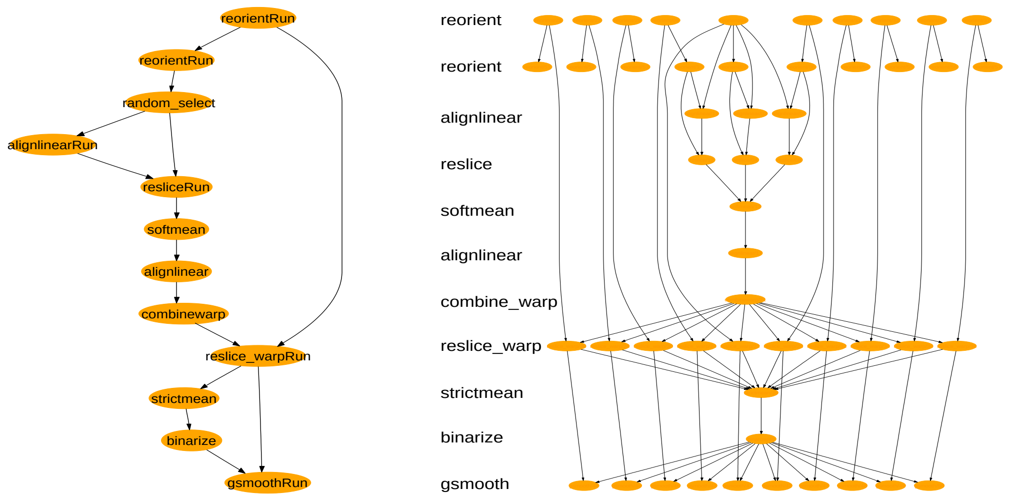


Figure 66. Workload DAGs of the fMRI applications

5.3.2.2 Evaluation Results

MATRIX client can generate a specific workload DAG, given the input parameters such as DAG type (BOT, Fan-In, Fan-Out, and Pipeline), DAG parameter (in-degree, out-degree, and pipe size). Table 7 gives the experiment setups, which were the same for all the DAGs: the executor had 2 executing threads with each one executing 1000 tasks on average (a weak-scaling configuration); the tasks had an average running time of 50ms (0 to 100ms) and outputted an average data size of 5MB (0 to 10MB), both were generated with uniform random distribution; we set the threshold *t* to 0.5, a ratio of 0.5 of the data-transferring time to the task running time, the time threshold *tt* of the *FLDS* policy to 10 sec, and the polling interval upper bound to 50 sec; the DAG parameters (in-degree, out-degree and pipe size) were set to 10. Figure 67 shows the results of scheduling all the DAGs in MATRIX using the *FLDS* policy up to 200 executing threads.

Table 7. Experiment Setup

Workload Parameters			**DAG Parameters**			**FLDS Policy Parameters**		
# task per core	*average length (ms)*	*average output (MB)*	*Fan-Out degree*	*Fan-In degree*	*Pipeline pipe size*	*t*	*tt (sec)*	*Polling interval upper bound (sec)*
1000	50	5	10	10	10	0.5	10	50

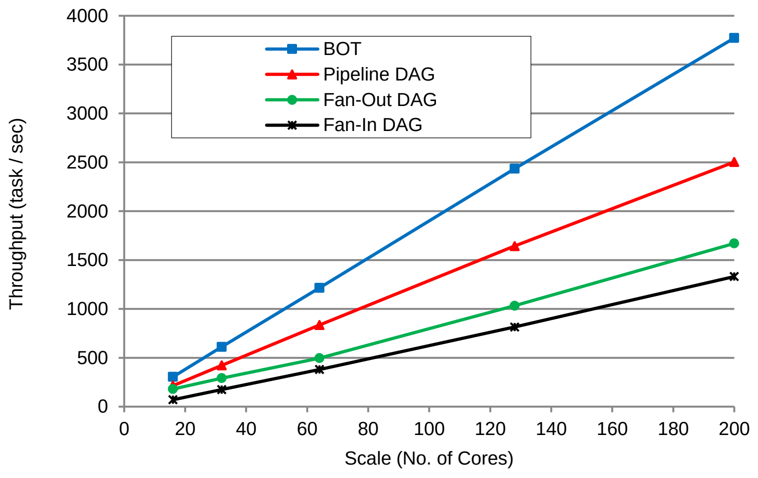


Figure 67. Evaluations using benchmarking workload DAGs

We see that BOT achieved nearly optimal performance with the throughput numbers implying a 90%+ efficiency at all scales. This is because tasks were run locally without requiring any data. The other three DAGs showed great scalability, as the throughput was increasing linearly with the scale. Comparing the three DAGs with data dependencies, Pipeline showed the highest throughput, because each task needed at most one data from the parent. Fan-Out experienced a long ramp-up period, as at the beginning, only the root task was ready to run. As time increased, more tasks were ready resulting in better utilization. Fan-In DAG was the opposite. At the beginning, tasks were run fast, but it got slower and slower due to the lack of tasks, leading to a long tail that had worse effect than the slow ramp-up period of Fan-Out.

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

5.3.3 Exploring the Scalability through SimMatrix. We explore the scalability of the proposed DAWS technique through SimMatrix up to extreme scales. The same benchmarking workload DAGs, the *FLDS* policy, and the experiment setups as specified in section 5.3.4.2 for MATRIX are used in SimMatrix. We first validate SimMatrix against MATRIX up to 200 cores; then show the scalability of the DAWS technique in SimMatrix up to 128K cores; and finally show how close the performance achieved through DAWS technique to that is from the sub-optimal upper bound up to 128K cores.

5.3.3.1 Validation of SimMatrix against MATRIX

Before using SimMatrix to gain valuable insights for the DAWS technique, we choose to validate our simulation results gained from SimMatrix against those achieved from MATRIX up to 200 cores for all the DAGs. To make the simulation results more convincible, each simulation experiment is run 10 times and the throughput result is the average throughput of the 10 runs for every workload. The validation results are shown in Figure 68.

The solid lines are the results of MATRIX, the round dotted lines represent the results of SimMatrix, and the dash dotted lines show the normalized difference between them (*abs*(MATRIX-SimMATRIX)/MATRIX). The results of BOT, Pipeline, Fan-Out and Fan-In are interpreted with the colors of blue, red, green, and black, respectively. We also display the 95% confidence intervals (both positive and negative) of the throughputs of all workloads in SimMatrix, shown as error bars in the figure. The 95% confidence interval is calculated as: $\pm t_{n-1,0.975} \times \sigma/n$, in which n is the sample size (10 in our case), σ is the standard deviation, and $t_{9,0.975} = 1.96$ according to the standard normal distribution table. The values of the 95% confidence intervals are negligible when comparing with the throughput values. This indicates that our simulation results are stable and convincible. Furthermore, we see that the throughput results of SimMatrix match those of MATRIX with an average difference of less than 10% at all scales for all the workload DAG patterns. The small differences show that SimMatrix is accurate enough in simulating the fully distributed scheduling architecture and the DAWS scheduling technique.

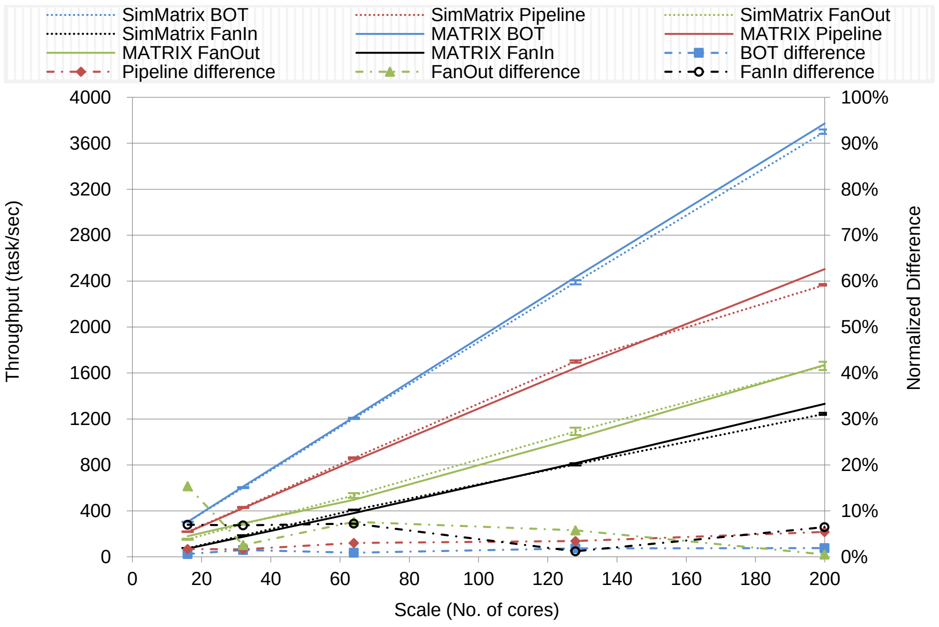


Figure 68. Validation of SimMatrix against MATRIX

One thing to notice is that SimMatrix shows smaller throughput values than MATRIX for all the workloads at increasingly large scales, indicating that SimMatrix has larger simulated overheads. The reason is that the simulated communication messages in SimMatrix have larger message sizes than the real messages in MATRIX. In SimMatrix, messages are defined in the "Message" class that is general for all kinds of messages, thus including many unnecessary attributes for simple messages. For example, in work stealing, the messages of querying the load information of neighbors should be simple (e.g. the string "query load"). However, the general "Message" class of SimMatrix include many other attributes, such as "*String* content", "*Object* obj", "*long* eventId", which enlarge the actual message sizes, leading to larger simulated communication overheads. However the small difference between SimMatrix and MATRIX indicate that this implementation has little impact on the accuracy of SimMatrix.

5.3.3.2 Scalability Exploration through SimMatrix

Since we have validated the results of SimMatrix, we explore the scalability of the fully distributed scheduling architecture and the DAWS technique towards extreme scales up to 128K cores in SimMatrix running weak-scaling experiments for all the benchmarking DAGs. The results are shown in Figure 69.

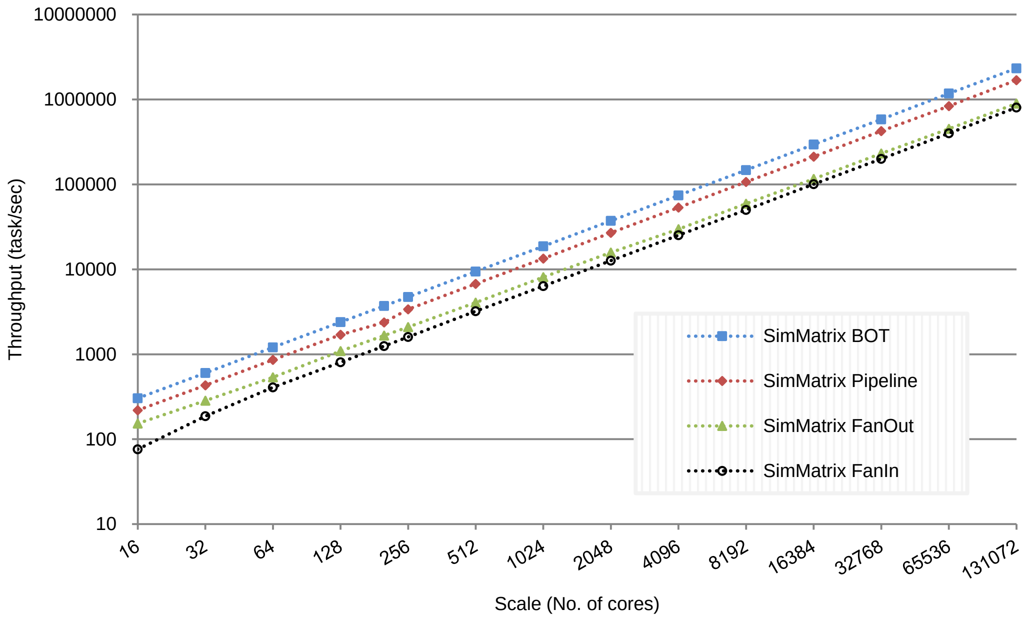


Figure 69. Scalability of the DAWS technique up to 128K cores (128M tasks)

For the four benchmarking workloads, SimMatrix shows the same relative throughput results as MATRIX: BOT performs the best, and the second best is Pipeline, followed by Fan-Out, with Fan-In performing the worst. These results further validate SimMatrix and justify our explanations of the MATRIX results in section 5.3.2.2. The throughput trends with respect to the scale indicate that the DAWS technique has great scalability for all the workloads. At 128K-core (128M tasks) scale, the DAWS technique achieves extremely 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. These throughput numbers satisfy the scheduling needs of MTC data-intensive applications towards extreme scales. The trend is likely to hold towards million-core scales, and we will validate in the future. One fact is that our SimMatrix simulator does not have accurate network interconnection models. Currently, the communication overhead of a message is modeled as: $\frac{message\ size}{Network\ Bandwidth} + latency$, and the *latency* is set as a fixed value (i.e. 10us) for all scales. To further explore the impact of network topology on SimMatrix, we leverage the data presented in [129], which is the state-of-the-art research work of modeling and simulating the exascale communication networks. The models and simulations were built on top of the ROSS parallel discrete event simulator [130] that has been validated extensively. We refer to the maximum latency values of torus network with different configurations at 256K-node scale, presented in [129]. We summarize the statistics of the maximum latency value, the torus network configuration, the message rate, of both SimMatrix with the four workloads and the referred work, in Table 8.

From Table 8, we see that the referred torus network simulation had much higher message rate than SimMatrix. To amplify the impact of the torus network topology on

SimMatrix, we choose to replace the latency value (i.e. 10us) of SimMatrix with the maximum "maximum latency" value (i.e. 1500 us) of the torus network at the scales larger than 200 cores that were validated in Figure 68. Notice that this is exaggerating the network communication overhead in SimMatrix, because SimMatrix has much lower message rate, which contributes significantly to the maximum latency.

Table 8. Statistics of SimMatrix and the referred simulation research

	SimMatrix				***Referred Torus Network***		
	BOT	*Pipeline*	*FanIn*	*FanOut*	*512-ary 2-cube*	*64-ary 3-cube*	*8-ary 6-cube*
Message rate (per nanosecond)	0.05 5	0.12	0.052	0.079	2.5	14	54
Maximum latency (microsecond)	10	10	10	10	1500	450	325

We rerun our simulations configured with the 1500us latency at scales larger than 200 cores, and compare the new simulation results with the 10us-latency results presented in Figure 69. Figure 70 shows the comparison results up to 128K cores. The solid lines are the 10us-latency throughput results of the four workloads, the round dotted lines represent the 1500us-latency results (Torus SimMatrix), and the dash dotted lines show the normalized performance degradation between them (SimMatrix-Torus SimMatrix) / SimMatrix). We see that the performance degradation due to the exaggerated network communication overhead is at most 12% for all the workloads at all scales. The small degradation values indicate that the time taken to execute all the tasks dominates the overall running of the workloads, and the network communication overheads are a minor factor. These results show that our simulation framework is accurate, even at extreme scales that have larger network communication overheads.

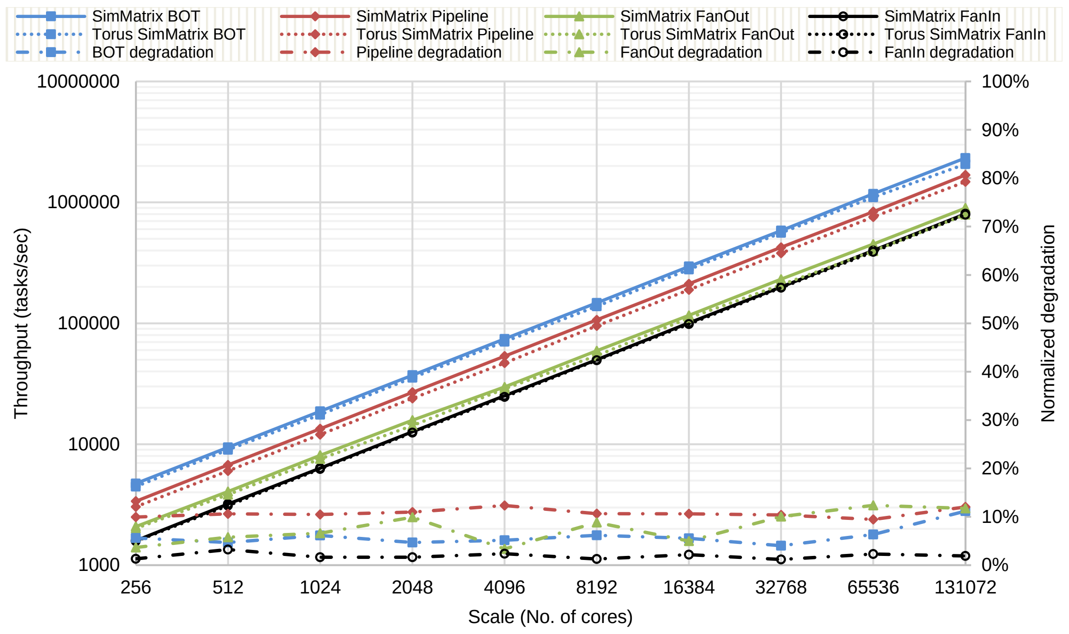


Figure 70. Comparing SimMatrix with SimMatrix of Torus network topology

Through simulations, we have shown that *the fully distributed scheduling architecture and the proposed DAWS scheduling technique have the potential to scale up to the level of exascale computing.*

5.3.4 Simulation vs. Theoretical Sub-optimal Solution. We have shown that the DAWS technique has great scalability towards extreme scales for the MTC data-intensive workloads through simulations. The question to ask is what the quality of the results achieved with the DAWS technique is. We measure the quality by comparing the simulation results with those achieved through the analytical sub-optimal solution that we have devised in section 5.2.3. The task running time $t_{exec}(v)$, waiting time $t_{qwait}(v)$, and the size of the output data $d(v)$ for each individual task are set the same as the workloads used in SimMatrix.

We apply dynamic programming to calculate the earliest time ($t_{ef}(v)$) to finish a task. The earliest finishing time of the last task is the overall timespan to finish all the tasks, and we compute the throughput based on this. We compare the results of SimMatrix with those of the analytical sub-optimal solution for the four benchmarking workloads up to 128K cores, shown in Figure 71.

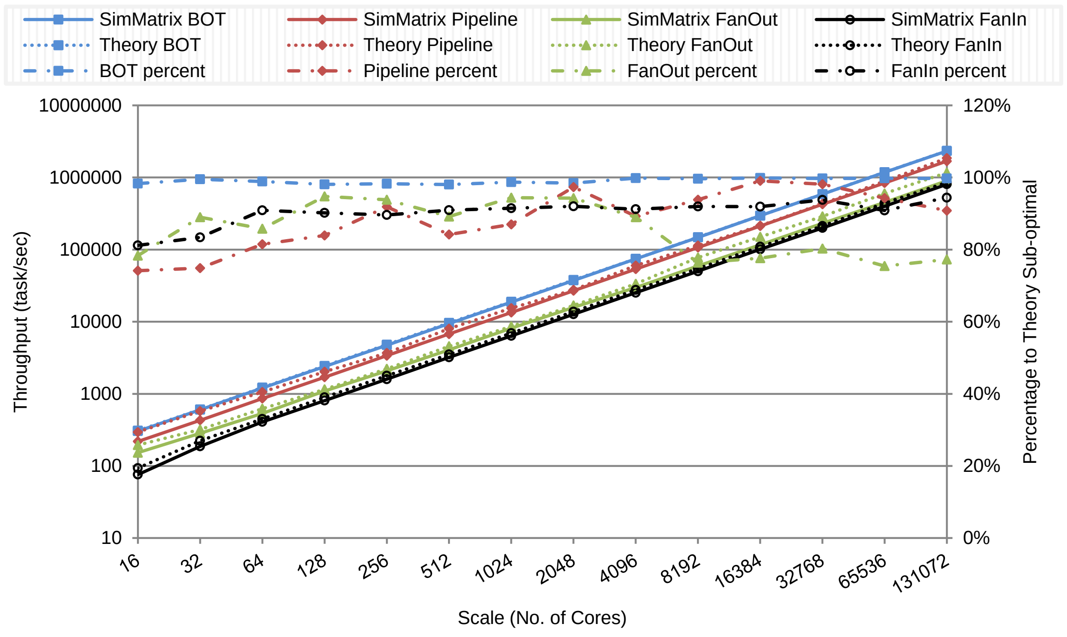


Figure 71. Comparison between SimMatrix and the analytical sub-optimal solution

The solid lines are the results of SimMatrix, the round dotted lines represent the results achieved from the sub-optimal solution (Theory), and the dash dotted lines show the ratio of the simulation results to those of sub-optimal solutions. Similar to Figure 68, the results of BOT, Pipeline, Fan-Out and Fan-In are interpreted with the colors of blue, red, green, and black, respectively. We see that for all the workloads at different scales, our DAWS technique could achieve performance within 15% compared with the analytical sub-optimal solution. This relatively small percentage of performance loss indicates that the DAWS technique has the ability to achieve performance with high quality (85%+). In order to highlight the numbers, we list the average percentage of performance that can be achieved through the DAWS technique for all the workloads in Table 9. We see that For the BOT workload, our technique works almost as good as the sub-optimal solution with a quality percentage of 99%, due to no data movement. For the other three workloads, the DAWS technique achieves about 85.4% of the sub-optimal for Fan-Out, 88.6% of the sub-optimal for Pipeline, and 90.3% of the sub-optimal for Fan-In, on average. The reason that the Pipeline and Fan-Out DAGs show bigger performance loss when comparing with the Fan-In DAG is because the former two DAGs have worse load balancing. For Fan-Out, every task (except for the leaf tasks) produces data of 5MB on average for 10 children, which may end up be run on the same node as their parent for minimizing the data movement overhead, leading to poor load balancing. Our *FLDS* policy mitigates this issue significantly. The same situation happens for the Pipeline DAG, but is less severe, because every task has only one child. However, the DAWS technique, configured with the *FLDS* policy, is still able to achieve 85.4% and 88.6% of the sub-optimal solution for the Fan-Out and Pipeline DAG, respectively, which demonstrate the high-quality performance.

Table 9. Comparing achieved performance with the sub-optimal

Workloads	***BOT***	***Pipeline***	***FanOut***	***FanIn***
Percentage of sub-optimal	99.043804%	88.572047%	85.381356%	90.2602799%

These results show that *the DAWS technique is not only scalable, but is able to achieve high quality performance results that are close to the bounded sub-optimal results within at most 15% performance loss.*

5.3.5 Analysis of the DAWS Technique. We have shown that the DAWS technique is not only scalable, but is able to achieve high quality performance within 15% of the sub-optimal solution on average for different benchmarking workload DAGs. We justify that the technique is applicable to a general class of MTC data-intensive workloads. This is due to the adaptive property of the DAWS technique. The adaptive property refers to the ability of adjusting the parameters for the best performance during the runtime according to system states. We have enunciated how the adaptive work stealing works, how to choose the best scheduling policies, individually. These should be considered together, along with the other parameters, such as the ratio threshold (*t*), and the upper bound of the execution time of the *LReadyQ* (*tt*) in the *FLDS* policy. The DAWS technique could always start with the *FLDS* policy. Based on the data volumes transferred during runtime, it is able to switch to other policies. If too much data is transferred, the technique would switch to the *MDL* policy; on the other hand, the technique would switch to the *MLB* policy if few data is transmitted. In addition, we can set the initial *tt* value in the *FLDS* policy, double the value when moving ready tasks from the *LReadyQ* to the *SReadyQ*, and reduce the value by half when work stealing fails. In order to cooperate with the *FLDS* policy, after the polling interval of work stealing hits the upper bound (no work stealing anymore), we set the polling interval back to the initial small value only if the threshold *tt* becomes small enough. This would allow to do work stealing again.

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

5.4 Overcoming Hadoop Scaling Limitations through MATRIX

Data driven programming models like MapReduce have gained the popularity in large-scale data processing in the Internet domains. Although great efforts through the

Hadoop implementation and framework decoupling (e.g. YARN, Mesos) have allowed Hadoop to scale to tens of thousands of commodity cluster processors, the centralized designs of the resource manager, task scheduler and metadata management of HDFS file system adversely affect Hadoop's scalability to tomorrow's extreme-scale data centers. This section aims to address the YARN scaling issues through a distributed task execution framework, MATRIX, which is originally designed to schedule the executions of data-intensive scientific applications of many-task computing (MTC) on supercomputers. We propose to leverage the distributed design wisdoms of MATRIX to schedule arbitrary data processing applications in the cloud domain. We compare MATRIX with YARN in processing typical Hadoop workloads, such as WordCount, TeraSort, RandomWriter and Grep, as well as the Ligand application in Bioinformatics on the Amazon Cloud. Experimental results show that MATRIX outperforms YARN by 1.27X for the typical workloads, and by 2.04X for the real application [227]. We also run and simulate MATRIX with fine-grained sub-second workloads. With the simulation results giving the efficiency of 86.8% at 64K cores for the 150ms workload, we show that MATRIX has the potential to enable Hadoop to scale to extreme-scale data centers for fine-grained workloads.

5.4.1 Motivations. Applications in the Internet and Cloud domains (e.g. Yahoo! weather [161], Google Search Index [162], Amazon Online Streaming [163], and Facebook Photo Gallery [164]) are evolving to be data-intensive that process large volumes of data for interactive tasks. This trend has led to the programming paradigm shifting from the compute-centric to the data driven. Data driven programming models [165], in the most cases, decompose applications to embarrassingly parallel tasks that are structured as Direct Acyclic Graph (DAG) [166]. In a specific application DAG, the vertices are the discrete tasks, and the edges represent the data flows from one task to another. This is the same for the data-intensive many-task computing application.

MapReduce [76] is the representative of the data driven programming model that aims at processing large-scale data-intensive applications in the Cloud domains on commodity processors (either an enterprise cluster, or private/public Cloud). In MapReduce, applications are divided into two phases (i.e. Map and Reduce) with an intermediate shuffling procedure, and the data is formatted as unstructured (key, value) pairs. The programming framework is comprised of three major components: the resource manager manages the global compute nodes, the task scheduler places a task (either a map task or a reduce task) on the most suitable compute node, and the file system stores the application data and metadata.

The first generation Hadoop [118] (Hadoop_v1, circa 2005) was the open-source implementation of the MapReduce. In Hadoop_v1, the centralized job tracker plays the roles of both resource manager and task scheduler; the HDFS is the file system [117] to store the application data; and the centralized namenode is the file metadata server. In order to promote Hadoop to be not only the implementation of MapReduce, but one standard programming model for a generic Hadoop cluster, the Apache Hadoop community developed the next generation Hadoop, YARN [112] (circa 2013), by decoupling the resource management infrastructure with the programming model. From this point, when we refer to Hadoop, we mean YARN.

YARN utilizes a centralized resource manager (RM) to monitor and allocate resources to an application. Each application delegates a centralized per-application master (AM) to schedule tasks to resource containers managed by the node manager (NM) on the allocated computing nodes. The HDFS file system and centralized metadata management remain the same. Decoupling of the resource management infrastructure with the programming model enables Hadoop to run different application frameworks (e.g. MapReduce, Iterative application, MPI, and scientific workflows) and eases the resource sharing of the Hadoop cluster. Besides, as the scheduler is separated from the RM with the implementation of the per-application AM, the Hadoop has achieved unprecedented scalability. Similarly, the Mesos [106] resource sharing platform is another example of the scalable Hadoop programming frameworks.

However, there are inevitable design issues that prevent Hadoop from scaling to extreme scales, the scales that are 2 to 3 orders of magnitude larger than that of today’s distributed systems; similarly, today’s scales do not support several orders of magnitude fine grained workloads (e.g. sub-second tasks). The first category of issues come from the centralized paradigm. Firstly, the centralized RM of YARN is a bottleneck. Although the RM is lightweight due to the framework separation, it would cap the number of applications supported concurrently as the RM has limited processing capacity. Secondly, the centralized per-application AM may limit the task placement speed when the task parallelism grows enormously for the applications in certain domains. Thirdly, the centralized metadata management of HDFS is hampering the metadata query speed that will have side effects on the task placement throughput for data-locality aware scheduling. The other issue comes from the fixed division of Map and Reduce phases of the Hadoop jobs. This division is simple and works well for many applications, but not so much for more complex applications, such as iterative MapReduce [12] that supports different levels of task parallelism, and the irregular applications with random DAGs. Finally, the Hadoop framework is not well suited for running fine-grained workloads with task durations of sub-seconds, such as the lower-latency interactive data processing applications [21]. The reason is twofold. One is that the Hadoop employs a pull-based mechanism. The free containers pull tasks from the scheduler; this causes at least one extra Ping-Pong overhead per-request in scheduling. The other one is that the HDFS suggests a relatively large block size (e.g. 64MB) when partitioning the data, in order to maintain efficient metadata management. This confines the workload's granularity to be tens of seconds. Although the administrators of HDFS can easily tune the block size, it involves manual intervention. Furthermore, too small block sizes can easily saturate the metadata server.

This work proposes to utilize MATRIX and the DAWS technique to do scalable task placement for Hadoop workloads, with the goal of addressing the Hadoop scaling issues. We leverage the distributed design wisdoms of MATRIX in scheduling data processing applications in clouds. We compare MATRIX with YARN using typical Hadoop workloads, such as WordCount, TeraSort, RandomWriter, and Grep, as well as an application in Bioinformatics, on Amazon Cloud up to 256 cores. We also run and simulate MATRIX with fine-grained sub-second workloads and MATRIX shows the potential to enable Hadoop to scale to extreme scales.

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

5.4.2.1 Centralized resource manager

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

5.4.2.2 Application task scheduler

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

5.4.2.3 Centralized metadata management

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

5.4.2.4 Limited data flow pattern

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

These inherent design issues prevent the Hadoop framework scaling up to extreme scales, which should be addressed sooner rather than later.

5.4.3 Leveraging MATRIX Distributed Design Wisdoms. Although MATRIX was designed for scheduling fine-grained MTC data-intensive applications on supercomputers, we show how to leverage the MATRIX distributed design wisdoms to overcome the Hadoop scaling limitations for arbitrary data processing applications.

5.4.3.1 Distributed Resource Management

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

5.4.3.2 Distributed Task Scheduling

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

5.4.3.3 Distriuted Metadata Management

We can leverage distributed KVS, as ZHT used in MATRIX, to offer a flat namespace in managing the task and file metadata. Comparing with other ways, such as sub-tree partitioning [167] and consistent hashing [29], flat namespace hashing has the ability to achieve both good load balancing, and faster metadata accessing rate with zero-hop routing. In sub-tree partitioning, the namespace is organized in sub trees rooted as individual directory, and the sub trees are managed by multiple metadata servers in a distributed way. As the directories may have wide distribution of number of files and file sizes, the sub-tree partitioning may result in poor load balancing. In consistency hashing, each metadata server has partial knowledge of the others. This partial connectivity leads to extra routing hops needed to find the right server that can satisfy a query. For example, MATRIX combines the task and file metadata as (key, value) pairs that represent the data dependencies and data file locations and sizes of all the tasks in a workload. For each task, the key is the “taskId”, and the value specifies the “parent” and “child” tasks, as well as the names, locations, and sizes of the required data files. One may argue that the full connectivity of the KVS will be an issue at extreme scales. However, our pervious simulation results [27] showed that in a fully connected architecture, the number of communication messages required to maintain a reliable service is trivial when comparing with the number of request-processing messages.

5.4.3.4 Fault Tolerance

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

5.4.3.5 Elastic Property

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

5.4.3.6 Applicability to the MapReduce programming Model

MATRIX is applicable to run the MapReduce applications. The applications are represented as a simple DAG that has two phases. In the first phase, each task takes the same amount of data of input files to do analysis and generates an output result. In the second phase, each task requires all the output results and aggregates a final output. We will show how MATRIX performs for typical Hadoop applications.

5.4.3.7 Support of arbitrary application DAG

MATRIX can support much broader categories of data-intensive applications with various data flow patterns, such as the Hadoop jobs, the iterative applications, and the irregular parallel applications. The MATRIX clients take any arbitrary application DAG as input. Before submitting the tasks, the clients insert the initial task dependency information into ZHT. Later, the schedulers update the dependency with the added data size and locality information when executing tasks, through the ZHT interfaces. We believe that MATRIX could be used to accelerate a large number of parallel programming systems, such as Swift [15] [67], Pegasus [69], and Charm++ [92].

5.4.4 Evaluation of MATRIX with Hadoop Workloads. In this section, we evaluate MATRIX by comparing it with YARN in processing typical Hadoop workloads, such as WordCount, TeraSort, RandomWriter and Grep, as well as an application in Bioinformatics, on the Amazon EC2 Large testbed. We first run YARN with these workloads and obtain the trace files. Through the trace files, we generate workload DAGs that become the inputs of MATRIX. We also evaluate the scalability of MATRIX through simulations running on the Fusion machine up to extreme scales with sub-second workloads. We aim to show that MATRIX is not only able to perform better than YARN for the workloads that are tailored for YARN, but also has the ability to enable Hadoop to scale to extreme scales for finer-grained sub-second workloads.

5.4.4.1 YARN Configuration

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

5.4.4.2 Benchmarking Hadoop Workloads

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

(1) WordCount

The WordCount is a typical two-phase Hadoop workload. The map task count the frequency of each individual word in a subset data file, while the reduce task shuffles and collects the frequency of all the words. Figure 72 and Figure 73 show the performance comparisons between MATRIX and YARN.

From Figure 72, we see at all scales, MATRIX outperforms YARN by 1.26X on average for tasks with average lengths ranging from 13 to 26 sec. As scale increases from 2 cores to 256 cores, YARN’s efficiency drops by 13%, while MATRIX’s drops by only 5% and maintains 93%. These results indicate that MATRIX is more scalable than YARN, due to the distributed scheduling architecture and technique that optimizes both load balancing and data locality.

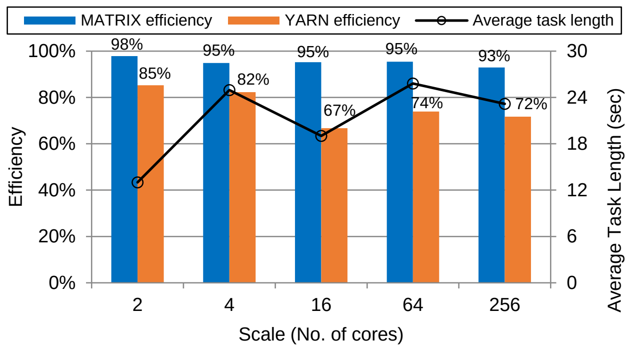


Figure 72. Efficiency for WordCount

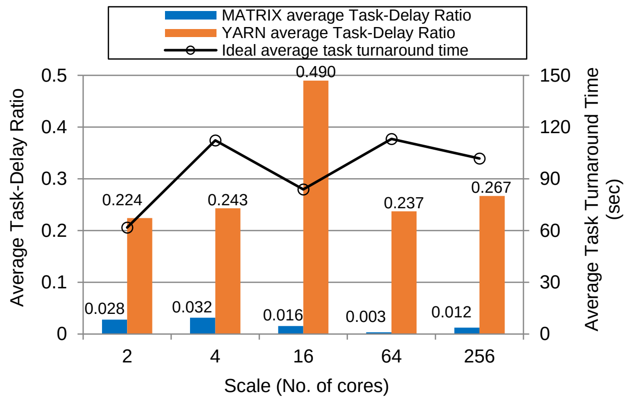


Figure 73. Average Task-Delay Ratio for WordCount

Figure 73 compares the average Task-Delay Ratio between MATRIX and YARN, and shows the ideal average task turnaround time for all the scales. MATRIX achieves performance that is quite close to the ideal case. The added overheads (quantified by the average Task-Delay Ratio) of MATRIX are much more trivial (20X less on average) than that of YARN. This is because each scheduler in MATRIX maintains task queues, and all the ready tasks are put in task ready queues as fast as possible. On the contrary, YARN applies a pull-based model that lets the free containers pull tasks from the application master, incurring significant Ping-Pong overheads and poor data-locality.

(2) TeraSort

TeraSort is another two-phase Hadoop workload that performs in-place sort of all the words of a given data file. Figure 74 and Figure 75 present the comparison results between MATRIX and YARN.

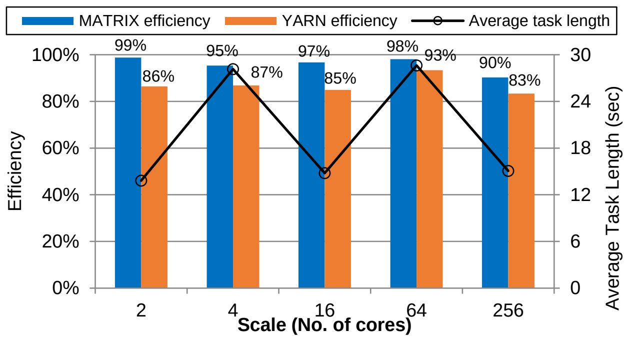


Figure 74. Efficiency for TeraSort

Figure 74 illustrates the efficiency comparisons. We see that YARN can achieve performance that is close to MATRIX, however, there is still a 10% discrepancy on average. This is because in TeraSort, the time spent in the reduce phase dominates the whole process. The final output data volume is as large as the initial input one, but the number of reduce tasks is much less than that of the map tasks (In our configurations, there are 8 reduce tasks at 256 cores, and 1 reduce task at all other scales). Therefore, load balancing is less important.

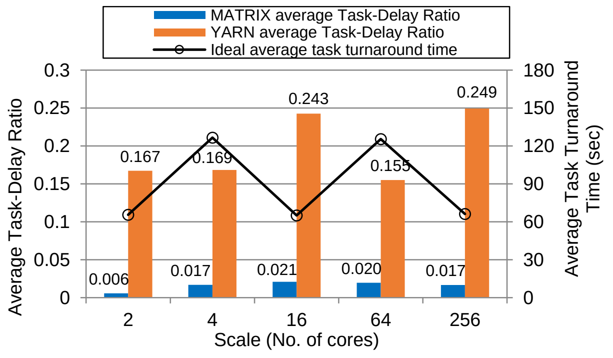


Figure 75. Average Task-Delay Ratio for TeraSort

However, in terms of the task turnaround time for each task, MATRIX can still achieve much faster responding time with way small overheads (10X less on average) than YARN, according to Figure 75.

(3) *RandomWriter*

The RandomWriter workload is consist of only map tasks, and each task writes an amount of random data to the HDFS with a summation of 10GB data per instance. Figure 76 and Figure 77 give the performance comparison results.

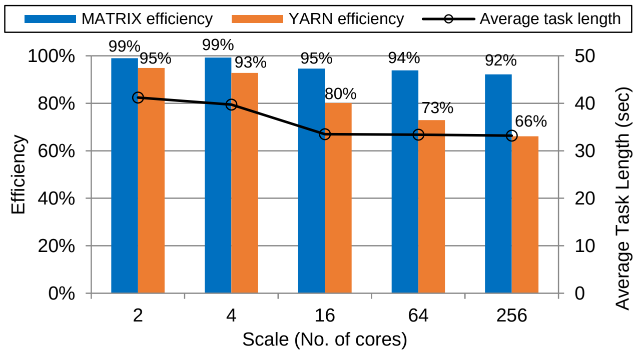


Figure 76. Efficiency for RandomWriter

Figure 76 shows that at all scales, MATRIX achieves much better performance (19.5% higher efficiency on average) than YARN. In addition, as the scale increases, YARN's efficiency drops dramatically, from 95% at 2 cores to only 66% at 256 cores. The trend indicates that at larger scales, YARN efficiency would continue to decrease. On the contrary, MATRIX has the ability to maintain high efficiency at large scales and the efficiency-decreasing rate is much slower comparing with YARN. We believe that as the scales keep increasing to extreme-scales, the performance gap between MATRIX and YARN would be getting bigger and bigger.

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

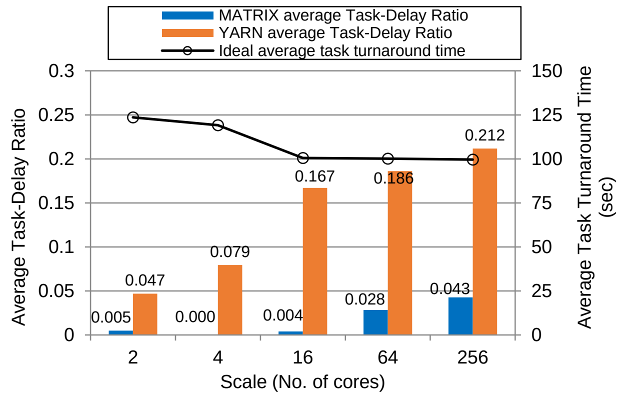


Figure 77. Average Task-Delay Ratio for RandomWriter

In terms of the average Task-Delay Ratio presented in Figure 77, again, MATRIX can response to per-task much faster than YARN, due to the pushing mechanism used in the MATRIX scheduler that eagerly pushes all the ready tasks to the task ready queues.

(4) *Grep*

The last Hadoop benchmark is the Grep workload that searches texts to match the given pattern in a data file. In YARN, the Grep workload is divided into 2 Hadoop jobs, namely search and sort. Both jobs have a two-phase MapReduce data pattern. However, MATRIX converts the entire Grep workload to one application DAG that has a four-phase data flow pattern. The output of the reduce phase of the search job is the input of the map phase of the sort job. The comparison results between MATRIX and YARN with the Grep workload are shown in Figure 78 and Figure 79.

Figure 78 shows that MATRIX performs much better than YARN at all scales with a performance gain of 53% on average (1.53X speedup). YARN achieves relatively low efficiency, even at 2-core scale. The reasons are two-folds. First, the Grep workload has a wide distribution of task lengths. Based on the targeting text pattern, map tasks may execute significantly different amounts of times and generate results ranging from empty to large volumes of data when given different parts of the data file. This huge heterogeneity of task length leads to poor load balancing in YARN. The other reason is that YARN needs to launch 2 Hadoop jobs for the Grep workload, which doubles the job launching overheads.

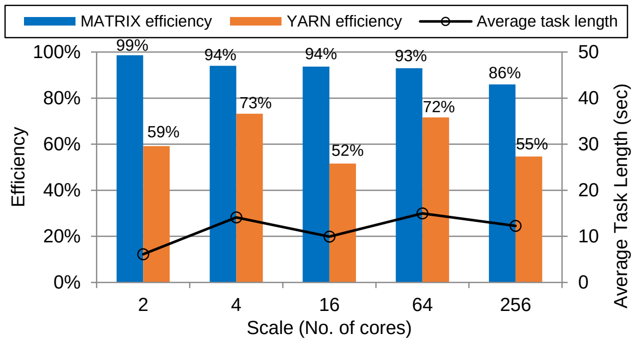


Figure 78. Efficiency for Grep

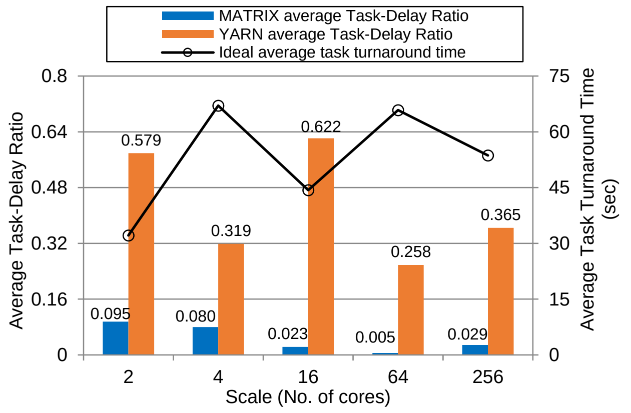


Figure 79. Average Task-Delay Ratio for Grep

However, MATRIX can optimize both load balancing and data-locality during runtime through the work stealing technique. This is preferable for heterogeneous workloads. Besides, MATRIX decomposes the workload as one DAG and launches all tasks once as fast as possible, introducing much less overheads. Figure 79 validates this justification by showing that MATRIX responses 20X faster on average than YARN.

(5) Ligand Clustering Application in Bioinformatics

The previous comparisons use typical benchmarking workloads, and MATRIX has shown better scalability than YARN for all the workloads. In this section, we show how they perform in processing a real data-intensive application in bioinformatics, namely the Ligand Clustering application [168].

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

Like the benchmarking workloads, in this application, the input data size is 256MB per instance based on the 59.7MB real application data. We apply 5 iterative Hadoop jobs, including 10 phases of map and reduce tasks. We run the application in both MATRIX and YARN, and the performance results are given in Figure 80 and Figure 81.

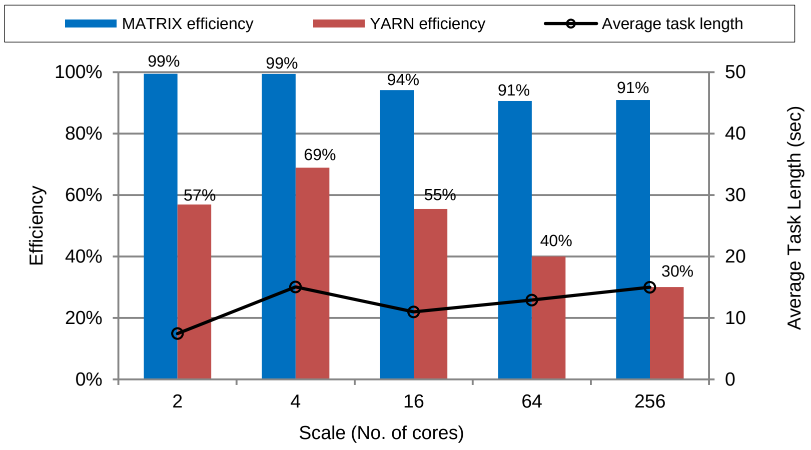


Figure 80. Efficiency for the Bioinformatics application

From Figure 80, we see that as the scale increases, the efficiency of YARN is decreasing significantly. At 256-core scale, YARN can only achieve 30% efficiency, which is one third of that achieved (91%) through MATRIX. The decreasing trend is likely to hold towards extreme-scales for YARN. On the other hand, the efficiency of MATRIX has a much slower decreasing trend and is becoming stable at 64-core scale. Even though it sounds too rush to conclude that MATRIX has the ability to maintain efficiency as high as 90% at extreme-scales, these results show the potential extreme scalability of MATRIX, which we will explore through simulations later.

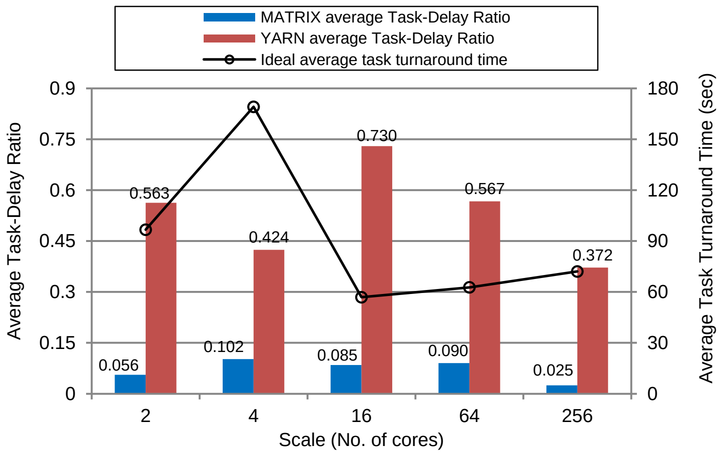


Figure 81. Average Task-Delay Ratio for the application

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

Like the Grep workload, YARN launches the 5 iterative Hadoop jobs of this application individually one by one, incurring tremendously large amount of launching and Ping-Pong overheads, whilst MATRIX launches the whole application DAG once. This difference is another factor that causes the speed gap between YARN and MATRIX in responding to per-task shown in Figure 81 (MATRIX achieves 9X faster than YARN on average).

(6) Fine-grained Data Processing Workloads

We have shown that MATRIX is more scalable than YARN in processing both typical benchmarking workloads and a real application in Bioinformatics. All the workloads evaluated so far are relatively coarse-grained (e.g. average task length of tens of seconds), comparing with the Berkeley Spark MapReduce stack [160] that targets finer-grained workloads (e.g. average task length of hundreds of milliseconds). We have identified that YARN is not well suited for running fine-grained workloads, due to the pull-based task scheduling mechanism and the HDFS centralized metadata management. To validate this, we run a fine-grained workload test of YARN at 64 cores for the Terasort workload by reducing the block size of HDFS to 256KB, 100X smaller than the previous course-grained experiments. The average task length should be only 1/100 of that of the course-grained workloads (286ms according to Figure 74). However, YARN's logs show that the average task execution time decreases only by half with about 14 sec, leading to the efficiency as low as 2.04% (286/14000).

On the contrary, MATRIX is designed to process the fine-grained sub-second workloads. In Chapter 3, MATRIX showed the ability to achieve 85%+ efficiency for the workload with an average task length of 64ms at 4K-core scales on an IBM BG/P supercomputer. In addition, we have compared MATRIX with the Spark sparrow scheduler with NOOP sleep 0 tasks and MATRIX was 9X faster than sparrow in executing the tasks.

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

We run MATRIX with this workload up to 256 cores. Besides, we conduct simulations of MATRIX for this workload at 64K-core scale, through the SimMatrix simulator. We feed the fine-grained workloads in SimMatrix. Beyond 256 cores, we increase the number of tasks linearly with respect to the system scales by repeating the workloads at 256 cores.

We show the efficiency results of MATRIX running both the fine-grained and coarse-grained workloads, as well as SimMatrix running the fine-grained workloads in Figure 82. When the granularity increases by 100X, the efficiency only drops about 1.5%

on average up to 256 cores (blue line vs. the solid red line). This shows great scalability of MATRIX in processing the fine-grained workloads. From the simulation's perspective, we run SimMatrix up to 64K cores and validate SimMatrix against MATRIX within 256-core scales. The normalized difference (black line) between SimMatrix and MATRIX within 256 cores is only 4.4% on average, which shows that SimMatrix is accurate and the simulation results are convincible. At the 64K cores, the efficiency maintains 86.8% for the workload of average task length of 150ms. According to the efficiency trend, we can predict that at the 1M-core scale, the efficiency will be 85.4% for this fine-grained workload.

These results show that MATRIX has the potential to enable Hadoop to scale to extreme scales, even for the fine-grained sub-second workloads.

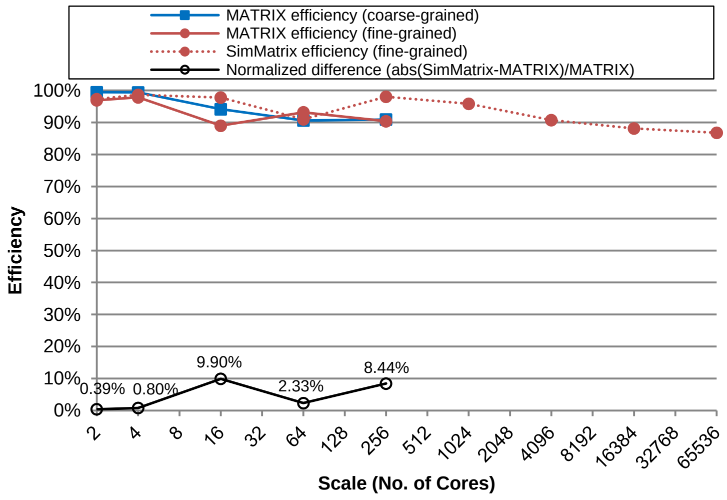


Figure 82. MATRIX for sub-second Bioinformatics workloads

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

We compared MATRIX with YARN using typical Hadoop workloads and the application in Bioinformatics up to 256 cores on Amazon. Table 10 summarizes the average performance results of both MATRIX and YARN for different workloads for all scales. On average of all the Hadoop workloads, MATRIX outperforms YARN by 1.27X. For the application in Bioinformatics (BioApp), MATRIX outperforms YARN by 2.04X.

Table 10. Efficiency results summary of MATRIX and YARN

Workloads	***Average Task Length***	***MATRIX Efficiency***	***YARN Efficiency***	***Average Speedup (MATRX/YARN)***
WordCount	21.19sec	95.2%	76.0%	1.26
TeraSort	20.09sec	95.9%	87.0%	1.10
RandomWriter	36.20sec	95.8%	81.4%	1.20
Grep	11.48sec	93.0%	62.0%	1.53
BioApp	12.26sec	94.9%	50.2%	2.04
Fine-grained BioApp	128.5ms	93.5%	N/A	N/A

We also explored the scalability of MATRIX through both real system and simulations at extreme scales for fine-grained sub-second workloads. The simulations indicate 86.8% efficiency at 64K-core scale for 150ms workloads. We predict that MATRIX has the potential to enable MapReduce to scale to extreme scale distributed systems.

5.5 Conclusions and Impact

Data-intensive MTC applications for exascale computing need scalable distributed task scheduling system to deliver high performance, posing urgent demands on both load balancing and data-aware scheduling. This work combined distributed load balancing with data-aware scheduling through a data-aware work stealing technique. We implemented the technique in both the MATRIX task execution framework and the SimMatrix simulator. We devised an analytic model to analyze the performance upper bound of the proposed techniques. We evaluated our technique under four different scheduling policies with different workloads, and compared our technique with the data diffusion approach. We also explored the scalability and performance quality of the proposed data-aware work stealing technique up to 128K cores. Results showed that our technique is scalable to achieve both good load balancing and high data-locality. Simulation results showed that the technique is not only scalable, but is able to perform within 15% of the sub-optimal solution, towards extreme scales. We further compared MATRIX with YARN in processing the typical Hadoop workloads, and MATRIX showed significant performance gains and promising potential to overcome the scalability limitations of the Hadoop framework.

With the advent of the big data era, applications run on both supercomputers and Internet domains are experiencing data explosion, involving archiving, processing and moving ever-growing amount of data volumes. The United States government has recently begun its multi-agency Big Data Initiative to address various big data issues [186]. We anticipate that the research contributions of this work will have profound influence in addressing the big data challenges through developing and deploying highly scalable and fault tolerant, and high performance big data infrastructures and scheduling frameworks.

The proposed scheduling architecture and technique are innovative, due to the distributed nature and the integration with scalable key-value store for state management. This work will open doors to many research topics that are related to big data computing, including complicated parallel machine learning models [183] that extract important information from big data volumes, data compression mechanisms [184] that can save storage spaces with a high compression ratio, erasure coding methods [20] that can replace the N-way replication one to maintain both fault tolerance and high storage utilization, data-locality aware scheduling for data-intensive high-performance computing applications with the aid of the burst buffer [187] intermediate storage layer, in situ data processing frameworks [185] for efficient scientific visualization, and many others. We believe that this work will be beneficial to many scheduling platforms by helping them address their scalability challenges in scheduling big data applications, such as the traditional data arrogant HPC resource management systems, the Hadoop framework, the Spark big data computing software stack, the Hive data warehouse [188].

CHAPTER 6

SLURM++ WORKLOAD MANAGER

As supercomputers are growing exponentially, one way of efficiently utilizing the machines is to support a mixture of applications in various domains, such as traditional large-scale HPC, the HPC ensemble runs, and the fine-grained many-task computing (MTC). Delivering high performance in resource allocation, scheduling and launching for all types of jobs has driven us to develop Slurm++, a distributed workload manager directly extended from the Slurm centralized production system. Slurm++ employs multiple controllers with each one managing a partition of compute nodes and participating in resource allocation through resource balancing techniques. A distributed key-value store is integrated to manage the system states in a scalable way. To achieve dynamic resource balancing within all the partitions, we propose two resource stealing techniques, namely a random one and an improved weakly consistent monitoring-based one that aims to address the resource deadlock problem occurring in distributed HPC job launch for big jobs and under high system utilization. We implement the techniques in Slurm++ and enable Slurm++ to run real MPI applications by preserving the entire Slurm communication layer. We compare Slurm++ with Slurm using micro-benchmark workloads, workloads from application traces, and real MPI applications. Slurm++ shows 10X faster than Slurm in allocating resources and launching jobs, and the performance gap is expected to grow as the jobs and system scales increase. We also evaluate Slurm++ through simulations showing promising performance results towards extreme scales.

6.1 Distributed HPC Scheduling

Predictions are that supercomputers will reach exascale with up to a billion threads of execution in less than a decade [2]. Exascale computing will enable the unraveling of significant mysteries for a diversity of scientific applications, such as Astronomy, Biology, Chemistry, Earth Systems, Neuroscience and Physics [2][39]. Given this extreme magnitude of concurrency, one way of efficiently utilizing the whole machine without requiring full-scale jobs is to categorize the applications into different domains and to support the running of a mixture of applications in all these domains. The application domains include traditional large-scale high performance computing (HPC), HPC ensemble runs, and fine-grained loosely coupled many-task computing (MTC).

Scientific applications in the traditional HPC domain typically require many computing processors (e.g. half or full-size of the whole machine) for a long time (e.g. days or weeks) to achieve the experimental and simulation goals. Examples of these applications are exploring the structures of biological gene sequences, chemical molecules, and human brains; and the simulations of the origin of the universe, spacecraft aerodynamics, and nuclear fusion [141]. The jobs are tightly coupled, and use the message-passing interface (MPI) programming model [64] to communicate and synchronize among the processors.

Although it is necessary to support HPC applications that demand the computing capacity of an exascale machine, only a few applications can scale to exascale. Most will be decomposed as ensemble runs of applications that have uncertainty in high-dimension parameter space. Ensemble runs [142] decompose applications into many small-scale and short-duration coordinated jobs with each one doing a parameter sweep in a much lower-resolution parameter space using MPI in parallel, thus enabling a higher system utilization.

These high-order low-order methods can be used either in an integrated fashion, in a single job, or in ensemble to reduce the uncertainty of the high-resolution method. In [143], ensemble runs have been defined for exascale simulations: “One approach to dealing with uncertainty is to perform multiple ensemble runs (parameter sweeps) with various combinations of the uncertain parameters”. For example, to address the uncertainty in weather forecast, each of the parameters, such as the current weather, the atmosphere condition, the human interaction, can be simulated as an ensemble run to examine the effects of the parameter on the weather. Another example is scheduling regression tests used by many applications to validate changes in the code base. These tests are small-scale and scheduled on a daily basis.

Furthermore, the high intra-node parallelism at exascale requires supporting embarrassingly parallel fine-grained many-task computing (MTC) [39] [40] workloads to increase utilization with efficient backfilling strategies. In MTC, applications are decomposed as orders of magnitude larger number (e.g. millions to billions) of fine-grained tasks in both size and duration, with data-dependencies. Workloads are represented as DAG. The tasks do not require strict coordination of processes at job launch, as do the traditional HPC workloads. The algorithm paradigms well suited for MTC are Optimization, Data Analysis, Monte Carlo and Un-certainty Quantification. Applications that demonstrate the MTC characteristics range from astronomy, bioinformatics, climate modeling, chemistry, economics, medical imaging, neuroscience, pharmaceuticals, to physics [53].

To better support our ascertain that tomorrow’s exascale supercomputers will likely be used to run all categories of the scientific applications of various job sizes, we analyze the published trace data about the application jobs that were run on the IBM Blue Gene / P supercomputer in Argonne National Laboratory during an 8-month period (from Jan. 2009 to Sept. 2009) [145]. The machine is comprised of 40960 homogeneous nodes with each one having 2 CPU cores. The trace includes 68,936 jobs with a wide range of job sizes and durations. We draw a CDF graph of the number of jobs with respect to the job size (represented as the percentage to the system scale of 40960 nodes) for all the jobs in Figure 83, in which, the changing points are also explicitly displayed.

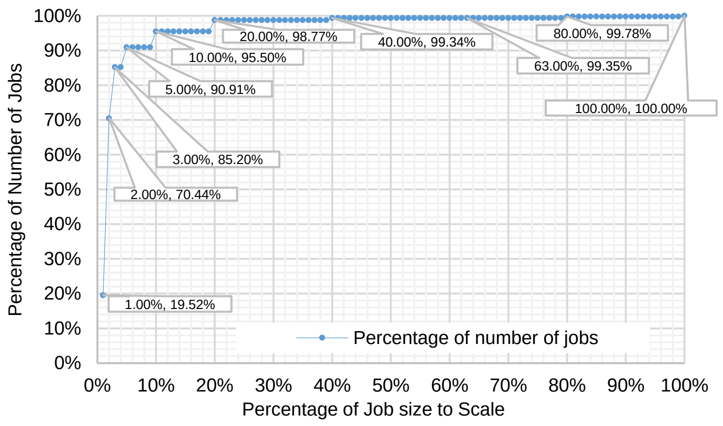


Figure 83. CDF graph of the number of jobs with respect to the job size

We observe that 19.52% jobs have job sizes no bigger than 1% of the system scale, 90.01% jobs have job sizes no bigger than 5% of the system scale, and 99.78% jobs have job sizes no bigger than 80% of the system scale. This distribution indicates that in the workload, most of the jobs are small, with small amount of big jobs, and few full-scale jobs (152 out of 68,936). We believe that for exascale supercomputers, the distributions of the job sizes will likely be the same as those of the BG/P machine. In addition, we expect that more policies will be proposed to ensure much fairer resource sharing in the HPC research community, which will lead to orders of magnitudes more number of jobs and an increasing share of the small jobs.

Running a mixture of applications in all domains on an exascale machine is considerably different from running merely traditional HPC workloads, and poses significant scalability challenges on resource management systems (or workload managers), especially the resource allocation scheduler and the state manage-ment component. The majority of the state-of-the-art workload managers (e.g. Slurm [30], PBS [84], Condor [80] [83], Cobalt [114] and SGE [85]) for large-scale HPC machines have a centralized architecture where a controller manages all the compute daemons and is in charge of the activities, such as node partitioning, state management, resource allocation, job scheduling, and job launching. For example, the centralized controller of Slurm is comprised of three subcomponents: the Node Manager monitors the state of nodes; the Partition Man-ager allocates resources to jobs for scheduling and launching; and the Job Manager manages the job state. This architecture may still work at moderate scales for large-scale HPC workloads, given the facts that, at to-day's scales, the number of jobs is small and each job is large and has long duration. Therefore, there are not many scheduling decisions to be made, nor is there much system state data to be maintained.

However, the exascale machines will have system sizes one or two orders of magnitude larger, and be used to run many more numbers of jobs with wider distributions of job sizes and durations. These tremendous growths in both the job amount and granularity will easily overwhelm the centralized controller. The throughput metric makes sense, as so many small-scale short-duration jobs make the scheduler a bottleneck. The Slurm production-level workload manager claimed a maximum throughput of 500 jobs/sec [79]; the latest version (13.0) of the PBS Professional workload manager reported throughput of 100 jobs/sec on 100K nodes [153]. However, future exascale machines, along with a miscellaneous collection of applications, will demand orders of magnitudes higher job-delivering rates to make full utilization of the machine. The scalability problem is growing even though the upcoming systems have relatively small numbers of nodes, such as Sierra at LLNL(5K-nodes with GPUs) [191] and Summit at ORNL(3.4K-nodes) [192], because the intra-node parallelism is increased by 100X to 1000X requiring much finer re-source allocation units than those of the current super-computers. As exascale machines will likely not follow IBM’s “many-thin-node” pattern, a minimum scheduled resource allocation of 64 nodes is perhaps not a good future choice due to the large intra-node parallelism. The allocation unit should be fine-grained, down to a node, a NUMA domain, a core, even a hardware thread, requiring distributed resource allocation de-signs. This is also where a distributed scheduling sys-tem makes sense. In addition, the centralized system state management will become a bottleneck for fast queries and updates of the job and resource metadata, due to limited processing capacity and memory footprint on a single node. These challenges drive us to design and implement Slurm++, the next generation distributed workload manager that can maintain high job-scheduling rate and system state accessing speed.

In this chapter, we propose a distributed partition-based architecture for the HPC resource management [234]. We devise two resource stealing techniques to achieve dynamic resource balancing within all the partitions, namely a random one and an improved weakly consistent monitoring-based one that aims to address the resource deadlock problem occurring in distributed HPC job launch for big jobs and under high system utilization. We develop the Slurm++ workload manager that implements the architecture and techniques. Slurm++ extends Slurm by applying multiple controllers to allocate job resources, schedule and launch jobs. Each controller manages a partition of compute daemons and balances resources among all the partitions through resource stealing techniques when allocating job resources. Slurm++ utilizes ZHT [51], a distributed key-value store (KVS) [27], to keep the resource state information in a scalable way. *This chapter makes the following contributions:*

- Propose a distributed architecture for exascale HPC resource management.
- Devise two resource stealing techniques to achieve dynamic resource balancing, namely a random one and a weakly consistent monitoring-based one.
- Develop both Slurm++ real resource management system and the SimSlurm++ simulator that implement the distributed architecture and techniques.
- Evaluate Slurm++ by comparing with Slurm up to 500-nodes with micro-benchmarks, workloads from real application traces, as well as real MPI applications, and through SimSlurm++ up to 1 million nodes.

6.2 Distributed Architecture for Exascale HPC Resource Management

6.2.1 Partition-based Distributed Architecture. As we have motivated that a centralized workload manager does not scale to extreme-scales for all the applications, we propose a partition-based distributed architecture of the Slurm++ workload manager, as shown in Figure 84.

Slurm++ employs multiple controllers with each one managing a partition of compute daemons (cd), as opposed to using a single controller to manage all the cd as Slurm does. Here, a cd is a process running on a physical node that can have multiple concurrent cd. By default, each node is configured with one cd. The partition size (the number of cd a controller manages) is configurable, and can vary according to needs. For example, for large-scale HPC jobs that require many nodes, we can configure each controller to manage thousands of cd, so that the jobs are likely to be satisfied within the partition; for MTC applications where a task usually requires one node, or one core of a node, we may have a 1:1 mapping between the controller and the cd. We can even have heterogeneous partition sizes to support workloads with wide job-size distributions, and with special requirements, such as only run on the partitions that have GPUs, InfiniBand, or SSDs. The users can submit jobs to any controller.

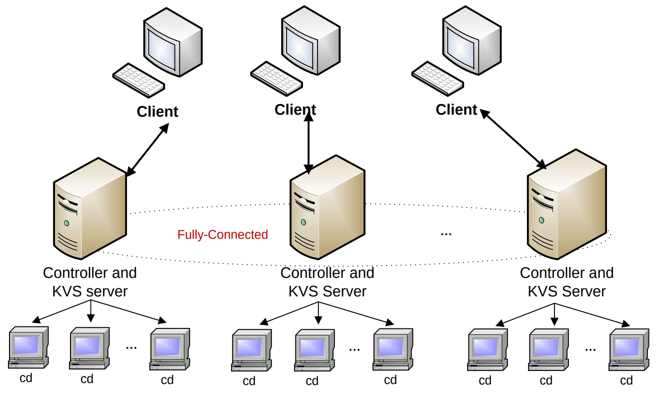

Client
Client
Client
Fully-Connected
...
Controller and KVS server
Controller and KVS Server
Controller and KVS Server
cd
cd
...
cd
cd
cd
...
cd
cd
cd
...
cd

Figure 84. Slurm++ distributed workload manager

Note that Slurm also divides the system into multiple partitions. However, this is different from the partition management of Slurm++. Firstly, Slurm layers the partitions hierarchically up to the centralized controller, while Slurm++ is distributed by employing one dedicated controller to manage a partition independently from other partitions. The hierarchical layout leads to longer latency as a job may need to go through multiple hops. What’s more, the root controller is still a central piece with limited capacity. Secondly, in Slurm, when a job is scheduled on one partition, it can only get allocation within that partition. This results in long job queueing time in over-loaded partitions, and poor utilization if loads are not balanced among the partitions.

6.2.2 Key-Value Store for State Management. Another important component of Slurm++ is the distributed key-value store (KVS), the ZHT zero hop distributed KVS in our case. Slurm++ deploys ZHT on the machine to manage the entire resource metadata and system state information in a scalable way. The job and resource metadata are archived as (*key*, *value*) pairs that are stored in ZHT servers by hashing the *key* on all the ZHT servers through a modular function that enables load balancing. ZHT is fully connected, and each ZHT client has a global membership list of all the ZHT servers. By hashing the *key*, a ZHT client can know and talk directly (zero-hop) to the exact server that manages the data an operation (e.g. *lookup*, *insert*) is working on. ZHT is scalable, and has been evaluated on the IBM BG/P supercomputer in ANL up to 32K cores. ZHT showed extreme throughput as high as 18M operations/sec [51]. One typical configuration is to co-locate a ZHT server with a controller forming 1:1 mapping, such as shown in Figure 84. Each controller is initialized as a ZHT client, and uses the simple client APIs (e.g. *lookup, insert*) to communicate with ZHT servers to query and modify the job and resource metadata information (some examples are listed in Table 11) transparently. In this way, the controllers do not need to communicate explicitly with each other.

Table 11. 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

In addition, Slurm++ is fault tolerant by relying on ZHT, which already implemented failure/recovery, replication and consistency models, to maintain reliable distributed service. As the controllers are stateless, when a controller fails, the resource is still available to others through the replication and resource stealing protocols, no reboot is required. The controllers only keep weakly consistent temporary data comparing with that stored in ZHT. This eases the development and deployment.

6.3 Dynamic Resource Balancing

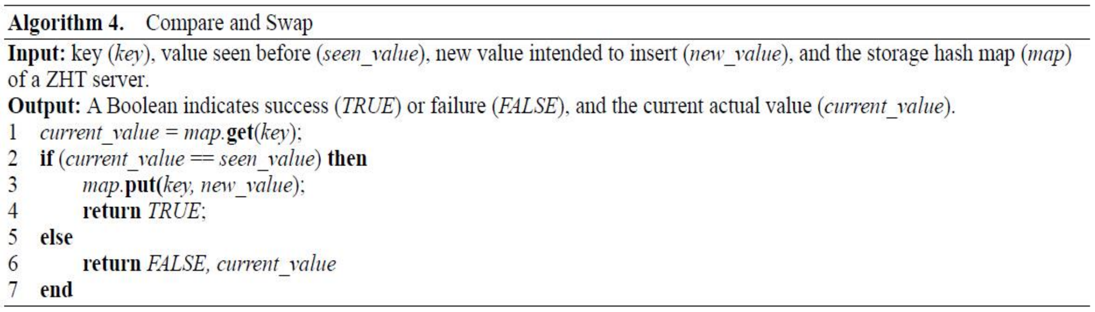
Resource balancing means to find the required number of free nodes in all the partitions fast to satisfy a job. It is trivial in the centralized architecture, as the controller has a global view of the system state. However, for the distributed architecture, resource balancing is a critical goal, and should be achieved dynamically in a distributed fashion by all the controllers during runtime, in order to maintain an overall high system utilization.

Inspired by the work stealing technique [52] that achieves distributed dynamic load balancing, we introduce the *resource stealing* concept [235] to achieve distributed dynamic resource balancing. Resource stealing refers to a set of techniques of stealing free nodes from other partitions if the local one cannot satisfy a job in terms of job size. When a controller allocates nodes for a job, it first checks the local free nodes. If there are enough free nodes, then the controller directly allocates the nodes; otherwise, it allocates whatever resources the partition has, and queries ZHT for other partitions (neighbors) to steal resources. For systems with heterogeneous interconnections that have different latencies among compute nodes, we can improve resource stealing by distinguishing the “near-by” and “distant” neighbors. The technique always tries to steal resources from the “nearby” partitions first, and will turn to the “distant” ones if necessary. For ex-ample, in a Torus network, we can set an upper bound of number of hops between two controllers. If the hop count is within the upper bound, the two partitions are considered “nearby”; otherwise, they are “distant”. In a fat-tree network, “nearby” partitions can be the sibling controllers with the same parent.

6.3.1 Resource Conflict. Resource conflict happens when different controllers try to modify the same resource. By querying ZHT, different controllers may have the same view of a specific resource of a partition. They may need to modify the resource concurrently based on the current view, either to shrink by allocating nodes, or to expand by releasing nodes. As the controllers have no knowledge about each other’s modification, the resource conflict would happen.

One naive way to solve the resource conflict problem is to add a global lock for each queried (*key*, *value*) record in ZHT. This approach is not scalable considering the tremendously large volume of data record stored. Another scalable approach is to implement an atomic operation in ZHT that can tell the controllers whether the resource modification succeeds or not. Using the traditional compare and swap atomic instruction [144], we implement a specific compare and swap algorithm, shown as Algorithm 4 in Figure 85, which can address the resource conflict problem.

When a controller attempts to modify the resource of a partition, it first looks up the current resource (*see_value*) of that partition. Then, it updates the resource to an attempted new resource value (*new_value*), and sends a *compare and swap* request to ZHT. The ZHT server executes the Algorithm 4 by first checking whether the current resource value has been changed (lines 1 – 2). If hasn’t, then the serve updates the resource value to the new value and returns success to the controller (lines 3 – 4); otherwise, the server would return failure, along with the current resource value (*curent_value*) to the client (line 6). The client then uses the *curent_value* as the *seen_value* in the next round. This procedure continues repeatedly until the modification succeeded. We have implemented the *compare and swap* operation in ZHT. ZHT applies the "epoll" event driven model to process requests sequentially in each server. The atomic operation, along with the serialization of the requests, guarantees that at any time, there is only one controller modifying the resource of a partition.



Algorithm 4. Compare and Swap

Input: key (*key*), value seen before (*seen_value*), new value intended to insert (*new_value*), and the storage hash map (*map*) of a ZHT server.

Output: A Boolean indicates success (*TRUE*) or failure (*FALSE*), and the current actual value (*current_value*).

```
1  current_value = map.get(key);
2  if (current_value == seen_value) then
3        map.put(key, new_value);
4        return TRUE;
5  else
6        return FALSE, current_value
7  end
```

Figure 85. Algorithm of compare and swap

6.3.2 Random Resource Stealing Technique. The simplest resource stealing technique does random stealing, which is given as Algorithm 5 in Figure 86.

As long as a job has not been allocated enough nodes, the controller randomly selects a partition and tries to steal free nodes from it (lines 4 – 16). Every time when the selected controller has no available nodes, the launching controller sleeps some time (*sleep_length*) and retries. We implemented an exponential back-off technique for the *sleep_length* to increase the chance of success (line 25). If the controller experiences several (*num_retry*) failures 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 problem (lines 20 – 23), which happens when two controllers hold part of the resource for each job, but neither can be satisfied.

Algorithm 5. Random Resource Stealing

Input: number of nodes required (*num_node_req*), number of nodes allocated (**num_node_alloc*), number of controllers/partitions (*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*[]).

```
1  num_try = 0; num_ctl_inv = 0;
2  default_sleep_length = sleep_length;
3  while *num_node_alloc < num_node_req do
4        sel_ctl_idx = ctl_id[Random(num_ctl)];
5        sel_ctl_node = zht_lookup(sel_ctl_id);
6        if (sel_ctl_node != NULL) then
7              num_more_node = num_node_req – *num_node_alloc;
8  again:      num_free_node = sel_ctl_node.num_free_node;
9              if (num_free_node > 0) then
10                   num_try = 0; sleep_length = default_sleep_length;
11                   num_node_try = num_free_node > num_more_node ? num_more_node : num_free_node;
12                   list_node = allocate(sel_ctl_node, num_node_try);
13                   if (list_node != NULL) then
14                         *num_node_alloc += num_node_try;
15                         part_node[num_ctl_inv++] = list_node;
16                         list_ctl_id_inv.add(remote_ctl_id);
17                   else
18                         goto again;
19                   end
20             else if (++num_try > num_retry) do
21                   release(list_ctl_id_inv, part_node);
22                   *num_node_alloc= 0;
23                   sleep_length = default_sleep_length;
24             else
25                   sleep(sleep_length); sleep_length *= 2;
26             end
27       end
28 end
29 return list_ctl_id_inv, part_node;
```

Figure 86. Algorithm of random resource stealing

6.3.2.2 Limitation Analysis through Simulation

We implemented the random resource stealing technique in Slurm++ v0. The technique works fine for jobs whose sizes are small. Besides, the free resources could be balanced during runtime due to the distributed and random features. However, for big jobs, it may take forever to allocate enough nodes. Another case the random algorithm will have poor performance is under high system utilization where the system has few free nodes.

To illustrate the scalability issues of the random technique, we consider a simplified case in which the system has only one job. We range the job size to be 25%, 50%, 75%

and 100% of the system scale, meaning that if the system has *n* nodes, the job size is 0.25*n*, 0.5*n*, 0.75*n*, and *n*, respectively. We also range the utilization to be 0%, 25%, 50%, and 75%, meaning that 0%, 25%, 50%, 75% of all the nodes are occupied, respectively. We simulate the random stealing scenario and count the number of stealing operations needed to satisfy a job for different combinations of job size and utilization. The results are shown in Figure 87.

The notation (j,u) means the job size ratio and the utilization. We set the partition size to be 1024 and range the number of partitions from 1 to 1024. We see that the number of stealing operations is increasing linearly as the system scale increases. This does not mean the random technique is scalable, as there is only one job. We also observe that the stealing operation count is increasing exponential with both the job size and utilization. This justifies the scalability problems of the random technique for big jobs and under high system utilization. At 1M-node scale, a full-scale job needs about 8000 stealing operations to satisfy the allocation. Things could be much worse when there are multiple big jobs competing for resources.

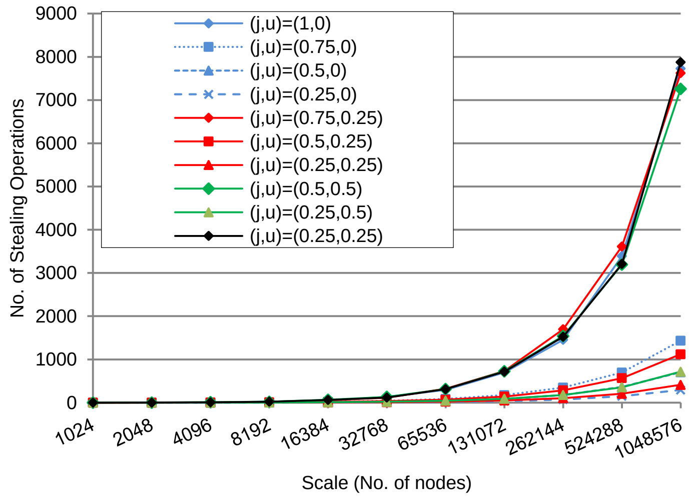


Figure 87. Stealing operation count for different job sizes and utilizations

6.3.2.3 Limitation Analysis through Analytical Model

To understand the random technique in depth, we devise a mathematical model that analyzes the probability of satisfying the allocations for one, two, and more full-scale jobs with distinct large amounts of stealing operations.

Assume that the job needs k steps of stealing operations to satisfy only one full-scale job, we show that the possibility of taking λk steps to satisfy λ full-scale jobs concurrently is negligible. For the case of 2 full-scale jobs competing to get a full allocation individually, p_k represents the possibility that one job has got full allocation after k steps. Without losing generality, we assume that job 1 gets the full-allocation after k steps. This means job 2 must fail at each step and at each step, job 2 would try to steal from the partitions whose nodes have been stolen by job 1. Let n be the number of all nodes, and after step i, job 1 has allocated c_i compute nodes, in which, $1 \leq i \leq k$ and $1 \leq c_i < n$, $c_i \leq c_{i+1}$, $c_k = n$. As in random resource stealing, all of the compute nodes have the same probability to be allocated, therefore, the probability that job 1 gets the full-allocation after k steps and job 2 failed at each step, notated as $p_k = \frac{\prod_{i=1}^{k} c_i}{n^k}$.

Assuming after m steps, job 1 starts with an allocation of half of all compute nodes, meaning that: $\frac{n}{2} \leq c_i < n$ $(m \leq i < k)$, and $1 \leq c_i < \frac{n}{2}$ $(1 \leq i < m)$. Therefore:

$$p_k \cong \frac{\prod_{i=1}^{m} c_i}{n^m} \cong \frac{\left(\frac{n}{2}\right)^m}{n^m} = \frac{1}{2^m}.$$

As our simulations show that the relationship between m and k is $m \cong \frac{k}{10}$, at large scales when the number of partition and the number of all compute nodes n are big, k is large (for example, at 1M nodes' scale with 1024 partitions, $k \cong 8000$). Therefore, m is large, indicating that $p_k \cong 0$. This means it is almost impossible to satisfy one full-scale job after k steps if two full-scale jobs compete for resources. Therefore, there is little chance that the two jobs can be satisfied after $2k$ steps.

Similarly, we can prove that if there are λ full-scale jobs competing for resources, there is little chance that they can be satisfied after λk steps. Therefore, it is better to allocate resources sequentially that can satisfy λ full-scale jobs after λk steps.

6.3.3 Weakly Consistent Monitoring-based Resource Stealing Technique. This section proposes a weakly consistent monitoring-based resource stealing technique that has the potential to resolve the problems of random stealing. The proposed technique relies on a centralized monitoring service and each controller conducts two phases of tuning: macro-tuning phase, and micro-tuning phase.

6.3.3.1 Monitoring Service

One of the reasons that the random technique is not scalable is because the controllers have no global view (even weakly consistent) of the system state. One alternative to enable all of the controllers to have global view is to alter the partition-based architecture so that the controllers know all the compute daemons. Then, there will be merely one (*key*, *value*) record of the global resource stored in a specific ZHT server. This method of strongly consistent global view is not scalable because a single ZHT server that stores the global resource (*key*, *value*) pair processes all the frequent KVS operations on the resources. Hence, we apply a monitoring service (MS) to query the free resources of all the partitions periodically described as Algorithm 6 in Figure 88.

In each round, the MS looks up the free resource of each partition in sequence (lines 4 – 10), and then gathers them together as global resource information and put the global information as one (*key*, *value*) record in a ZHT server (line 12). This (*key*, *value*) record offers a global view of resource states for all the controllers. This is different from the alternative mentioned above in that the frequency of querying this global (*key*, *value*) record is much less. Though the MS is centralized and queries all the partitions, we believe that it should not be a bottleneck. Because the number of partitions for large-scale HPC applications is not that many (i.e. 1K), and with the right granularity of frequency (*sleep_length*) of updating and gathering the global resource information, the MS should be scalable. The MS could be implemented either as a standalone process on one compute node or as a separate thread in a controller. In Slurm++, the MS is implemented as the latter case.

```
Algorithm 6.    Monitoring Service Logic
Input: number of controllers or partitions (num_ctl), controller membership list (ctl_id[num_ctl]), update frequency
(sleep_length).
Output: void.
1   global_key = "global resource specification"; global_value = "";
2   while TRUE do
3         global_value = "";
4         for each i in 0 to num_ctl – 1; do
5               cur_ctl_res = zht_lookup(ctl_id[i]);
6               if (cur_ctl_res == NULL) then
7                     exit(1);
8               else
9                     global_value += cur_ctl_res;
10              end
11        end
12        zht_insert(global_key, global_value); sleep(sleep_length);
13  end
```

Figure 88. Algorithm of the monitoring service

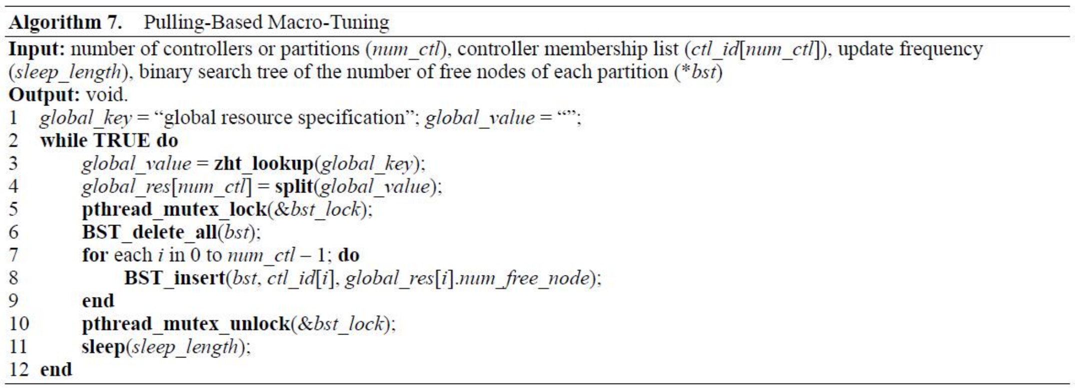
6.3.3.2 Two-Phase *Tuning*

Each controller would conduct a two-phase tuning procedure of updating resources in the aid of allocating resources to jobs.

(1) Phase 1: Pulling-based Macro-Tuning

As the MS offers a global view of the system free nodes, each controller will periodically pull the global resource information by a ZHT *lookup* operation. In each round when the controller gets the global resources, it organizes the resources of different partitions as a binary-search tree (BST). Each data item of the BST contains the controller id (char*) and the number of free nodes (int) of a partition. The data items are organized as a BST based on the number of free nodes of all partitions. The BST guides a controller to steal resources from the most suitable partitions. The logic of is given as Algorithm 7 in Figure 89.

We call this phase macro-tuning as it evicts the cached free resource information of all partitions in BST (line 6), and updates the BST with the new information (lines 7 - 9). This update is globally consistent for all the controllers as the resource information is pulled from a single place (the record is inserted by the MS) by all the controllers (line 3). Each controller pulls the global resource before it is too obsolete to offer valuable information.



Algorithm 7. Pulling-Based Macro-Tuning

Input: number of controllers or partitions (*num_ctl*), controller membership list (*ctl_id*[*num_ctl*]), update frequency (*sleep_length*), binary search tree of the number of free nodes of each partition (**bst*)

Output: void.

```
1  global_key = "global resource specification"; global_value = "";
2  while TRUE do
3      global_value = zht_lookup(global_key);
4      global_res[num_ctl] = split(global_value);
5      pthread_mutex_lock(&bst_lock);
6      BST_delete_all(bst);
7      for each i in 0 to num_ctl – 1; do
8          BST_insert(bst, ctl_id[i], global_res[i].num_free_node);
9      end
10     pthread_mutex_unlock(&bst_lock);
11     sleep(sleep_length);
12 end
```

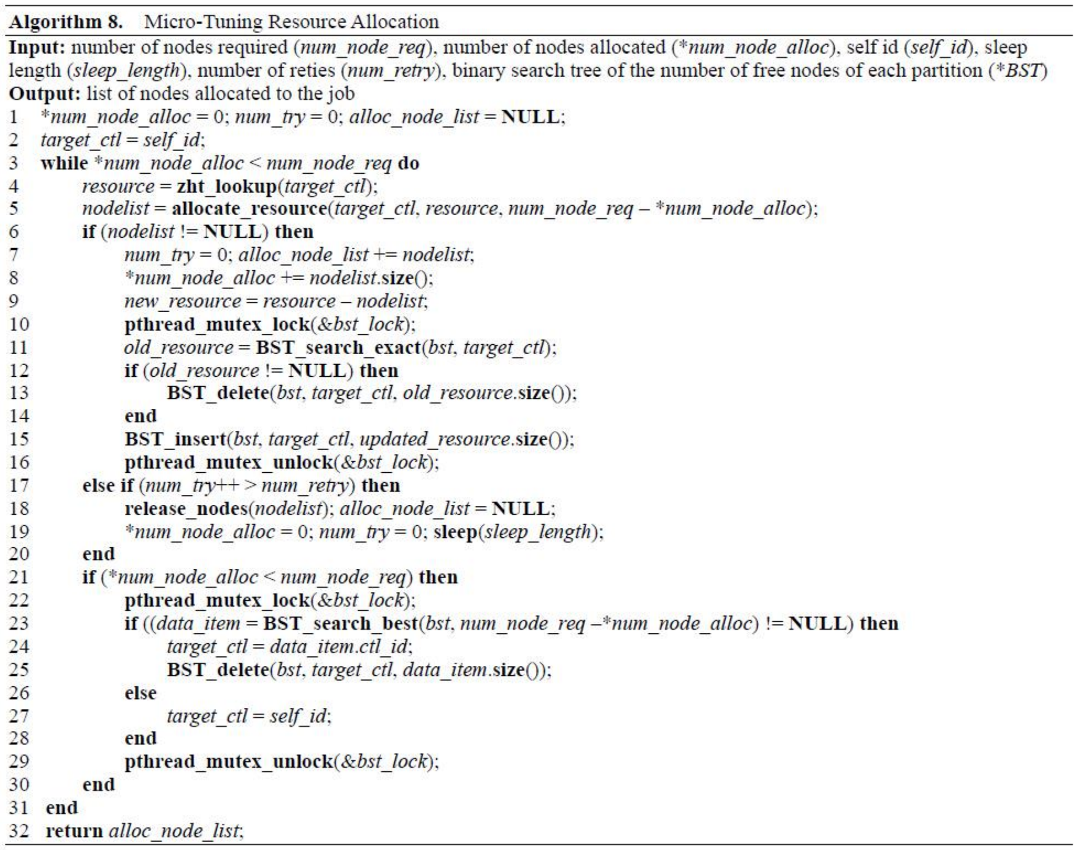
Figure 89. Algorithm of the pulling-based macro-tuning

(2) Phase 2: Weakly Consistent Micro-Tuning

The controller uses the BST as a guide to choose the most suitable partitions to steal resources when allocating nodes for a job. We implement the following operations for the BST structure to best serve the job resource allocation:

- BST_insert(BST*, char*, int): insert a data item to the BST data structure specifying the number of free nodes of a partition. This operation has already been used in Algorithm 7 (line 8).
- BST_delete(BST*, char*, int): delete a data item from the BST data structure.
- BST_delete_all(BST*): evict all the data items from the BST data structure for all the partitions. This operation has already been used in Algorithm 7 (line 6).
- BST_search_best(BST*, int): for a given number of required compute nodes, this operation searches for the most suitable partition to steal free nodes. There are 3 cases: (1) multiple partitions have enough free nodes; (2) only one partition has enough free nodes; (3) none of the partitions have enough free nodes. For case (1), it will choose the partition that has the minimum number of free nodes among all the partitions that have enough free nodes. For case (2), it will choose the exact partition that has enough free nodes. For case (3), it will choose the partition that has the maximum number of free nodes.
- BST_search_exact(BST*, char*): given a specific controller id, this operation searches the resource information of that partition.

Algorithm 8 depicted in Figure 90 gives the complete resource allocation procedure. When a job is submitted, a controller first tries to allocate free nodes in its partition (line 2). As long as the allocation is not satisfied, the controller searches for the most suitable partition to steal resources from the BST (lines 23, 24, 27). The data of that partition is then deleted (line 25) to prevent different job allocations competing for the resources from the same partition in a controller. The controller then queries the actual free resource of that partition via a ZHT *lookup* operation (line 4). After that, the controller tries to allocate free nodes for the job through a ZHT *compare and swap* operation (line 5). If the allocation succeeds, the controller will insert the updated free node list of that partition to the local BST (lines 6 – 16). Otherwise, if the controller experiences several failures in a row, it releases all the allocated free nodes, waits some time, and tries the resource allocation procedure again (lines 17 – 20).



```
Algorithm 8.   Micro-Tuning Resource Allocation
Input: number of nodes required (num_node_req), number of nodes allocated (*num_node_alloc), self id (self_id), sleep
length (sleep_length), number of reties (num_retry), binary search tree of the number of free nodes of each partition (*BST)
Output: list of nodes allocated to the job
1   *num_node_alloc = 0; num_try = 0; alloc_node_list = NULL;
2   target_ctl = self_id;
3   while *num_node_alloc < num_node_req do
4         resource = zht_lookup(target_ctl);
5         nodelist = allocate_resource(target_ctl, resource, num_node_req – *num_node_alloc);
6         if (nodelist != NULL) then
7               num_try = 0; alloc_node_list += nodelist;
8               *num_node_alloc += nodelist.size();
9               new_resource = resource – nodelist;
10              pthread_mutex_lock(&bst_lock);
11              old_resource = BST_search_exact(bst, target_ctl);
12              if (old_resource != NULL) then
13                    BST_delete(bst, target_ctl, old_resource.size());
14              end
15              BST_insert(bst, target_ctl, updated_resource.size());
16              pthread_mutex_unlock(&bst_lock);
17        else if (num_try++ > num_retry) then
18              release_nodes(nodelist); alloc_node_list = NULL;
19              *num_node_alloc = 0; num_try = 0; sleep(sleep_length);
20        end
21        if (*num_node_alloc < num_node_req) then
22              pthread_mutex_lock(&bst_lock);
23              if ((data_item = BST_search_best(bst, num_node_req –*num_node_alloc) != NULL) then
24                    target_ctl = data_item.ctl_id;
25                    BST_delete(bst, target_ctl, data_item.size());
26              else
27                    target_ctl = self_id;
28              end
29              pthread_mutex_unlock(&bst_lock);
30        end
31  end
32  return alloc_node_list;
```

Figure 90. Algorithm of micro-tuning resource allocation

We call the procedure the micro tuning phase because only the data of the resource of one partition is changed during one attempted stealing. Every controller updates its BST individually when allocating resources. As time increases, the controllers would have inconsistent view of the free resources of all the partitions. In the meantime, the controller is updating the whole BST with the most current resources of all the partitions (Macro-tuning phase as Algorithm 6). With both macro tuning and micro tuning, the resource stealing technique has the ability to balance the resources among all the partitions dynamically, to aggressively allocate the most suitable resources for big jobs, and to find the free resources quickly under high system utilization.

Going back to the problem of finding the number of stealing operations required satisfying one, two, and more concurrent full-scale jobs. It is straightforward that one full-scale job needs only n stealing operations, where n is the number of partitions. For k concurrent full-scale jobs, assuming all the controllers have the same pace of processing speed, it will take $\Theta(kn)$ stealing operations to satisfy all the jobs. This is much better than the random resource stealing technique.

6.4 Implementation Details

This section describes the Slurm production system and the implementation details of Slurm++ distributed workload manager.

6.4.1 Slurm Production System. Slurm is a centralized workload manager for launching large-scale HPC workloads on several of the top supercomputers in the world. Slurm has a centralized controller (slurmctld) that manages all the compute daemons (slurmd) located on the compute nodes. Slurm keeps all the job and resource metadata in global data structures in both memory and a centralized file system.

Upon receiving a job, the slurmctld first looks up the global resource data structure to allocate resource for the job. In this paper, we focus on the interactive jobs submitted by the "srun" command. All the jobs have the format: *srun -N$num_node* [-*n$num_process_per_node*] [--*mpi=pmi2*] *program parameters*, for example *srun -N2 sleep 10* is a job that requires 2 nodes to sleep 10 seconds; *srun -N10 -n2 --mpi=pmi2 pi*

1000000 is an MPICH2 MPI job that requires 2 nodes with each one using 2 processes to calculate the value of pi with 1M points. Once a job gets its allocation, it can be launched via a tree-based network rooted at rank-0 slurmd by “srun”. When a job is done, the “srun” notifies the slurmctld to release the resource. In addition, all the slurmds return to slurmctld, which then terminates the job on all the involved slurmds. The input to Slurm is a configuration file read by slurmctld and all the slurmds, which specifies the identities of the slurmctld and slurmds so that they can communicate with each other. All the slurmds first register by sending a registration message to the slurmctld, and then wait for jobs to be executed.

6.4.2 Slurm++ Workload Manager. We implement the Slurm++ workload manager in two steps. In the first step, we develop a simple orphan prototype that implements the random resource stealing technique. In the second step, we implement the weakly consistent monitoring-based resource stealing technique and enable to run real MPI applications. From this point, we distinguish the two implementations as Slurm++ v0 and Slurm++ in the two steps, respectively.

6.4.2.1 Slurm++ v0 implementation

In Slurm++ v0, we discard the bulk of the slurmctld functionality, and developed a lightweight controller that talks to slurmds directly. We moved the standalone “srun” command into the controller and modified it to allocate resources and launch jobs as a thread in the controller. Different partitions have different configuration files; a controller knows which slurmds it manages, and a slurmd knows which controller to talk to.

Each controller was initialized as a ZHT client, and called the client APIs to query and modify the job and resource information. Upon receiving all the slurmds’ registration messages in a partition, the controller inserted the available nodes to ZHT. The controller implemented the random resource stealing technique.

6.4.2.2 Slurm++ implementation

The Slurm++ v0 implementation broke the communication links among slurmctld, srun, and slurmds in Slurm because of the added simplified controller. We managed to enable Slurm++ v0 to run micro-benchmark sleep jobs, but it failed to run real MPI applications. In Slurm++, we preserve all the communication protocols and components of Slurm and majorly modify the code in the slurmctld source file folder. We add the initialization as ZHT client in the controller; replace the resource allocation part in Slurm with the ZHT KVS interactions and the proposed weakly consistent monitoring-based resource stealing technique. The modifications turn into about 5000 lines of code added to the Slurm codebase. The Slurm++ source code is made open source on GitHub repository: https://github.com/kwangiit/SLURMPP_V2. Slurm++ has several dependencies, such as Slurm [30], ZHT [51], and Google Protocol Buffer [95].

We examined developing a resource allocation plugin for Slurm to reduce the tedious labor of touching and modifying the Slurm code directly. However, this was unrealistic as the resource allocation is the core part and has been designed in a centralized manner in Slurm.

6.5 Performance Evaluation

We evaluate Slurm++ by comparing it with Slurm++ v0 and Slurm. Then, we compare Slurm++ with Slurm using workloads from real application traces and real MPI applications. All of the systems were run on Kodiak. Furthermore, we explore the scalability of Slurm++ through simulations up to millions of nodes on the Fusion machine.

6.5.1 Micro-Benchmark Workloads. This section aims to show how the proposed resource stealing technique is compared with the random one, and how much performance gain achieved by Slurm++ comparing with Slurm.

We use the following workloads to compared Slurm++ with Slurm++ v0 and Slurm: the partition size is set to be 50, a reasonable number to insure a sufficient number of controllers. It is the default value of the following experiments, unless specified explicitly. At the largest scale of 500 nodes, there are 10 controllers. The workloads include the simplest possible NOOP “sleep 0” jobs that require different number of nodes per job. The workloads have 3 different distributions of job sizes: all one-node jobs; jobs with sizes having uniform distribution that has an average of half partition – 25 (1 to 50), referred to half-partition jobs; and jobs with sizes having uniform distribution that has an average of full partition – 50 (25 to 75), referred to full-partition jobs. We do weak-scaling experiments for all the workloads, and compare the throughputs of all the workload managers up to 500 nodes. Then, we focus on Slurm++ and show the speedups between Slurm++ and Slurm for the three workloads.

6.5.1.1 One-node Jobs

In this workload, each controller runs 50 jobs and all the jobs require only one node. The throughput comparison results of Slurm++, Slurm++ v0, and Slurm are shown in Figure 91.

We see that the throughput of Slurm increases to a saturation point (51.6 jobs/sec at 250-node scale), and after that experiences a decreasing trend as the system scales up. This is because the centralized “slurmctld” has limited processing capacity that is unable to allocate resources fast enough to keep up with the increasing numbers of jobs and system scales, leading to longer waiting time of resource allocation when scaling up, even though there are enough resources to satisfy all the jobs. The throughputs of both Slurm++ and Slurm++ v0 have a linearly increasing trend up to 500 nodes, because more controllers are added when scaling up. In addition, Slurm++ performs constantly better than Slurm++ v0 by about 10%, due to the better resource stealing technique in Slurm++. Within 200-node scales, Slurm has the highest throughput because the distributed architecture has overheads, such as communications through ZHT. As the scale increases, the overhead is dwarfed by the performance gain of multiple controllers. At 500-node scale, Slurm++ can launch jobs 2.61X faster than Slurm. We believe that the performance gap will be bigger at larger scales, given the throughput trends.

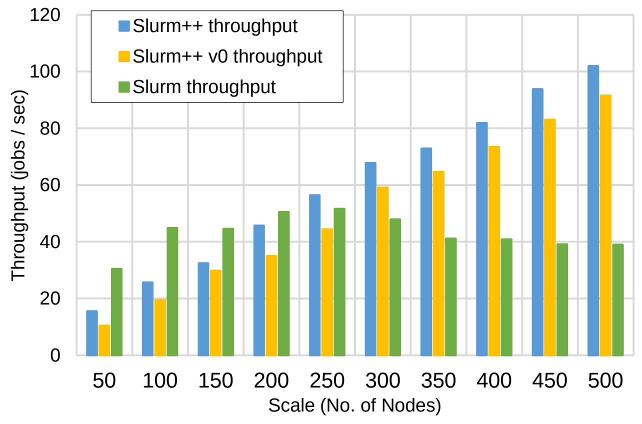


Figure 91. Throughput comparison with one-node jobs

6.5.1.2 Half-partition jobs

In this set of experiments, each controller runs 50 jobs that have an uniform distribution of jobs sizes with an average of half partition – 25 (1 to 50). Figure 92 illustrates the throughput comparison results with this workload. From Figure 92, we see that like the one-node job workload, the throughput of Slurm first ramps up to a saturation point (7 jobs/sec at 250 nodes) and then decreases when scaling up. Slurm has the lowest throughput at all scales. Again, for both Slurm++ and Slurm++ V0, the throughputs are increasing linearly up to 500 nodes, except that Slurm++ has a lower throughput number than Slurm++ at all scales.

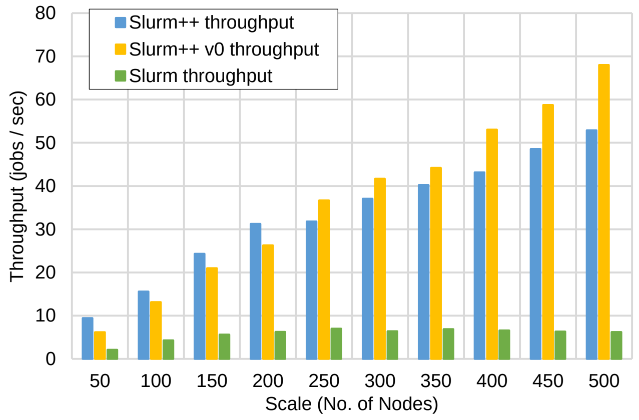


Figure 92. Throughput comparison with half-partition jobs

This does not mean the proposed resource stealing technique in Slurm++ performs worse than the random one in Slurm++ v0. This is because the implementation of the Slurm++ v0 replaced the complex “slurmctld” by a simplified lightweight controller, which reduced a large portion of communication messages when launching a job, such as the returning messages to the slurmctld from all the slurmds after a job is done. For this workload, the reduced messages dominate the communication overheads instead of the resource stealing technique. The resource stealing operation is not triggered too often because all of the jobs require not more than a full partition of nodes. This will be validated in the full-partition job workload results. Still, at 500 nodes, Slurm++ launches jobs 8.5X faster than Slurm; and the speedup will be bigger at larger scales.

6.5.1.3 Full-partition jobs

In this workload, each controller runs 20 jobs that have an uniform distribution of jobs sizes with an average of full partition – 50 (25 to 75). Figure 93 depicts the throughput comparison results of the three systems with this workload. Not surprisingly, Slurm saturates after 400 nodes, with a maximum throughput of 4.3 jobs/sec at 500 nodes. Also, Slurm has the lowest throughput at all scales. On the other hand, both Slurm++ and Slurm++ v0 are able to keep a linear throughput increasing tread up to 500 nodes, and will keep this trend at larger scales. Furthermore, Slurm++ launches jobs 2.25X faster than Slurm++ v0 at 500-node scale. This validates the analysis of the performance difference between Slurm++ and Slurm++ v0 in section 6.5.1.2, and indicates that the proposed resource stealing technique performs better than the random one, especially for big jobs. At 500 nodes, Slurm++ is able to launch jobs 10.2X faster than Slurm; and again, the trends show that the speedup will be bigger at larger scales.

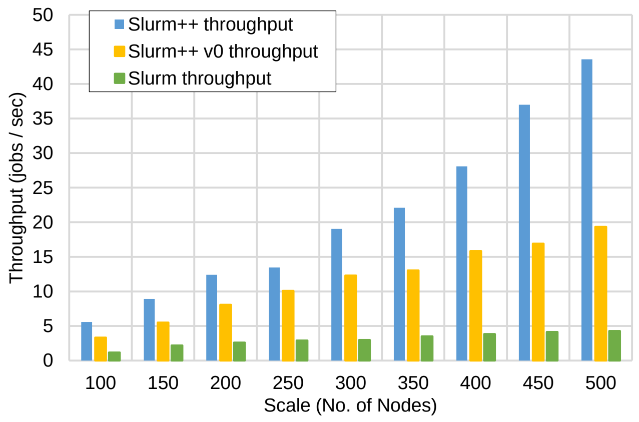


Figure 93. Throughput comparison with full-partition jobs

6.5.1.4 Speedup Summary

In order to understand the impacts of different job sizes on the performance of job launching, we summarize the throughput speedups between Slurm++ and Slurm of the three workloads with different resource stealing intensities in Figure 94.

Figure 94 shows that for all the workloads at all scales (except for the one-node job workload within 200 nodes), Slurm++ is able to launch jobs faster than Slurm. Besides, the performance slowdown (9.3X from one-node jobs to full-partition jobs) of Slurm due to increasingly large jobs is much more severe than that (2.3X from one-node jobs to full-partition jobs) of Slurm++. This highlights the better scalability of Slurm++. In addition, the speedup is increasing as the scale increases for all the workloads, indicating that at larger scales, Slurm++ would outperform Slurm even more. Another important fact is that as the job size increases, the speedup is also increasing. This trend proves that the proposed resource stealing technique can overcome the problems of the random method, and has great scalability for big jobs.

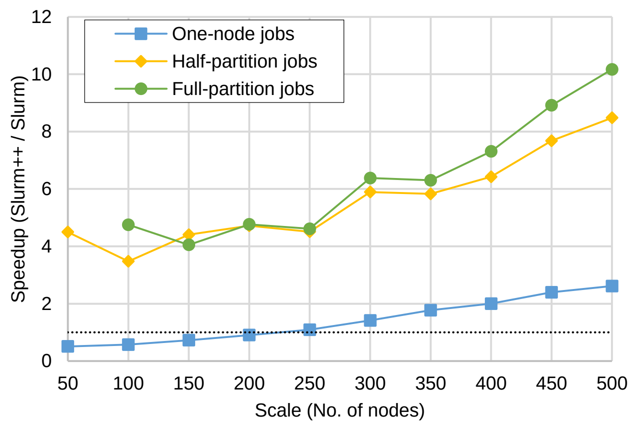


Figure 94. Speedup summary between Slurm++ and Slurm

6.5.2 Workloads from Application Trace. The micro-benchmark jobs are relatively small, even the largest-size jobs in the full-partition workload require only 75 nodes. To further show how Slurm++ performs under workloads that have a mixture distribution of job sizes including full-scale jobs, we run both Slurm++ and Slurm using workloads obtained from the real parallel application trace, whose job size duration is shown in Figure 83.

At each scale, we generate a workload that include all the 68,936 jobs and preserves the job size distribution of the original workload trace by applying the job size percentage of the machine size of each job. For example, in the original workload trace, if a job requires k nodes, then at the scale of n nodes of our experiments, that job will require k/40960*n where 40960 is the number of nodes of the BG/P machine. Besides, we reduce the job duration of each job by 1M times to save the running time of the whole experiments. More importantly, this is to mimic the scheduling challenges of extreme-scale systems through a 500-node scale. As we do not have exascale machines yet, we make the scheduling granularity much finer. We believe the scheduling challenges at exascale with the current runtime could only be worse than that at 500-node scale with runtime of 1M times shorter. Because the job runtime should have decreasing trends as the nodes keep becoming more powerful. In the end, all the jobs have sub-second lengths. However, Slurm is only able to handle 10K jobs (crash beyond 10K jobs). Therefore, at each scale, we only run the first 10K jobs of the total 68,936 jobs.

Figure 95 shows the performance comparison results between Slurm++ and Slurm with this workload. The experiments are strong scaling in terms of number of jobs and job length, and weak scaling in terms of job size. We see that as the system scales up, the throughput of Slurm has a decreasing trend, indicating that Slurm has increasing overheads to handle bigger jobs, and is not scalable to handle big and short jobs. On the contrary, the throughput of Slurm++ is increasing almost linearly with respect to the system scale, as well as the job sizes, showing the great scalability of Slurm++. At 500-node scale, Slurm++ outperforms Slurm by 8X, and this performance gain will be larger at larger scales.

To understand the scalability of the proposed resource stealing technique, we show the average ZHT message count per job of Slurm++ in Figure 95, which presents a linear increasing relationship with respect to the system scale. This makes perfect sense as the job size is also increasing linearly as the system scales up. It also confirms our induction that the proposed resource stealing technique takes $\Theta(kn)$ number of request to satisfy big jobs.

In prior work on evaluating ZHT [51], micro-benchmarks showed ZHT achieving more than 1M ops/sec at 1024K-node scale. We see that at 500-node scale, the average ZHT message count per job is 51 (or about 510K messages for 10K jobs). Even with this workload achieving 36.9 jobs/sec at 500-node scale, ZHT generates throughput of 1882 ops/sec, which is far from being a bottleneck for the workload and scale tested.

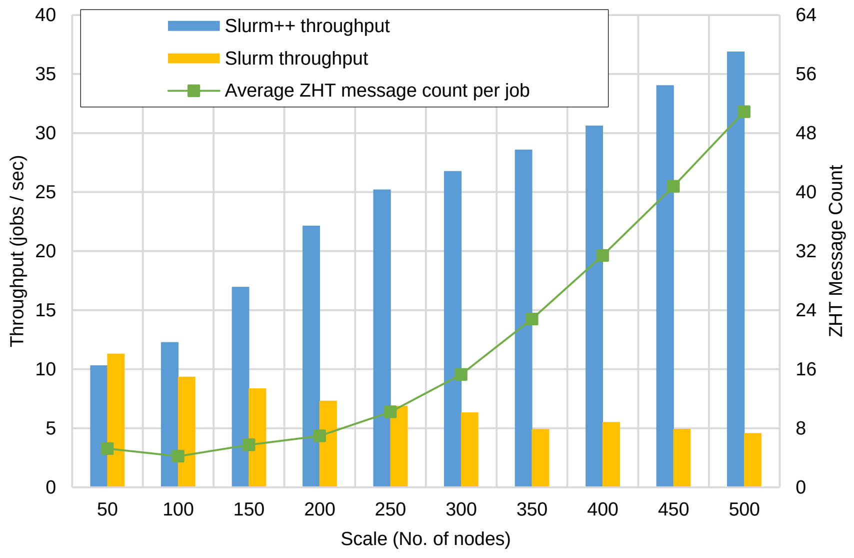


Figure 95. Comparisons with workloads from real application traces

6.5.3 Real MPI Applications. The ultimate goal is to enable Slurm++ to support the running of real scientific HPC applications that use MPI for synchronization and communication. PETSc library [152] is a portable, extensible toolkit for scientific numeric computation, developed by ANL. The library offers scientific programmers simple interfaces to operate on numeric vectors and matrices, and encapsulates the MPI communication layer to allow transparent and automatic parallelism without programmers' laborious efforts. One example of the scientific problems is to solve the complex partial differential equations (PDE).

We studied the PETSc library and extracted 1000 PETSc jobs with different scientific applications, ranging from Physics, Chemistry, to Mathematics that include the matrix multiplication and the linear solutions of the equation Ax=b, which use MPI underneath for multi-processor scalable computing. We compile all the application programs, and use both Slurm++ and Slurm to launch and execute the jobs. We set the job size distribution the same as that shown in Figure 83, and each node forks 2 processes. Figure 96 shows the performance comparison between Slurm++ and Slurm with the real MPI applications.

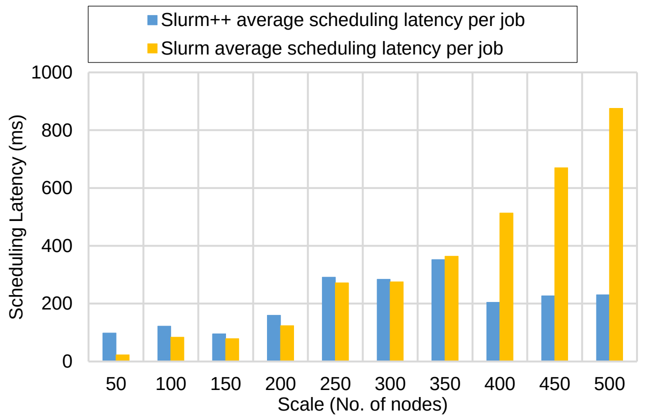


Figure 96. Comparisons with real MPI applications

From Figure 96, we see that, within 300 nodes, Slurm has a lower average scheduling latency than Slurm++. This is because at small scales for Slurm++, the overheads of the distributed resource state and scheduling are relatively big comparing with the performance gain of distributed multiple schedulers. However, as the system scale increases, the average scheduling latency of Slurm++ increases with a much slower speed than that of Slurm. At 350 nodes and beyond, Slurm++ has lower average scheduling latency than Slurm, and the difference is getting bigger at larger scales. At 500 nodes, Slurm has average scheduling latency 3.8X longer than Slurm++, and the gap will increase at even larger scales. The trend indicates that Slurm++ is much more scalable towards larger scales for real scientific applications. This attributes to the partition-based distributed architecture and the proposed resource stealing technique.

6.5.4 Exploration of Different Partition Sizes of MTC. This subsection explores the effects of different partition sizes on the performance of Slurm++. We focus on the NOOP "sleep 0" MTC workloads that all the jobs require 1 node. We do weak-scaling experiments launching 100 jobs per node; at 500 nodes, there are 50K jobs. We vary the partition sizes to be 1, 4, 16, 64 and 100, and show the throughput results in Figure 97. We see that as the partition size increases, the throughput decreases. Slurm++ gets the best performance when partition size is 1, meaning a 1:1 mapping between the controller and the slurmd. At 500 node-scale with 1:1 mapping, Slurm++ achieves throughput as high as 3509 jobs/sec, and the linearly increasing trend will likely remain at larger scales. The reason that bigger partition size performs worse is that the bigger partition size leads to bigger (*key*, *value*) record size in ZHT, and results in a longer communication time per ZHT operation.

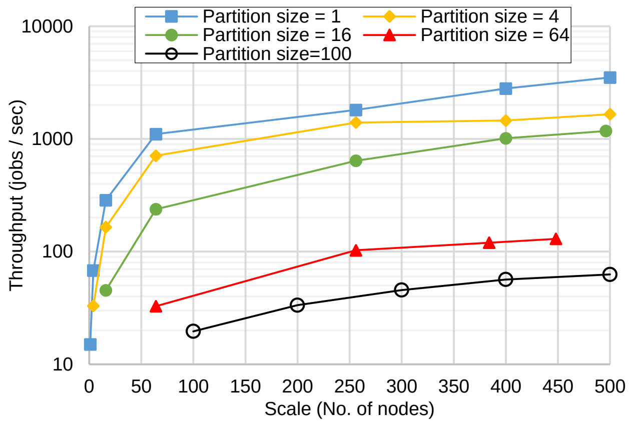


Figure 97. Performances of different partition sizes with MTC workloads

Therefore, for MTC application, we conclude that *the partition size should be set close to the average job size with a uniform size distribution.* For a workload that has a large distribution of job sizes, it is better to configure heterogeneous partition sizes with a few large partition sizes to handle big jobs, and a lot more small partition sizes for other jobs.

6.6 Exploring Scalability through the SimSlurm++ Simulator

We have shown that Slurm++ is able to outperform Slurm by 10.2X regarding job throughput and by 3.8X in terms of average scheduling latency up to 500 nodes. This section explores the scalability of the proposed architecture and the resource stealing techniques towards extreme-scales via the SimSlurm++ simulator.

6.6.1 SimSlurm++ Overview. SimSlurm++ is a sequential discrete event simulator (DES) [119] that was built on top of PeerSim [49], a lightweight and scalable peer-to-peer simulator that offers the framework and functionality of simulating distributed systems, developed in Java. SimSlurm++ simulates the exact behavior of the Slurm++ workload manager. All the activities in the system are modeled as events that are tagged with occurrence times, and are sorted chronologically in a global event queue. At each iteration, the simulation engine fetches the first event, executes the corresponding actions, and may insert the following events in the queue. The simulation terminates when the queue is empty.

The simulator consists of many simulated nodes (a node is an object instance). Some nodes are controlling nodes that simulate controllers and ZHT servers with 1:1 mapping; while others are computing nodes that simulate slurmds. All the nodes are identified with consecutive integers, ranging from 1 to the number of nodes. The ZHT server we simulated has an in-memory hash table data structure that keeps all the (*key*, *value*) pairs, and can process 3 requests (*lookup*, *insert*, *compare and swap*) to the hash table. We implement the proposed resource stealing technique in SimSlurm++. The connectives among all the nodes are modeled as the same as those in Slurm++: a controller manages a partition of slurmds, and a slurmd has a dedicated controller to talk to.

The input to SimSlurm++ is a configuration file that specifies the parameters, such as the simulation scale (number of nodes), the simulation network environment (e.g. network speed, latency, etc.), the partition size, the workload file, etc. The parameters are set based on the values of running Slurm++ on the Kodiak machine. We run SimSlurm++ on the “fusion” machine. SimSlurm++ is single-threaded, and at 64K nodes, it requires 30GB memory and takes 5 hours.

6.6.2 Validation of the SimSlurm++ Simulator. Before using SimSlurm++ to explore the scalability of Slurm++, we validate SimSlurm++ against Slurm++ to make sure it is accurate to gain valuable insights. The validation is conducted up to 500 nodes using the same configurations and workloads as in section 6.5.2. Figure 98 depicts the validation results.

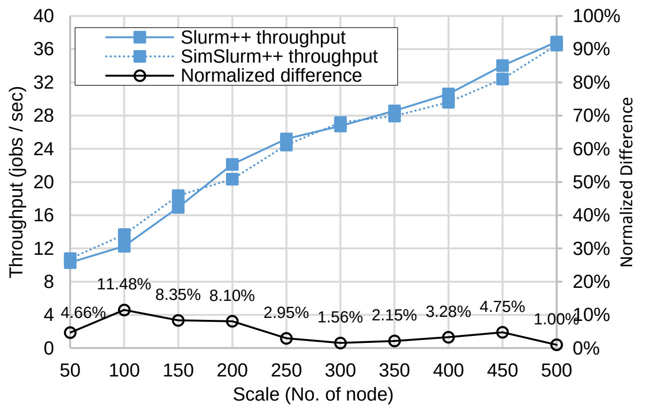


Figure 98. Validation of SimSlurm++ againt Slurm++

We label the normalized difference in percentage in the figure, which is calculated as: *Math.abs*(Slurm++ *throughput* – SimSlurm++ *throughput*) / Slurm++ *throughput*. Smaller difference percentage indicates a more accurate simulation. We see that the differences are small at all scales, with an average of 4.8% and a maximum of 11.48% (at 100 nodes). We believe the relatively small differences demonstrate that SimSlurm++ is accurate enough to produce convincible results.

6.6.3 Exploring the Random Resource Stealing. We explore the scalability of the random resource stealing technique with both the 1024:1 HPC and 1:1 MTC configurations, up to millions of nodes.

6.6.3.1 SimSlurm++ HPC Configuration (1024:1)

We evaluate SimSlurm++ with HPC orientation of 1024:1 mapping between controller and slurmd. The workload used follows the same distribution as in section 6.5.2. We run experiments up to 64K nodes. Figure 99 shows the results of SimSlurm++ with HPC configuration.

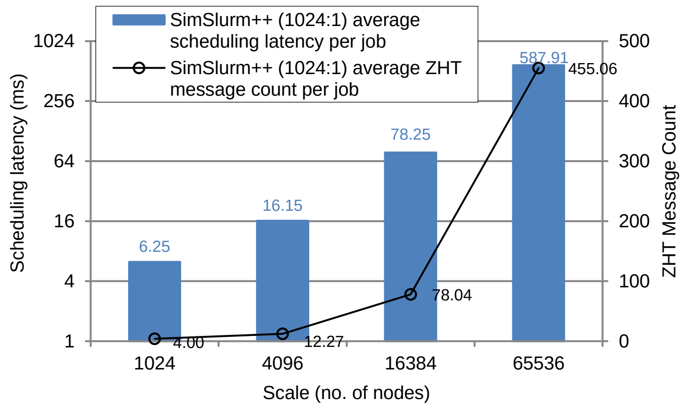


Figure 99. SimSlurm++ (1024:1) latency and ZHT messages

We see that the average scheduling latency increases super-linearly with respect to the system scales, partition sizes, and job sizes (the large the system is, there are more partitions and the average job size is bigger). At 64K-node scale, the average scheduling latency increases to 588ms from 6ms at 1K nodes, which is 1.5X of the average job length (i.e. 429ms). These results show the scalability problem of the random resource stealing technique for increasingly large jobs and more partitions. The increasing of average scheduling latency is due to the longer waiting time needed to get allocation at larger scales.

We also show the average ZHT message count in Figure 99. The average message count is increasing with the same trend as that of the average scheduling latency, from 4 ZHT messages at 1K nodes to 455 ZHT messages at 64K nodes. The increased messages are mainly *lookup* messages that are involved in the resource stealing procedure to find available resources, especially for the full-scale jobs. The bigger a job is, the longer it takes to find enough resources. This also exposes the scalability problems of the random resource stealing algorithm.

6.6.3.2 SimSlurm++ MTC Configuration (1:1)

The random resource stealing technique is supposed to achieve good performance for small jobs. We validate this through SimSlurm++ with the MTC configuration of 1:1 mapping up to millions of nodes. The workload we used is micro-benchmark: each controller launches 10 “sleep 0” jobs, and each job requires 1 or 2 nodes randomly. 1-node jobs are processed locally, while 2-node jobs need to steal resources from other controllers. This workload limits the jobs to small MTC jobs, and at the meanwhile preserves the overheads of stealing resources among controllers.

Figure 100 shows the results of SimSlurm++ with MTC configuration. We see that the throughput is increasing linearly with the system scale. At 1M-node scale with MTC configuration, SimSlurm++ achieves throughput as high as 1.75M jobs/sec, which is very promising. At the same time, the average scheduling latency increases trivially from 4-node scale (233ms) to 1M-node scale (329ms), and keeps within half-second bound. These numbers satisfy the requirements of high throughput and low latency of next-generation resource management systems towards exascale computing with MTC workloads.

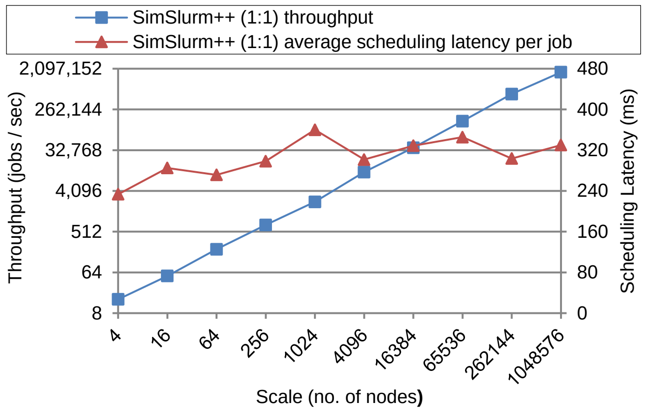


Figure 100. SimSlurm++ (1:1) throughput and latency

We also show the average ZHT message count information in Figure 101. The average message count is relatively stable, and keeps almost constant at large scales. At 1M-node scale, on average, each job requires only 25.6 message exchanges between controllers and ZHT servers. This small amount of messages shows great scalability of using the key-value store as the metadata storage system in resource management systems.

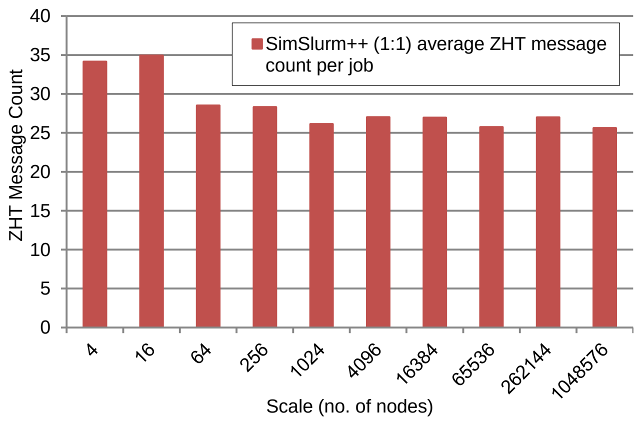


Figure 101. SimSlurm++ (1:1) ZHT message count

6.6.4 Exploring the Weakly Consistent Monitoring-based Resource Stealing. We also explore the scalability of the weakly consistent monitoring-based resource stealing technique through SimSlurm++ with the 1024:1 HPC configuration. The workloads used are the same as those in section 6.6.3.1: 10K jobs that have the same size duration as shown in section 6.5.2 at all scales. Figure 102 shows the simulation results of *average scheduling latency per job* and *average ZHT message per job* up to 65536 nodes (64 partitions).

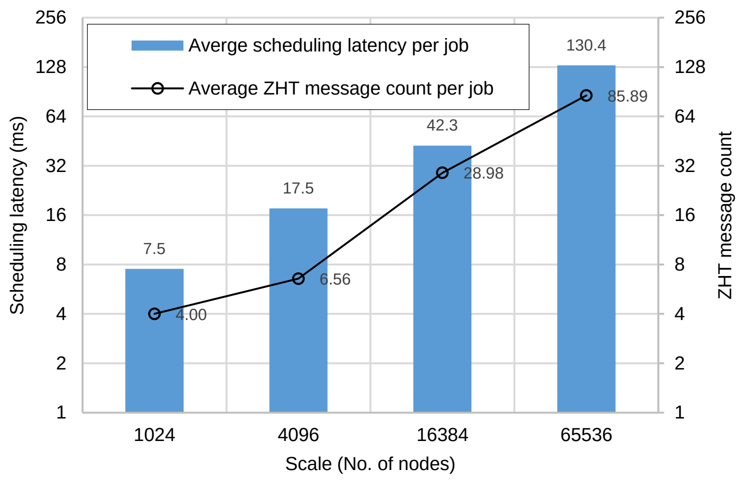


Figure 102. Simulation results, partition size = 1024

Both of the scheduling latency and the message count are increasing sub-linearly with respect to the number of partitions and the average job sizes. These results indicate the excellent scalability of the distributed architecture and the proposed resource stealing technique. The increased overheads (not so much) are resulted from extra query messages needed to satisfy the increasingly big jobs at larger scales. At 64K-node scale, the average scheduling latency is about 130ms, and the average message count is about 86. Comparing these numbers with those of the random resource stealing technique shown in Figure 99, the average scheduling latency is only 22.1% (130ms vs 587ms), and the average ZHT message count is only 18.7%, proofing that the weakly consistent monitoring-based technique scales up much better than the random one, and has quite low overheads considering the extreme scale of the system and job sizes.

The conclusions we can draw from the simulations are that *the distributed architecture, along with the proposed resource stealing technique, is scalable towards extreme-scales with more partitions and increasingly big jobs.*

6.7 Conclusions and Impact

Exascale supercomputers require next-generation workload managers to deliver jobs with much higher throughput and lower latency for a mixture of applications. We have shown that key-value store is a valuable building block to allow scalable system state management. With the distributed architectures and scheduling techniques, Slurm++ showed performance 10X better than the Slurm production system. Furthermore, simulations showed that Slurm++ can launch jobs that have a wide distribution of sizes with an average scheduling latency of 130ms, and an average message count of 86 at 64K-node scale. The distributed architecture, along with the weakly consistent monitoring-based resource stealing technique, is scalable towards extreme-scales with more partitions and increasingly big jobs.

We expect our work of Slurm++ to have revolutionary impacts on tomorrow's exascale supercomputers in doing scalable HPC resource management and job scheduling. The exascale machines are only several years away, and yet we still do not have certain answers to how the resource management system software will be architected and designed to maintain high performance. It is super urgent and vital to start exploring scalable distributed resource management systems to prepare for exascale machines. Without solid research in this space, we may be left with exascale machines that can only handle a few extremely large jobs at once for days long. Not even to mention the challenges of decreasing mean-time-to-failure and increasing cost of checkpointing, which will likely render the exascale machines unusable. Even though the concept of running large-scale long-duration HPC applications on supercomputers is inveterate, the ideal of supporting a mixture of applications with wide distributions in both job sizes and durations seems controversial at present, we believe this will be the case for exascale machines that are expected to have billion-way parallelism, given the fact that few applications have the ability to scale up to exascale. Without efficiently supporting large numbers of small to moderate HPC jobs and MTC workloads, the future of exascale computing is not very bright.

It is maybe immature to jump into the conclusion that “distributing” is the solution to address the resource management and scheduling challenges of exascale computing, However, Slurm++ is one of the pioneer work that seeks revolutionary and paradigm-shifting solutions, comparing with the decades-old centralized ones. Slurm++ opens the doors to and encourages a wide range of research directions to find scalable non-centralized solutions, such as system state management, resource sharing and balancing, resource allocation, job queueing, job scheduling, job launching, monitoring. We hope these research contributions will become handy upon the advent of exascale machines.

CHAPTER 7

RELATED WORK

This chapter discusses the related work that covers six areas, namely system software taxonomy, fine-grained task scheduling systems, load balancing, data-aware scheduling, batch-scheduling RMS, and distributed system simulations.

7.1 System Software Taxonomy

Work that is related to the simulation of system software includes an investigation of peer-to-peer networks [54], telephony simulations [55], simulations of load monitoring [56], and simulation of consistency [57]. However, none of the investigations focused on HPC, or combine distributed features of replication, failure/recovery and consistency. The survey in [58] investigated six distributed hash tables and categorized them in a taxonomy of algorithms; the work focused on the overlay networks. In [59], p2p file sharing services were traced and used to build a parameterized model. Another taxonomy was developed for grid computing workflows [60]. The taxonomy was used to categorize existing grid workflow managers to find their common features and weaknesses. Nevertheless, none of these work targeted HPC workloads and system software, and none of them use the taxonomy to drive features in a simulation. M. Schwarzkopf et al. [107] proposed a taxonomy of resource management systems of clusters. Using the taxonomy, they defined three scheduling architectures, and conducted simulations to study the design choices. However, the taxonomy only focuses on cluster schedulers, cannot be generalized.

Examples of the types of system software of interest in HPC are listed in Table 1. They include runtime systems (e.g. Charm++ [92], Legion [12], STAPL [14], and HPX

[13]); resource management systems (e.g. SLURM [30], Slurm++ [140], and MATRIX [23]); the I/O forwarding system, IOFSL [31], which is a scalable, unified I/O forwarding framework for HPC systems; the interconnect fabric managers, OpenSM [61], which is an InfiniBand subnet manager; and the data aggregation system, MRNet, which is a software overlay network that provides multicast and reduction communications for parallel and distributed tools. These are the types of system software that will be targeted for design explorations with our simulator.

Distributed key-value stores (KVS) are a building block for extreme-scale system software. Dynamo [34] is a highly available and scalable KVS of Amazon. Data is partitioned, distributed and replicated using consistent hashing, and eventual consistency is facilitated by object versioning. Voldemort [36] is an open-source implementation of Dynamo developed by LinkedIn. Cassandra [35] is a distributed KVS developed by Facebook for Inbox Search. ZHT [51] is a zero-hop distributed hash table for managing the metadata of future exascale distributed system software. Our simulator is flexible enough to be configured to represent each of these key-value stores.

7.2 Fine-grained Task Scheduling Systems

Fine-grained task scheduling systems of MTC and cloud computing have also been under development for years, such as Falkon [78], Mesos [106], YARN [112], Omega [107], Sparrow [81], CloudKon [82]).

Falkon is a centralized task scheduler with the support of hierarchical scheduling for MTC applications, which can scale and perform orders of magnitude better than centralized HPC cluster schedulers perform. 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. Besides, the resource provisioner dynamic creations and releases executors on compute node to execute tasks under the guidance of a centralize dispatcher, incurring significant overheads of network communication and forking and killing processes.

YARN [112] and Mesos [106] are two frameworks that decouple the resource management infrastructure from the task scheduler of the programming model to enable efficient resource sharing in general commodity Hadoop clusters for different data-intensive applications. Both of them apply a centralized resource manager to allocate resources to applications. The application master then will be in charge of scheduling tasks onto the allocated compute nodes. The difference between them is that Mesos employs an application master for one category of applications, while YARN is much finer grained in that it uses an application master per application, which, in theory, should be more scalable. Although they have improved the scalability and efficiency of the resource sharing in Hadoop clusters significantly with the separation, the centralized resource manager is still a barrier towards extreme scales or of the support for fine-grained workloads. We have compared YARN with MATRIX in scheduling Hadoop workloads, and MATRIX outperformed YARN by 1.27X for typical workloads, and by 2.04X for the real application.

Omega [107] is a cluster scheduling system that employs specific distributed schedulers for different applications. Like Slurm++, Omega shares global resource information to all the schedulers through a centralized master, and the schedulers cache the global information and use an atomic operation to resolve resource conflict on all the compute nodes. Unlike Slurm++, Omega does not group the nodes into partitions, each scheduler directly talks to potentially large number of compute nodes. Neither does Omega use key-value store to manage the resource and job metadata. Instead, Omega keeps the metadata in the centralized master, leading to poor scalability.

Another research direction aims to improve the Hadoop schedulers. Most of work focuses on optimizing the scheduling policies to meet different requirements in a centralized task scheduler. The Hadoop default schedulers include the Capacity Scheduler (CS) [169], the Fair Scheduler (FS) [170] and the Hadoop On Demand (HOD) Scheduler (HS) [171]. Each of them has a different design goal: the CS aims at offering resource sharing to multiple tenants with the individual capacity and performance SLA; the FS divides resources fairly among job pools to ensure that the jobs get an equal share of resources over time; the HS relies on the Torque resource manager to allocate nodes, and allows users to easily setup Hadoop by provisioning tasks and HDFS instances on the nodes. Rasooli and Down proposed a hybrid scheduling approach [172] that can dynamically select the best scheduling algorithm (e.g. FIFO, FS, and COSHH [173]) for heterogeneous systems. To optimize fairness and locality, Zaharia et. al proposed a delay scheduling algorithm [174] that delays the scheduling of a job for a limited time until highly possible to schedule the job to where the data resides. These efforts have limited advancement to the scalability because they work within a single scheduler. Some early work towards distributed resource management was GRUBER [175], which focused on distributed brokering of Grid resources.

Sparrow [81] is similar to our work of MATRIX 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 many 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 [96] due to early binding of tasks to worker resources. We have compared Sparrow and MATRIX using heterogeneous workloads in [82], 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 [82] has similar architecture as MATRIX, except that CloudKon focuses on the Cloud environment, and relies on the Cloud services, SQS [108] to do distributed load balancing, and DynamoDB [34] as the distributed key-value stores 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 Cloud services, which makes its adoption in high-end computing impractical.

7.3 Load Balancing

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 [77]. Centralized load balancing has been extensive studied in the past (JSQ [131], least-work-left [132], SITA [133]), but they all suffered from poor scalability and resilience.

Distributed Load balancing employs multiple schedulers to spread out computational and communication loads evenly across processors of a shared-memory parallel machine, or across nodes of a distributed system (e.g. clusters, supercomputers, grids, and clouds), so that no single processor or node is overloaded. Clients are able to submit workload to any scheduler, and each scheduler has the choice of executing the tasks locally, or forwarding the tasks to another scheduler based on some function it is optimizing. Although distributed load balancing is likely a more scalable and resilient solution towards extreme scales, there are many challenges that must be addressed (e.g. utilization, partitioning). Fully distributed strategies have been proposed, including neighborhood averaging scheme (ACWN) [101], [103], [104]. In [104], several distributed load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model (GM) and a Hierarchical Balancing Method (HBM). Other hierarchical strategies are explored in [103] and [134].

Work stealing is an efficient distributed load balancing technique that has been used at small scales successfully in parallel languages such as Cilk [105], X10 [125], Intel TBB [139] and OpenMP, to balance workloads across threads on shared memory parallel machines [86] [87]. Theoretical work has proved that a work stealing scheduler can achieve execution space, time, and communication bounds all within a constant factor of optimal [86]. But the scalability of work stealing has not been well explored on modern large-scale systems. In particular, concerns exist that the randomized work stealing can lead to long idle times and poor scalability on large-scale clusters [87]. The work done by Diana et. al in [87] scaled work stealing to 8K processors using the PGAS programming model and the RDMA technique. A hierarchical technique that improved Diana's work described work stealing as retentive work stealing. This technique scaled work stealing to over 150K cores by utilizing the persistence principle iteratively to achieve the load balancing of task-based applications [135]. However, these techniques considered only load balancing, not data-locality. On the contrary, our work optimized both work stealing and data-locality. Besides, through simulations, our work shows that work stealing with optimal parameters works well at exascale levels with 1-billion cores.

Charm++ [77] supports centralized, hierarchical and distributed load balancing. It has demonstrated that centralized strategies work at scales of thousands of processors for NAMD. In [77], 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 [137] 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 [138] 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.

Swift [15] [67] is a parallel programming system for running workflow applications. Tasks in Swift may be implemented as external programs, library calls, and script fragments in Python, R, or Tcl. Swift uses in-memory functions to compose applications, and the Turbine scheduler to schedule tasks. Turbine has a distributed load balancer, called ADLB [190], which applies work stealing to achieve load balancing. Swift has the ability to execute its dataflow-based programming model over an MPI-based runtime, or execute the tasks using the MPI library internally. Swift has been scaled to distribute up to 612 million dynamically load balanced tasks per second at scales of up to

262,144 cores [176]. Swift works at the programming laungauge level, and this extreme scalability would absolutely advance the progress of making MATRIX supporting large-scale scientific application at extreme-scales.

7.4 Data-aware Scheduling

Falkon implemented a data diffusion approach [53] 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 [177] 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 [75] 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 [118], 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 [122] is a scalable locality-aware adaptive work stealing scheduler that supports both work-first and help-first policies [123] 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.

The other work about data-aware work stealing technique improved data locality across different phases of fork/join programs [136]. This work relied on constructing a sample pre-schedule of work stealing tree, and the workload execution followed the pre-schedule. This involved overheads of creating the sample and was not suitable for irregular applications. Furthermore, both this work and SLAW focused on the single shared-memory environment.

Another work [124] 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 [125] 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.

Another related research direction is scalable metadata management. To optimize the metadata management and usage for small files, Machey et al. provided a mechanism that utilizes the Hadoop "harballing" compression method to reduce the metadata memory footprint [178]. Zhao et al. presented a metadata-aware storage architecture [179] that utilizes the classification algorithm of merge module and efficient indexing mechanism to merge multiple small files into Sequence File, aiming to solve the namenode memory bottleneck. However, neither work touched the base of addressing the scalability issues of the centralized namenode in processing the tremendously increased large amount of metadata access operations. Haceph [180] is a project that aims to replace the HDFS by the

Ceph file system [181] integration with the Hadoop POSIX IO interfaces. Ceph uses a dynamic subtree partitioning technique to divide the name space onto multiple metadata servers. This work is more scalable, but may not function well under failures due to the difficulty of re-building a tree under failures.

7.5 Batch-Scheduling Resource Management Systems

Traditional batch-scheduling systems are batch-sampled specific for large-scale HPC applications. Examples of these systems are SLURM [30], Condor [83], PBS [84], and Cobalt [114]. 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. Condor was developed as one of the earliest RMSs, 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 [182] 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 RMSs, such as Cobalt [114], 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.

There have also been several projects that addressed efficient job launch mechanisms. In STORM [146], the researchers leveraged the hardware collective available in the Quadrics QSNET interconnect. They then used the hardware to broadcast the binaries to the compute nodes. Though this work is as scalable as the intercon-nect, the server itself is still a single-point of failure.

BPROC [147] was a single system image and single pro-cess space clustering environment where all process ids were managed and spawned from the head node, and then distributed to the compute nodes. BPROC trans-parently moved virtual process spaces from the head node to the compute nodes via a tree spawn mecha-nism. However, BPROC was a centralized server with no failover mechanism.

LIBI/LaunchMON [148] is a scalable lightweight bootstrapping service specifically to disseminate con-figuration information, ports, addresses, etc. for a ser-vice. A tree is used to establish a single process on each compute node, this process then launches any subse-quent processes on the node. The tree is configurable to various topologies. This is a centralized service with no failover or no persistent daemons or state, therefore if a failure occurs they can just re-launch.

PMI [149] is the process management layer in MPICH2. It is close to our work in that it uses a KVS to store job and system in-formation. But the KVS is a single server design rather than distributed and therefore has scalability as well as resilience concerns.

ALPS [150] is Cray's resource man-ager that constructs a management tree for job launch, and controls separate daemon with each one having a specific purpose. It is multiple single-server architec-ture, with many single-point of failures.

ORCM [151] is an Open Resilient Cluster Manger originated from the Cisco resilient runtime system for monitoring enterprise-class routers, and is under develop-ment in Intel to do resource monitoring and scalable tree-based job launching for high-end computing clusters. Currently, the state management of ORCM is centralized in the top layer aggregator, which is not scalable. The use of key-value stores to manage the state simi-lar to Slurm++ is a good alternative for ORCM.

Borg [154] is Google's cluster manager that offers a general interface for various workloads, such as the low-latency production jobs and long-running batch jobs. Borg applies a centralized resource manager (BorgMaster) to monitor and manage resources, and a separate scheduler process to allocate resources and schedule jobs. Although Borg managed to improve the scalability of the centralized architecture through techniques such as score caching, equivalence classes, and relaxed randomization, we believe that the continued growths of system size and applications will eventually hit the ultimate scalability limit, if the centralized architecture remains.

7.6 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 [109], SimGrid [88] and PeerSim [49]; and packet level, which models the details of the underlying network, such as OMNET++ [46], and OverSim [47].

SimGrid [88] 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 [109] is developed based on SimJava [110] 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. We have examined the resource consumpitons of both SimGrid and GridSim, and the results showed that SimGrid simulated only 64K nodes while consuming 256GB memory, and GridSim could simulate no more than 256 nodes.

PeerSim [46] 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++ [46] is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators. Built on top of OMNeT++, OverSim [47] uses DES to simulate exchange and processing of network messages. All of PeerSim, OMNet++ and OverSim have standard Chord [28] protocol implementation, we compared them and found that PeerSim is much faster and consumes much less memory.

ROSS [130] is a parallel discrete event simulator that aims at simulating large-scale systems with up to millions of objects on supercomputers. ROSS simulator is a collection of logical processes (LP) with each one modeling a component of the simulated system. The LPs communicate with each other by exchanging events that are synchronized through the Time Warp mechanism using a detection-and-recovery protocol. Reverse Computation technique is used in Ross to achieve extremely high parallelism. Ross has been run on the BG/P supercomputer up to 13K processors, and reported event-rate of 12.26 billion events per second for the PHOLD benchmark at 64K processors. We have begun work in using the ROSS simulator to simulate distributed scheduling algorithms.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This chapter highlights the research contributions, draws the conclusions, presents the future work, and envisons the long-term impacts of this dissertation.

8.1 Research Contributions

This dissertation has focused on delivering scalable resource management system software for tomorrows's extreme-scale distributed systems and diverse applications. The current resource management systems employ the decades-old centralized design paradigm, thus are unlikely to scale up to extreme scales, leading to inefficient use of extreme-scale distributed systems. This dissertation has taken the paradigm-shifting distributed approaches to reimplement the resource management systems. We have evaluated the proposed architectures and techniques extensively and they showed huge performance gains over the centralzied ones. Furtheremore, our simulation work has demonstrated that these architectures and technqiues are scalable up to extreme scales. *The research contributions of this dissertation are both envoluntionary and revolutionary, which are highlighted as follows :*

- We devised a comprehensive system software taxonomy by decomposing system software into their basic building blocks. As distributed systems approach exascale, the basic design principles of scalable and fault-tolerant system architectures need to be investigated for system software implementations. Instead of exploring the design choices of each system software individually and in an ad hoc fashion, this taxonomy has the ability to reason about general system software as follows: (1) it gives a systematic way to decompose system software into their basic components; (2) it allows one to categorize system software based on the features of these components, and finally, (3) it suggests the configuration spaces to consider for evaluating system designs via simulation or real implementation. We have shown quite a few examples of system software that can be beneficial from this taxononmy. This taxonomy has also led us architecting, designing, simulating and implementing the next generation resource management systems successfully.

- We proposed that key-value stores (KVS) are a viable buiding blocks for and can accelerate the development and deployment of extreme-scale system software. KVS had not yet been widely used in HPC environment before when neither of the system scale and the application metadata was large. This proposal is meaningful and enlightening to the HPC community as both the system and application scales are evolving towards extreme scalses. Many HPC system software have started to explore the use of KVS. In addition, both the MATRIX task scheduling system and the Slurm++ workload manager are built on top of the ZHT KVS, resulting in great performance.
- We developed a KVS simulator that has been used to explore different architectures and distributed design choices up to extreme scales of millions of nodes. Our KVS simulator is configurable, and can be tuned to simulate many types of KVS. We have shown how easily to simulate a centralized KVS, a hiearchial KVS, distributed KVS with different configurations (e.g. Dynamo, ZHT) with our simulator. This simulator can be used as a basic tool to explore

the scalability and design choices of any KVS implementations. We have seen that the KVS simulator offered valuable insights to the ZHT KVS.

- We proposed a data-aware work stealing technique to optimize the goals of achiving both load balancing and data locality for distributed MTC scheduling. The technique is crucial in making the distributed achitecture more scalable than the centralized one, and in enabling the MTC paradigm scalable towards extreme scales.
- We developed the SimMatrix simulator of MTC task execution framework. Through SimMatrix, we explored the work stealing technique up to exascale of 1-million nodes, 1-billion cores, and 100-billion tasks, and identified the optimal parameter configurations that scale work stealing up to exascale. The optimal parameters are: *to steal half the number of tasks from the neighbors, to use the square root number of dynamic random neighbors, and to use a dynamic poll interval.* We also studied the data-aware work stealing technique through SimMatrix up to 128K-core scale, and showed that the technique is highly scalable. In addition, we devised an alytical model, showing that the achieved performance of the technique is close to the sub-optimal solution. These simulation studies are insightful, help us understand the scalability of the techniques, and guide us to implement real systems.
- We implemented a real task execution framework of many-task computing, MATRIX, which has applied the distributed scheduling architecture and the data-aware work stealing technique, and integrated the ZHT key-value store, for scheduling both the fine-grained MTC workloads and the Hadoop

workloads. We have deployed MATRIX on various platforms, compared MATRIX with other fine-grained task scheduling systems, and MATRIX showed huge performance gains over others. These results will encourage us to move towards the distributed designs on our way to approach extreme-scale computing.

- We proposed a partition-based distributed architecture, and resource stealing techniques to achieve dynamic resource balancing among all the partitions, for the resource management system of future exascale supercomputers. These distributed designs are similar to those of MATRIX, with the differences of targeting the HPC applications. Notice that MATRIX targets the MTC applications and big data application in clouds, has a fine-grained fully distributed architecture instead of a partion-based one, and focuses on balancing workload tasks instead of computing resources.
- We implemented both a real HPC workload manager named Slurm++ that is directly extended from the SLURM production system, and a simulator of Slurm++ named SimSlurm++. Slurm++ has implmented the partition-based distributed architecture and the resource balancing techniques, and has integrated the ZHT key-value store to keep resource and job metadata. We have evaluated Slurm++ by comparing it with SLURM, and Slurm++ showed up to 10X speedup of performance. More importantly, Slurm++ has been able to perform well for increasingly large jobs. We have also explored the scalability through SimSlurm++ up to 65K nodes, and results showed that Slurm++ has the potential to scale up to extreme scales.

8.2 Conclusions

We believe that the workloads of distributed systems are becoming diverse, including the traditional large-scale HPC application, the HPC ensemble runs, the MTC applications, as well as the big data applications across all the application domains, as the distributed systems are evolving towards extreme scales. These developing trends pose significant challenges on distributed system software, such as concurrency and locality, resilicence, memory and storage, energy and power, which need to be addressed sooner than later.

Resource management system is a vital system software for distributed systems. It is responsible for managing, monitoring, and allocating resources, and scheduling, launching and managing jobs. Current RMS are still designed around the decades-old centralized paradigm, thus not scalable towards extreme scales due to the capped processing capacity and single-point-of failure. There are urgent needs of developing next generation RMS that are signifticanly more scalable and fault tolerant. Distributed designs are the ways to overcome the shortcomings of the centralized ones, for both the tightly coupled HPC environment and the loosely coupled MTC and cloud computing envrionments. Challenges of distributed designs, such as maintaining failure/recovery, replication and consistency protocols, load balancing, data locality-aware scheduling, resource balancing, need to be studied in depth through both simulations and real implementations, in order to perform themselves better than the centralized ones and enable scalability up to extreme scales.

This dissertation has achieved the goal of *delivering scalable resource management system software that can manage the numerous computing resources of extreme-scale* *systems and efficiently schedule the workloads onto these resources for execution, in order to maintain high performance*. Our early work on the general system software taxonomy and KVS simulations is fundamental and necessary. The conclusions drawn from the early work are very meaningful and enlighting in guiding the designs and implementations of distributed RMS, such as KVS is a viable building block, fully distributed architecture is scalable under the circumstance where the client requrests dominate the communications. For distributed MTC scheduling, fine-grained fully distributed architecture is necessary with the challenges of achieving both load balancing and data-locality. The data-aware work stealing technique is scalable and can address both challenges in an optimized way. For distributed HPC scheduling, the partition-based distributed architecture is a great option, with the challenges of achieving resource balancing and gathering global state. The proposed weakly consistent monitoring-based resource stealing technique can address the challenges efficiently. We believe that the research contributions and conclusions of this dissertation are meaningful to the extreme-scale system software community.

8.3 Future Work

My future work plans to run MATRIX with more scientific applciations and Hadoop workloadss, and to study HPC data-aware scheduling through Slurm++.

We plan to integrate MATRIX with Swift to run more data-intensive scientific applciations. Swift [15] [67] is a parallel programming system and workflow engine for MTC applications. 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. Swift has been scaled to distribute up to 612 million dynamically load balanced tasks per second at scales of up to 262,144 cores [176]. This extreme scalability would absolutely advance the progress of making MATRIX supporting large-scale scientific application at extreme-scales. We also plan to use integrate MATRIX with the FusionFS [42] distributed file system to better support data-intensive applciations. FusionFS assumes that each compute node will have local storage, and all the file servers are writing the data files locally to eliminate the data movement overheads introduced by stripping the data files as the HDFS does. MATRIX will rely on FusionFS for scalable data management.

To run more Hadoop applications with MATRIX, we plan to integrate MATRIX with the Hadoop application layer, by developing interfaces between the Hadoop applications and MATRIX and exposing the same APIs as YARN does, and by developing interfaces between MATRIX with the HDFS and exposing the HDFS APIs to MATRIX, so that the Hadoop application codes can be run in MATRIX seamlessly.

We also believe that data-aware scheduling is important for data-intensive HPC appliatins. The storage systems of the HPC machines are involving to include high bandwidth and low latency storage devices (e.g. SSD, NVRAM), refer to burst buffer [187], in the I/O forwarding layer. The burst buffer enables overlapping the computations with I/O requests, which makes great contributions to reduce the runtime of data-intensive applications. However, the HPC resource managers are data ignorance, resulting in dwarfing the significance of the burst buffer. We plan to build models to quantify the overheads of scheduling with different layers of storage systems, and implement the models in Slurm++ to schedule data-intensive HPC workloads on the Trinity machine [189] with burst buffer hardware that will be deployed in Los Alamos National Laboratry.

8.4 Long-Term Impacts

We believe that the reserach contributions of this dissertation are both envoluntionary and revoluntionary. We have high hope that this dissertation will have long-term impacts on the extreme-scale system software commuty, and can lay the foundations of addressing the extreme-scale computing challenges through building scalable resource management system software that is able to effciently harness the extreme paralleslim and locality. The radical distributed architecture and scheduling techniques of the resource management system make extreme-scale computing more tractable, and will also open doors to many research directions, such as developing more inclusive taxonomies to address the challenges of the HPC system software at all layers of the software stack ; developing scalable key-value stores for HPC environments ; topology-aware scheduling; power-aware scheduling; many big data computing areas, including complicated parallel machine learning models [183], data compression mechanisms [184], erasure coding methods [20], in situ data processing frameworks [185] for efficient scientific visualization, and many others.

BIBLIOGRAPHY

[1] "Architectures and Technology for Extreme Scale Computing," Dept. Energy, Nat. Nuclear Security Administration, San Diego, CA, USA, 2009.

[2] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, 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," Sep. 28, 2008.

[3] M. A. Heroux, "Software challenges for extreme scale computing: Going from petascale to exascale systems, " *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 437–439, 2009.

[4] Department of Energy, "Top Ten Exascale Research Challenges," DOE ASCAC Subcommittee Report, Feb. 10, 2014.

[5] Department of Energy, "Exascale Operating System and Runtime Software Report," DOE Office of Science, Office of ASCR, Nov. 1, 2012.

[6] R. Wisniewski. (2014, Apr. 23). "HPC system software vision for exascale computing and beyond," in *Proc. Salishan Conf.* [Online]. Available: http://www.lanl.gov/conferences/salishan/salishan2014/Wisniewski.pdf

[7] X. Besseron and T. Gautier, "Impact of over-decomposition on coordinated checkpoint/rollback protocol," in *Proc. Int. Conf. Euro-Par Parallel Process. Workshops*, 2012, pp. 322–332.

[8] R. Rabenseifner, G. Hager, and G. Jost. (2013, Nov. 17). "Hybrid MPI and OpenMP parallel programming," *Int. Conf. High Perform. Comput., Netw., Storage Anal.* [Online]. Available: http:// openmp.org/sc13/HybridPP_Slides.pdf

[9] Argonne Nat. Lab. (2015, May 16). *ZeptoOS project* [Online]. Available: http://www.zeptoos.org/

[10] Sandia Nat. Lab. (2015, June 2). *Kitten Lightweight Kernel* [Online]. Available: https://software.sandia.gov/trac/kitten

[11] L. Kale and A. Bhatele, *Parallel science and engineering applications: The Charm++ approach*, Boca Raton, FL, USA: CRC Press, 2013.

[12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, p. 66.

[13] H. Kaiser, T. Heller, B. A. Lelbach, A. Serio, and D. Fey, "HPX—A task based programming model in a global address space," in *Proc. 8th Int. Conf. Partitione Global Address Space Programm. Models*, 2014, p. 6.

[14] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thoms, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard template adaptive parallel library," in *Proc. 3rd Annu. Haifa Experimental Syst. Conf.*, 2010, p. 14.

[15] J. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. Foster, "Swift/T: Large-scale application composition via distributed-memory data flow processing," in *Proc. 13th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2013, pp. 95−102.

[16] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX an advanced parallel execution model for scaling-impaired applications," in *Proc. Int. Conf. Parallel Process. Workshops*, 2009, pp. 394–401.

[17] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger, "ARMI: A high level communication library for STAPL," *Parallel Process. Lett.*, vol. 16, no. 2, pp. 261–280, 2006.

[18] J. Lidman, D. Quinlan, C. Liao, and S. A. McKee, "Rose: Fttransform-a source-to-source translation framework for exascale fault- tolerance research," in *Proc. 42nd Int. Conf. Dependable Syst. Netw. Workshops*, 2012, pp. 1–6.

[19] M. Kulkarni, A. Prakash, and M. Parks. (2015, Mar. 13). *Semantics-rich libraries for effective exascale computation or SLEEC* [Online]. Available: https://xstackwiki.modelado.org/SLEEC

[20] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. 1st Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 328–338.

[21] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[22] K. Wang, X. Zhou, H. Chen, M. Lang, and I. Raicu, "Next generation job management systems for extreme scale ensemble computing," in *Proc. 23rd Int. Symp. High Perform. Distrib. Comput.*, 2014, pp. 111–114.

[23] K. Wang, A. Rajendran, and I. Raicu. (2013, Aug. 23). *MATRIX: Many-task computing execution fabRIc at eXascale* [Online]. Available: http://datasys.cs.iit.edu/reports/2013_matrix_TR.pdf

[24] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proc. IEEE Int. Conf. BigData*, 2014, pp. 119–128.

[25] I. Raicu, I. T. Foster, and P. Beckman, "Making a case for distributed file systems at exascale," In *Proceedings of the third international workshop on Large-scale system and application performance*, San Jose, California, USA, 2011, pp. 11–18.

[26] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang, and I. Raicu. (2012, Nov. 17). "Paving the road to exascale with many-task computing," in *Proc. IEEE/ACM Supercomput. Doctoral Showcase* [Online]. Available: http://datasys.cs.iit.edu/publications/2012_SC12-MTC-paper.pdf

[27] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale systems services," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, p. 9.

[28] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, 1983.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *Sigcomm. Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, 2001.

[30] M. Jette, A. Yoo, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Proc. 9th Int. Workshop Job Scheduling Strategies Parallel Process.*, 2003, pp. 44–60.

[31] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan," Scalable I/O forwarding frame- work for high-performance computing systems," in *Proc. Int. Conf. Cluster Comput. Workshops*, 2009, pp. 1–10.

[32] A. Vishnu, A. Mamidala, J. Hyun-Wook, and D. K. Panda, "Performance modeling of subnet management on fat tree infiniband networks using openSM," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp.Workshop*, 2005.

[33] P. Roth, D. Arnold, and B. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proc. ACM/ IEEE Supercomput.* , 2003, p. 21.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM Symp. Oper. Syst. Principles*, 2007, pp. 205–220.

[35] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[36] A. Feinberg. (2013, Apr. 14). "Project voldemort: Reliable distributed storage," *Int. Conf. Data Eng.* [Online]. Available: http://www.devoxx.com/display/Devoxx2K10/Project+Voldemort+Reliable+distributed+storage

[37] M. A. Heroux. (2013, Sept. 16). *Toward resilient algorithms and applications* [Online]. Available: http://www.sandia.gov/rvmaherou/docs/HerouxTowardResilientAlgsAndApps.pdf

[38] I. Raicu, I. Foster, M. Wilde, Z. Zhang, K. Iskra, P. Beckman, Y. Zhao, A. Szalay, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, and D. Thain, "Middleware support for many-task computing," *Cluster Comput.*, vol. 13, no. 3, pp. 291–314, Sep. 2010.

[39] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Proc. IEEE Workshop Many-Task Comput. Grids Supercomput. Workshop*, 2008, pp. 1–11.

[40] I. Raicu, "Many-task computing: Bridging the gap between high- throughput computing and high-performance computing," Ph.D. dissertation, Univ. Chicago, Chicago, IL, 2009.

[41] I. Raicu, P. Little, I. Foster, C. M. Moretti, Y. Zhao, A. Chaudhary, A. Szalay, and D. Thain, "Towards data intensive many-task computing," in *Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management*, Hershey, PA: IGI Global Publishers, 2009.

[42] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "FusionFS: Towards supporting data- intensive scientific applications on extreme-scale high-performance computing systems," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 61–70.

[43] R. M. Fujimoto, "Parallel discrete-event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30-53, Oct. 1999.

[44] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 654–663.

[45] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004.

[46] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proc. 1st Int. Conf. Simul. Tools Techn. Commun., Netw. Syst. Workshops*, 2008, p. 60.

[47] I. Baumgart, B. Heep, and S. Krause, "Oversim: A flexible overlay network simulation framework," in *Proc. IEEE Global Internet Symp.*, 2007, pp. 79–84.

[48] T. Vignaux and K. Muller. (2015, May 26). *Simpy: Documentation* [Online]. Available: http: //simpy.sourceforge.net/SimPyDocs/index.html

[49] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. 9th Int. Conf. Peer-to-Peer*, 2009, pp. 79–84.

[50] K. Wang, K. Brandstatter, and I. Raicu, "SimMatrix: Simulator for many-task computing execution fabric at exascale," in *Proc. Symp. High-Perform. Comput.*, p. 9.

[51] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 775–787.

[52] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[53] I. Raicu, I. Foster, Y. Zhao, P. Little, C. Moretti, A. Chaudhary, and D. Thain, "The quest for scalable support of data intensive workloads in distributed systems," in *Proc. 18th Int. Symp. High Perform. Distrib. Comput.*, 2009, pp. 207–216.

[54] T. Tuan, A. Dinh, G. Theodoropoulos, and R. Minson, "Evaluating large scale distributed simulation of p2p networks," in *Proc. 12th IEEE/ACM Int. Symp. Distrib. Simul. Real-Time Appl.*, 2008, pp. 51–58.

[55] I. Diane, I. Niang, and B. Gueye, "A hierarchical DHT for fault tolerant management in P2P-SIP networks," in *Proc. IEEE 16th Int. Conf. Parallel Distrib. Syst.*, 2010, pp. 788–793.

[56] B. Ghit, F. Pop, and V. Cristea, "Epidemic-style global load monitoring in large-scale overlay networks," in *Proc. Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.*, 2010, pp. 393–398.

[57] M. Rahman, W. Golab, A. Young, K. Keeton, and J. Wylie, "Toward a principled framework for benchmarking consistency," in *Proc. 8th USENIX Conf. Hot Topics Syst. Dependability*, 2012, p. 8.

[58] E. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes, " *IEEE Commun. Surveys Tuts.* vol. 7, no. 2, pp. 72–93, Second Quarter 2005.

[59] K. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan, "Measurement, modeling, and analysis of a peer-to- peer file-sharing workload," in *Proc. 19th ACM Symp. Symp. Oper. Syst. Principles*, 2003, pp. 314–329.

[60] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *J. Grid Comput.*, vol. 3, nos. 3/4, pp. 171–200, 2005.

[61] A. Vishnu, A. R. Mamidala, H. W. Jin, and D. K. Panda, "Performance modeling of subnet management on fat tree infiniband networks using opensm," in *Proc. 19th IEEE Int. Symp. Parallel Distrib. Process.*, 2005, p. 296.

[62] P. M. Kogge and T. J. Dysart, "Using the TOP500 to Trace and Project Technology and Architecture Trends," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2011, pp. 1-11.

[63] A. Szalay, J. Bunn, J. Gray, I. Foster and I. Raicu, "The Importance of Data Locality in Distributed Computing Applications," *NSF Workflow Workshop*, 2006.

[64] M. Snir, S. Otto, S. H. Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, Volume 1: The MPI Core, 2nd ed. Cambridge, MA: MIT Press., 1994.

[65] D. Zhao, D. Zhang, K. Wang, and I. Raicu, "RXSim: Exploring Reliability of Exascale Systems through Simulations," in *Proc. 21st High Performance Computing Symposia*, Bahia Resort, San Diego, CA, 2013, pp. 1-9.

[66] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward loosely coupled programming on petascale systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2008, pp. 1-12.

[67] Y. Zhao, M. Hategan, B. Clifford, I. T. Foster, G. Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," in *Proc. IEEE Workshop on Scientific Workflows*, 2007.

[68] D. Abramson, B. Bethwaite, C. Enticott, S. Garic, and T. Peachey, "Parameter Space Exploration Using Scientific Workflows," in *Proc. of the 9th Int. Conference on Computational Science*, Springer-Verlag Berlin, Heidelberg, 2009.

[69] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *J. Scientific Programming*, vol. 13, pp. 219-237, July 2005.

[70] M. Livny. (2011, July 17). *The Condor DAGMan (Directed Acyclic Graph Manager)* [online]. Available: http://www.cs.wisc.edu/condor/dagman

[71] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, Kevin Glover, Mattew R. Pockck, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *J. Bioinformatics*, vol. 20, Nov. 2004.

[72] I. Taylor, M. Shields, I. Wang, and A. Harrison, "Visual Grid Workflow in Triana," *J. Grid Computing*, vol. 3, pp. 153-169, Sep. 2005.

[73] I. Altintas, C. Berkley, E. Jaeger, M. Johes, B. Ludascher and S. Mock, "Kepler: An Extensible System for Design and Execution of Scientific Workflows," in *Proc. of the 16th Int. Conference on Scientific and Statistical Database Management*, 2004, p. 423.

[74] G. V. Laszewski, M. Hategan, and D. Kodeboyina, "Java CoG Workflow," in *Proc. of Workflows for eScience*, 2007, pp. 340-356.

[75] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proc. of the 2nd ACM European Conference on Computer Systems*, 2007, pp. 59-72.

[76] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters," in *Proc. of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004, pp. 10-10.

[77] G. Zheng, E. Meneses, A. Bhatele and L. V. Kale, "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers," in *Proc. of the 39th Int. Conference on Parallel Processing Workshops*, 2010, pp. 436-444.

[78] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2007, pp. 1-12.

[79] M. A. Jette and D. Auble, "SLURM: Resource Management from the Simple to the Sophisticated," *Lawrence Livermore Nat. Lab., SLURM User Group Meeting*, Oct., 2010.

[80] J. Frey, T. Tannenbaum, M. Livny, I. Foster and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *J. Cluster Computing*, vol. 5, pp. 237-246, 2002.

[81] K. Ousterhout, P. Wendell, M. Zaharia and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proc. of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 69-84.

[82] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, and I. Raicu, "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing," in *Proc. of the 14th EEE/ACM Int.Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 404-413.

[83] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," *J. Concurrency and Computation: Practice and Experience – Grid Performance*, vol. 17, pp. 323-356, 2005.

[84] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson, "The portable batch scheduler and the maui scheduler on linux clusters," in *Proc. of the 4th annual Linux Showcase & Conference*, 2000, pp. 27-27.

[85] W. Gentzsch, "Sun Grid Engine: Towards Creating a Compute Power Grid," in *Proceedings of the 1st Int. Symposium on Cluster Computing and the Grid*, 2001, p. 35.

[86] V. Kumar, A. Y. Grama and N. R. Vempaty, "Scalable load balancing techniques for parallel computers," *J. Parallel and Dist. Comput.*, vol. 22, no. 1, pp. 60-79, 1994.

[87] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy and J. Nieplocha, "Scalable work stealing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2009, pp. 1-11.

[88] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: A Generic Framework for Large-Scale Distributed Experiments," in *Proc. of the 10th Int. Conference on Computer Modeling and Simulation*, 2008, pp. 126-131.

[89] V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Trans. on Commun.*, vol. 22, pp. 637-648, 1974.

[90] R. L. Kruse, J. L. Stone, K. Beck and E. O'Dougherty, *Data Structures & Program Design second edition,* Prentice-Hall: Inc. div. of Simon & Schuster, 1984, p. 150.

[91] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, K. Iskra, P. Beckman, and I. Foster, "Extreme-scale scripting: Opportunities for large task parallel applications on petascale computers," *J. Physics: Conference Series*, 2009.

[92] L. V. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataramanz, L. Wesolowski, and G. Zheng, "Charm++ for Productivity and Performance," *A Submission to the 2011 HPC Class II Challenge*, Nov. 2011, pp. 11-49.

[93] C. L. Dumitrescu, I. Raicu, and I. Foster, "Experiences in running workloads over grid3," in *Proc. of the 4th international conference on Grid and Cooperative Computing*, 2005, pp. 274-286.

[94] Y. Zhao, X. Fei, I. Raicu, and S. Lu, "Opportunities and Challenges in Running Scientific Workflows on the Cloud," in *Proc. of the Int. Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery,* 2011, pp. 455-462.

[95] Google. (2015, Feb. 18). *Google Protocol Buffer* [online]. Available: https://developers.google.com/protocol-buffers/

[96] K. Wang, Z. Ma, and I. Raicu, "Modelling Many-Task Computing Workloads on a Petaflop IBM Blue Gene/P Supercomputer," in *Proc. 2nd Int. Workshop on Workflow Models, Systems, Services and Applications in the Cloud*, 2013, pp. 11-20.

[97] P. E. Ross. (2012, Apr. 16). "Why CPU Frequency Stalled," *IEEE Spectrum* [online]. Available: http://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled

[98] G. E. Moore, "Cramming more components onto integrated circuits," *Book of Readings in computer architecture*, San Francisco, CA: Morgan Kaufmann Publishers Inc., 2000.

[99] Intel. (2014, Dec. 12). *Intel MIC* [online]. Available: http://hothardware.com/Reviews/Intel-Announces-MIC-Xeon-Phi-Aims-For-Exascale-Computing/

[100] NVIDIA. (2012, July 20). *NVIDIA GPU* [online]. Available: http://www.nvidia.com/object/gpuventures.html

[101] L. V. Kale, "Comparing the performance of two dynamic load distribution methods," in *Proc. Conf on Parallel Processing*, 1988, pp. 8-12.

[102] W. Shu, and L. V. Kale, "A dynamic scheduling strategy for the Chare-Kernel system," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 1989, pp. 389-398.

[103] A. B. Sinha, and L. V. Kale, "A load balancing strategy for prioritized execution of tasks," in *Proc. of the 7th Int. Parallel Processing Symposium*, Apr. 1993, pp. 230-237.

[104] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. on Parallel and Dist. Systems*, vol. 4, no. 9, pp. 979-993, Sept. 1993.

[105] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. of the ACM conference on Programming language design and implementation*, 1998, pp. 212-223.

[106] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proc. of the 8th USENIX conference on networked systems design and implementation*, 2011, pp. 295-308.

[107] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. of the 8th ACM European Conference on Computer Systems*, 2013, pp. 351-364.

[108] Amazon. (2013, Oct. 18). *Amazon Simple Queue Service* [online]. Available: http://aws.amazon.com/documentation/sqs/

[109] R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *J. Concurrency and Computation: Practice and Experience*, vol. 14, no. 13, pp. 1175-1220, 2002.

[110] W. Kreutzer, J. Hopkins, and M. V. Mierlo, "SimJAVA—a framework for modeling queueing networks in Java," in *Proc. of the 29th conference on Winter simulation*, 1997, pp. 483-488.

[111] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and Evaluation of the GeMTC Framework for GPU-enabled Many-Task Computing," in *Proc. of the 23rd Int. symposium on High-performance parallel and distributed computing*, 2014, pp. 153-164.

[112] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: yet another resource negotiator," in *Proc. of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1-16.

[113] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay, "Accelerating large-scale data exploration through data diffusion," in *Proc. of the Int. workshop on Data-aware distributed computing*, 2008, pp. 9-18.

[114] Argonne Nat. Lab. (2011, Jan. 28). *Cobalt* [online]. Available: http://trac.mcs.anl.gov/projects/cobalt

[115] J. D. Ullman, "NP-complete scheduling problems," *J. Computer and System Sciences*, vol. 10, no. 3, pp. 384-393, June, 1975.

[116] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proc. of the 7th annual ACM symposium on Parallelism in algorithms and architectures*, 2005, pp. 21 – 28.

[117] K. Shvachko, H. Huang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, May, 2010, pp. 1-10.

[118] A. Bialecki. (2013, Mar. 13). *Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware* [online]. Available http://lucene.apache.org/hadoop/

[119] J. Banks, J. Carson, B. Nelson and D. Nicol, "Discrete-event system simulation", 4th ed.: Pearson, 2005.

[120] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "PRObE: A thousand-node experimental cluster for computer systems research," *J. Usenix login*, vol. 38, no. 3, pp. 37-39, 2013.

[121] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, "All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 1, pp. 33-46, 2010.

[122] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *Proc. of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 341-342.

[123] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proc. of the IEEE Int. conference on Parallel & Distributed Processing Symposium (IPDPS)*, 2009, pp. 1-12.

[124] J. Paudel, O. Tardieu and J. Amaral, "On the merits of distributed work-stealing on selective locality-aware tasks," in *Proc. of the 42nd Int. Conference on Parallel Processing*, 2013, pp. 100-109.

[125] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *Proc. of ACM Conference on Object-oriented Programming Systems Languages and Applications*, 2005, pp. 519–538.

[126] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, and M. Wilde, "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments," *Grid Computing Research Progress*: Nova Publisher, 2008.

[127] S. A. Huettel, A. W. Song, and G. McCarthy, *Functional Magnetic Resonance Imaging*, 2nd ed. Massachusetts: Sinauer, 2009.

[128] J. V. Horn, J. Dobson, J. Woodward, M. Wilde, Y. Zhao, J. Voeckler, and I. Foster. "Grid-Based Computing and the Future of Neuroscience Computation," *Methods in Mind*: MITPress, 2006.

[129] N. Liu and C. D. Carothers, "Modeling Billion-Node Torus Networks Using Massively Parallel Discrete-Event Simulation," *in Proceedings of the IEEE Workshop on Principles of Advanced and Distributed Simulation*, 2011.

[130] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: a high-performance, low memory, modular time warp system," in *Proc. of the 4th workshop on Parallel and distributed simulation*, 2000.

[131] H. C. Lin and C. S. Raghavendra, "An approximate analysis of the join the shortest queue (JSQ) policy," *IEEE Trans. on Parallel and Dist. Systems*, vol. 7, no. 3, pp. 301-307, 1996.

[132] M. Harchol-Balter, "Job placement with unknown duration and no preemption," *ACM SIGMETRICS Performance Evaluation Review*, vol. 28, no. 4, pp. 3-5, 2001.

[133] E. Bachmat and H. Sarfati, "Analysis of size interval task assignment policies," *ACM SIGMETRICS Performance Evaluation Review*, vol., no. 2, pp. 107-109, 2008.

[134] M. Furuichi, K. Taki, and N. Ichiyoshi, "A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi," in *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[135] J. Liander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proc. of the Int. symposium on High-performance parallel and distributed computing*, 2012.

[136] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *Proc. of the Int. Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[137] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proc. of the Int. Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[138] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proc. of the 6th annual ACM Symposium on Principles of distributed computing*, 1987.

[139] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA: O'Reilly & Associates, Inc, 2007.

[140] K. Wang, X. Zhou, K. Qiao, M. Lang, B. McClelland, and I. Raicu, "Towards Scalable Distributed Workload Manager with Monitoring-Based Weakly Consistent Resource Stealing," in *Proc. of the 24th Int. symposium on High-performance parallel and distributed computing*, 2015.

[141] E. L. Goh. (2014, Nov. 19). *SC14 HPC matters Plenary* [online]. Available http://www.hpcwire.com/2014/11/19/hpc-matters/

[142] A. Basermann, and K. Solchenbach, "Ensemble Simulations on highly Scaling HPC Systems (EnSIM)," in *Proc. of the Competence in High Performance Computing*, Schwetzingen, Deutschland, June 2010.

[143] H. Simon, T. Zacharia, R. Stevens, Modeling and Simulation at the Exascale for Energy and the Environment," *Report on the Advanced Scientific Computing Research Town Hall Meetings on Simulation and Modeling at the Exascale for Energy, Ecological Sustainability and Global Security*, 2007.

[144] T. L. Harris, K. Fraser, and I. A. Pratt, "A Practical Multi-Word Compare-and-Swap Operation," in *Proc. of the 16th International Symposium on Distributed Computing*, 2002, pp. 265-279.

[145] D. Feitelson. (2014, Jan. 28). *Logs of Real Parallel Workloads from Production Systems* [online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html

[146] E. Frachtenberg, F. Petrini, J. Fernández, and S. Pakin, "Storm: Scalable resource management for large-scale parallel computers," *IEEE Trans. on Comp.*, vol. 55, no. 12, pp. 1572-1587, 2006.

[147] E. Hendriks, "BProc: The Beowulf distributed process space," in *Proc. of the 16th international conference on Supercomputing*, June 2002, pp. 129-136.

[148] J.D. Goehner, D.C. Arnold, D.H. Ahn, G. L. Lee, B. R. Supinsk, M.P. LeGendre, B.P. Miller, and M. Schulz, "LIBI: A Framework for Bootstrapping Extreme Scale Software Systems," *J. Parallel Computing*, vol. 39, no. 3, pp. 167-176, 2013.

[149] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur, "PMI: A scalable parallel process-management interface for extreme-scale systems," in *Proc. of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, 2010, pp. 31-41.

[150] M. Karo, R. Lagerstrom, M. Kohnke, C. Albing, "The application level placement scheduler," *Cray User Group*, 2006, pp. 1-7.

[151] Intel. (2014, Nov. 21). *ORCM: Open Resilience Cluster Manager* [online]. Available: https://github.com/open-mpi/orcm

[152] Argoone Nat. Lab. (2014, Sept. 23). *PETSc* [online]. Available: http://www.mcs.anl.gov/petsc/

[153] PBS Professional. (2014, Dec. 16). *PBS Professional 13.0* [online]. Available: http://www.pbsworks.com/pdfs/PBSPro_13.0_Whats_new_SC14_letter_103014_WEB.pdf

[154] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. of the European Conference on Computer Systems*, 2015.

[155] J. L. Stone, K. Beck, and E. O'Dougherty, *Data Structures & Program Design*, 2nd ed, Englewood Cliffs, New Jersey: Prentice-Hall, Inc. div. of Simon & Schuster, 1984, p. 150.

[156] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed.: The MIT Press, 2009.

[157] Oracle. (2011, May 16). *TreeSet* [online]. Available: http://download.oracle.com/javase/6/docs/api/java/util/TreeSet.html

[158] R. Barr, Z. J. Haas, and R. V. Renesse, "JiST: an efficient approach to simulation using virtual machines," *Softw. Pract. Exper.* vol. 35, no. 6, pp. 539-576, May 2005.

[159] K. Brandstatter, T. Li, X. Zhou, and I. Raicu, "NoVoHT: a Lightweight Dynamic Persistent NoSQL Key/Value Store," *2nd Greater Chicago Area System Research Workshop*, 2013.

[160] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets, " in *Proc. the 2nd USENIX conference on Hot topics in cloud computing*, 2010, p. 10.

[161] Yahoo! Labs. (2014, Dec. 21). *Yahoo weather* [online]. Available: http://labs.yahoo.com/news/researchers-explain-the-science-powering-yahoo-weather/

[162] L. A. Barroso, J. Dean, U. Holzle, "Web search for a planet: The Google cluster architecture," *Micro, IEEE*, vol. 23, no. 2, pp. 22-28, Apr. 2003.

[163] W. S. Coats, V. L. Feeman, J. G. Given, and H. D. Rafter, "Streaming into the Future: Music and Video Online," *Loy. LA Ent. L. Rev*. vol. 20, no. 4, pp. 285-308, 2000.

[164] Z. Stone, T. Zickler, and T. Darrell, "Autotagging facebook: Social network context improves photo annotation," in *Proc. of the Computer Vision and Pattern Recognition Workshops*, 2008.

[165] F. Darema, "Dynamic data driven applications systems: A new paradigm for application simulations and measurements," in *Proc. of the 4th Int. Conference on Computational Science,* Springer Berlin Heidelberg, 2004, pp. 662-669.

[166] H. Ligang, S.A. Jarvis, D.P. Spooner, D. Bacigalupo, T. Guang, and G. R. Nudd. "Mapping DAG-based applications to multiclusters with background workload," in *Proc. of the IEEE Int. Symposium on Cluster Computing and the Grid*, 2005, pp. 855-862.

[167] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," in *Proc. of the ACM/IEEE Conference on Supercomputing*, 2004, p. 4.

[168] B. Zhang, D. Yehdego, K.L. Johnson, M.Y. Leung, and M. Taufer, "Enhancement of Accuracy and Efficiency for RNA Secondary Structure Prediction by Sequence Segmentation and MapReduce," *J. BMC Structural Biology*, vol. 13, no. 1, 2013.

[169] A. Raj, K. Kaur, U. Dutta, V. V. Sandeep, and S. Rao, "Enhancement of Hadoop Clusters with Virtualization Using the Capacity Scheduler," in *Proc. of the 3rd International Conference on Services in Emerging Markets,* 2012, pp. 50-57.

[170] Apache. (2014, Dec. 21). *Apache Hadoop 1.2.1 Documentation* [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html

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

[172] A. Rasooli and D. G. Down, "A Hybrid Scheduling Approach for Scalable Heterogeneous Hadoop Systems," in *Proc. of the High Performance Computing, Networking Storage and Analysis*, 2012.

[173] A. Rasooli and D. G. Down, "An adaptive scheduling algorithm for dynamic heterogeneous Hadoop systems," in *Proc. of the Conference of the Center for Advanced Studies on Collaborative Research*, 2011.

[174] M. Zaharia, D. Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of the 5th European conference on Computer systems*, 2010, pp. 265-278.

[175] C. Dumitrescu, I. Raicu, and I. Foster, "The Design, Usage, and Performance of GRUBER: A Grid uSLA-based Brokering Infrastructure," *J. Grid Computing*, 2007.

[176] T. G. Armstrong, J. M. Wozniak, M. Wilde, and Ian T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in *Proc. of the ACM/IEEE Conference on Supercomputing*, 2014.

[177] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proc. of the ACM Symposium on Operating Systems Principles*, 2009, pp. 261-276.

[178] G. Mackey, S. Sehrish, and W. Jun, "Improving metadata management for small files in HDFS," in *Proc. of the IEEE International Conference on Cluster Computing and Workshops,* 2009, pp. 1-4.

[179] X. Zhao, Y. Yang, L. L. Sun, and H. Huang. "Metadata-Aware small files storage architecture on Hadoop," in *Proc. of the Int. conference on Web Information Systems and Mining,* 2012.

[180] E. M. Estolano and C. Maltzahn, "Haceph: Scalable Meta- data Management for Hadoop using Ceph," *Post Session at the 7th USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[181] S. A. Weil, S. A. Brandt, E. L. Miller, D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proc. of the 7th symposium on Operating systems design and implementation*, 2006.

[182] IBM. (2012, Jan. 28). *LSF* [online]. Available: http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/index.html

[183] T. Condie, P. Mineiro, N. Polyzotis, and M. Weimer, "Machine learning for big data," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2013.

[184] A. H. Baker, H. Xu, J. M. Dennis, M. N. Levy, D. Nychka, S. A. Mickelson, J. Edwards, M. Vertenstein, and A. Wegener, "A methodology for evaluating the impact of data compression on climate simulation data," in *Proc. of the 23rd international symposium on High-performance parallel and distributed computing*, 2014.

[185] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova, "Scalable in situ scientific data encoding for analytical query processing," in *Proc. of the 22nd international symposium on High-performance parallel and distributed computing*, 2013.

[186] White House, "Big Data Research and Development Initiative," Office of Science and Technology Policy (OSTP), 2012.

[187] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proc. of IEEE 28th Symposium on Mass Storage Systems and Technologies*, 2012, pp. 1-11.

[188] A. Thusoo, J. S. Sarma, N. Jain, S. Zheng, P. Chakka, Z. Ning, S. Antony, L. Hao, and R. Murthy, "Hive - a petabyte scale data warehouse using Hadoop," in *Proc. of the IEEE 26th International Conference on Data Engineering*, 2010, pp. 996-1005.

[189] Los Alamos Nat. Lab. (2015, Apr. 9). *Trinity machine* [online]. Available: http://www.lanl.gov/projects/trinity/

[190] Argoone Nat. Lab. (2015, May 26) *ADLB: Asynchronous Dynamic Load Balancing* [online]. Available: http://www.mcs.anl.gov/project/adlb-asynchronous-dynamic-load-balancer

[191] Lawrence Livermore Nat. Lab. (2015, May 26). *Sierra | Computation* [online]. Available: http://computation.llnl.gov/computers/sierra

[192] Oak Ridge Nat. Lab. (2015, May 26). *SUMMIT: Scale new heights. Discover new solutions* [online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2014/11/Summit_FactSheet.pdf

[193] S.L. Chu and C.C. Hsiao, "OpenCL: Make Ubiquitous Supercomputing Possible," in *Proc of the 12th IEEE International Conference on High Performance Computing and Communications*, 2010, pp. 556-561.

[194] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13-27, 2008.

[195] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46-55, 1998.

[196] X. Jian, J. Sartori, H. Duwe, and R. Kumar, "High Performance, Energy Efficient Chipkill Correct Memory with Multidimensional Parity," *Computer Architecture Letters*, vol. 12, no. 2, pp. 39-42, 2013.

[197] B. Obama. (2009). "A Strategy for American Innovation: Driving Towards Sustainable Growth and Quality Jobs", *National Economic Council* [online], Available: http://www.whitehouse.gov/administration/eop/nec/StrategyforAmericanInnovation/

[198] S. L. Min, "Flash memory solid state disks," in *Proc. of the International Conference on Information and Emerging Technologies*, 2010, pp. 1-3.

[199] H. Li and Y. Chen, "An overview of non-volatile memory technology and the implication for tools and architectures," in *Proc of the Europe Conference on Design, Automation, Test and Exhibition*, 2009, pp. 731-736.

[200] *Top500*. (2015, May 29). *The Top500 List* [online]. Available: http://www.top500.org/blog/lists/2013/06/press-release/

[201] R. Ghosh, F. Longo, F. Frattini, S. Russo, and K. S. Trivedi, "Scalable Analytics for IaaS Cloud Availability," *IEEE Transactions on Cloud Computing*, vol. 2, no. 1, pp. 57-70, 2014.

[202] J. Rake-Revelant, O. Holschke, P. Offermann, and U. Bub, "Platform-as-a-Service for business customers," in *Proc. of 14th International Conference on the Intelligence in Next Generation Networks*, 2010, pp. 1-6.

[203] J. Gao, P. Pattabhiraman, X. Bai, and W. T. Tsai, "SaaS performance and scalability evaluation in clouds," in *Proc. of the IEEE 6th International Symposium on Service Oriented System Engineering*, 2011, pp. 61-71.

[204] IBM, "Overview of the ibm blue gene/p project," *IBM Journal of Research and Development*, vol. 52, nos. 1.2, pp. 199 –220, 2008.

[205] Google. (2015, May 27). *Goolge Search Statistics* [online]. Available: http://www.internetlivestats.com/google-search-statistics/.

[206] Forbes. (2015, May 28). [online]. Available: http://www.forbes.com/sites/hengshao/2014/11/11/9-3-billion-sales-recorded-in-alibabas-24-hour-online-sale-beating-target-by-15/, 2015.

[207] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, Inc., 2006.

[208] D. Bader and R. Pennington, "Cluster Computing: Applications," *Georgia Tech College of Computing*, 1996.

[209] H. Miyazaki, Y. Kusano, H. Okano, T. Nakada, K. Seki, T. Shimizu, N. Shinjo, F. Shoji, A. Uno, and M. Kurokawa, "K computer: 8.162 PetaFLOPS massively parallel scalar supercomputer built with over 548k cores," *Solid-State Circuits Conference Digest of Technical Papers*, 2012.

[210] F. Magoules, J. Pan, K. A. Tan, and A. Kumar, *Introduction to Grid Computing* 1st ed. Boca Raton, FL: CRC Press, Inc., 2009.

[211] I. Foster and E. C. Kesselman, “Chapter 2: Computational Grids," *The Grid: Blueprint for a Future Computing Infrastructure*,: Morgan Kaufmann Publishers, 1999.

[212] C. Catlett, et al. "TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications," *Advances in Parallel Computing*, vol. 16, pp. 225-249, 2006.

[213] F. Gagliardi, "The EGEE European Grid Infrastructure Project," in *Proc. of the 6th International Conference on High Performance Computing for Computational Science*, 2005, pp. 194-203.

[214] H. Jin, "ChinaGrid: making grid computing a reality," in *Proc. of the 7th international Conference on Digital Libraries: international collaboration and cross-fertilization*, 2004.

[215] H. Subramoni, L. Ping, R. Kettimuthu, and D.K. Panda, "High Performance Data Transfer in Grid Environment Using GridFTP over InfiniBand," in *Proc. of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 557-564.

[216] G. Aloisio, D. Conte, C. Elefante, G.P. Marra, G. Mastrantonio, and G. Quarta, "Globus Monitoring and Discovery Service and SensorML for Grid Sensor Networks," in *Proc. of the 15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2006, pp. 201-206.

[217] M. Rambadt and P. Wieder, "UNICORE-Globus interoperability: getting the best of both worlds," in *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 422.

[218] S. Krause, "Tutorial: Hands on Introduction to Amazon Web Services," in *Proc. of the IEEE/ACM 6th International Conference on Utility and Cloud Computing*, 2013.

[219] L. Zheng, L. OBrien, R. Ranjan, and M. Zhang, "Early Observations on Performance of Google Compute Engine for Scientific Computing," in *Proc. of the IEEE 5th International Conference on Cloud Computing Technology and Science* 2013, pp. 1-8.

[220] E. Roloff, F. Birck, M. Diener, A. Carissimi, and P.O.A Navaux, "Evaluating High Performance Computing on the Windows Azure Platform," in *Proc. o fthe IEEE 5th International Conference on Cloud Computing*, 2012, pp. 803-810.

[221] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 Spot Instance Pricing," in *Proc. of the IEEE 3rd International Conference on Cloud Computing Technology and Science*, 2011, pp. 304-311.

[222] M. Malawski, M. Kuźniar, P. Wójcik, and M. Bubak, "How to Use Google App Engine for Free Computing," *IEEE Internet Computing*, vol.17, no.1, pp. 50-59, 2013.

[223] C. L. Petersen, "New product development: salesforce and cycle time," in *Innovation in Technology Management - The Key to Global Leadership, Proc. of the Portland Int. Conference on Management and Technology*, 1997, p.478.

[224] Workday. (2015, May 29). *Workday* [online]. Available: http://www.workday.com/

[225] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proc. of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[226] P. Carns, W. B. Ligon, R. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters, " in *Proc. of the Extreme Linux Track: 4th Annual Linux Showcase and Conference*, Oct. 2000.

[227] K. Wang, N. Liu, I. Sadooghi, X. Yang, X. Zhou, M. Lang, Xian-He Sun, and I. Raicu, "Overcoming Hadoop Scaling Limitations through Distributed Task Execution," in *Proc. of the IEEE Int. Conference on Cluster Computing,* 2015.

[228] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Exploring the Design Tradeoffs for Extreme-Scale High-Performance Computing System Software," in *Proc. of the IEEE Trans. on Parallel and Dist. Systems*, vol. PP, no. 99, pp. 1, 2015.

[229] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, "A Dynamically Scalable Cloud Data Infrastructure for Sensor Networks," in *Proc. of the 6th Workshop on Scientific Cloud Computing*, 2015.

[230] K. Wang, and I. Raicu, "Scheduling Data-intensive Many-task Computing Applications in the Cloud," in *the NSFCloud Workshop on Experimental Support for Cloud Computing*, 2014.

[231] K. Wang and Ioan Raicu. (2014, Oct. 7). "Towards Next Generation Resource Management at Extreme-Scales," *Ph.D. Comprehensive Exam, Computer Science Department, Illinois Institute of Technology* [online]. Available: http://datasys.cs.iit.edu/publications/2014_IIT_PhD-proposal_Ke-Wang.pdf

[232] A. Kulkarni, K. Wang, and M. Lang. (2012, Aug. 17). "Exploring the Design Tradeoffs for Exascale System Services through Simulation," *Summer Student Research Workshop at Los Alamos National Laboratory* [online]. Available: https://newmexicoconsortium.org/component/com_jresearch/Itemid,146/id,31/task,show/view,publication/

[233] K. Wang, J. Munuera, I. Raicu, and H. Jin. (2011, Aug. 23). *Centralized and Distributed Job Scheduling System Simulation at Exascale* [online]. Available: http://datasys.cs.iit.edu/~kewang/documents/summer_report.pdf

[234] X. Zhou, H. Chen, K. Wang, M. Lang, and I. Raicu. (2013, Dec. 15). *Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System* [online]. Available: http://datasys.cs.iit.edu/reports/2013_IIT-CS554_dist-slurm.pdf

[235] K. Ramamurthy, K. Wang, and I. Raicu. (2013, Dec. 15). *Exploring Distributed HPC Scheduling in MATRIX* [online]. Available: http://www.cs.iit.edu/~iraicu/teaching/CS554-F13/best-reports/2013_IIT-CS554_MATRIX-HPC.pdf

[236] K. Wang and I. Raicu. (2014, May 23). "Achieving Data-Aware Load Balancing through Distributed Queues and Key/Value Stores," in *the 3rd Greater Chicago Area System Research Workshop* [online]. Available: http://datasys.cs.iit.edu/reports/2014_GCASR14_paper-data-aware-scheduling.pdf