MATRIX: MANY-TASK COMPUTING EXECUTION FABRIC FOR EXTREME SCALES

BY

ANUPAM RAJENDRAN

DEPARTMENT OF COMPUTER SCIENCE

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

Approved _____
Adviser

Chicago, Illinois
May 2013

ACKNOWLEDGEMENT

It gives me a great pleasure to thank all those people who have helped me and contributed for the successful completion of this work.

First of all, I would like to thank my advisor Dr. Ioan Raicu for giving me the opportunity to work under him on this project. The course I took under him triggered my passion in the field of distributed systems. His constant support and encouragement throughout my two years of graduate school has made my thesis possible. I also thank him for his patience, expertise and allowing me to work independently which eased me throughout my thesis.

I would like to thank my parents Mr. Rajendran and Mrs. Sowdhamini Rajendran as well as my sister Mrs. Sheetal Balajee who have constantly supported me in boosting my morale. I dedicate this work to them.

I would also to thank Dr. Lan and Dr. Sun for everything they taught me about Distributed Systems through their invaluable courses that broadened my knowledge.

I want to thank Dr. Lan for being a member of the thesis examining committee.

I am also grateful to my lab members Tonglin Li, Ke Wang and Dongfang Zhao for their advice, contributions and helping me patiently throughout my thesis. It has been a pleasure working with them.

Finally, I would like to thank the Almighty for being on my side all the time and showing me the right path.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Definition
θ	Complexity
m	Number of nodes in the system
k	kth neighbor to be chosen at random
p_r	Probability that a neighbor chosen already is chosen again
E_c	Expected number of times a neighbor is chosen
k_{max}	Maximum dynamic number of neighbors

ABSTRACT

Scheduling large amount of jobs/tasks over large-scale distributed systems play a significant role to achieve high system utilization and throughput. Today's state-of-the-art job management/scheduling systems have predominantly Master/Slaves architectures, which have inherent limitations, such as scalability issues at extreme scales (e.g. petascales and beyond) and single point failures. In designing the next-generation job management system that addresses both of these limitations, we argue that we must distribute the job scheduling and management; however, distributed job management introduces new challenges, such as non-trivial load balancing.

This thesis proposes an adaptive work stealing technique to achieve distributed load balancing at extreme scales, those found in todays' petascale systems towards tomorrow's exascale systems. This thesis also presents the design, analysis and implementation of a distributed execution fabric called MATRIX (MAny-Task computing execution fabRIc at eXascales). MATRIX utilizes the adaptive work stealing algorithm for distributed load balancing and distributed hash tables for managing task metadata. MATRIX supports both high-performance computing (HPC) and many-task computing (MTC) workloads. We have validated it using synthetic workloads up to 4K-cores on a IBM BlueGene/P supercomputer. Results show that high efficiencies (e.g. 90%+) are possible with certain workloads. We study the performance of MATRIX in depth, including understanding the network traffic generated by the work stealing algorithm. Simulation results are presented up to 1M-node scales which show that work stealing is a scalable and efficient load balancing approach for many-core architectures to extreme-scale distributed systems.

CHAPTER 1

INTRODUCTION

The goal of job scheduling system is to efficiently manage the distributed computing power of workstations, servers, and supercomputers in order to maximize job throughput and system utilization. With the dramatically increase of the scales of today's distributed systems, it is urgent to develop efficient job schedulers. Predictions are that by the end of this decade, we will have exascale system with millions of nodes and billions of threads of execution [1]. Unfortunately, today's schedulers have centralized Master/Slaves architecture (e.g. Slurm [2], Condor [3][4], PBS [5], SGE [6]), where a centralized server is in charge of the resource provisioning and job execution. This architecture has worked well in grid computing scales and coarse granular workloads [63], but it has poor scalability at the extreme scales of petascale systems with fine-granular workloads [13][29].

One approach to mitigate the issues arising from centralized architectures is to distribute the centralized job manager in either hierarchical or fully distributed architectures. Although this addresses potential single point of failures, and increases the overall performance of the scheduling system, issues can arise in load balancing work across all schedulers and compute nodes.

Load balancing is the technique of distributing workloads evenly across processors of a parallel machine, or across nodes of a supercomputer, so that no single processor or computing node is overloaded. Although extensive research about load balancing has been done with centralized or hierarchical methods, we believe that distributed load balancing techniques are potential approaches to extreme scale. This

work adopts work stealing [7][8][9][10] to achieve distributed load balancing, where the idle processors steal tasks from the heavily-loaded ones. There are several parameters affecting the performance of work stealing, such as the number of tasks to steal, the number of neighbors of a node from whom it can steal tasks, static/dynamic neighbors, and the polling interval. We explore these parameters through a light-weight job scheduling system simulator, SimMatrix [11]. We explore work stealing as an efficient method for load balancing jobs/tasks across a variety of systems, from 1000-core many-core processors with 2D/3D mesh interconnect, to a billion-core exascale system. We also explore the performance of work stealing in a real system, MATRIX, at scales of 1K-nodes and 4K-cores.

This work is motivated by the Many-Task Computing (MTC) paradigm [12] [13][58][62] which bridges the gap between High Performance Computing (HPC) and High Throughput Computing (HTC). MTC was defined in 2008 to describe a class of applications that did not fit easily into the categories of traditional HPC or HTC. Many MTC applications are structured as graphs of discrete tasks, with explicit input and output dependencies forming the graph edges. In many cases, the data dependencies will be files that are written to and read from a file system shared between the compute resources; however, MTC does not exclude applications in which tasks communicate in other manners. MTC applications often demand a short time to solution, may be communication intensive or data intensive [59]. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled.

The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large.

For many applications, a graph of distinct tasks is a natural way to conceptualize the computation. Structuring an application in this way also gives increased flexibility. For example, it allows tasks to be run on multiple different supercomputers simultaneously; it simplifies failure recovery and allows the application to continue when nodes fail, if tasks write their results to persistent storage as they finish; and it permits the application to be tested and run on varying numbers of nodes without any rewriting or modification. Examples of MTC systems are various workflow systems (e.g. Swift [14][61][64], Nimrod [15], Pegasus [16], DAGMan [17], BPEL [18], Taverna [19], Triana [20], Kepler [21], CoG Karajan [22], Dryad [23]). Other examples of MTC are MapReduce systems (e.g. Google's MapReduce [24], Yahoo's Hadoop [25], Sector/Sphere [26]), and distributed run-time systems such as Charm++ [27], ParalleX [28]. Finally, light-weight task scheduling systems also fit in this category for enabling MTC applications (e.g. Falkon [29] , Condor GlideIns [4] , Coaster [30], Sparrow [31]).

The rest of the chapter explains the Many-Task Computing (MTC) paradigm and also discusses the different challenges at exascale computing followed by the contributions to this thesis and its outline.

1.1 Many-Task Computing

Many-Task Computing (MTC) was introduced by Raicu et al. [12][13] in 2008 to describe a class of applications that did not fit easily into the categories of traditional high-performance computing (HPC) or high-throughput computing (HTC). Many MTC applications are structured as graphs of discrete tasks, with explicit input and output

dependencies forming the graph edges. In many cases, the data dependencies will be files that are written to and read from a file system shared between the compute resources; however, MTC does not exclude applications in which tasks communicate in other manners.

MTC applications often demand a short time to solution, may be communication intensive or data intensive, and may comprise of a large number of short tasks. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. For many applications, a graph of distinct tasks is a natural way to conceptualize the computation. Structuring an application in this way also gives increased flexibility. For example, it allows tasks to be run on multiple different supercomputers simultaneously; it simplifies failure recovery and allows the application to continue when nodes fail, if tasks write their results to persistent storage as they finish; and it permits the application to be tested and run on varying numbers of nodes without any rewriting or modification.

The hardware of current and future large-scale HPC systems, with their high degree of parallelism and support for intensive communication, is well suited for achieving low turnaround times with large, intensive MTC applications. Hardware and software for MTC must be engineered to support the additional communication and I/O, must minimize task dispatch overheads, queue management, and support resource management at finer granularity (e.g. at the core level, or node level, as opposed to the partition level). The MTC paradigm has been defined and built with the scalability of

tomorrow's systems as a priority and can address many of the HPC shortcomings at extreme scales.

1.2 Challenges at Exascales

The era of many-core and exascale computing will bring new fundamental challenges in how we build computing systems and their hardware, how we manage them, and how we program them. 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 four most significant challenges of exscale computing are: Energy and Power; Memory and Storage; Concurrency and Locality; Resiliency. Any one of these challenges, if left unaddressed, could halt progress towards exascale computing.

The Energy and Power challenge is the most pervasive of the four, which refers to the ability to keep the power consumption at a reasonable level, so that the cost to power of a system does not dominate the cost of ownership. The DARPA Exascale report [42] defined probably the single most important metric, namely the energy per flop. Given the energy consumption of current state-of-the-art technologies which uses 12.7MW of power, the increase in performance by 100X (to reach exascales), and the upper cap of 20MW of power for a single supercomputer, we can conclude that we need to reduce the energy per flop by 50X to 100X to make exascale computing viable.

The Memory and Storage challenge refers to optimizing and minimizing data movement through the memory hierarchy (e.g. persistent storage, solid state memory, volatile memory, caches, and registers). Exascales 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), 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.

The Concurrency and Locality challenge refers to how we will harness the many magnitude orders of increased parallelism fueled by the many-core computing era, and minimize the data movements among billions of threads of execution. The largest supercomputers have increased in parallelism at an alarming rate. Many have said that the "free ride" software had for many decades, has finally come to a halt, and a new age is upon us which paints a bleak picture unless revolutionary progress is made in the entire computing stack.

The Resilience challenge refers to the capability of making both the infrastructure (hardware) and applications fault tolerant in face of a decreasing mean-time-to-failure (MTTF). In order to achieve exascales, revolutionary advancements must be made in the programming paradigm. A more abstract and modern programming paradigm could allow parallelism to be harnessed with greater ease, as well as making applications fault tolerant diminishing the effects of the decreasing system MTTF.

The MTC paradigm can address four of the five major challenges of exascale computing, namely concurrency, resilience, memory and storage, and heterogeneity.

1.3 Contributions

The main contributions of this thesis are as follows:

1. Design and implement MATRIX (a distributed execution fabric for MTC workloads at extreme scales) to support distributed task scheduling and management through an adaptive work stealing algorithm
2. Evaluate the functionality of MATRIX using different workload types such as Bag of Tasks, Fan-In DAG, Fan-Out DAG, Pipeline DAG and Complex Random DAG
3. Performance evaluation of MATRIX, including analyzing the behavior of the load balancing algorithm and network traffic involved at scales of 64 nodes up to 1024 nodes for synthetic workloads of fine to medium granularity (from 64ms to 8 seconds per task)
4. Validate and compare the performance of MATRIX with SimMatrix (a simulator implementing the work stealing algorithm), and with Falkon (a light weight task execution framework that supports both a centralized and hierarchical architecture)

1.4 Thesis Outline

The thesis is organized in several chapters as follows: Chapter 2 describes the background information and related work about job scheduling systems, load balancing, and work stealing. In Chapter 3 we present the design and implementation of the adaptive work stealing algorithm in a real system MATRIX. We show the evaluation and

experimental results in Chapter 4. Conclusions are drawn and future work is envisioned in Chapter 5.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter covers essential background information on the topics of clusters, grids, supercomputers, high throughput computing and high performance computing. This chapter also discusses related work in two main areas: 1) job scheduling, and 2) distributed load balancing. Then we present the adaptive work stealing algorithm and give a brief overview of the simulator SimMatrix.

2.1 Clusters, Grids, and Supercomputers

A computer cluster is a collection of computers, connected together by some networking fabric, and is composed of commodity processors, network interconnects, and operating systems. Clusters are usually aimed to improve performance and availability over that of a single computer; furthermore, clusters are typically more cost-effective than a single computer of comparable speed or availability. Middleware such as MPI allows cluster computing to be portable to a wide variety of cluster architectures, operating systems, and networking offering high performance computing over commodity hardware. [32] High throughput computing [33] has also seen good success on clusters, as the needed computations are more loosely coupled and most scientists can be satisfied by commodity CPUs and memory, essentially making high efficiency not playing a major role.

Grids tend to be composed of multiple clusters, and are typically loosely coupled, heterogeneous, and geographically dispersed. The term “the Grid” was coined in the mid-

1990s to denote a proposed distributed computing infrastructure for advanced science and engineering [34]. Foster et al. describes the definition of Grid Computing to be about large scale resource sharing, innovative applications, and high performance computing. It is meant to offer flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources, namely virtual organizations. Some examples of grids are TeraGrid [35], Open Science Grid (OSG) [36], and Enabling Grids for E-sciencE (EGEE) [37]. Some of the major grid middleware are the Globus Toolkit [38] and Unicore [39]. Grids have also been used for both HPC and HTC, just as clusters have; HPC is more challenging for grids as resources can be geographically distributed which can increases latency significantly between nodes, but it can still be done effectively with careful tuning for some HPC applications.

A supercomputer is a highly-tuned computer clusters using commodity processors combined with custom network interconnects and typically customized operating system. The term supercomputer is rather fluid, as today's supercomputer usually ends up being an ordinary computer within a decade. Figure 1 shows the Top500 [40] trends in the fastest supercomputers for the past fifteen years, and projecting out for the next decade.

We have currently surpassed the petaflop/s rating, and it is predicted that we will surpass an exaflop/s within the next decade. Much of the predicted increase computing power comes from the prediction of increasing the number of cores per processor (see Figure 1), which is expected to be in the hundreds to thousands within a decade [41]. Until recently, supercomputers have been strictly HPC systems, but more recently they have gained support for HTC as well.

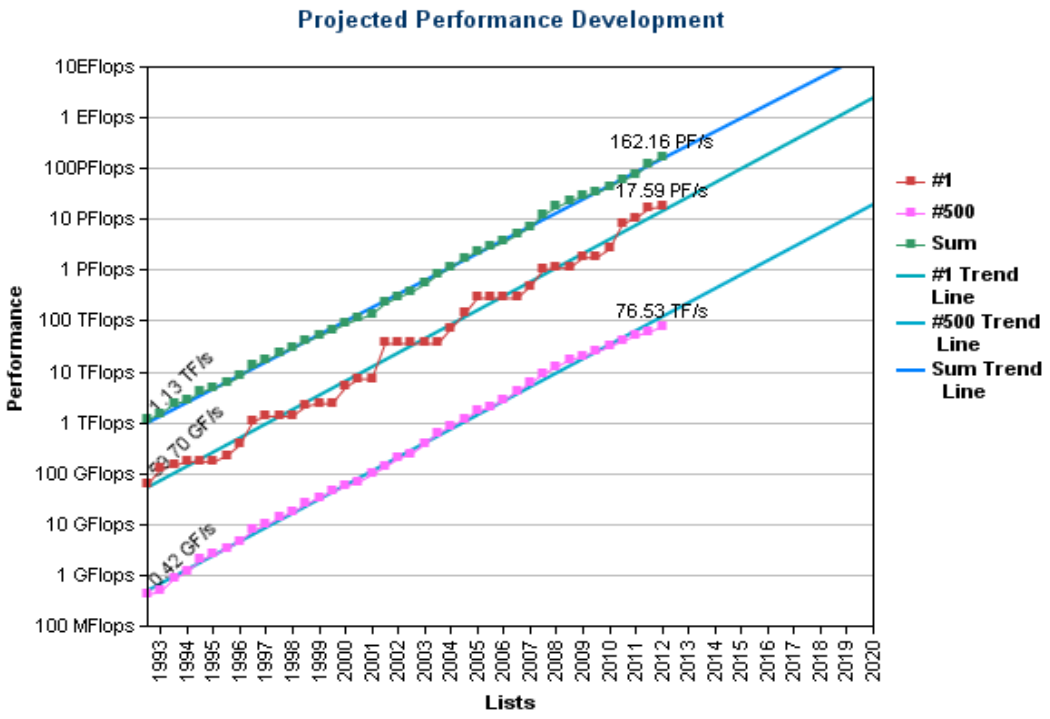

Figure 1. Projected Performance of Top500, November 2012

2.2 Related Work

The job schedulers could be centralized, where a single dispatcher manages the job submission, and job execution state updates; or hierarchical, where several dispatchers are organized in a tree-based topology; or distributed, where each computing node maintains its own job execution framework. The University of Wisconsin developed one of the earliest job schedulers, Condor [3], to harness the unused CPU cycles on workstations for long-running batch jobs. Slurm [2] is a resource manager designed for Linux clusters of all sizes. It allocates exclusive and/or non-exclusive access to resources to users for some duration of time so they can perform work, and provides a framework for starting, executing, and monitoring work on a set of allocated nodes. Portable Batch System (PBS) [5] 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 [43] is the load-sharing and batch-queuing component of a set of workload management tools from Platform Computing of Toronto. All these systems target as the HPC or HTC applications, and lack the granularity of scheduling jobs at node/core level, making them hard to be applied to the MTC applications. What's more, the centralized dispatcher in these systems suffers scalability and reliability issues. In 2007, a light-weight task execution framework, called Falkon [29] was developed. Falkon also has a centralized architecture, and although it scaled and performed magnitude orders better than the state of the art, its centralized architecture will not even scale to petascale systems [13]. A hierarchical implementation of Falkon was shown to scale to a petascale system in [13], the approach taken by Falkon suffered from poor load balancing under failures or unpredictable task execution times.

Although distributed load balancing at extreme scales is likely a more scalable and resilient solution, there are many challenges that must be addressed (e.g. utilization, partitioning). Fully distributed strategies have been proposed, including neighborhood averaging scheme (ACWN) [44][45][46][47]. In [47], several distributed and hierarchical load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model (GM) and a Hierarchical Balancing Method (HBM). Other hierarchical strategies are explored in [46]. Charm++ [27] supports centralized, hierarchical and distributed load balancing. In [27], 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.

Work stealing has been used at small scales successfully in parallel languages such as Cilk [48], to load balance threads on shared memory parallel machines [8][9][10].

Theoretical work has proved that a work-stealing scheduler can achieve execution space, time, and communication bounds all within a constant factor of optimal [8][9]. However, the scalability of work stealing has not been well explored on modern large-scale systems. In particular, concerns exist that the randomized nature of work stealing can lead to long idle times and poor scalability on large-scale clusters [10]. The largest studies to date of work stealing have been at thousands of cores scales, showing good to excellent efficiency depending on the workloads [10].

2.3 Adaptive Work Stealing Algorithm

We present the adaptive work stealing algorithm, which applies dynamic multiple random neighbor selection and adaptive poll interval techniques. These adaptive natures of the work stealing algorithm are critical to achieve good scalability and efficiency.

2.3.1 Dynamic Multi-Random Neighbor Selection. In work stealing, the selection of neighbors from which an idle node could steal tasks could be static or dynamic/random. In dynamic case, traditional work stealing randomly selects one neighbor to steal tasks [10]. Choosing a single neighbor to steal could yield poor performance at extreme scales. We propose a multiple random neighbor selection strategy, which randomly selects several neighbors instead of one, and chooses the most heavily loaded neighbor to steal tasks. We identify the optimal number of neighbors for both static and dynamic selection. The multiple-random neighbor selection algorithm is given in Algorithm 1.

Algorithm 1. Dynamic Multi-Random Neighbor Selection for Work Stealing

DYN-MUL-SEL(num_neigh, num_nodes)

```
1.  let selected[num_nodes] be boolean array initialized false except the node itself
2.  let neigh[num_neigh] be array of neighbors
3.  for i = 1 to num_neigh
4.          index = random () % num_nodes
5.          while selected[index]  do
6.                  index = random() % num_nodes
7.          end while
8.          selected[index] = true
9.          neigh[i] = node[index]
10. end for
11. return neigh
```

When a node is idle, it randomly selects several neighbors from its membership list to communicate for stealing work. The time complexity of Algorithm 1 is Θ(n), where n is the number of neighbors. The proof is as follows:

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 be chosen is chosen again is: $p_r = \frac{k-1}{m}$. So, Let *i* be the number of times for selecting the kth neighbor, the actual number of times is: $i \times (p_r)^{i-1} \times (1 - p_r)$. $\mathrm{i} \times (\mathrm{p_r})^{\mathrm{i-1}} \times (1\text{-}\mathrm{p_r})$

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

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

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 $p_r = \frac{k_{max}-1}{\mathrm{m}} \approx \frac{k_{max}}{m} = \frac{1}{1000}$. So, for E_c , after $i = 3$,

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

which is negligible. So, we just need to consider $i = 1$, and $= 2$,

$$E_c \approx 1 \times \left(\frac{1}{1000}\right)^0 \times \left(1 - \frac{1}{1000}\right) + 2 \times \left(\frac{1}{1000}\right)^1 \times \left(1 - \frac{1}{1000}\right) \approx 1 \qquad (3)$$

So, the time complexity to choose n neighbors is: Θ(n).

2.3.2 Adaptive Poll Interval. When a node fails to steal tasks from the selected neighbors because all selected neighbors have no more tasks, or fails to steal tasks when the most heavily loaded neighbor told that it had tasks, but which had already been executed at the time when actual stealing happens, the node waits for a period of time and then it does stealing again. We call this wait time the poll interval.

We implement an adaptive poll interval policy in order to achieve reasonable performance while still keeping the work stealing algorithm responsive. Without this policy, we observed that under idle conditions, many nodes would poll neighbors to do work stealing, which would ultimately fail and would lead to more work stealing requests. If the polling interval was set large enough to limit the number of work steal events, work stealing would not respond quickly to change conditions, and lead to poor load balancing. Therefore, we change the poll interval of a node dynamically by doubling it each time when work stealing fails; and setting the poll interval back to the default small value whenever the node steals some tasks successfully. This technique is similar to the

exponential back-off approach in the TCP networking protocol [49]. We set the default poll interval to be small value (e.g. 1 ms). The specification of the adaptive work stealing algorithm is given in Algorithm 2. Whenever a node has no tasks in its task waiting queue, it signals the adaptive work stealing algorithm, which first randomly selects several neighbors using the Algorithm 1, and then selects the most heavily loaded neighbor to steal tasks. If work stealing fails, the node would double the poll interval and wait for that period of time, after which the node tries to do work stealing again. This procedure continues until the node finally successfully steals tasks from a neighbor, and at which point, it sets the poll interval back to the initial small value (e.g. 1 sec).

At the beginning, just one node (id = 0) has tasks, all the other nodes signal work stealing. Let's say we have m nodes, and each one talks to n neighbors, so within logn (m) steps, ideally the tasks should be distributed across all the nodes. At the very end, if there are just a few tasks left in the system, the work stealing doesn't help much, instead it would introduce more communication overhead. One way to solve this problem is to set an upper bound of the poll interval. After reaching the upper bound, the node would stop doing work stealing.

Algorithm 2. Adaptive Work Stealing Algorithm

ADA-WORK-STEALING(num_neigh, num_nodes)

1. Neigh = *DYN-MUL-SEL (num_neigh, num_nodes)*
2. most_load_node = Neigh[0]
3. for i = 1 to num_neigh
4. if most_load_node. load < Neigh[i]. load then

```
5.                  most_load_node = Neigh[i]
6.          end if
7.  end for
8.  if most_load_node.load = 0 then
9.          sleep (poll_interval)
10.         poll_interval = poll_interval * 2
11.         ADA-WORK-STEALING(num_neigh, num_nodes)
12. else
13.         steal tasks from most_load_node
14.         if num_task_steal = 0 then
15.                 sleep (poll_interval)
16.                 poll_interval = poll_interval * 2
17.                 ADA-WORK-STEALING(num_neigh, num_nodes)
18.         else
19.                 poll_interval = 1
20.                 return
21.         end if
22. end if
```

2.4 SimMatrix

SimMatrix is a scalable discrete event simulator for MTC execution fabric, which enables the exploring of work stealing at exascales. The design, implementation and resource requirements are presented in [11]. SimMatrix simulates the exact behavior as mentioned in the implementation of MATRIX above except that it does not support task dependency. So it can simulate only the Bag of Tasks type workloads, and also has just one ready queue and the other two are not required. SimMatrix supports both centralized (best case scenario) and distributed scheduling (worst case scenario). Whenever a worker node has no tasks in the ready queue, it triggers work stealing for load balancing.

The work [57] presents the investigation of adaptive work stealing algorithm at exascale levels through simulations. Through the SimMatrix simulator, a wide range of parameters important to understanding work stealing is explored at up to exascale levels, such as number of tasks to steal number of neighbors of a node, and static/dynamic neighbors. Experiment results show that adaptive work stealing configured with optimal parameters could scale up to 1 million nodes and 1 billion cores, while achieving 85%+ efficiency running on real MTC workload traces obtained from a 17-month period on a petascale supercomputer. It provides evidence that work stealing is a scalable method to achieve distributed load balancing, even at exascales with millions of nodes and billions of cores. Some of the results are presented in the evaluation section for comparing MATRIX with SimMatrix. MATRIX is validated against Falkon and the simulator SimMatrix and the results show 90% efficiency for synthetic workloads. However, more work is needed to scale up the prototype many orders of magnitude needed to help validate and prove the simulation results of SimMatrix.

CHAPTER 3

DESIGN AND IMPLEMENTATION OF MATRIX

This work investigates the usability of work stealing towards exascale levels of parallelism. There are several parameters which could affect the performance of work stealing to achieve load balancing, such as steal tasks from global space or just some neighbors, how to select neighbors, how many number of neighbors a node could have, how many tasks to steal, and the length of waiting time after which a node signals work stealing again, if the node fails to steal tasks from others. Previous work done using SimMatrix [11] investigated the search for an ideal set of parameters (e.g. worker's connectivity, number of tasks to steal, static/dynamic neighbors) needed to make work stealing a viable and efficient distributed load balancing mechanism. Through simulations, it was concluded that work stealing with the right parameters, would work well at exascale levels of millions of nodes, billions of cores, and hundreds of billions of tasks. This work seeks to validate the work stealing algorithm by implementing it, using the optimal parameters suggested in previous works, in a real system called MATRIX, a distributed execution fabric for MTC workloads at exascales.

MATRIX is a distributed many-Task computing execution framework, which utilizes the adaptive work stealing algorithm to achieve distributed load balancing. MATRIX uses ZHT (a distributed zero hop key-value store) [50] for job metadata management, to submit tasks and monitor the task execution progress. We have a functional prototype implemented in C/C++, and have scaled this prototype on a BG/P supercomputer up to 1024-nodes (4K-cores) with good results.

3.1 Architecture

The components of MATRIX and the communication signals among all the components are shown in Figure 2. For the purpose of evaluation, there are two kinds of components, the client and the compute node. The client is just a benchmarking tool that issues request to generate a set of tasks to be executed on a cluster of nodes. The benchmarking tool has a task dispatcher for which allows the client to submit workload to the compute nodes. A compute node can also be referred as worker node and can be used interchangeably. Each compute node has a task execution unit along with a NoSQL data store for managing the metadata of every task. The task execution unit is the core component of MATRIX and the data store is possible through ZHT.

3.2 Design and Implementation

The current version of MATRIX supports synthetic workloads. Essentially, it can be a set of "sleep" tasks. MATRIX also supports task dependency. This means the order of execution among the tasks in the workload can be specified as a part of task description and MATRIX would execute the tasks in the right order. For example the workload can be a *Directed Acyclic Graph (DAG)* [54] where each node in the DAG is a task and the edges among the nodes specify the dependency. This can be easily translated to give the order of execution among tasks similar to topological sort. The system is tested for different types of workloads such as Bag of Tasks, Fan-In DAG, Fan-Out Dag, Pipeline DAG and a complex random DAG. These different DAGs are shown in Figure 4.

Initially at the time of booting, each compute node records its identity and location information to a membership list which can be read by all other compute nodes

and the client. This membership list ensures that there is an N-N communication possible among the compute nodes.

Upon request from the client, the task dispatcher initializes the workload of given type and submits the workload to the one or more compute nodes. With the help of ZHT, the task dispatcher could submit tasks to one arbitrary node, or to all the nodes in a balanced distribution. The compute nodes execute the tasks in the given order. In the background all compute nodes distribute the workload among themselves until the load gets balanced using the adaptive work stealing algorithm. The client periodically monitors the status of workload until all the tasks present in the workload gets executed by the compute nodes.

ZHT records the instantaneous information of every task and this information is distributed across all the compute nodes. Every time when a task is moved from one compute node to other due to work stealing, this information is updated instantly. Thus task migration can be considered as an atomic process that involves updating ZHT followed by the actual movement of task from one compute node to other. So the client can look up the status information of any task by performing a “lookup” operation on ZHT.

3.3 Types of Messages

There are different kinds of messages caused by the work stealing algorithm as shown in Figure 2.

ZHT Insert: The metadata of tasks such as task-id, task-description, task submission time etc. are inserted into ZHT before submitting the actual tasks for execution.

MATRIX Insert: After the metadata of all tasks is stored in ZHT, then the workload can be submitted to the queue of execution unit to execute the workload.

ZHT Lookup: This provides an interface to retrieve the existing information from ZHT. For instance, it can be used by the execution unit to check for a given task, if all the dependency conditions are met so that the task is ready to be executed or to get the task description when the task is about to be executed.

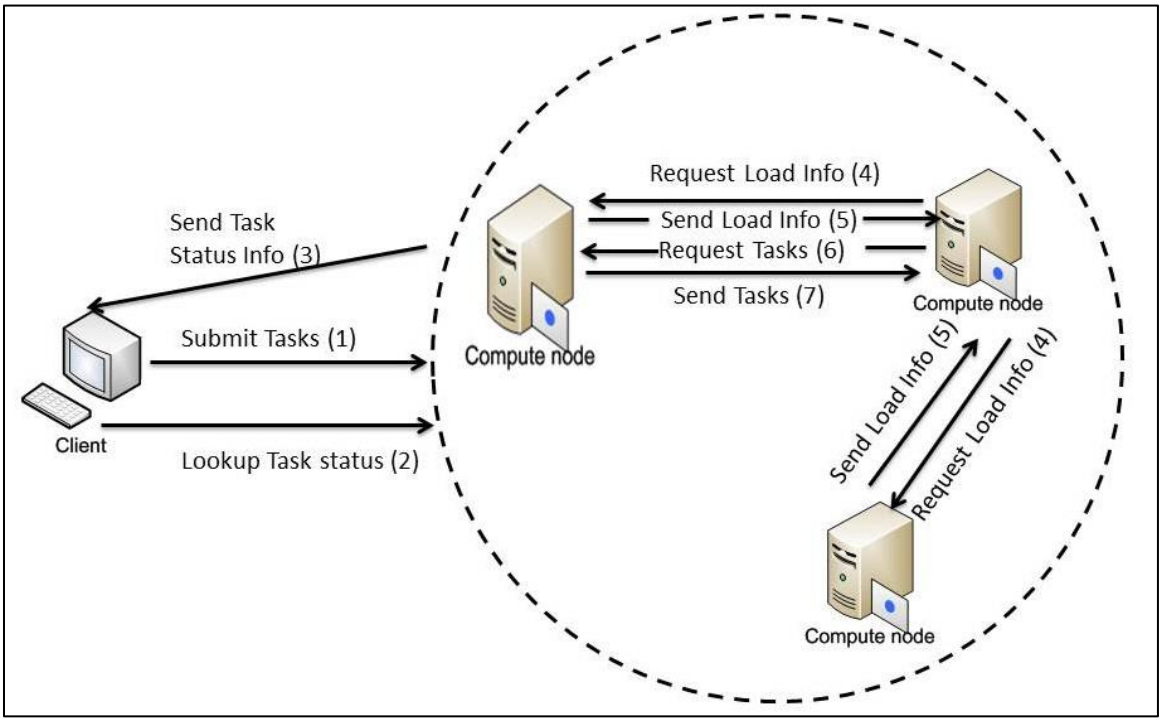

Figure 2. MATRIX components and communications among components

ZHT Update: If any part of metadata of the task existing in ZHT needs to modified, then this API can be used to update the given field. For instance after a task has finished its execution, the execution unit can signal all the children tasks that were waiting for this task to finish its execution. This is also useful, to update the current node information of a task, when it is migrating to a different compute node due to work stealing.

Load Information: Idle nodes can poll a subset of compute nodes for knowing the load on all those nodes.

Work Stealing: After getting the load information of neighboring nodes, the idle node can pick the node with maximum load and send a request to steal tasks from that node. Then the chosen node will send some of its load to the requested node.

Client Monitoring: The client periodically monitors the system utilization and the rates of completion of task execution.

If the system is configured to run only Bag of Tasks that have no dependency criteria, then no ZHT operations are involved. The only operations to be performed are MATRIX Insert, Load Information, Work Stealing and Client Monitoring. This is because for running of Bag of Tasks, the system can be optimized for executing the workload by storing only the essential information directly in the MATRIX queue and hence cutting down all ZHT operations. This difference can be seen in the results show in the evaluation section.

3.4 Task Assignment

Broadly, MATRIX supports two types of task assignment: best case and worst case assignment. The architectures are shown in Figure 3. For simplicity, the ids of all nodes are represented as consecutive integer numbers ranging from 0 to the number of nodes N-1.

In the best case situation (Figure 3 left part), the dispatcher initializes the tasks and submit them one by one to the compute nodes in round-robin fashion. This is possible due to the hashing mechanism in ZHT that maps each task to a compute node based on the task-id. This is the best case situation in terms of system utilization because

the hashing function of ZHT does most of the load balancing. Another way to realize this situation is to have as many dispatchers as compute nodes and divide the total tasks among the dispatchers so that there is 1:1 mapping between a task dispatcher and a compute node. Then let each task dispatcher submit the tasks to corresponding compute node. Here work stealing is useful only at the end of the experiment, when there are very few tasks left to be executed, and they are concentrated at only few compute nodes.

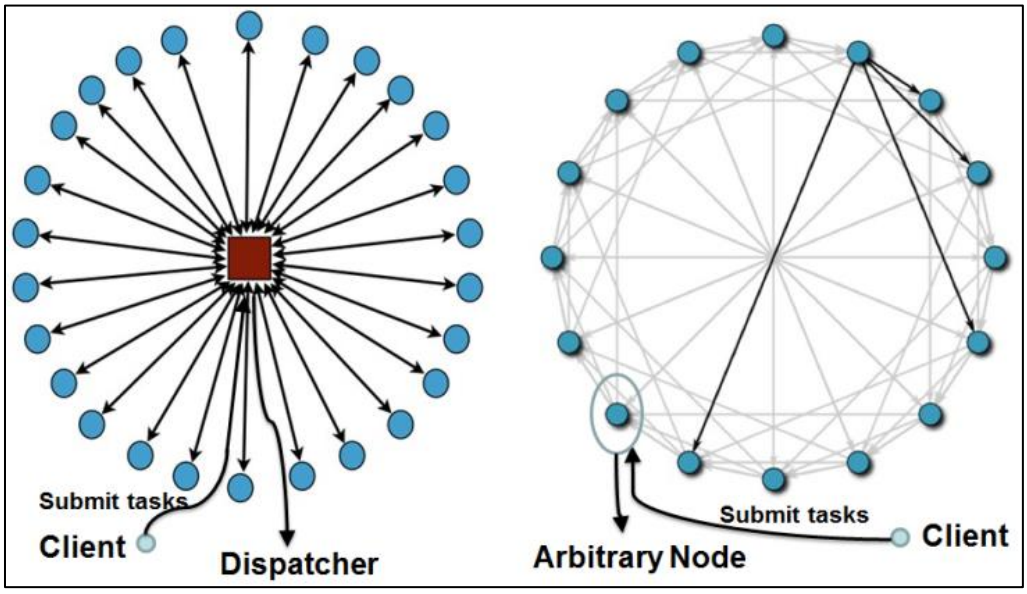

Figure 3. MATRIX architecture for both best case and worst case scheduling

In the worst case situation (Figure 3 right part) there is only one dispatcher which initializes all tasks into a single package and submits the package to a single arbitrary compute node. This is the worst case situation in terms of system utilization because the entire load is on a single compute node. Thus work stealing thread runs from the start to ensure that all the tasks get quickly distributed among all compute nodes evenly to reduce the time for completing the execution of a workload. Thus it generates considerable network traffic for initial load balancing when compared to the best case situation. Then throughout the experiment the network traffic caused by work stealing reduces as the

system has converged with evenly distribution of workload. It increases again at the end of experiment similar to the best case situation when there are very few tasks left to be executed which might be concentrated at only few compute nodes and needs to be balanced evenly among all other compute nodes.

The system can also be configured for tuning the number of dispatchers. It can be equal to square root, or logarithm-base-2 of number of compute nodes. The best case and worst case situation can thus be treated as special cases. The greater the number of dispatchers the faster the load gets distributed evenly among the workers.

3.5 Execution Unit

The worker running on every compute node maintains three different queues: a wait queue, a ready queue and a complete queue. The wait queue is used to hold all the incoming tasks. The tasks remain in the wait queue as long as they have dependency conditions than needs to be satisfied. Once they are satisfied, the tasks can be moved from wait queue to the ready queue. Once in ready queue, the execution unit can then execute them one by one in the FIFO way. After completing task execution, the task is then moved to the complete queue. For each task in the complete queue, the execution unit is responsible for sending the ZHT update messages to all children tasks of that particular task to satisfy the dependency requirements.

3.6 Load Balancing

Anytime when a node has no waiting tasks, it will ask the load information of all the neighbors in turn, and try to steal tasks from the heaviest loaded neighbor. When a node receives a load information request, it will send its load information to the neighbor. If a node receives work stealing request, it then checks its queue, if which is not empty, it

will send some tasks (e.g. half of the 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 execution unit can be configured to perform work stealing for any queue.

Each compute node has the knowledge of every other nodes, and can choose to have a subset of neighbors to for work stealing. The amount of neighbors is same for every worker and is configured at the time of initialization. The number of neighbors from which to steal and the number of tasks to steal were set as concluded in the paper [57]. It concluded that the optimal parameters for the MTC workloads and adaptive work stealing are to steal half the number of tasks from their neighbors, and to use the square root number of dynamic random neighbors. These parameters are tunable though and are set during initialization time.

One modification in the implementation of the work stealing algorithm mentioned in previous section is in the implementation of the exponential back-off approach. In the algorithm mentioned in previous section the adaptive poll interval keeps doubling every time a worker node fails to steal task. But in the implementation we found out increasing adaptive poll interval indefinitely can cause starvation if the poll interval becomes too high. So we set a cap (e.g. 1 sec). Thus the adaptive poll interval increases from initial start (e.g. 1ms) to this cap value (e.g. 1 sec). Beyond this the poll interval does not change even if the work stealing fails. It resets itself to the initial start value if the worker node is successful in stealing tasks. Although it can avoid starvation, this condition can cause problems at the end of experiment. It is due to the fact that at the end of experiment there are very few tasks and it is difficult to load balance very few tasks especially when each

task runs for a very short time (e.g. 64ms). So having this cap can cause lot of failed work stealing messages at the end of the experiment. This leads to poor utilization of network in the perspective of work stealing. So we also maintain a counter that keeps track of consecutive failed attempts. It the counter goes beyond a threshold value (e.g. logarithm base-2 of number of worker nodes) then the worker node finds itself doing no useful work other than causing network congestion. So it just stops its work stealing function and exits as there were no tasks in its ready queue in the first place. Also, if it is possible to know the total runtime of the entire workload beforehand, then the work stealing algorithm can be tuned to be less aggressive at the start and end of experiment, which means the initial start and cap may be set to a higher value (e.g. instead of 1ms to 1s, it can be 100ms to 10s). This may sound counter intuitive for load balancing but we learned through experiments that it can reduce network congestion also possibility of network communication failures.

3.7 Monitoring

Regardless of the number of dispatchers used to submit tasks to compute nodes, only one dispatcher keeps monitoring the system utilization and status of submitted tasks, while all other task dispatchers exit. The monitoring dispatcher periodically sends requests messages to determine the current load on all compute nodes and calculate the number of tasks that have completed its execution. The termination condition is that all the tasks submitted by client are finished.

CHAPTER 4

EVALUATION AND EXPERIMENTAL RESULTS

This chapter presents the experimental hardware and software environments, the evaluation metrics, the essential results of the simulation from SimMatrix paper [57], the workloads generation, the throughput of dispatcher, the throughput of the run-time system, the validation of MATRIX against Falkon [29] and SimMatrix [11], and the study of scalability, load balancing and network traffic caused by the adaptive work stealing algorithm.

4.1 Experiment Environment, Metrics, Workloads

4.1.1 Testbed. MATRIX is implemented in C++; the dependencies are Linux, ZHT [50], NoVoHT , Google Protocol Buffer [52], and a modern gcc compiler.

All the experiments in this section were performed on the IBM BlueGene/P supercomputer [51]. Each node on the BG/P uses a quad-core, 32-bit, 850 MHZ IBM Power PC 450 with 2GB of memory. A 3D torus network is used for point-to-point communication among computing nodes. For validation of MATRIX against Falkon, Falkon runs on the IBM BlueGene/P supercomputer [51] on a scale of 64 nodes up to 1024 nodes in powers of 2.

All the simulations for SimMatrix were performed on fusion.cs.iit.edu, which boasts 48 AMD Opteron cores at 1.93GHz, 256GB RAM, and a 64-bit Linux kernel 2.6.31.5.

4.1.2 Metrics. We use important metrics to evaluate the performance of the adaptive work stealing algorithm. They are listed below:

- *Throughput*: Number of tasks finished per second. Calculated as total-number-of-tasks/ execution -time.

- *Efficiency*: the ratio between the ideal execution time of completing a given workload and the real execution time. The ideal execution time is calculated by taking the average task execution time multiplied by the number of tasks per core.
- *Load Balancing*: We adopted the coefficient variance [53] of the number of tasks finished by each node as a measure of the load balancing. The smaller the coefficient variance is, the better the load balancing would be. It is calculated as the standard-deviation/average of number of tasks finished by each node.
- *Scalability*: Total number of tasks, number of nodes, and number of cores supported.
- *Utilization*: This is another way of looking the efficiency of load balancing by visualizing the utilization of the compute nodes in the system.
- *Number of messages*: is the count of the various messages flowing across the network caused by the work stealing algorithm.

4.1.3 Workloads. First for testing the functionality of MATRIX we used five different workload: Bag of tasks, Fan-In DAG, Fan-Out DAG, Pipeline DAG and a complex random DAG. All the tasks in the DAG had a run-time of 8 seconds. These different workloads are shown in Figure 4.

Bag of Tasks: This is the simplest workload where there are no dependencies among the tasks and every task is always ready to execute. For such a workload, the task dispatcher inserts them directly into the ready queue instead of inserting them into wait queue first and then moving to ready queue. So some of the ZHT operations are skipped for the Bag of Tasks workload. Hence, in terms of efficiency, this gives the best performance among all the workloads.

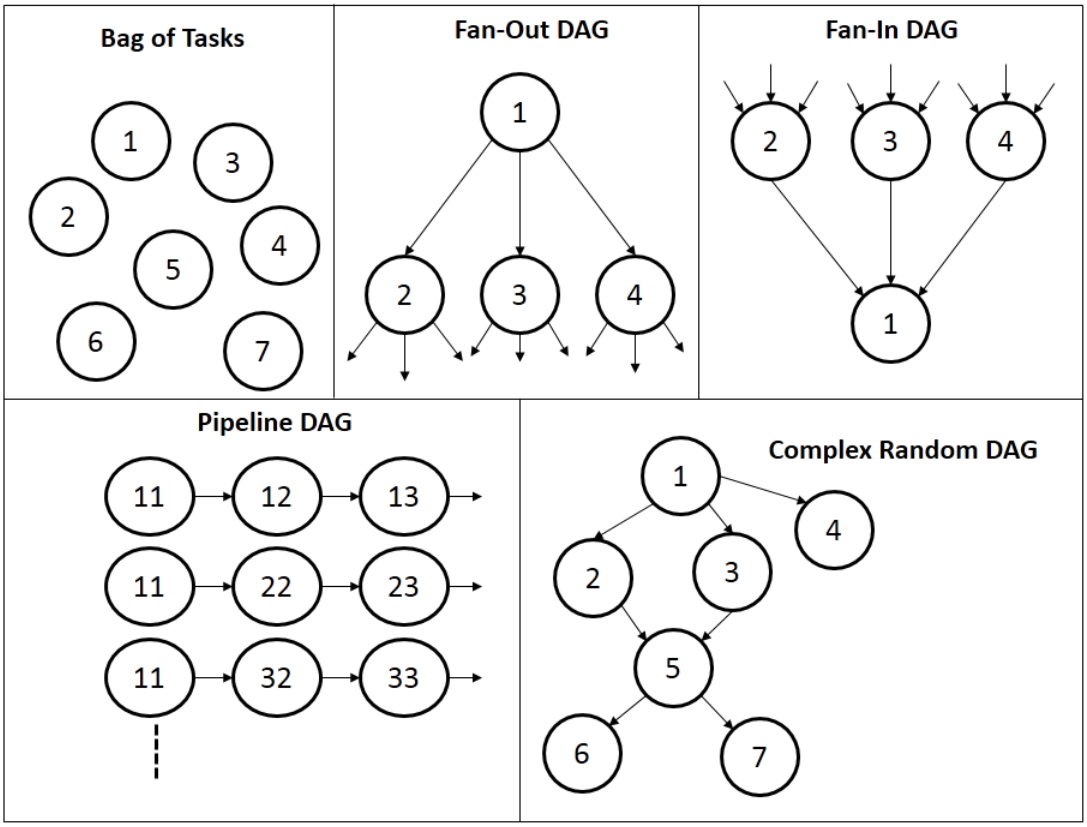

Figure 4. Different Types of Workload used for evaluating functionality of MATRIX

Fan-In DAG: This workload introduces simpler dependencies among the tasks in the workload and thus adds a little bit of complexity for the execution system. Since not all the tasks in the workload are readily available at any given instant of time, the system utilization is lesser when compared to Bag of Tasks workload. The performance of the execution unit depends also the number of tasks that are ready to execute at any instant of time and since initially there are large number of ready tasks, the Fan-In workload also has a better efficiency.

Fan-Out DAG: This workload is similar to Fan-In DAG, except the Fan-Out DAG is obtained be reversing the Fan-In DAG. Initially since there is only one ready task, it takes slightly longer to get full utilization which depends on the out-degree of every node in the graph.

Pipeline DAG: This workload is a collection of "pipes" where each task in a pipe is dependent on the previous task. Here 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 due to the fact that only one task in a pipeline can execute at any given time. This workload has a lower efficiency than the above three.

Complex Random DAG: This workload has the largest number of interdependencies among tasks. Since this DAG is formed randomly, it is hard to predict the performance. The workload has the lowest efficiency among all. All other workloads can be treated as a special case of this workload.

In order to study the adaptive work stealing algorithm through MATRIX, the experiments were run using synthetic workloads composed of sleep tasks of different durations. We tested the scalability of MATRIX using two sets of sleep tasks. First we tested it will sleep tasks of duration 1 second up to 8 seconds on a scale of 64 nodes up to 1024 nodes. We also tested the scalability of the system for fine granular workload using sub-second tasks of duration 64ms up to 512ms on a scale of 1 node up to 1024 nodes.

In both cases, the sleep duration and the type of workload is specified as an argument to task dispatcher which then initializes the set of tasks of the given type and submit the workload to the one or more worker nodes.

To amortize the potential slow start and long trailing tasks at the end of experiment, we fixed the number of tasks such that the each experiment runs for about 1000 seconds. For example, for an experiment with a workload of tasks of 1 second duration, and with 1024 nodes, where each node has 4 cores, the number of tasks in the workload would be 4M (1024×4×1000).

4.2 Studying the Adaptive Work Stealing Algorithm through simulations

This section explains the essential results obtained through simulations such as number of tasks to steal number of neighbors of a node, and static/dynamic neighbors.[57]

4.2.1 Number of Tasks to Steal. In the five groups of experiments, steal_1, steal_2, steal_log, steal_sqrt, steal_half means steal 1, 2, logarithm base-2, square root, and half number of tasks respectively. Every node was set to have 2 static neighbors. The changes of the efficiency of each group with respect to the scale are shown in Figure 5. 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 [10].

The reason that steal_half is not perfect (efficiency is very low at large scale) for these experiments is that 2 neighbors of a node is not enough, and starvation can occur for some nodes that are too far in the id namespace from the original compute node who is receiving all the task submissions.

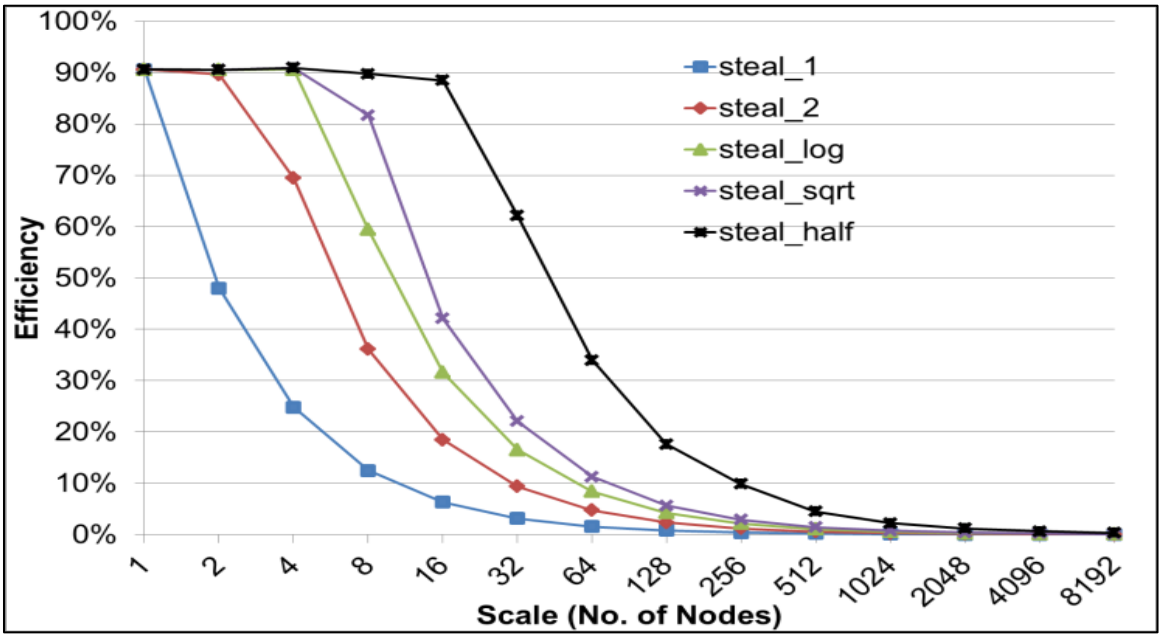

Figure 5. Efficiencies of the number of tasks to steal with respect to the scale

4.2.2 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 ids) are determined at first and never change; dynamic random neighbors mean that each time when does work stealing, a node randomly selects some neighbors.

In the experiments involving *Static Neighbors*, nb_2, nb_log, nb_sqrt, nb_eighth, nb_quar, nb_half means 2, logarithm base-2, square root, eighth, a quarter, half neighbors of all nodes, respectively. In this case, neighbors are chosen as consecutive ids at the beginning, and will not change. The result in Figure 6 shows that when the number of neighbors is no less than eighth of all nodes, the efficiency will keep at the value of higher than 87% within 8192 nodes' scale. For other numbers of static neighbors, the efficiencies could not remain, and will drop down to very small values.

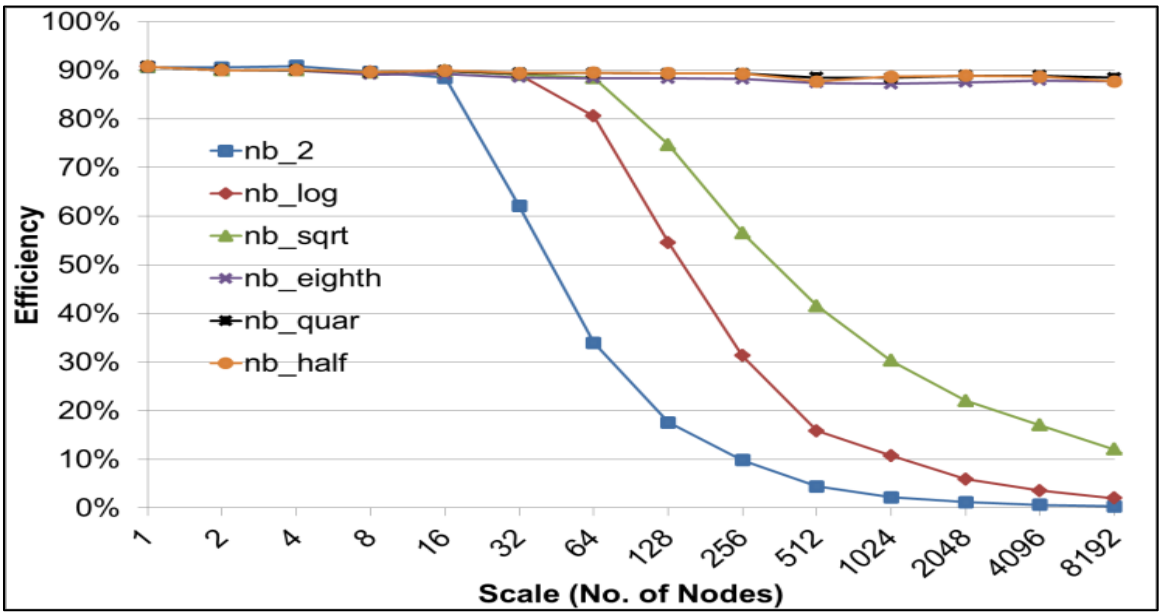

Figure 6. Efficiencies of number of static neighbors with respect to the scale

Thus the optimal number of static neighbors is eighth of all nodes, as more neighbors do not improve performance significantly. However, in reality, an eighth of neighbors will likely lead to too many neighbors to be practical, especially for an

exascale system with millions of nodes (meaning 128K neighbors). In the search for a lower number of needed neighbors, dynamic multiple random neighbor selection technique was explored.

In *dynamic random neighbors*, there were 4 groups of experiments, nb_1, nb_2, nb_log, nb_sqrt. First nb_1 experiment was done until it started to saturate (the efficiency is less than 80%), then at which point, nb_2 was started, then nb_log, and nb_sqrt at last. The results are shown in Figure 7.

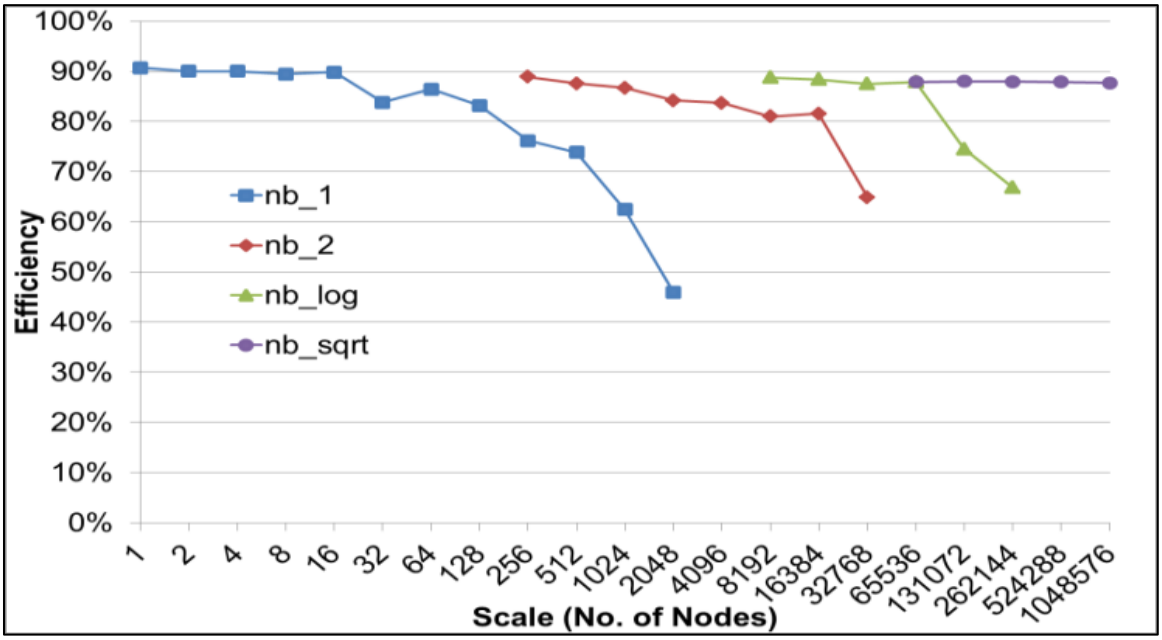

Figure 7. Efficiencies of dynamic random neighbors with respect to the scale

Even with 1M-nodes in an exascale system, the square root implies having 1K neighbors, a reasonable number of nodes for which each node to keep track of with modest amounts of resources. The conclusion drawn from this section about the 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.

4.3 Evaluating the Adaptive Work Stealing Algorithm with MATRIX

All the experiments in this section were performed using Bag of Tasks workload.

4.3.1 Validation: MATRIX vs. SimMatrix. Before evaluating the performance of the work stealing in the real system, the throughput of the system on a sleep 0 workload was compared with SimMatrix. Figure 8 shows the validation results comparing SimMatrix and MATRIX for raw throughput on a sleep 0 workload. The real performance data matched the simulation with 5.8% difference.

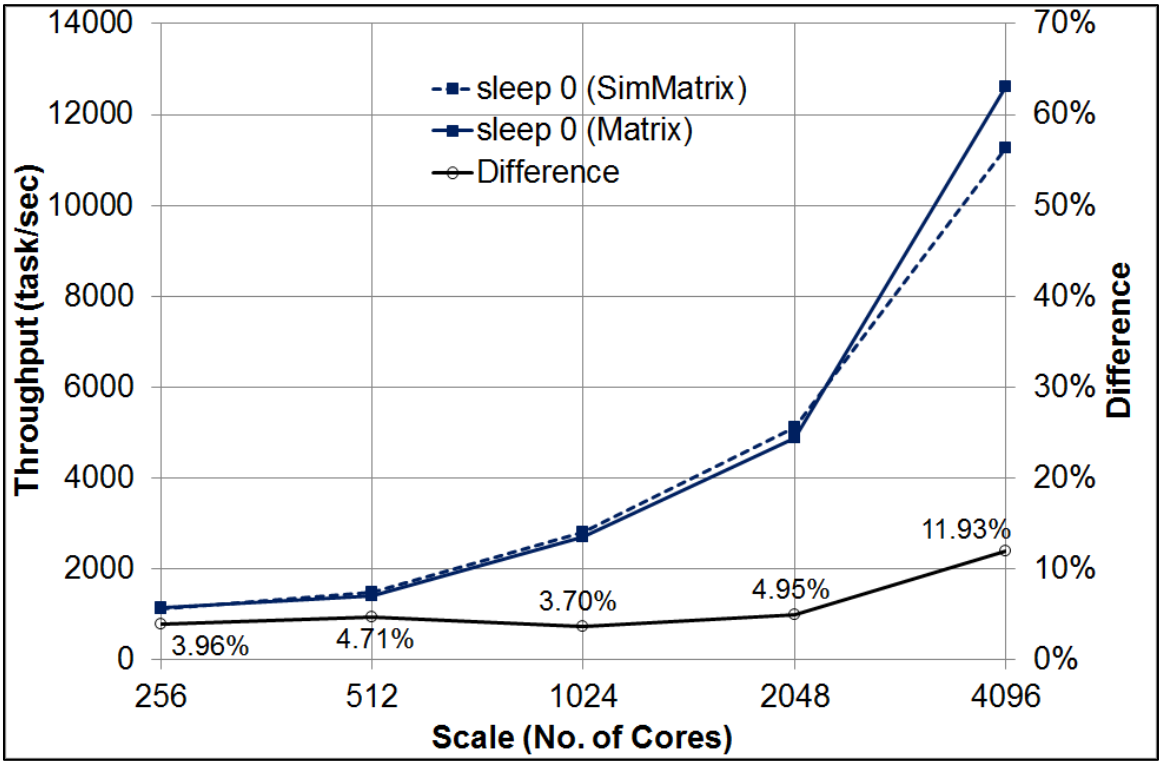

Figure 8. Comparison of MATRIX with SimMatrix throughput of 100K sleep 0 tasks

We also validated the performance of the implementation against SimMatrix up to 1024 nodes. Since the average runtime of every task in the MTC workload used for simulation was 95.20 seconds, we evaluated MATRIX with a workload of sleep tasks where each task runs for 100 seconds. The comparison is shown in Figure 9.

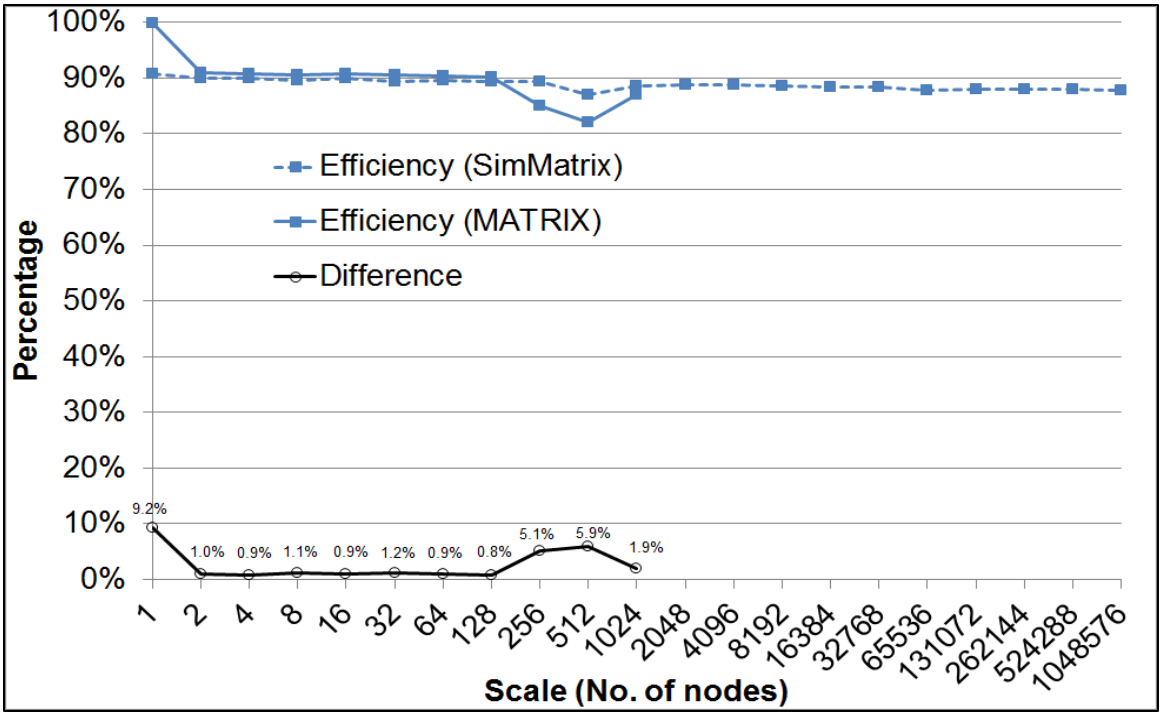

Figure 9. Comparison of MATRIX with SimMatrix performance up to 1024 nodes (4 cores per node)

The real performance data matched the simulation with 2.6% difference. This shows that the work stealing algorithm has the potential to achieve distributed load balancing, even at exascales with millions of nodes and billions of cores.

4.3.2 Comparison: MATRIX vs. Falkon. Figure 10 shows the results from a study of how efficient we can utilize up to 2K-cores with varying size tasks using both MATRIX and the distributed version of Falkon (which used a naïve hierarchical distribution of tasks); MATRIX are the solid lines, while Falkon are the dotted lines. We see MATRIX outperform Falkon across the board with across all size tasks, achieving efficiencies starting at 92% up to 97%, while Falkon only achieved 18% to 82%.

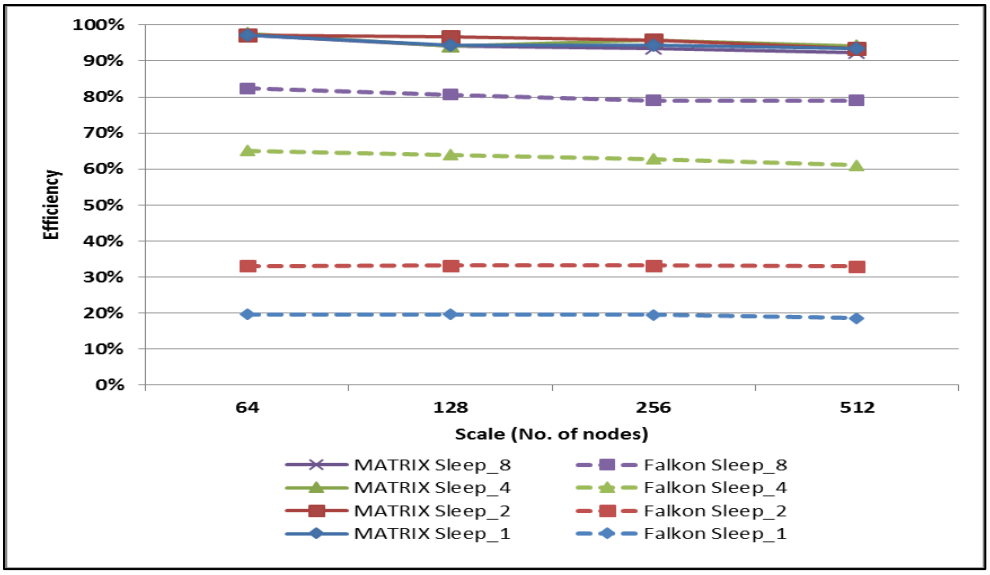

Figure 10. Comparison of MATRIX with Falkon efficiency of 256K to 2M sleep tasks across 256 to 2K-cores

Figure 11 shows the comparison of average efficiencies between MATRIX and Falkon for different duration.

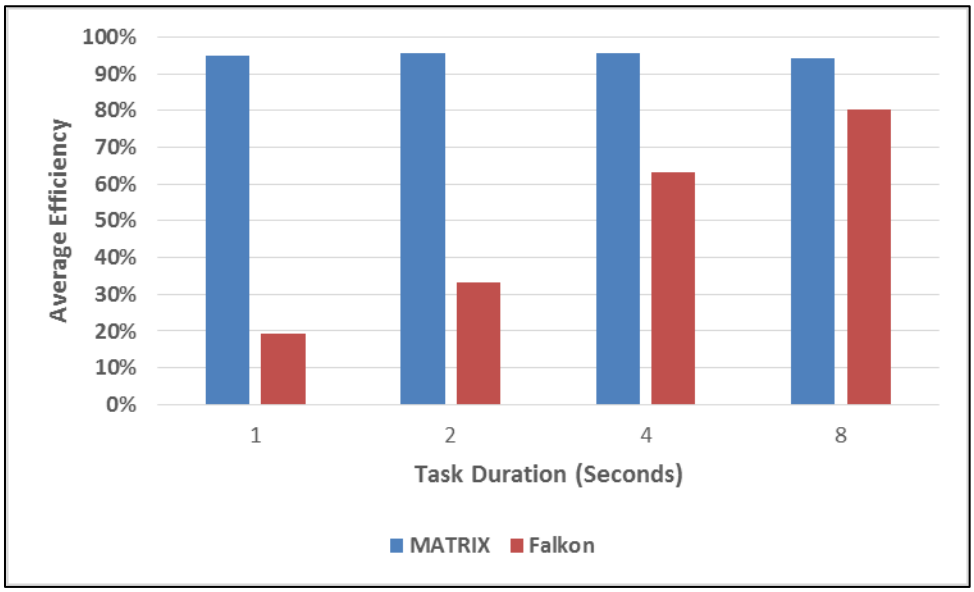

Figure 11. Average Efficiency comparison of MATRIX with Falkon

4.3.3 Scalability of Adaptive Work Stealing. In all the following experiments, we use the sleep workloads, where each node has 4 cores, and the number of tasks is 1000 times

of the number of cores. Figure 12 shows the scalability of the adaptive work stealing up to 1024 nodes for tasks of duration 1 second up to 8 seconds, in terms of efficiency and coefficient variance.

The results show that up to 1024 nodes, the adaptive work stealing actually works quite well, given the right work stealing parameters. We see an efficiency of 88% at a 1024 node scale, with a co-variance of less than 0.05 (e.g. meaning that the standard deviation of the number of tasks run being a relatively low 500 tasks when on average each node completed 4K tasks).

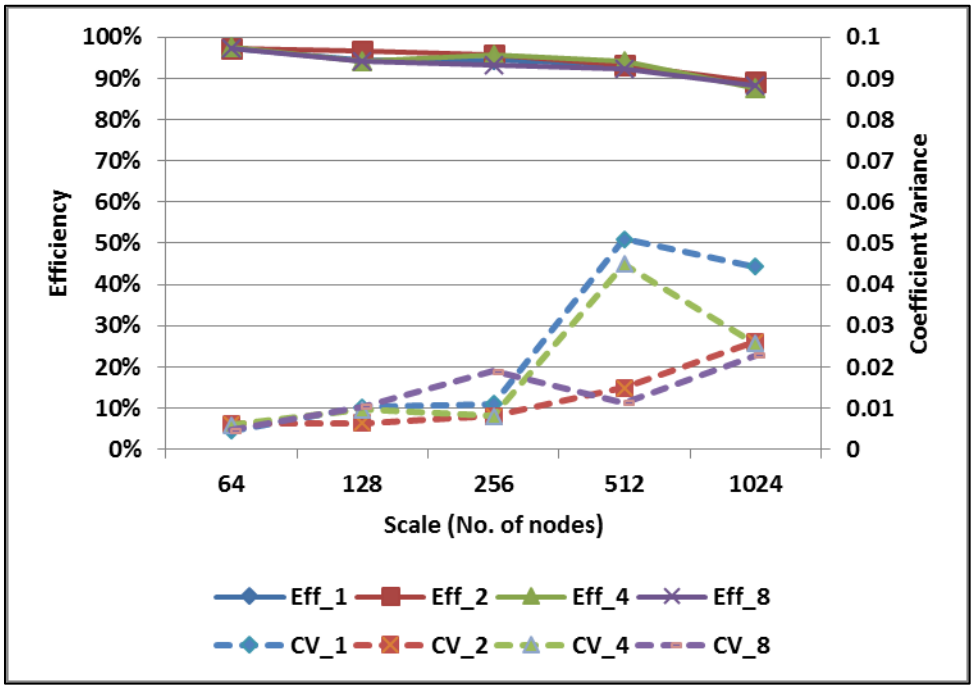

Figure 12. Scalability of Adaptive Work Stealing algorithm for sleep tasks (1s to 8s) on a scale of 64 nodes up to 1024 nodes (4 cores per node)

The efficiency drops from 100% to 88% (12 percentages) from 1 node to 1024 nodes. The reason that efficiency decreases with the scale is because the run time of 1000 seconds is still not perfectly enough for amortizing 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, within an upper bound (there are communication overheads, such as the time taken to submit all the tasks from the client), but the longer it takes to run large scale experiments. We found that run time of 1000 seconds (or a workload of 4 million tasks – 1000 tasks * 1024 nodes * 4 cores) could balance well between the efficiency (88%) and the running time to run large scale experiments.

Figure 13 shows the scalability of the adaptive work stealing up to 1024 nodes for fine granular tasks of duration 64ms up to 512ms, in terms of efficiency and coefficient variance.

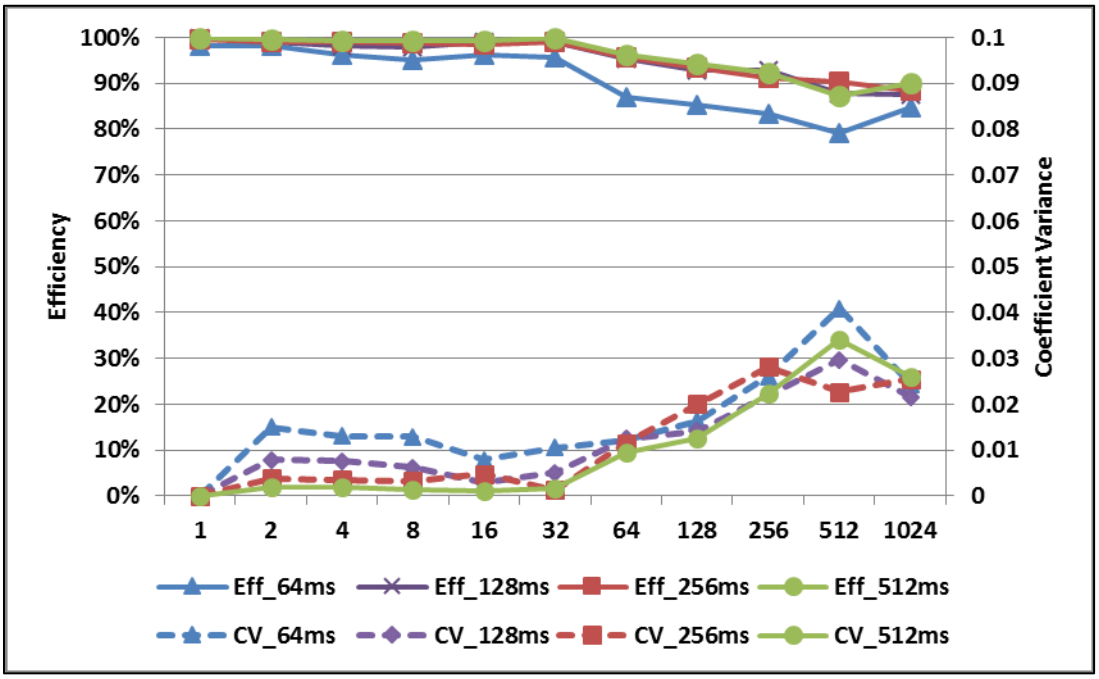

Figure 13. Scalability of Adaptive Work Stealing algorithm for short tasks (64ms to 512ms) up to 1024 nodes (4 cores per node)

Again the results show that up to 1024 nodes, the adaptive work stealing actually works quite well even for sub-second tasks starting at 79% up to 98% with the right parameters.

4.3.4 Evaluating functionality of MATRIX. This section explains the experiments performed to test the functionality of MATRIX. Based on the workload type, the task dispatcher generates a Directed Acyclic Graph for that type and then submits it to the execution unit. All tasks in the workload were sleep tasks and had a run-time of 8 seconds. The efficiency of system was measured and is shown in the Figure 14.

The Bag of tasks has highest efficiency because there is no dependency among any tasks and each task is always in the ready queue. So the system utilization for bag of tasks reaches maximum immediately at the start of the experiment.

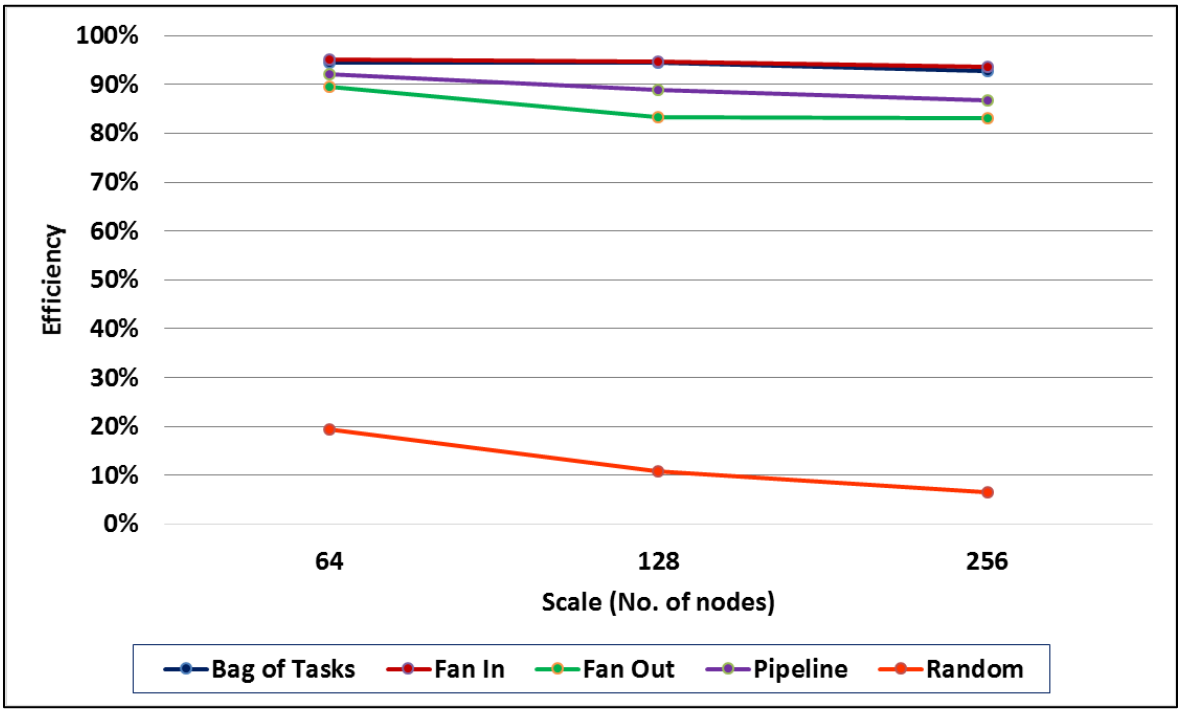

Figure 14. Analysis of work stealing using different workload types via MATRIX

For Fan-In and Fan-Out DAG, completion of one task might satisfy the dependencies of many other tasks thus providing lot of ready tasks at any instant of time. This number keeps increasing till the point where the system utilization can reach its maximum.

For Pipeline DAG, the efficiency depends on system utilization which is in turn depends on the number of pipelines. So if we have greater number of pipelines, then the efficiency will be greater.

For random DAG, the efficiency is the least because the generated DAG might contain tasks that have lot of dependencies and is thus hard to have lot of ready tasks at any instant of time. So we tried evaluating the random DAG at small scales with a smaller workload on different testbed called HEC which has 64 nodes each having 8 AMD Opteron cores at 2.7GHz, 8GB RAM, and a 64-bit Linux kernel 2.6.28.10. This is shown in Figure 15. The efficiency is higher for lower scales and lower number of tasks (100 tasks per core instead of 1000 tasks per core), probably because the generated smaller workload had fewer dependencies and thus higher number of readily available tasks at any instant of time. Also, although smaller, HEC is more powerful than BGP.

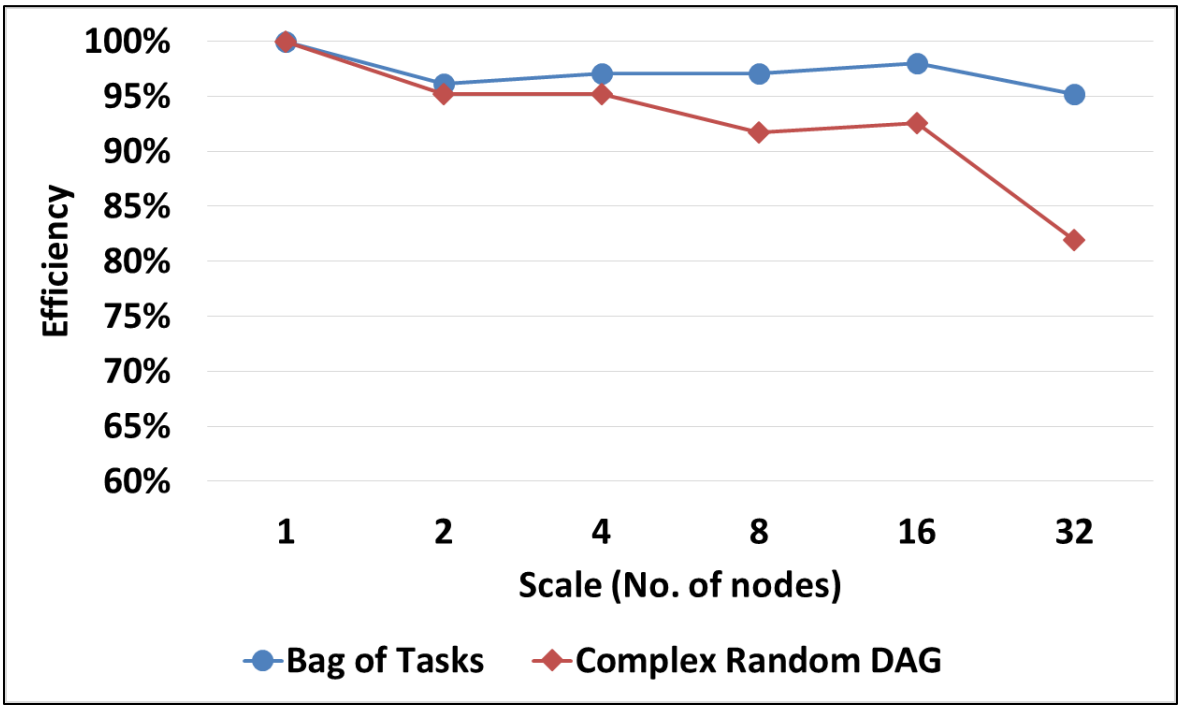

Figure 15. Evaluating Complex Random DAG at small scales

4.3.5 Visualization of Load Balancing. In order to understand more about the behavior of the algorithm, the system utilization was observed to get an idea of how well the load is balanced across all the compute nodes. Figure 16 shows the system utilization when running the eperiment different with different workload types.

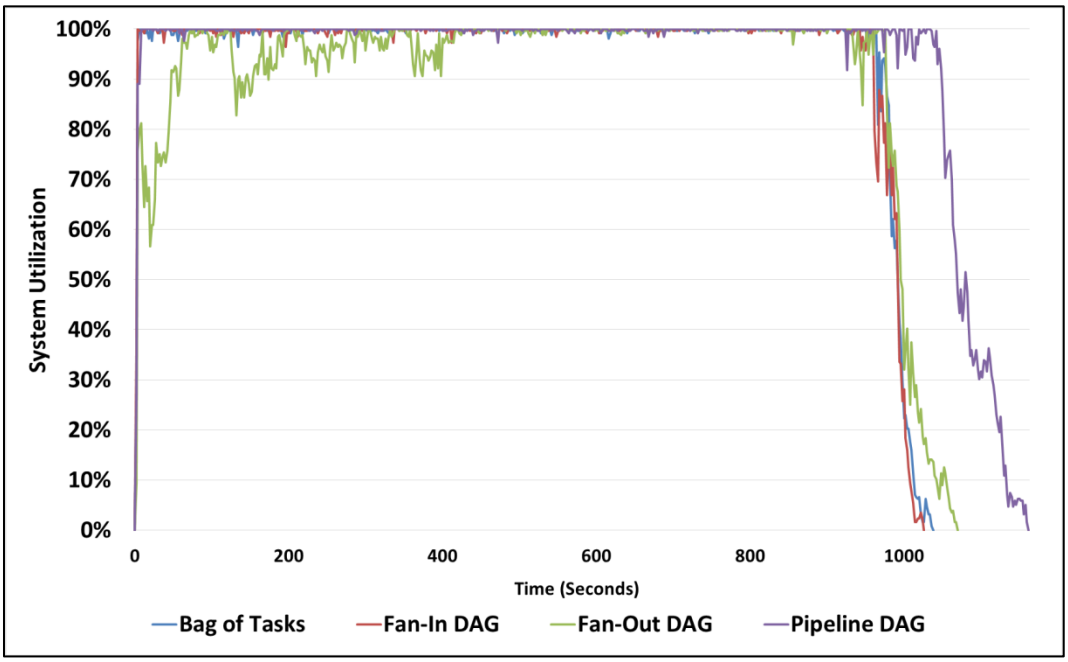

Figure 16. System Utilization for different types of workload

For the Bag of Tasks and Fan-In workload, 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 such a low utilization for Pipeline and Complex Random workload is that, there are not sufficient tasks available to keep the entire system busy. This can be improved by increasing the number of tasks in the workload.

A more detailed analysis was made for Bag of Tasks workload in terms of duration of each task. As seen from Figure 17, it is generally difficult to get higher

efficiency for shorter jobs since load balancing is not perfect for shorter running jobs like Sleep 32ms.

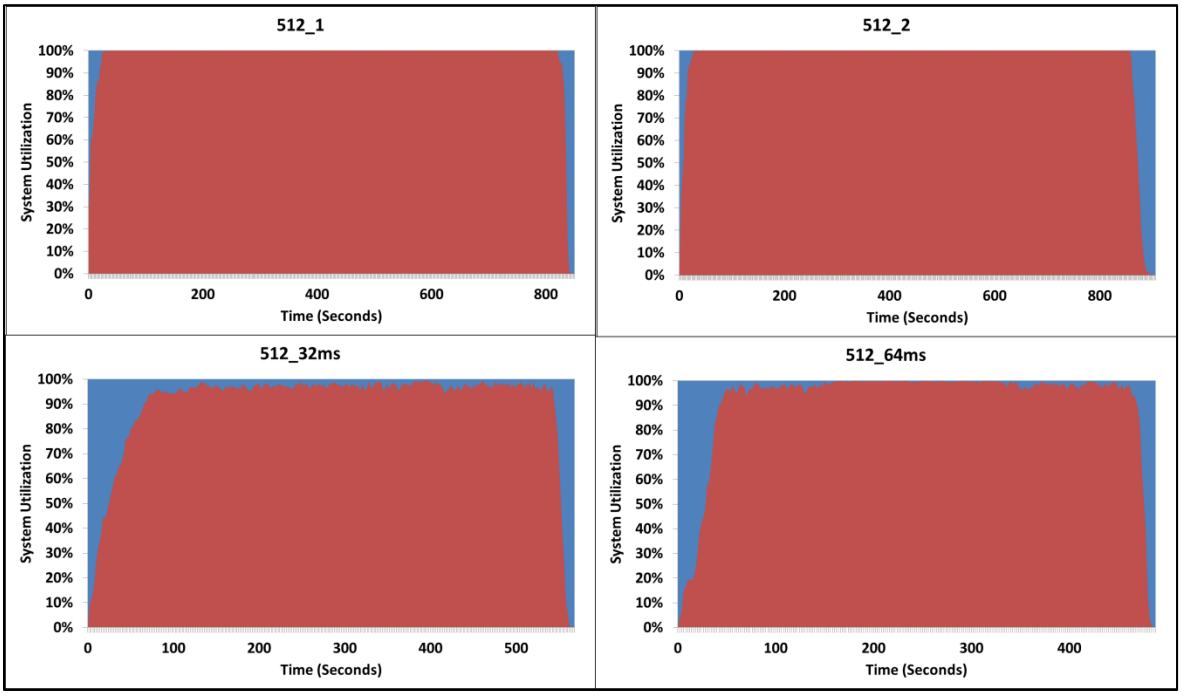

Figure 17. Visualization of load balancing for Bag of Tasks workload

Figure 18 shows a comparison of load balancing for 1s and 32ms tasks on 512 nodes. The Start figures (Figure 18 top half) indicate the convergence of the load i.e. how quick the entire load get balanced on 512 nodes. The 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 32ms workload the time taken for the entire workload to get distributed evenly is more than double than that required for 1s workload. The End figures (Figure 18 lower half) tells us about the end of experiment when there are very few tasks left to be executed.

There can be two reasons for such behavior. The 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 gets executed at the

node where it is present i.e. the queues are changing state so fast that by the time they want to steal work, there is nothing to steal anymore. Tasks migration is very low as they all run fast, and any work stealing that tries to occur likely fails.

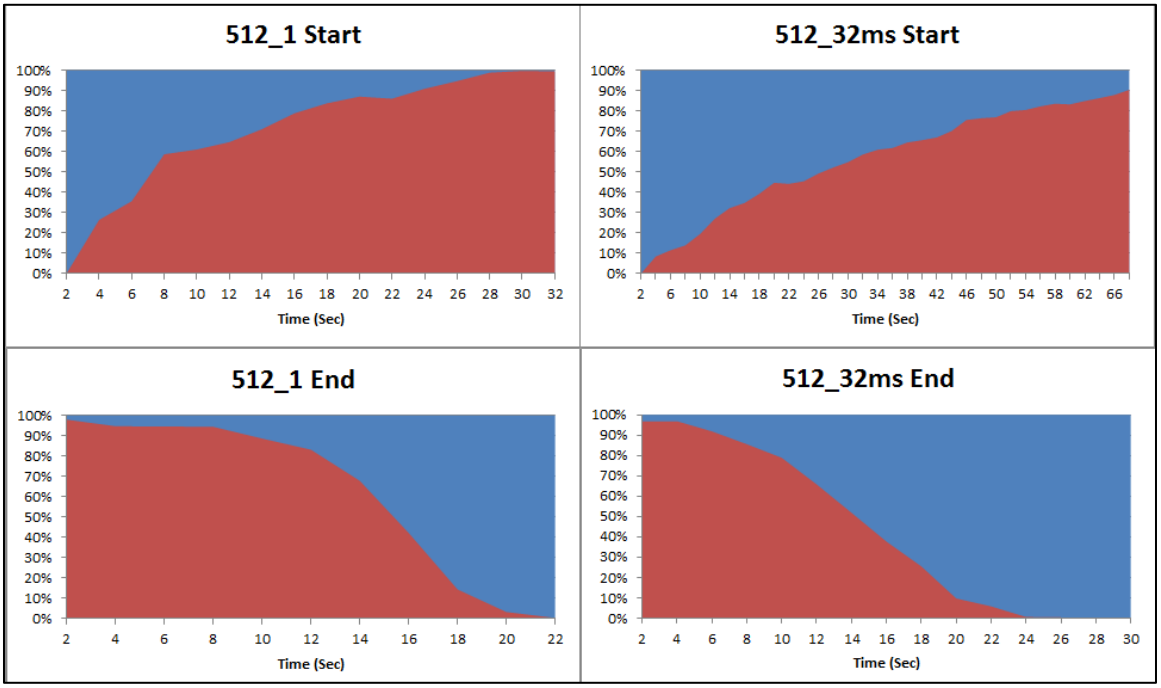

Figure 18. Comparison of work stealing algorithm performance for 1s and 32ms workload at the beginning and the end of the experiment on 512 nodes

4.3.6 Network Traffic generated by Work Stealing. The last thing that was measured as a part of evaluation of MATRIX is the number of messages per task involved in testing of different workloads and the percentage of each message type. Figure 19 shows the the number of messages for different workload types. This shows that although with increase in the number of nodes the total number of messages is increased, the number of messages per task has remained almost constant which means work stealing is a stable algorithm. These numbers were broken down to see the

individual contribution of each message type. This is shown in Figure 20.

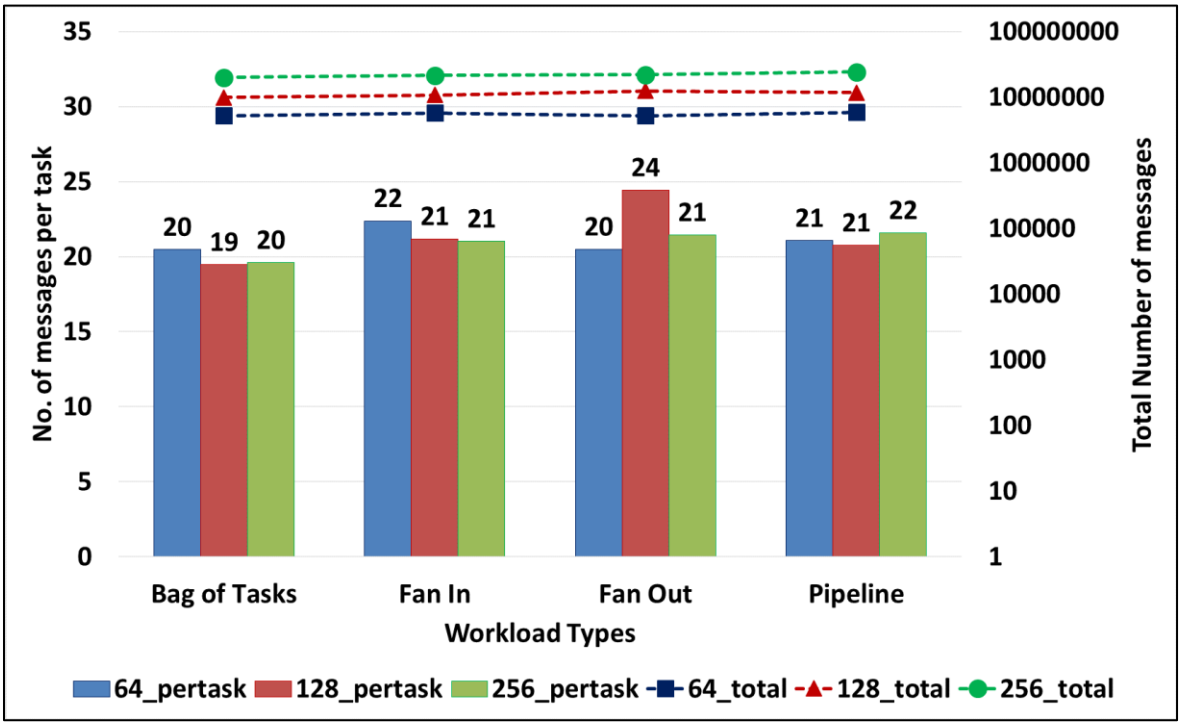

Figure 19. Number of messages per task for different types of workload

As seen, the most of the messages were of the *ZHT Update* and *Load Information* type. This is understood because in DAGs all the tasks have dependencies and for each task completion an update message is sent. *ZHT Update* message is also sent when task migration occurs due to work stealing to update the current node information. Also the idle nodes try to steal tasks but all the stealing attempts are unsuccesful. A *Load Information* message can be treated as success only if it finds a node with tasks. So it should be immediately followed by a *Work Stealing* message. Approximately only 1% of *Load Information* messages are succesful. This is a concern and needs to be improved in the future version of MATRIX by introducing "*random stealing*". Here the work stealing algorithm can be modified to randomly choose one neighbor and steal tasks from if the

neighbor has tasks in its queue. This will significantly improve performance for fine grained tasks significantly.

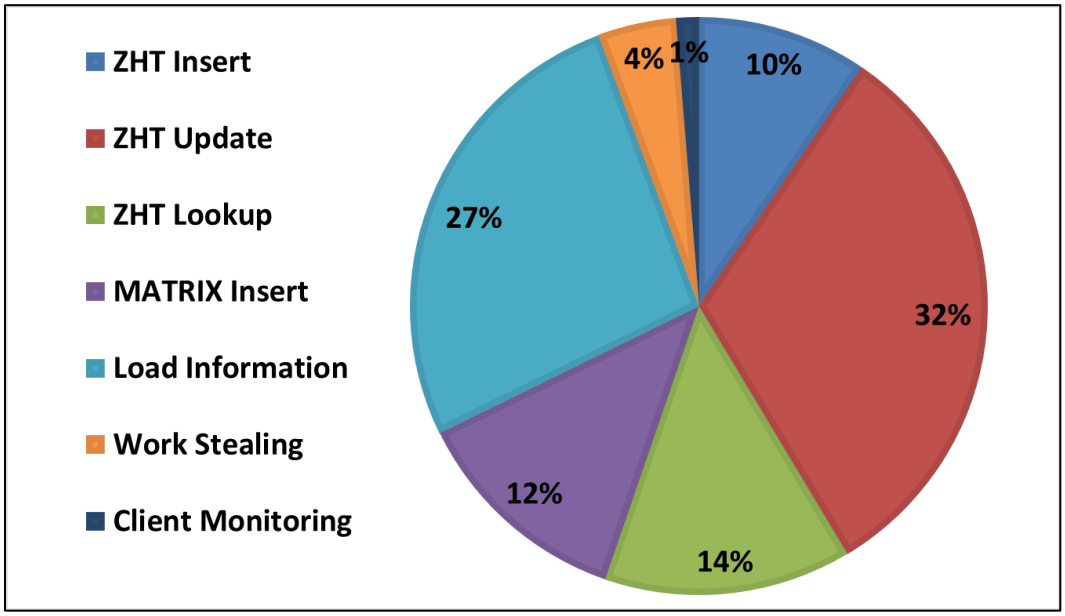

Figure 20. Breakdown of total number of messages per task into individual message types

Although the number of messages seems high, it is due to the runtime of the experiment which was fixed to 1000 seconds for amortizing the slow start and long trailing tasks at the end. The total runtime of the experiment is based on the number of tasks in the workload.

Throughout the experiment, apart from the initial insert from the task dispatcher in the beginning, all the inserts happen when a worker sends tasks from its ready queue upon receiving work stealing request. So essentially, there will be as many inserts as the stealing. Also work stealing requests occur only after determining the heaviest loaded worker which is shown by the green line. But not all time the load query is successful. The successful requests are shown by the black line. Thus the use of adaptive poll interval to minimize the network traffic by regulating the number of load query requests seems to be working better for Bag of Tasks than other workload types. As mentioned

above, the “random stealing” can help improve the performance for fine grained tasks significantly.The monitoring information gathered periodically by the dispatcher is used to determine the status of submitted tasks and the load on all workers for the validation purposes. The dispatcher can be configured to this less frequently so as to reduce the monitoring traffic.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Large scale distributed systems require efficient job scheduling system to achieve high throughput and system utilization. Distributed load balancing is critical for designing job schedulers. Work stealing is a potential technique to achieve distributed load balancing across many concurrent threads of execution, from many-core processors to exascale distributed systems. The work stealing algorithm was implemented in a real system called MATRIX, and a preliminary evaluation up to 4K-core scales was performed for different types of workload namely Bag of Tasks, Fan-In DAG, Fan-Out DAG, Pipeline DAG and Complex Random DAG. The parameters of adaptive work stealing algorithm was configured using the simulation-based results from SimMatrix [57] (the number of tasks to steal is half and there must be a squared root number of dynamic neighbors) and its performance was analyzed in terms of system utilzation and network traffic.

We modified the MATRIX implementation and developed a job launch framework to add scheduling support for HPC workloads. We plan to evaluate it and integrate it with the Slurm job manager [2]. We plan to integrate MATRIX with Swift [14] (a data-flow parallel programming systems) for running real application Directed Acyclic Graphs.

Some of the features in MATRIX such as Message Batching, Atomic updates, Distribued Queue and selective lookups can be added to ZHT [50] to make it more general so that many new applications can benefit from it. Also the current version of ZHT has a N-N network topology where each compute node can communicate with

every other node. We plan to add new network topologies such as logarithmic topology to allow each compute node select neighbors based on location. This can help optimize the network traffic.

We will also continue to develop the MATRIX system. Based on the simulation results, we expect that MATRIX should scale to 160K-cores on the IBM BlueGene/P supercomputer we conducted our preliminary evaluation. We also plan to test it on the newly built IBM BlueGene/Q supercomputer at a full 768K-core (3M hardware threads) scale. MATRIX will also be integrated with other projects, such as large-scale distributed file systems [60] FusionFS [56] and large scale programming runtime systems Charm++ [27]. A potential future software stack is shown in Figure 21. The gray areas represent the traditional HPC-stack. The green areas are additional components, such as support for many-task computing applications, using lower level components such as MATRIX, ZHT [50], and FusionFS [56]. The yellow areas represent the simulation components aiming to help explore peta/exascales levels on modest terascale systems. Once SimMatrix is extended more complex network topologies, we could address the remaining challenge of I/O and memory through data-aware scheduling. [55].

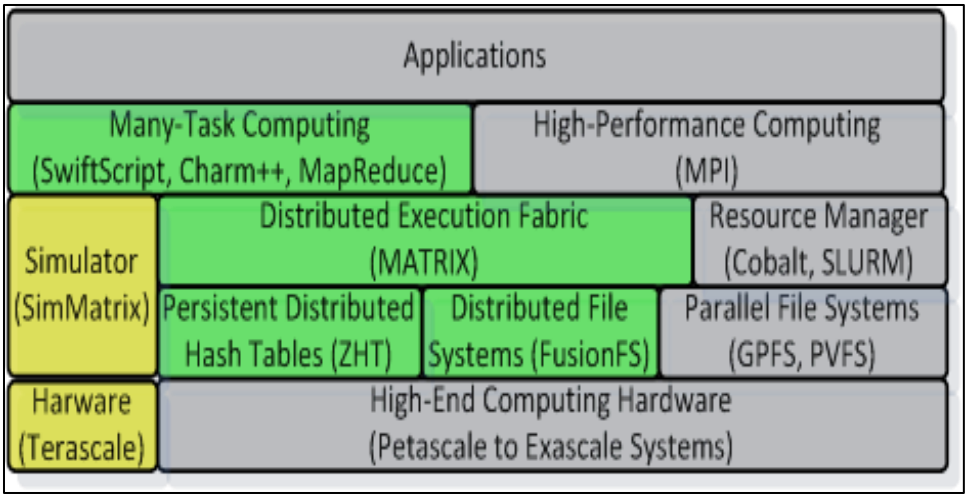

Figure 21. Building blocks for future parallel programming systems and distributed applications

BIBLIOGRAPHY

[1] P. Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” 2008.

[2] M. A. Jette et. al, Slurm: Simple linux utility for resource management. In In Lecture Notes in Computer Sicence: Proceedings of Job Scheduling Strategies for Prarallel Procesing (JSSPP) 2003 (2002), Springer-Verlag, pp. 44-60.

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

[4] J. Frey, T. Tannenbaum, I. Foster, M. Frey, S. Tuecke. “Condor-G: A Computation Management Agent for Multi-Institutional Grids,” Cluster Computing, 2002.

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

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

[7] Work Stealing: www.cs.cmu.edu/~acw/15740/proposal.html, 2012.

[8] R. D. Blumofe, et. al. “Scheduling multithreaded computations by work stealing,” In Proc. 35th FOCS, pages 356–368, Nov. 1994.

[9] V. Kumar, et. al. “Scalable load balancing techniques for parallel computers,” J. Parallel Distrib. Comput., 22(1):60–79, 1994.

[10] J. Dinan et. al. “Scalable work stealing,” In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), 2009.

[11] K. Wang, K. Brandstatter, I. Raicu, “SimMatrix: Simulator for Many-Task computing execution fabRIc at eXascales,” technical report, http://datasys.cs.iit.edu/~kewang/publications.html.

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

[13] I. Raicu, et. al. “Toward Loosely Coupled Programming on Petascale Systems,” IEEE SC 2008.

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

[15] D. Abramson, et. al. "Parameter Space Exploration Using Scientific Workflows," Computational Science—ICCS 2009, LNCS 5544, Springer, 2009, pp. 104-113.

[16] E. Deelman, et. al. "Pegasus: A Framework for Mapping ComplexScientific Workflows onto Distributed Systems." Scientific Programming, 13 (3). 219-237.

[17] The Condor DAGMan (Directed Acyclic Graph Manager), http://www.cs.wisc.edu/condor/dagman, 2007.

[18] Business Process Execution Language for Web Services, Version 1.0, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.

[19] T. Oinn, et. al. A Tool for the Composition and Enactment of Bioinformatics Workflows Bioinformatics Journal, 20 (17). 3045-3054.

[20] I. Taylor et. al. "Visual Grid Workflow in Triana." Journal of Grid Computing, 3 (3-4). 153-169.

[21] I. Altintas, et. al. "Kepler: An Extensible System for Design and Execution of Scientific Workflows." 16th Intl. Conf. on Scientific and Statistical Database Management, (2004).

[22] G. Laszewski, et. al. Java CoG Kit Workflow. in Workflows for eScience, 2007, 340-356.

[23] M. Isard, et. al. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," ACM SIGOPS Operating System Rev., June 2007, pp. 59-72..

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Comm. ACM, Jan. 2008, pp. 107-113.

[25] Hadoop Overview: wiki.apache.org/hadoop/ProjectDescription, 2012.

[26] Y. Gu and R.L. Grossman, "Sector and Sphere: The Design and Implementation of a High Performance Data Cloud," Philosophical Trans. Royal Society A, 28 June 2009, pp. 2429-2445.

[27] G. Zhang, et. al, "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers," In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW 10, pages 436-444, Washington, DC, USA, 2010.

[28] H. Kaiser, M. Brodowicz, T. Sterling. "ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications", Parallel Processing Workshops, 2009.

[29] I. Raicu, et. al. "Falkon: A Fast and Light-weight tasK executiON Framework," IEEE/ACM SC 2007.

[30] K. Maheshwari, et. al. “Flexible Cloud Computing through Swift Coasters.” In Proceedings of Cloud Computing and its Applications, Chicago, April 2011. Open Cloud Consortium.

[31] K. Ousterhout et. al. “Batch Sampling: Low Overhead Scheduling for Sub-Second Prallel Jobs.” University of California, Berkeley, 2012.

[32] D. Bader, R. Pennington. "Cluster Computing: Applications". Georgia Tech College of Computing, June 1996

[33] M. Livny, J. Basney, R. Raman, T. Tannenbaum. “Mechanisms for High Throughput Computing,” SPEEDUP Journal 1(1), 1997

[34] I. Foster and C. Kesselman, Eds., “The Grid: Blueprint for a Future Computing Infrastructure”, “Chapter 2: Computational Grids.” Morgan Kaufmann Publishers, 1999

[35] C. Catlett, et al. “TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications,” HPC 2006

[36] Open Science Grid (OSG), https://www.opensciencegrid.org/bin/view, 2013

[37] F. Gagliardi, The EGEE European Grid Infrastructure Project, LNCS, Volume 3402/2005, p. 194-203, 2005

[38] I. Foster, “Globus Toolkit Version 4: Software for Service-Oriented Systems,” Conference on Network and Parallel Computing, 2005

[39] D. W. Erwin and D. F. Snelling, “UNICORE: A Grid Computing Environment”, Euro-Par 2001, LNCS Volume 2150/2001: p. 825-834, 2001

[40] Top500, November 2012, http://www.top500.org/lists/2012/11/, 2013

[41] P. Helland, Microsoft, “The Irresistible Forces Meet the Movable Objects”, November 9th, 2007

[42] V. Sarkar, S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, T. Sterling, “ExaScale Software Study: Software Challenges in Extreme Scale Systems”, ExaScale Computing Study, DARPA IPTO, 2009.

[43] LSF: http://platform.com/Products/TheLSFSuite/Batch, 2012.

[44] L. V. Kal´e et. al. “Comparing the performance of two dynamic load distribution methods,” In Proceedings of the 1988 International Conference on Parallel Processing, pages 8–11, August 1988.

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

[46] A. Sinha and L.V. Kal´e, “A load balancing strategy for prioritized execution of tasks,” In International Parallel Processing Symposium, pages 230–237, April 1993.

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

[48] M. Frigo, et. al, “The implementation of the Cilk-5 multithreaded language,” In Proc. Conf. on Prog. Language Design and Implementation (PLDI), pages 212–223. ACM SIGPLAN, 1998.

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

[50] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu, “ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, to appear at the 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2013

[51] IntrBlue Gene/P Solution: http://www.top500.org/system/176322, 2013.

[52] Google Protocol Buffers: https://developers.google.com/protocol-buffers/, 2013

[53] Coefficient Variance: http://en.wikipedia.org/wiki/Coefficient_of_variation, 2013.

[54] Directed Acyclic Graph: http://en.wikipedia.org/wiki/Directed_acyclic_graph, 2013

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

[56] FusionFS: Fusion Distributed File System, http://datasys.cs.iit.edu/projects/FusionFS/, 2013.

[57] K. Wang, I. Raicu. "SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascales", Oral Qualifier, Illinois Institute of Technology, 2012

[58] I. Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", Computer Science Dept., University of Chicago, Doctorate Dissertation, March 2009

[59] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C.M. Moretti, A. Chaudhary, D. Thain. “Towards Data Intensive Many-Task Computing”, book chapter in Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, IGI Global Publishers, 2011

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

[61] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, K. Iskra, P. Beckman, I. Foster. “Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers”, Scientific Discovery through Advanced Computing Conference (SciDAC09), 2009

[62] I. Raicu, I. Foster, M. Wilde, Z. Zhang, A. Szalay, K. Iskra, P. Beckman, Y. Zhao, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, D. Thain. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010

[63] C. Dumitrescu, I. Raicu, I. Foster. “Experiences in Running Workloads over Grid3”, The 4th International Conference on Grid and Cooperative Computing (GCC 2005)

[64] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, M. Wilde. “Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments”, book chapter in Grid Computing Research Progress, ISBN: 978-1-60456-404-4, Nova Publisher 2008