SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascales

Ke Wang[1], Ioan Raicu[1,2]
kwang22@hawk.iit.edu, iraicu@cs.iit.edu

[1]*Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA*
[2]*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL, USA*

ABSTRACT
Exascale computers will enable the unraveling of significant scientific mysteries. Predictions are that by 2019, supercomputers will reach exascales with millions of nodes and billions of threads of execution. Many-task computing (MTC) is a new viable distributed paradigm for extreme-scale supercomputing. The MTC paradigm can address four of the five major challenges of exascale computing, namely concurrency, resilience, heterogeneity, and I/O and memory; this work specifically addresses the first three major challenges. This paper presents a new light-weight and scalable discrete event simulator, SimMatrix, that enables the exploration of distributed scheduling for MTC workloads at exascale levels with up to 1 million nodes and 1 billion cores. SimMatrix is validated against real MTC workloads executed under Falkon at petascale levels, with 40K nodes and 160K-cores. Centralized scheduling is compared and contrasted to distributed scheduling; this work adopts work stealing, as an efficient and scalable approach to distributed load balancing. It explores a wide range of parameters important to understand work stealing at exascale levels, such as number of tasks to steal, number of neighbors of a node, static or dynamic neighbors, and different workloads. Experiment results show that the centralized scheduling saturates at small number of nodes, while the distributed scheduler configured with optimal parameters could scale up to 1 million nodes and 1 billion cores without any explicit upper bound. SimMatrix is light-weight and scalable, having been tested up to 1 billion cores and 10 billion tasks with modest resources (e.g. 200GB of memory and 256-core hours).

Categories and Subject Descriptors
C.2.4 [Distributed Systems]; C.5.1 [Large and Medium (``Mainframe") Computers]; D.4.8 [Performance]

General Terms
Management, Performance.

Keywords
Exascale, Many-Task Computing, MTC, Scheduling, Work Stealing, Load Balancing.

HPDC'12, June 18–22, 2012, Delft, The Netherlands.

1. INTRODUCTION
Exascale (i.e. 10^{18} operations/sec) computers will enable the unraveling of significant scientific mysteries. The US President made reaching exascales a top national priority, claiming it will "dramatically increase our ability to understand the world around us through simulation". [1] There are many domains (e.g. weather modeling, global warming, national security, energy, drug discovery, etc.) that will achieve revolutionary advancements due to exascale computing. Predictions are that 2019 will be the year of exascales, with millions of nodes and billions of threads of execution. [2][3][4]

1.1 Challenges at Exascale
The era of manycore and exascales computing will bring new fundamental challenges in how we build computing systems and its 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 five most significant challenges of exascale computing are: concurrency, resilience, I/O and memory, heterogeneity, and energy. Any one of these challenges, if left unaddressed, could halt progress towards exascale computing.

Concurrency refers to programmability, and how we will harness the many magnitude orders of increased parallelism fueled by the manycore computing era. The largest supercomputers have increased in parallelism at an alarming rate. In 1993, the largest supercomputers had 1K-cores (0.00006PF/s), in 2004 8K-cores (0.035PF/s) and in 2011 688K-cores (10.5PF/s); by 2019, supercomputers will likely reach billions of threads/cores (~1000PF/s). [2] 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. Today's programming languages are inadequate to automatically harness even modest parallelism. Popular programming languages (e.g. C/C++, Java) are unlikely to scale to manycore levels given the level of expertise needed to parallelize applications; furthermore, their imperative nature makes them difficult to parallelize automatically.

Resilience refers to the capability of making both the infrastructure (hardware) and applications fault tolerant in face of a decreasing mean-time-to-failure (MTTF). The MPI programming model [5] is unlikely to survive in its current form, given how brittle the programming paradigm is due to

its synchronous nature. MPI was designed in the 1980s, when parallelism was on the order of 10s of processors; MPI has already evolved significantly, however it is facing difficulty in scaling up on large machines due to an increasing cost to checkpoint (the state-of-the-art in HPC reliability) and decreasing MTTF [6]. In order to achieve exascale levels with millions of nodes and billions of threads of execution, 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.

I/O and memory refers to optimizing and minimizing data movement through the memory hierarchy (e.g. persistent storage, solid state memory, volatile memory, caches, and registers). Exascale will bring unique challenges to the memory hierarchy never seen before in supercomputing, such as a significant increase in concurrency at both the node level (number of cores is increasing at a faster rate than the memory subsystem performance), and at the infrastructure level (number of cores is increasing at a faster rate than persistent storage performance). The memory hierarchy will change with new technologies (e.g. non-volatile memory), 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.

Heterogeneous systems offer the opportunity to exploit the extremely high performance heterogeneous computing resources (e.g. accelerators, GPUs, MIC, FPGA) while still providing a general purpose platform. In the November 2011 Top500 rankings, four of the top ten supercomputers had a heterogeneous architecture. A recent study [3] has also shown that exascale systems will more likely have heterogamous architectures, as opposed to strawman or heavyweight architectures.

Power refers to the ability to keep the power consumption at a reasonable level, so that the cost to power a system does not dominate the cost of ownership. The DARPA Exascale report [2] defined probably the single most important metric, namely the energy per flop. 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.

1.2 Defining Many-Task Computing

Many-Task Computing (MTC) was introduced by Raicu et al. [7][8] 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 have features that distinguish them from typical HTC applications. HTC applications have traditionally run on platforms such as grids and clusters, through either workflow systems or parallel programming systems. MTC applications, in contrast, will 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 and is often a natural way to build the application. 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. However, HPC systems often lack a dynamic resource provisioning feature, are not ideal for task communication via the file system, and have an I/O system that is not optimized for MTC-style 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 tomorrows systems as a priority and can address many of the HPC shortcomings at extreme scales.

1.3 Contributions

The main contributions of this paper are as follows:

(1) Develop a new light-weight and scalable discrete event simulator that enables distributed scheduling for MTC workloads at exascales.

(2) Provide evidence that work stealing is a scalable method to achieve load balance, even at exascales.

(3) Identified optimal parameters affecting the performance of work stealing; at the largest scales, in order to achieve the best work stealing performance, we found the number of tasks to steal is half and there must be a squared root number of dynamic neighbors (e.g. at 1M nodes, we would need 1K neighbors).

1.4 Organization

The rest of the paper is organized as follows: Section 2 gives some background information, which is necessary to make the paper self-contained. In Section 3, we propose the system architecture and the implementation of the simulator. We

show the evaluation and the experiment results of the performances of the simulator in Section 4. In Section 5, related works about job scheduling systems, work stealing and load balancing are discussed. Conclusions are drawn and future work is envisioned in Section 6.

2. BACKGROUND INFORMATION

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. Job management in MTC should support the granularity at the node/core level at extreme scales. The system could be centralized, where a single dispatcher manages the job submission, job assignment, 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. Centralized dispatcher suffers scalability, due to its limited processing capacity. Hierarchical dispatchers have the problem of long job turnaround time, because of the communications between different-layer dispatchers. Distributed scheduling with innovative load balancing techniques is an efficient way to maintain scalability, high performance and reliability at exascale systems.

Distributed Load balancing is the technique of distributing computational and communication loads evenly across processors of a parallel machine, or across nodes of a supercomputer, so that no single processor or computing node is overloaded. Load balancing strategies can be divided into two broad categories – those for applications where new tasks are created and scheduled during execution (i.e. task scheduling) and those for iterative applications with persistent load patterns. [9] Clients will be able to submit work to any queue, and each queue will have the choice of executing the work locally, or forwarding the work to another queue based on some function it is optimizing. Load balancing can be used to optimize resource utilization, data movement, power consumption, or any combination of these.

Work stealing [10] refers to a distributed load balancing approach in which processors needing work steal computational tasks from other processors. 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 if a node fails to steal takes from others.

3. PROPOSED SOLUTION

This work investigates the usability of work stealing towards exascale levels of parallelism, and investigate the optimal parameters (e.g. worker's connectivity, number of tasks to steal, static/dynamic neighbors, etc) needed to make work stealing a viable and efficient distributed load balancing mechanism. This work seeks to prove that given certain work stealing parameters, that good load balancing can be obtained in a finite amount of time, and that resource partitioning is unlikely to occur.

3.1 SimMatrix Architecture

SimMatrix supports both centralized and distributed scheduling, whose architectures are shown in Figure 1. For simplicity, we assign consecutive integer numbers as the ids of each node, ranging from 0 to the number of nodes N-1.

In the centralized situation, the clients submit tasks to the task waiting queue of the single dispatcher, which then assigns tasks to the first available node based on the load information of every node in the FIFO way. None nodes have task waiting queue. If all cores are occupied, the dispatcher will wait until some tasks are finished, and then send tasks again until all finished.

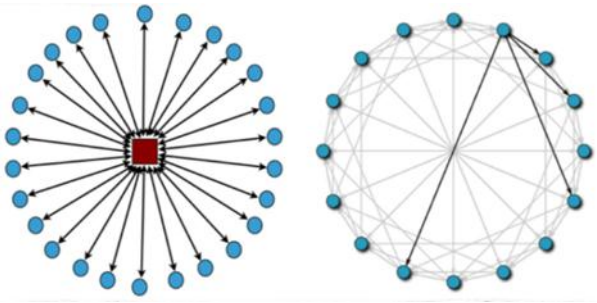

Figure 1: Simulation architectures; the left part is the centralized one with a single dispatcher connecting all nodes, the right part is the homogeneous distributed topology with each node having the same number of cores and neighbors

In the distributed scheduler case, the clients submit tasks to any arbitrary node. For simplicity, we let the clients submit tasks to the first node, whose id is 0. This is the worst scenario from a load balancing perspective. Every node has its own task waiting queue, and the same number of neighbors. Figure 1 shows a fully connected topology of the nodes; in this example, the neighbors of a node are just its several left and right nodes with consecutive ids. Anytime when a node has no tasks in its task waiting queue, it will ask the load information from all the neighbors one by one, and try to steal tasks from the one having the heaviest load. 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 task waiting queue, if which is not empty, it will send some tasks to the neighbor, or it will send information to signal a steal failure. When a node fails to steal tasks, it will wait some time, and then try again. We call this waiting time the poll interval. The termination condition is that all the tasks submitted by client are finished. We do this by setting a global counter which can be read by all simulator threads to signal the termination of the simulation.

3.2 Task Description

The tasks in our simulator are MTC per-core tasks, which are independent with each other. Each task has the attributes such as task length (the time taken by a core to finish the task), task size (data size required by the task), task timestamps recording the times when a task is submitted by client, when a task arrives the computing node, and when it is finished. We expect that some other higher level system is

managing all the task dependencies, such as some parallel programming system (e.g. Swift [13], Charm++ [14], etc).

3.3 Global Variables

There are several global variables in our simulator. These variables define the communication networks, the scale of the system, the work stealing parameters, etc. The names and descriptions of the variables are listed in Table 1. The bolded ones are specific for the distributed scheduling, while others are for both the centralized and distributed scheduling.

Table 1: Global Variables and Descriptions

Name	type	Description
numNode	int	Number of nodes of the system
linkSpeed	double	The link speed of the network
procTimePerTask	double	Time the server takes to determine which node to dispatch tasks
networkLatency	double	Network latency for every communication
numCoresPerNode	int	Number of cores of a node
logTimeInterval	double	The time interval to write log
numNeighbors	int	Number of neighbors a node has
numStealWork	int	Number of tasks to steal
StealInterv	double	The initial poll interval

3.4 Discrete Event Simulation

SimMatrix is built as a discrete event simulator as it was the only viable approach to ensuring scalability to exascales (millions of nodes and billions of cores) on a single shared memory system. The single shared memory system requirement came from aiming for a simple to implement and run simulator.

Before settling on SimMatrix being a discrete event driven simulator, we performed experiments to explore how many threads could be supported under Java, and we found that on our 48-core system with 256GB of memory, we were limited to 32K threads. Furthermore, at this scale of concurrent threads, the threads active state was so infrequent (as there were only 48 physical cores) that it made the simulator extremely slow and inaccurate. Since it was not feasible for us to run 1M threads in Java (or C/C++ which we also explored), we abandoned the idea of creating a separate thread per simulated node.

We therefore decided on creating a unique object per simulated node, and convert any behavior to an event. All events are put in a global event queue (see Section 3.5), and sorted based on the occurrence time.

3.5 Global Event Queue

The global event queue is the heart of the SimMatrix simulator, and it is used to keep the millions to billions of events active at any point in time (when simulating an exascale system) in an organized fashion. There is only one global event queue for the entire simulation, no matter how many nodes or tasks are being simulated. The first event in the queue is always the next event to be process. Every time an event is removed from the event queue for processing, we advance the simulation time to the occurrence time of the event.

There are several events in the simulator listed below, and the ones marked with an * are specific for the distributed scheduling:

- **TaskEnd:** Signals a task completion event (which inherently frees a processing core). This event causes the scheduler to advance to the next task to schedule. In the centralized scheduler, the compute node (with the available core) will wait for the dispatcher to assign more tasks. In the distributed scheduler, the compute node starts to execute another task (assuming it has tasks in the waiting queue) by inserting another 'TaskEnd' event with a future time (when the new task is expected to complete), or it invokes the work stealing algorithm to take tasks from its neighbors.
- **Submission:** In the centralized scheduler, the client submits some number of tasks to the centralized dispatcher.
- **Log:** Signals the record writing to a summary log file, including the information such as the simulation time, number of all cores, number of executing cores, waiting queue length, throughput, etc. The Log event can be used to generate periodic logs for monitoring and visualization purposes.
- ***Steal:** Signals the work stealing algorithm to invoke the steal operation. In particular, a node asks for tasks from its neighbors. First, the node will ask for the load information of its neighbors one by one, and then selects the one that has the heaviest load to steal tasks by inserting a 'TaskReception' event. If all neighbors have no tasks, the node will wait for some time to 'Steal' again.
- ***TaskDispatch**: Signals the task dispatch to a neighbor. If at the current time, the node happens to have no tasks, it will inform the neighbor to ask for tasks again, by inserting a 'Steal' event on the neighbor's side. Else, the node will dispatch a part of its waiting tasks to the neighbor by inserting a 'TaskReception' event on that neighbor's side.
- ***TaskReception:** Signals the receiving node to increase the length of its task waiting queue. The task received could be from the submitted client, or from a neighbor.
- ***Visualization**: It is used as an event to visualize the load information of all nodes.

The state diagram of all the events are shown in **Figure 2**, where each state is an event that is executing, and the next state is the event to be inserted in the event queue signaled after finishing current event. The performance of the event queue is central to that of the simulator. It has to be scalable to many events (billions), and be subjected to frequently updates. All these operations need to re-order the queue. In our implementation, we use the TreeSet data structure [15],

which is a set whose elements are ordered using their natural ordering, or by a comparator provided at set creation time. In SimMatrix, it is ordered by a comparator based on the event occurrence time, along with the node ids, task ids or the event ids in the distributed scheduling. The TreeSet is implemented based on Red-Black tree [16], which guarantees $\Theta(\log n)$ time for removing and inserting, and $\Theta(1)$ time for getting the first event.

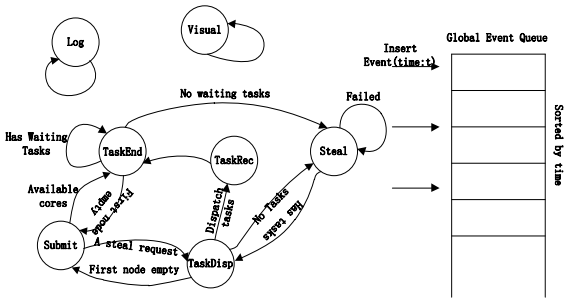

Figure 2: Event State Diagram

3.6 Node Load Information

In the centralized scheduler, the load information of all nodes is accessed by the dispatcher to determine the next node to assign tasks. The load is the number of busy cores ranging from 0 to the number of cores. The dispatcher can access the load information continuously as long as there are still waiting tasks. If we were to naively go through all the nodes to get the load information, the simulator would be highly inefficient when the number of nodes is large (e.g. 1 million). We implement the load information using a Hash Map [17]. The 'Keys' are the node load, while the 'Value' is corresponds to a hash set which contains the node ids whose loads are all equal to the 'Key'.

Each time when the dispatcher wants to assign some tasks to a node, it goes through all the 'Keys', and finds where the corresponding node's information is at. As the number of cores per node is relatively small (e.g. 1000 cores), we consider this lookup operation taking $\Theta(c)$ time, where c=1000. Once the right load level is identified, inserting, getting or removing an element in the nested hash set only takes $\Theta(1)$ time. This hierarchical nested data-structure helped reduce the practical time complexity by orders of magnitude, from a $\Theta(\log(n))$ to $\Theta(1)$.

In the distributed scheduling, the load of a node is the number of waiting tasks minus the number of idle cores. In order to keep programming easy, every node could access the load information of its neighbors directly. However, the simulator keeps track of the query and response overheads when asking for this load information.

3.7 Logs

In order to do statistical analysis, to help generate the results from Section 4, as well as for visualization purposes, we write some information into logs. We have two logs, one recording the per task information (can be very large for exascale simulations), while the other recording the summary over some defined unit of time (quite efficient regardless of scale of experiment). The per task log records information such as task ID, compute node ID, submission time, queue wait time, execution time, and exit code (whether it was successful or not). The summary log records information such as the 'simulation time', 'number of all cores', 'number of executing cores', 'wait queue length', 'throughput', etc.

The per task log is optional due to the potential large overhead and storage requirement. If enabled, a record gets logged whenever a 'TaskEnd' event happens. The summary log is mandatory, and is implemented by submitting events to the global event queue. At the beginning when simulation time is 0, we insert a 'Log' event. Every time when handling a 'Log' event, we remove it and insert the next 'Log' event which would happen some fixed simulation time later. In this way, we ensure that the increment of the simulation time between two consecutive records is constant.

3.8 Dynamic Task Submission

Both the centralized and distributed scheduling support dynamic task submission. Client could submit a couple of tasks to the dispatcher, or an arbitrary node dynamically when the number of waiting tasks is below some threshold. Dynamic task submission aims to reduce the memory foot-print of having more tasks submitted than available compute nodes/cores.

3.9 Poll Interval for Work Stealing

In the distributed scheduler, we implement a dynamic poll interval policy in order to achieve reasonable simulation performance while still keeping the work stealing algorithm responsive. Without a dynamic poll interval, 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, the work stealing algorithm would not respond quickly to changing conditions, and would lead to poor load balancing. Therefore, we change the poll interval of an idle node dynamically by doubling it each time when all of the neighbors have no tasks, and setting the poll interval back to the default small value whenever it steals some tasks successfully; this algorithm is similar to the exponential backoff algorithm in the TCP networking protocol. We set the default poll interval to be small value (e.g. 1 sec).

3.10 Implementation Details

SimMatrix has been developed in Java, and includes about 1500 lines of code. It has been tested on a variety of operating systems, from Linux to Windows. SimMatrix uses the uniform random distribution generator for producing workloads, and BufferedWriter for logging, as we found this to be the most efficient writer in Java (after comparing 6 ways to write to a file [18]). Also, we implement a Gamma Distribution random generator to produce the popular many-task computing workload. The source code is open source, and can be accessed from [12].

4. EVALUATION

This section present the evaluation methodology, metrics measured, the experimental hardware and software environments, as well as the results showing the scalability and performance of SimMatrix, plus the feasibility of utilizing work stealing at exascale levels.

Methodology: Since exascales is not anticipated until the end of the decade, we decided to explore work stealing through simulations to evaluate its feasibility at exascales with millions of nodes and billions of cores.

Hardware Environment: All experiments presented in this section are 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.

Software Environment: SimMatrix is developed 100% in JAVA; we used the Sun 64-bit JDK version 1.6.0_22. SimMatrix has no other dependencies.

4.1 Metrics

We use important metrics to evaluate the performance of our simulators. They are listed below:

- **Throughput:** Number of tasks finished per second. Calculated as total-number-of-tasks/simulation-time.
- **Efficiency:** the ratio between the ideal simulation time of completing a given workload and the real simulation time. The ideal simulation 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 [19] of the number of tasks finished by each node as a measure the load balancing. The smaller the coefficient variance, the better the load balancing is. It is calculated as the standard-deviation/average in terms of number of tasks finished by each node.
- **Scalability:** Total number of tasks, number of nodes, and number of cores supported.

4.2 Simulator Parameters

For all the experiments, we set some global variables to be appropriate constants, which are listed in Table 2.

Table 2: Values of Some Global Variables

Variables	Values
linkSpeed	10Gb/s
procTimePerTask	1 millisecond
networkLatency	10 microseconds
numCoresPerNode	1000
logTimeInterval	1 second

We choose values based on the BlueGene/P machine configured with Falkon [20] scheduler, and make the 'linkSpeed' 10 times faster, and 'networkLatency' ten times lower. This assumption is realistic given the rate of improvements in network performance.

4.3 Workloads

A variety of workloads, in terms of the task length, have been used in our experiments, including synthetic and real ones.

For synthetic workloads, we use uniform distributions with different average task lengths, such as 10s, 100s, 1000s, 5000s, 10000s, and 100000s. We name them ave_1, ave_10, ave_100, ave_5000, ave_10000, and ave_100000, respectively. Also, we use the workload where each task has the same length, i.e. 1s, and name it all_1.

For more realistic application workloads, we use the general 64±486s many task one, which has been shown to represent years of MTC workloads, comprising of hundreds of millions of tasks. [21] We generate this workload called mtc_64 by using a Gamma Distribution.

4.4 Validation

Before we try to explore MTC at exascales, we validated SimMatrix against the state-of-the-art MTC systems (e.g. Falkon), to ensure that the simulator can accurately predict the performance of current petascale systems. The results are shown in Figure 4 and Figure 4.

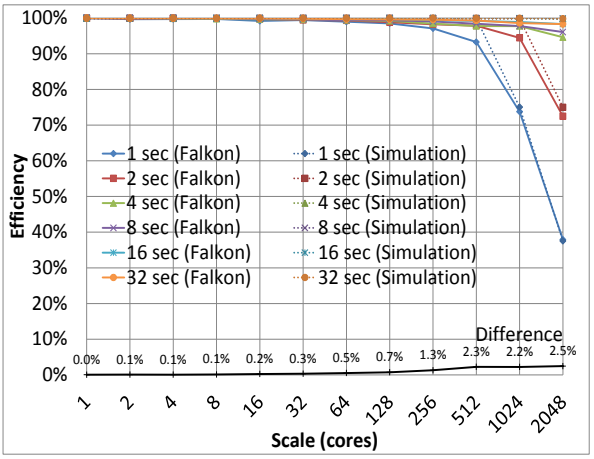

Figure 3: SimMatrix validation for the centralized scheduler, compared to the Falkon centralized scheduler up to 2K-cores [8]

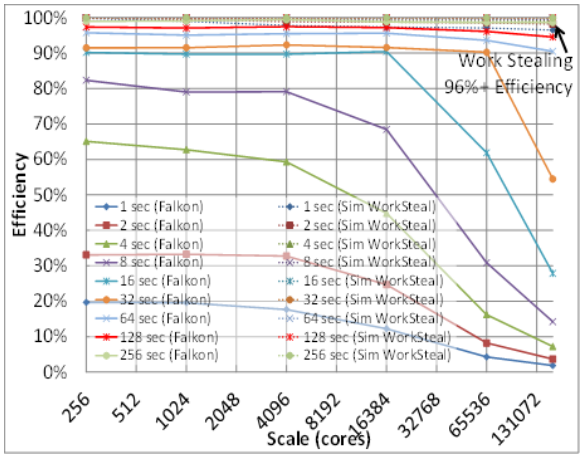

Figure 4: SimMatrix performance comparing work stealing efficiency to the Falkon naïve distributed scheduler up to 160K-cores [8]

Figure 3 presents the validation results when compared to the centralized Falkon scheduler up to 2K-cores; we measured a 2.5% difference in reported efficiency between our simulator (dotted lines) and Falkon (solid lines). In Figure 4, the work stealing approach is able to maintain a 96%+ efficiency even with 1 second tasks at full 160K-core scales, when Falkon was only able to achieve 2% efficiency with 1 sec tasks at

full scale, requiring task lengths of 256 seconds to achieve upper 90% efficiencies.For these experiments, we set the number of cores per node to be 4, as the case of BlueGene/P machine. The total number of tasks is 50000000, large enough to ensure that the experiments completed in a reasonable amount of time. The parameters of work stealing are set as the optimal values (as discovered by Sections 1.3). These results show that our simulator performs correctly (with less than 2.5% difference in predicted efficiency), and the distributed scheduling with work stealing configured with optimal parameters outperforms Falkon distributed dispatcher significantly (96% efficiency for work stealing when compared to 2% for Falkon).

4.5 Scalability of SimMatrix

We show our simulator's scalability, efficiency and resource requirement (time and memory) to run exascale experiments using ave_5000 workload. Figure 5 and Figure 6 show the results.

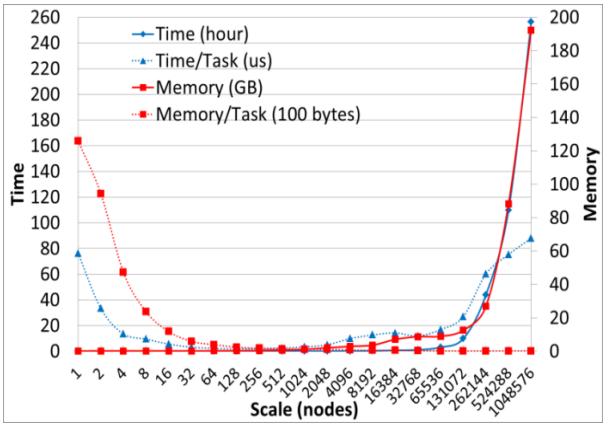

Figure 5: Scalability of the SimMatrix up to 1M nodes and 10B tasks

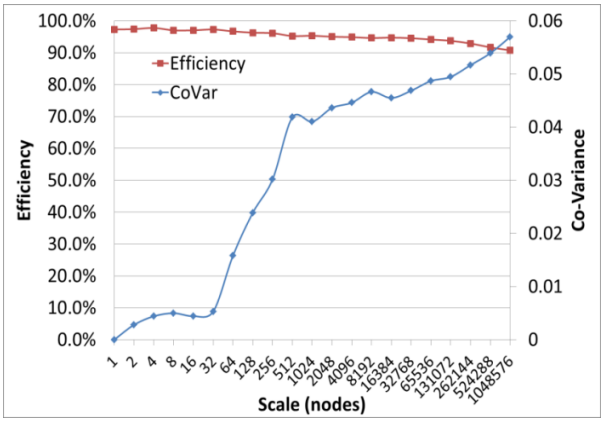

Figure 6: Efficiency and co-variance of work stealing at 1M nodes and 10B tasks

The results show that up to 1 million nodes, we could run workloads with 10 billion tasks in about 256 hours and 190GB memory. Work-stealing actually works quite well at extremely large scales, given the right work-stealing parameters. In Figure 6, we see an efficiency of 90%+ at a million node scale, with a co-variance of 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 10K tasks).

4.6 Work Stealing Parameter Space

There are several parameters that could affect the performance of work stealing, such as number of tasks to steal, number of neighbors of a node, static neighbors vs. dynamic random neighbors. We investigate them in great detail in this section. The experiments scale up to 8192 nodes, with each one 1000 cores. The number of tasks is 10 times of the number of cores. The ave_5000 workload is used. We do each experiment five times, and show the average efficiencies and standard deviations. We do weak scaling experiments.

4.6.1 Number of tasks to steal

In our five groups of experiments, steal_1, steal_2, steal_log, steal_sqrt, steal_half means steal 1, 2, logarithm base-2, square root, and half number of tasks respectively. We set numNeighbors = 2. The changes of the efficiency of each group with respect to the number of nodes are shown in Figure 7.

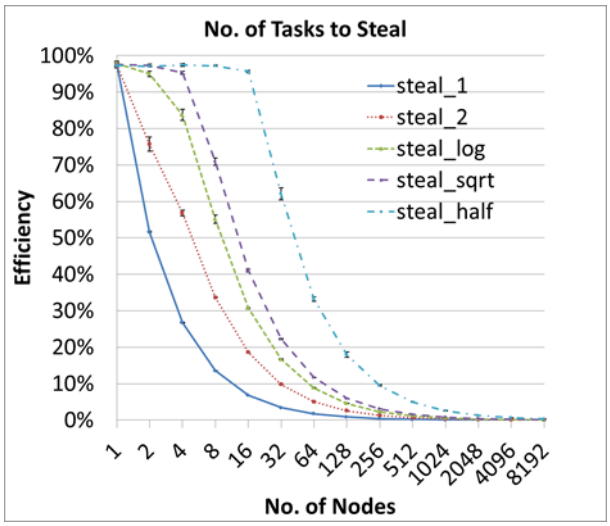

Figure 7: Efficiencies of different numbers of tasks to steal with respect to number of nodes

From Figure 7, we see that as the number of nodes increases, the efficiencies of steal_1, steal_2, steal_log, steal_sqrt decrease. The efficiency of steal_half keeps at the value of about 97% up to 8 nodes, and decreases after that. And the decrease speed of steal_half is the slowest. These results show that stealing half number of tasks is optimal, which confirms both our intuition and the results from prior work on work stealing [22]. The reason that steal_half is not perfect (efficiency is very low at large scale) for these experiments is that 2 neighbors of a node is not enough, and starvation can occur for some nodes that are too far in the ID namespace from the original compute node who is receiving all the task submissions. The conclusion of this experiment is that having a small number of static neighbors is not sufficient to achieve high efficiency even at modest scales. We also can generalize that stealing more tasks (less than half) generally produces higher efficiencies.

4.6.2 Number of neighbors of a node

4.6.2.1 Static Neighbors

In our experiments, 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 statically as consecutive ids in the ring topology at the beginning, and will not change. The changes of the efficiency of each group with respect to the number of nodes are shown in Figure 8.

The result shows that when the number of neighbors is no less than a quarter of all nodes, the efficiency will keep at the value of higher than 95% within 8192 nodes' scale. For other numbers of static neighbors, the efficiencies could not remain, and will drop down to very small values. We conclude that the optimal number of static neighbors is a quarter, as more neighbors do not improve performance significantly.

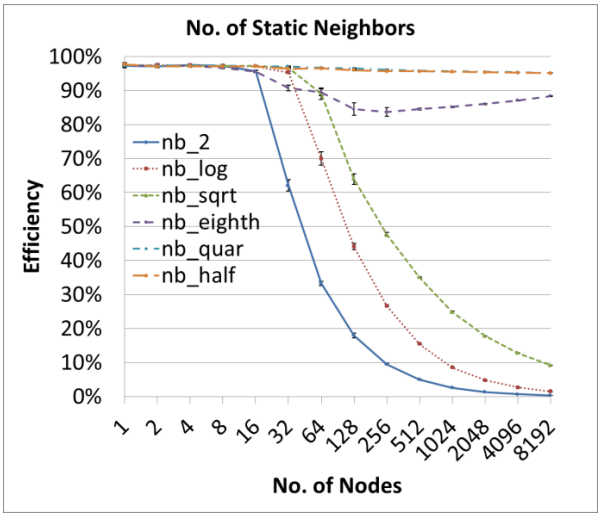

Figure 8: Efficiencies of different numbers of static neighbors with respect to number of nodes

However, in reality, a quarter neighbors is too many to make work stealing practical, especially for an exascale system with millions of nodes. In the search for a lower number of needed neighbors, we explore a dynamic random neighbor selection policy.

4.6.2.2 Dynamic Random Neighbors

For each node, whenever it does work stealing, it randomly selects some neighbors with uniform distribution. This policy will reduce the requirement of number of neighbors. We do 4 groups of experiments, nb_1, nb_2, nb_log, nb_sqrt. We first do nb_1 experiment until starting to saturate (the efficiency is less than 90%), then at that point, do nb_2, then nb_log, and nb_sqrt at last. The results are shown in Figure 9.

Figure 9 shows that nb_1 scales up to 512 nodes, nb_2 scales up to 2048 nodes, nb_log sclares up to 16384 nodes, and nb_sqrt scales up to 1 million nodes, remaining the efficiency at the value about 90%. Our conclusion is that dynamic random nb_sqrt is the best and could be used in general, but it might produce more neighbors than needed for 90% efficiency level. 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.

The optimal parameters for the aver_5000 workload and work stealing are to steal half the number of tasks from their neighbors, and to use the square root number of dynamic random neighbors.

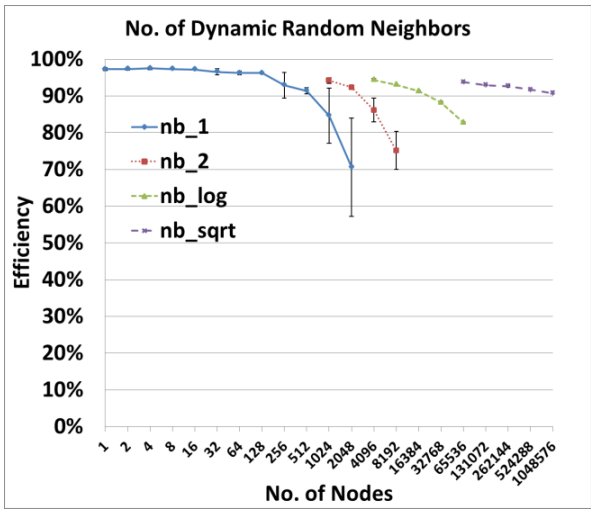

Figure 9: Efficiencies of different numbers of dynamic random neighbors with respect to number of nodes

4.7 Optimal Parameters for Different Workloads

We apply the optimal parameters to different workloads, such as ave_10, mtc_64, ave_100, ave_1000, ave_10000 to see how work stealing works. For ave_10, ave_100, ave_1000, we set the total number of tasks as 10 times of the number of cores. For mtc_64, we set the total number of tasks as 10000 times of the number of cores, that means for 16384 nodes, the number of tasks is 163,840,000,000 (163 billion tasks). The reason is that for mtc_64 workload, we generate it using Gama Distribution. It requires the sample space to be large enough to ensure the average is 64, and the standard deviation is 486, even for 1 node. The result is shown in Figure 10

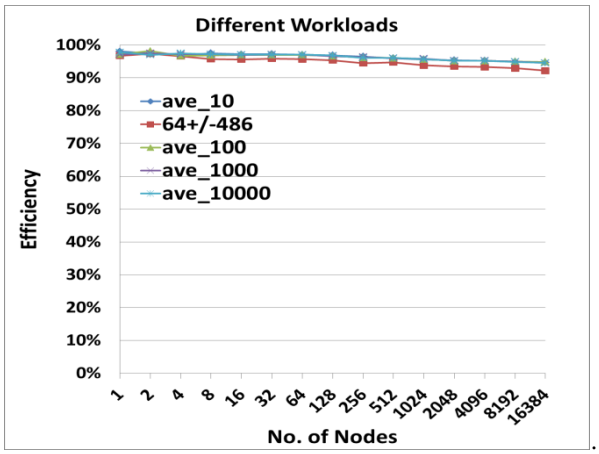

Figure 10: Efficiency of different workloads with same optimal parameters' configuration

From Figure 10, we see that, work stealing configured with the optimal parameters works quite well for all these 5 workloads within 16k nodes' scale (16,384,000 cores). Even for the real mtc_64 workload, given that the number of tasks is large enough, work stealing still works well. We do not expect the same trends to hold true as we scale up to 1M nodes. There results show that the optimal parameters we found for work stealing are general at some extent, and work stealing is a promising approach to load balancing at near exascale levels. We plan to extend this experiment in the final manuscript to show the efficiency of these various workloads up to 1M nodes.

4.8 Visualization of Load Balancing

In the experiments, we capture the changes of loads of all nodes with respect to the simulation time to visualize the performance of work stealing. Figure 11 shows 1024 nodes' situation.

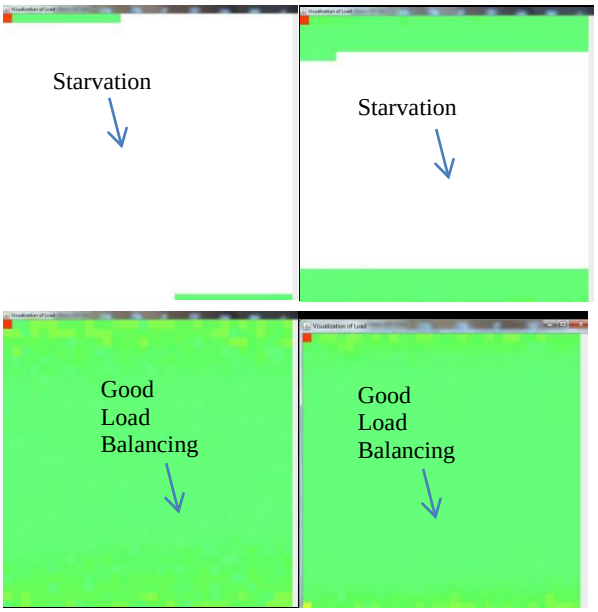

Figure 11: Visualization for 1024 nodes and ave_5000 workload for different number of neighbors; the upper left has 2 static neighbors, the upper right has square root static neighbors; the lower left has a quarter static neighbors, the lower right has square root dynamic random neighbors.

We present the representative graphs for different number of neighbors when the system is stable. The nodes are mapped to tiles depending just on their 'ids' and they are assigned in a row-wise manner. Due to our neighbors' distribution policy, it results that nodes that are beside each other row-wise are neighbors. We map the level of load of the nodes to colors, and each node it is represented as a tile in a canvas. The red square means the most loaded node (usually node 0, which accepted tasks from clients), the right ones are idle nodes, and the other ones are nodes with different loads represented with different colors. We see that when the number of neighbors is small, some nodes are starved (the top two figures that have some white tiles).

4.9 Comparison between centralized and distributed scheduling

We compare the centralized and distributed scheduling to understand the bottleneck of the centralized way, in terms of throughput. The number of tasks is 10 times of the number of cores. We do two groups of experiments. The first uses ave_5000 workload, and the second uses all_1, for both schedulers. The optimal parameters of work stealing are used. The throughputs with respect to the number of nodes are shown in Figure 12.

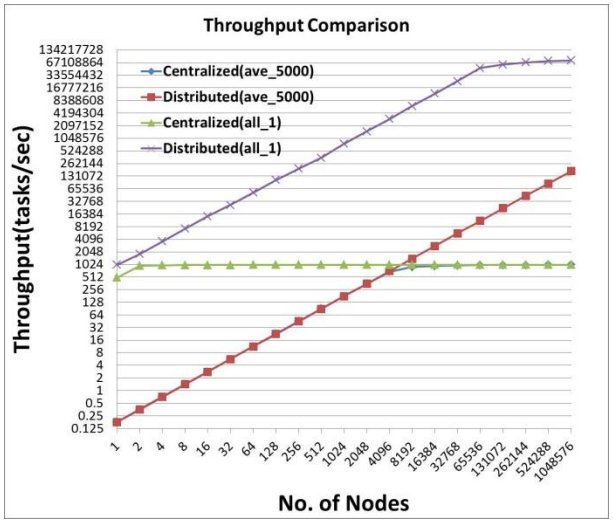

Figure 12: Throughput comparison between centralized and distributed scheduling

We see that for ave_5000 workload, after 8192 nodes, the centralized scheduling is saturated with upper bound throughput of about 1000, and the distributed one could perfectly scale up to one million nodes, the throughput doubles as the number of nodes doubles, and has no explicit upper bound. For all_1, the centralized scheduling saturates at about 32 nodes with upper bound throughput of about 1000, while the distributed one saturates at about 131072 nodes with throughput about 60M tasks/sec; it finally reaches 1M nodes with a throughput of 75M tasks/sec. The 1000 upper bound of the centralized scheduling is because of the processing time of the dispatcher per task (set to 1ms in the simulator). The reason that the distributed scheduler saturates is likely due to the growth of messages in relation to the number of nodes in the system. The average number of messages per task with respect to the system scale for the all_1 workload is shown in Figure 13 below. We see that after 131072 nodes, the average number of messages per task increases dramatically (and exponentially). We believe that having sufficiently long tasks to amortize the cost of these many number of messages would be critical to achieving good efficiency numbers at exascales.

Also, we give the summary plots showing the information regarding utilization and throughput in terms of simulation time in Figure 14. The utilization is calculated as the area_of_green_region / area_of_red_region. We could see that the distributed scheduling has about 100% utilization after about 12K seconds elapsed and big throughput for ave_5000 workload at exascales (~200K tasks/sec in the

steady state part of the experiment), while the centralized one (not shown below) has just about 0.5% utilization, and a much lower throughput (~1K tasks/sec) given the same configuration. The overall efficiency of the distributed scheduling is about 82% after taking into account the ramp up and down period of the experiment.

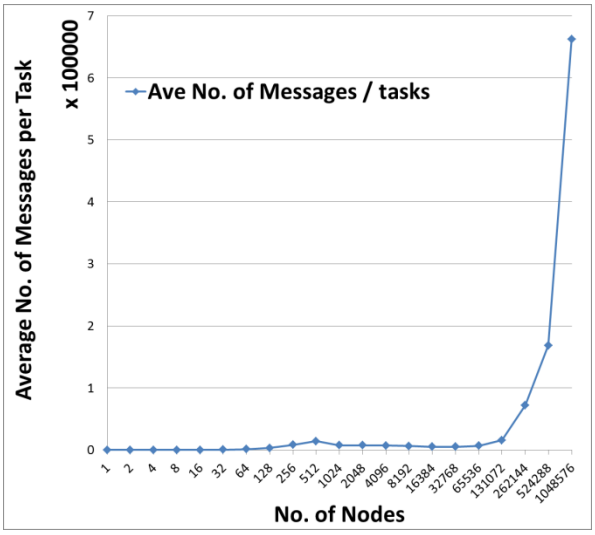

Figure 13: Average number of messages with respect to the system scale

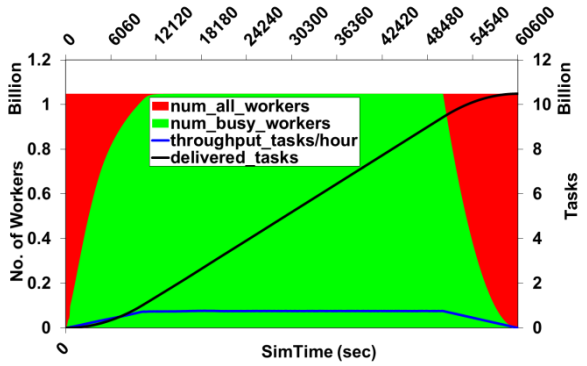

Figure 14: Summary plot for distributed scheduling at Exascales for ave_5000 workload

5. RELATED WORK

Since 1990s, a lot of work related to **job scheduling systems** has been done. The University of Wisconsin developed one of the earliest job management systems, Condor [23] to harness the unused CPU cycles on workstations for long-running batch jobs. Portable Batch System (PBS) [24] 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 [25] 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 [20] was developed by the University of Chicago and Argonne National Laboratory for MTC applications (this work was done by the authors of this work). Falkon also had 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. A naïve distributed Falkon implementation was shown to scale to a petascale system in [8], the approach taken by Falkon suffered from poor load balancing under failures or unpredictable task execution times.

For **simulators of job scheduling** systems, SimJava [26] and GridSim [27] are two of them. SimJava is developed by the University of Edinburgh, based on which, the University of Melbourne, Australia developed GridSim. They model nodes and networks in Grid heterogeneous environment using Java threads, which likely could just scale up to thousands of nodes, because it consumes too much resource to manage many threads, and our experiment shows that the maximum number of threads a process could create is about 32K.

Most parallel programming systems require **load balancing**. Centralized load balancing has been extensive studied in the past (JSQ [28], least-work-left [29], SITA [30]), but they all suffer from poor scalability and resilience. Although distributed load balancing at extreme scales of millions of nodes and billions of threads of execution is likely a more scalable and resilient solution, there are many challenges that must be addressed (e.g. utilization, partitioning). Fully distributed strategies have been proposed, including neighborhood averaging scheme (ACWN) [31][32][33][34]. In [34], 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 [35] and [36]. Charm++ [14] supports centralized, hierarchical and distributed load balancing. It has demonstrated that centralized strategies work still ok for 3000 processors for NAMD. In [9], the authors present an automatic dynamic hierarchical load balancing method for Charm++, which scales up to 16384 cores of Ranger (at TACC) for a synthetic benchmark.

Work stealing [37][38][22] has been used at small scales successfully in parallel languages such as Cilk [39], to load balance threads on shared memory parallel machines. Theoretical work has proved that a work-stealing scheduler can achieve execution space, time, and communication bounds all within a constant factor of optimal [37][38]. 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. [22]The largest studies to date of work stealing have been at thousands of cores scales, showing good to excellent efficiency depending on the workloads. [22] Our work from the simulation perspective shows that work stealing with optimal parameters works great for even exascale systems at millions of nodes and billions of cores.

6. CONCLUSIONS AND FUTURE WORK

Exascale systems bring great opportunities in unraveling of significant scientific mysteries. Also, there are challenges, such as concurrency, resilience, I/O and memory, heterogeneity, and energy, which require revolutions in programming languages and models, memory hierarchy technologies, job management systems, communication networks, and power efficiency. Many-Task Computing for exascale applications needs dynamic job scheduling system at the granularity of node/core levels. Distributed scheduling is likely the efficient way to achieve load balancing, leading to high job throughput and system utilization.

We developed a new light-weight and scalable discrete event simulator that enables distributed scheduling for MTC workloads at exascales. This work provides evidence that work stealing is a scalable method to achieve load balance, even at exascales. It also identified optimal parameters affecting the performance of work stealing; at the largest scales, in order to achieve the best work stealing performance, we found the number of tasks to steal is half and there must be a squared root number of dynamic neighbors (e.g. at 1M nodes, we would need 1K neighbors).

We see this work continue in many different directions. One such direction is to use the same base simulator to explore work stealing for many-core chips with thousands of cores. Instead of simulating an exascale system with millions of nodes and billions of cores, we plan to use the same simulator to simulate a single chip with 1000 cores. The challenge will be to model the network accurately with 2D meshes (or other expected network topologies). We have started exploring scheduling of direct acyclic graphs (DAGs) on many-core processors in [40] which takes a computer architecture perspective, and uses another simulator NIGRAM [40]. We believe that work stealing could be applied to improve the work presented in [40].

Another direction for future improvements of SimMatrix is to allow more complex network topologies for an exascale system, such as trees, 3D torus networks, daisy chained switches, etc. Having a more complex network model would allow us to explore another dimension, specifically on the placement of the neighbors, and how it might affect overall system efficiency. It would also allow us to study job dependency management with more realistic constraints.

The insight we are getting from SimMatrix will ultimately be used to develop MATRIX, a distributed task execution fabric. MATRIX will likely employ work stealing for distributed load balancing, among other techniques to enable extreme scalability and performance. We expect MATRIX to be integrated with other projects, such as Swift [13] (a data-flow parallel programming systems) and FusionFS [41] (a distributed file systems). We will also investigate the possibility of MATRIX benefiting other systems such as Charm++ [14]. MATRIX's goal would be to allow MTC applications to scale to extreme scales distributed systems, including exascale systems. A potential future software stack is shown in Figure 15.

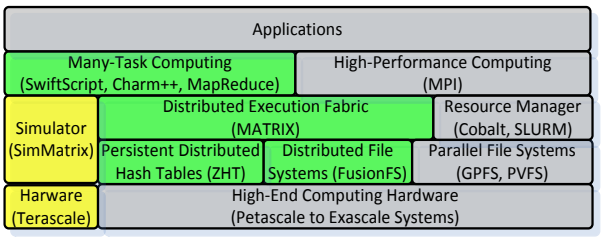

Figure 15: Proposed software stack

The gray areas represent the traditional HPC-stack, comprising of applications, MPI, resource managers, parallel file systems, and HEC hardware. The green areas are additional components, such as support for many-task computing applications, using lower level components such as MATRIX, ZHT[42], and FusionFS[41]. The yellow areas represent the simulation components (SimMatrix), aimed to help explore peta/exascales levels on modest terascale systems.

We envision the SimMatrix and MATRIX projects to be building blocks for future parallel programming systems, having the potential to address some of the hardest problems in exascale computing, namely concurrency and resilience. Once SimMatrix is extended with all of the features mentioned earlier this section, I/O and memory (e.g. data-aware scheduling), as well as heterogeneity could be addressed.

7. REFERENCES

[1] B. Obama. "A Strategy for American Innovation: Driving Towards Sustainable Growth and Quality Jobs", National Economic Council, http://www.whitehouse.gov/administration/eop/nec/StrategyforAmericanInnovation/, 2009

[2] 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

[3] P.M. Kogge, T.J. Dysart. "Using the TOP500 to Trace and Project Technology and Architecture Trends", IEEE/ACM Supercomputing 2011

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

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

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

[7] 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

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

[9] G. Zhang, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW 10, pages 436-444, Washington, DC, USA, 2010. IEEE Computer Society

[10] http://www.cs.cmu.edu/~acw/15740/proposal.html

[11] Keld Helsgaun. "Discrete Event Simulation in Java". Department of Computer Science Roskilde University, Denmark

[12] http://datasys.cs.iit.edu/projects/SimMatrix/index.html

[13] 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

[14] http://charm.cs.uiuc.edu/research/charm

[15] http://download.oracle.com/javase/6/docs/api/java/util/TreeSet.html

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction To Algorithms, Third Edition, The MIT Press, 2009

[17] http://download.oracle.com/javase/1.4.2/docs/api/java/util/HashMap.html

[18] http://tutorials.jenkov.com/java-io/index.html

[19] http://en.wikipedia.org/wiki/Coefficient_of_variation

[20] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde. “Falkon: A Fast and Light-weight tasK executiON Framework,” IEEE/ACM SC 2007

[21] Ioan Raicu, Ian Foster, Mike Wilde, Zhao Zhang, Yong Zhao, Alex Szalay, Pete Beckman, Kamil Iskra, Philip Little, Christopher Moretti, Amitabh Chaudhary, Douglas Thain. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010

[22] J. Dinan, D.B. Larkins, P. Sadayappan, S. Krishnamoorthy, J. Nieplocha. “Scalable work stealing”, In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), 2009

[23] Condor: http://www.cs.wisc.edu/condor/, 2012

[24] PBS: http://pbs.mrj.com, 2012

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

[26] SimJava: http://dcs.ed.ac.uk/home/hase/simjava/, 2012

[27] GridSim: http://www.buyya.com/gridsim/, 2012

[28] H.C. Lin, C.S. Raghavendra. An approximate analysis of the join the shortest queue (JSQ) policy, IEEE Transaction on Parallel and Distributed Systems, Volume 7, Number 3, pages 301-307, 1996

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

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

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

[32] 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

[33] 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

[34] 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

[35] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1990

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

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

[38] V. Kumar, A. Y. Grama, and N. R. Vempaty. “Scalable load balancing techniques for parallel computers”, J. Parallel Distrib. Comput., 22(1):60–79, 1994

[39] M. Frigo, C. E. Leiserson, and K. H. Randall. “The implementation of the Cilk-5 multithreaded language”, In Proc. Conf. on Prog. Language Design and Implementation (PLDI), pages 212–223. ACM SIGPLAN, 1998

[40] Ke Yue, Ioan Raicu. “Scheduling Direct Acyclic Graphs on Massively Parallel 1K-core Processors”, under review at ACM HPDC 2012.

[41] FusionFS: Fusion Distributed File System, http://datasys.cs.iit.edu/projects/FusionFS/, 2011

[42] T. Li, R. Verma, X. Duan, H. Jin, I. Raicu. “Exploring Distributed Hash Tables in High-End Computing”, ACM Performance Evaluation Review (PER), 2011