MATRIX: MAny-Task computing execution fabRIc at eXascale

Ke Wang, Anupam Rajendran, Ioan Raicu
Illinois Institute Technology
Technical Report, Sept. 2013

ABSTRACT

Efficiently scheduling large number of jobs over large-scale distributed systems is critical in achieving high system utilization and throughput. Today's state-of-the-art job management 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 of failure. In designing the next-generation distributed job management system, we must address new challenges such as load balancing. This paper presents the design, analysis and implementation of a distributed execution fabric called MATRIX (MAny-Task computing execution fabRIc at eXascale). MATRIX utilizes an 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, as well as task dependencies in the execution of complex large-scale workflows. We have evaluated it using synthetic workloads up to 4K-cores on an IBM Blue Gene/P supercomputer, and have shown high efficiency rates (e.g. 85%+) are possible with certain workloads with task granularities as low as 64ms. MATRIX has shown throughput rates as high as 13K tasks/sec at 4K-core scales (one to two orders of magnitude higher than existing centralized systems). We also explore the feasibility of adaptive work stealing up to 1M-node scale through simulations.

1. INTRODUCTION

The Many-Task Computing (MTC) paradigm [12][13][57][61] 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 [58]. 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][60][1], 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]). For completeness, job management systems (e.g. Slurm [2], Condor [3][4], PBS [5], SGE [6]) can also be used to support a subset of MTC workloads (e.g. HTC). Unfortunately, most of these systems have centralized Master/Slaves architecture (with the exception of Falkon supporting hierarchical architecture and Sparrow supporting a distributed architecture), where a centralized server is in charge of the resource provisioning and job/task execution. These systems have worked well in clusters, grids, and supercomputers with coarse granular workloads [1], but it has poor scalability at the extreme scales of petascale systems (and beyond) with fine-granular workloads [13][29].

With the dramatically increase of the scales of today's distributed systems, it is critical 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].One approach towards more efficient job managers is to depart from centralized scheduling towards complete decentralization. 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 many schedulers and compute nodes.

Load balancing is the technique of distributing workloads as evenly as possible 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][1] 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 [10]. We explore work stealing as an efficient method for load balancing tasks in a real system, MATRIX, at scales of 1K-nodes and 4K-cores. We study the performance of MATRIX in depth, including understanding the network traffic generated by the work stealing algorithm. We also use simulations to explore the feasibility of adaptive work stealing up to 1M-node scales (the expected scales at exascales).

The main contributions of this paper are as follows:

1. *Proposed an adaptive work stealing algorithm, which applies dynamic multiple random neighbor selection, and adaptive polling interval techniques, as well as identified parameters for best work stealing performance*
2. *Design and implement MATRIX to support distributed task scheduling and management through an adaptive work stealing algorithm for both MTC and HPC workloads*
3. *Evaluate the functionality of MATRIX using different workload types (e.g. Bag of Tasks, Fan-In DAG, Fan-Out DAG, Pipeline DAG and Complex Random DAG) and granularity (e.g. 64ms to 8 seconds per task) at scales of 64 nodes up to 1024 nodes*
4. *Compare MATRIX with Falkon (a light weight task execution framework that supports both a centralized and hierarchical architecture) at up to 1024 node scales*
5. *Explored the feasibility of adaptive work stealing at millions of nodes and billions of core scales through the SimMatrix simulator*

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][1]. 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 [1]. The largest studies to date of work stealing have been at thousands of cores scales, showing good to excellent efficiency depending on the workloads [1].

3. DESIGN AND IMPLEMENTATION OF MATRIX

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 Adaptive Work Stealing

3.1.1 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, as the traditional work stealing randomly which selects one neighbor to steal tasks [1], could yield poor performance at extreme scales, a multiple random neighbor selection strategy is used which randomly selects several neighbors instead of one, and chooses the most heavily loaded neighbor to steal tasks. The optimal number of neighbors for both static and dynamic selection are identified through SimMatrix [10]. The multiple-random neighbor selection algorithm is given in Algorithm 1. The time complexity of Algorithm 1 is $\Theta(n)$, where n is the number of neighbors.

3.1.2 Adaptive Poll Interval

When a node fails to steal tasks from the selected neighbors it is either because all selected neighbors have no more tasks, or the most heavily loaded neighbor had already executed all the tasks at the time stealing. To keep a check on continuous failing, every node on failing waits for a period of time called poll interval before which it will attempt to steal again. Adaptive poll interval policy helps to achieve reasonable performance while still keeping the work stealing algorithm responsive. If the polling interval is very large, work stealing would not respond quickly to change conditions, and would lead to poor load balancing. Therefore, the poll interval of a node is changed dynamically similar to the exponential back-off approach in the TCP networking protocol [49]. The default poll interval is set to be a small

value (e.g. 1 ms). Once a node successfully steals, the poll interval is set back to the initial small value. The specification of the adaptive work stealing algorithm is given in Algorithm 2.

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
```

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
```

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 for m nodes, each one talks to n neighbors, so within log(m) steps, ideally the tasks should be distributed across all the nodes.

3.2 Architecture Overview

The components of MATRIX and the communication signals among all the components are shown in Figure 1. 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.

MATRIX supports single core tasks, single node tasks, or multi-node tasks. It also supports task dependency, which means the order of execution among a workload's tasks can be specified as a part of task description and MATRIX would enforce the execution 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.

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 a specific order (based on task dependencies) or in arbitrary order (if no dependencies

are specified). In the background all compute nodes distribute the workload among themselves until the load is balanced using the adaptive work stealing algorithm. The client periodically monitors the status of workload until all the tasks present in the workload are executed and completed.

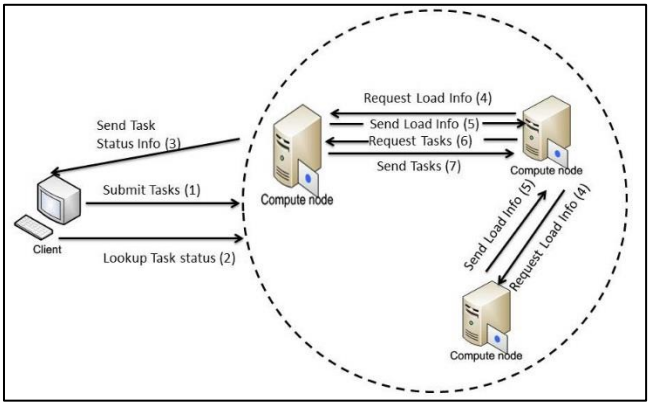

Figure 1. MATRIX components and communications among components

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 section 3.1 .

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.

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.

3.4 Task Assignment

Broadly, MATRIX supports two types of task assignment: best case and worst case assignment. The architectures are shown in Figure 2. 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 2 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.

In the worst case situation (Figure 2 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 log2 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.

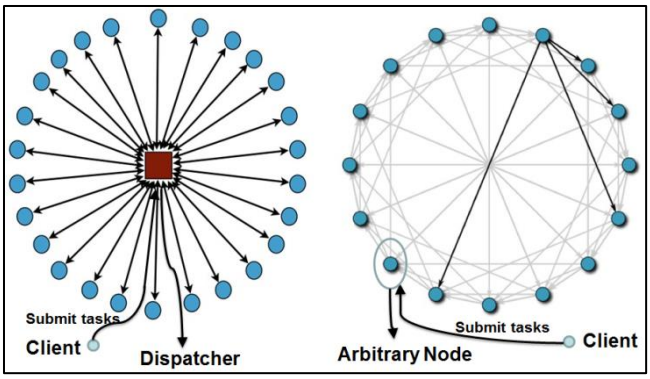

Figure 2. MATRIX architecture for both best case and worst case scheduling

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. This is shown in Figure 3. 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.

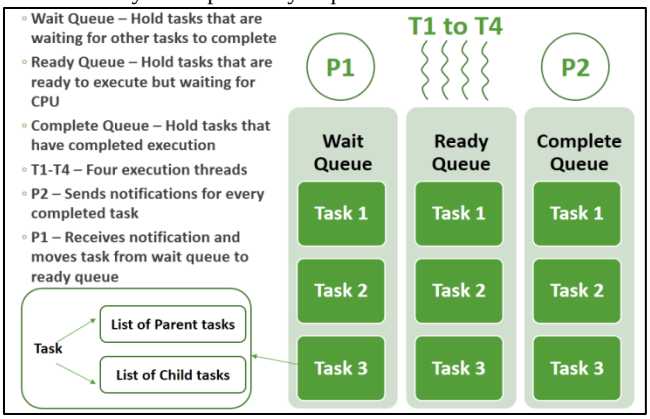

Figure 3. Execution Unit in MATRIX

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 to values discovered in Section 1.1. 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.

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. The monitoring dispatcher is only used for debugging and for ease of visualization of the system state. It is not used as part of the work stealing algorithm.

3.8 Distributed Scheduling for HPC Workloads

In order to support HPC workloads (multi-node jobs), we modified the adaptive work stealing algorithm to support mutli-node tasks. This allows the client to submit jobs that require multiple nodes. We modified the implementation by having a fixed number of compute slots on every node. Every job description has two key things: m - number of compute slots and the task to be run on every compute slot.

For every job in the ready queue, if the current node has the sufficient number of compute slots for that job, then the execution unit just launches the job in that node itself. Else, it does "resource stealing".

This is similar to work stealing with the difference that instead of stealing tasks, it steals compute slots. The algorithm for neighbor selection and stealing is the same. Every time when a node receives a resource-steal request, it locks half of its compute slots and sends these slot-ids to the node who requested for resource. The reason we have chosen the number of slots to be locked as half is because we believe that stealing half tasks in the work stealing gives optimal results. So we adopted the same number for resource stealing. After receiving the list of slot-ids from all neighbors, the execution unit checks if there are sufficient number of compute slots for the job. If unsuccessful, it releases all the locked slots. Otherwise it launches the job on the first m slots and releases any excess slots. Every compute slot, after the completion of task, sends the result back to the node which launched the job. This node after receiving the result from all the m slots, sends the acknowledgement to the client. Every compute slot after execution of a task, is automatically released.

We have verified the functionality of the resource stealing algorithm at small scales (16-nodes). However, more work is needed to scale up the HPC support to 1K-node scales as we have done for the MTC workload support.

4. PERFORMANCE EVALUATION

This section presents the experimental hardware and software environments, the evaluation metrics, the workloads generation, the throughput of dispatcher, the throughput of the run-time system, the comparison of MATRIX with Falkon [29] and SimMatrix [10], and the study of scalability of 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 Blue Gene/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 Blue Gene/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 four different workload: Bag of tasks, Fan-In DAG, Fan-Out DAG and Pipeline DAG. All the tasks in the DAG had a run-time of 1 second. These different workloads are shown in Figure 4.

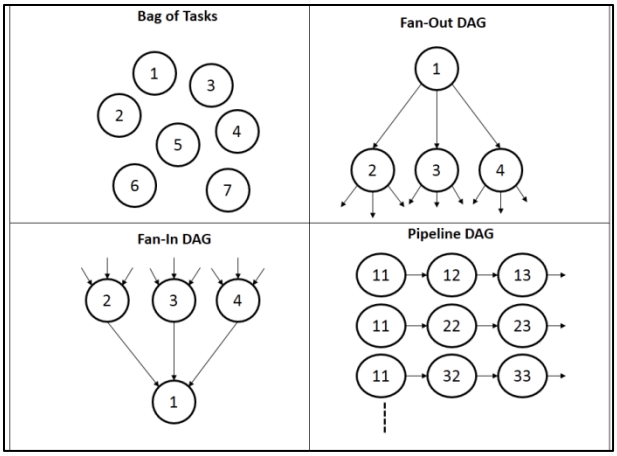

Figure 4. Different Types of Workload for evaluating functionality of MATRIX

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 due to less communication overhead to keep track of the dependencies.

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.

Fan-Out DAG: This workload is similar to Fan-In DAG, except the Fan-Out DAG is obtained be reversing the Fan-In DAG. The performance of this workload depends on the out-degree of nodes 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.

Second, 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 Evaluating the Adaptive Work Stealing Algorithm with MATRIX

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

4.2.1 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 5 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.

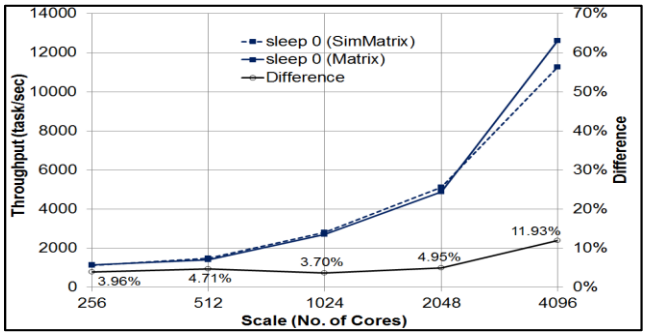

Figure 5. Comparison of MATRIX and SimMatrix throughput from 256 to 4K-cores

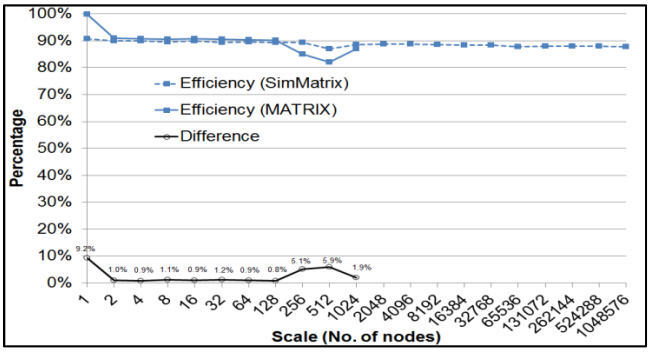

Figure 6. Comparison of MATRIX and SimMatrix performance up to 1024 nodes

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 6. 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.

4.2.2 Comparison: MATRIX vs. Falkon

Figure 7 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%.

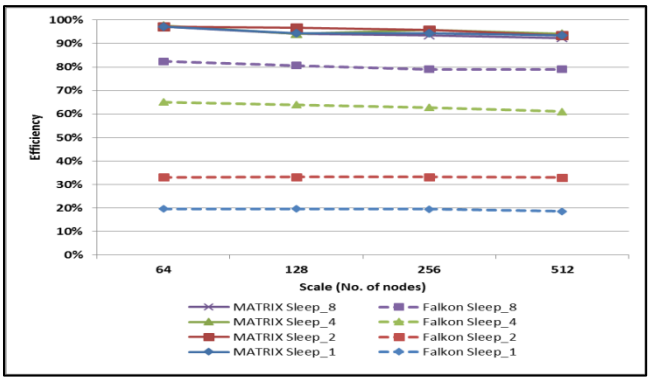

Figure 7. Comparison of MATRIX and Falkon efficiency across 256 to 2K-cores

4.2.3 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 8 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.

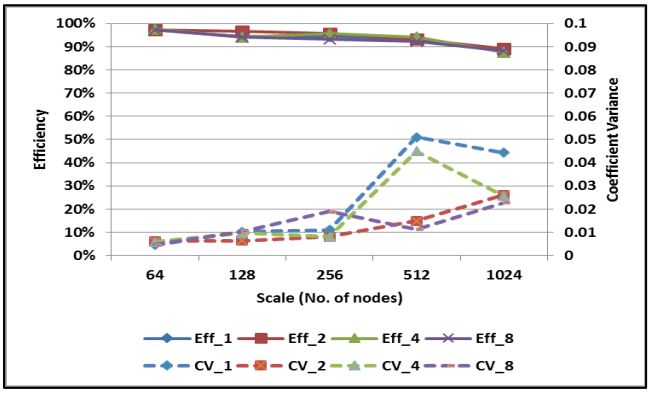

Figure 8. Scalability of Adaptive Work Stealing algorithm for long tasks

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).

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 9 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.

4.2.4 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 10.

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.

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.

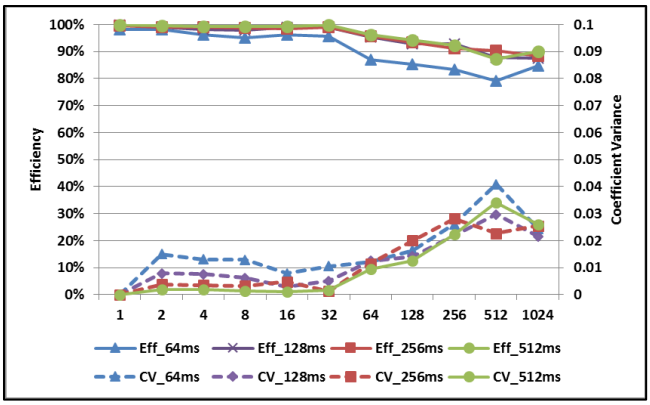

Figure 9. Scalability of Adaptive Work Stealing algorithm for fine granular tasks

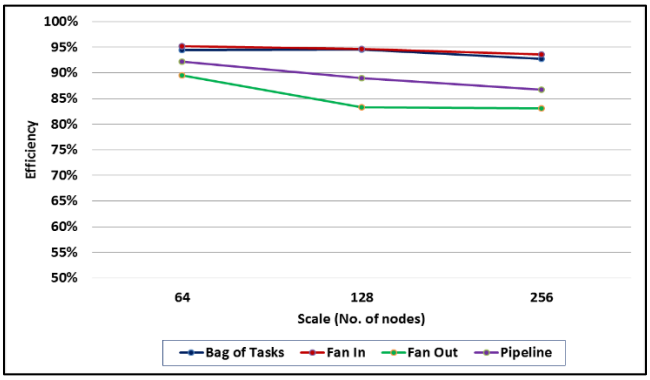

Figure 10. Analysis of work stealing using different workload types via MATRIX

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 [10] (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.

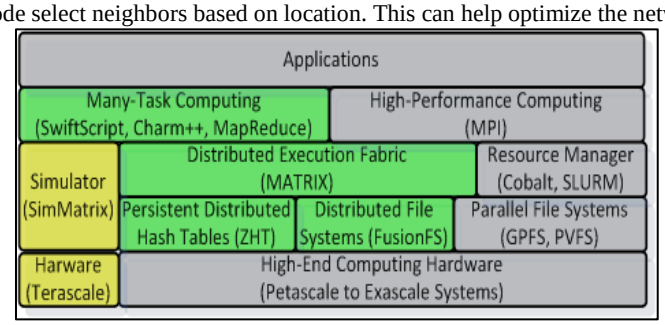

Figure 11. Building blocks for future parallel programming systems and distributed applications

We will also continue to develop the MATRIX system. Based on the simulation results, we expect that MATRIX should scale to 160Kcores on the IBM Blue Gene/P supercomputer we conducted our preliminary evaluation. We also plan to test it on the newly built IBM Blue Gene/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 [59] FusionFS [56] and large scale programming runtime systems Charm++ [27]. A potential future software stack is shown in Figure 11. 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].

6. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation grant NSF-1054974. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory. We also want to thank our collaborators for the valuable help that made this work possible, namely Tonglin Li, Kevin Brandstatter and Zhao Zhang.

7. REFERENCES

[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] Ke Wang, Kevin Brandstatter, Ioan Raicu. "SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascales", ACM HPC 2013

[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] 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

[58] 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

[59] 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

[60] 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

[61] 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

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