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

Ke Wang[†], Kevin Brandstatter[†], Ioan Raicu[†‡]
[†]Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA
[‡]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL, USA
kwang22@hawk.iit.edu, kbrandst@hawk.iit.edu, iraicu@cs.iit.edu

Keywords: Exascale, many-task computing, discrete event simulation, scheduling

Abstract

Exascale computers (expected to be composed of millions of nodes and billions of threads of execution) will enable the unraveling of significant scientific mysteries. Many-task computing is a distributed paradigm, which can potentially address three of the four major challenges of exascale computing, namely Memory/Storage, Concurrency/Locality, and Resiliency. Exascale computing will require efficient job scheduling/management systems that are several orders of magnitude beyond the state-of-the-art, which tend to have centralized architecture and are relatively heavy-weight. This paper proposes a light-weight discrete event simulator, SimMatrix, which simulates job scheduling system comprising of millions of nodes and billions of cores/tasks. SimMatrix supports both centralized (e.g. first-in-first-out) and distributed (e.g. work stealing) scheduling. We validated SimMatrix against two real systems, Falkon and MATRIX, with up to 4K-cores, running on an IBM Blue Gene/P system, and compared SimMatrix with SimGrid and GridSim in terms of resource consumption at scale. Results show that SimMatrix consumes up to two-orders of magnitude lower memory per task, and at least one-order of magnitude (and up to four-orders of magnitude) lower time per task overheads. For example, running a workload of 10 billion tasks on 1 million nodes and 1 billion cores required 142GB memory and 163 CPU-hours. These relatively low costs at exascale levels of concurrency will lead to innovative studies in scheduling algorithms at unprecedented scales.

1. INTRODUCTION

There are many domains (e.g. weather modeling, national security, energy) that will achieve revolutionary advancements due to exascale computing. Predictions are that by the end of the decade, supercomputers will reach exascale with millions of nodes and billions of threads of execution [1]. The era of exascale computing will bring fundamental challenges in how we build computing systems and hardware, how we manage and program them. The techniques designed decades ago will have to be dramatically changed to support the coming extreme-scale general purpose parallel computing. The four most significant challenges of exascale computing are: Energy and Power; Memory and Storage; Concurrency and Locality [33]; Resiliency [1]. One attempt to address these challenges is to take a radically different approach to the traditional HPC/MPI [2] programming paradigm, which is usually the source of these challenges. For example, many HPC applications use MPI for synchronous communication, making it hard to be resilient in face of a decreasing MTTF [3]. Checkpointing (the state-of-the-art mechanism to make HPC systems reliable) is increasingly less effective with larger systems for HPC [42]. One alternate programming model to HPC is Many-Task Computing (MTC) [4]**[**38**]**.

MTC [4] was introduced to describe a class of applications that did not fit easily into the categories of traditional HPC or HTC [41]; many such applications can be found in astronomy [35], medicine [6], biology [6], and many others. Many MTC applications are structured as graphs of discrete tasks, with explicit input and output dependencies forming the graph edges. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive [39]. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. For many applications, a graph of distinct tasks is a natural way to conceptualize the computation and build the application [4]. MTC can address three of the four major challenges (except for Energy and Power) of exascale computing. It offers better resiliency than HPC due to the asynchronous nature, which makes task level checkpointing easy, and failures only affect the tasks running on the failed nodes. Concurrency can be addressed more transparently, based on the data-flow model, as opposed to coded explicit parallelism by expert programmers of HPC. In order to address the challenges, MTC needs scalable storage systems to achieve asynchronous inter-process communication, and job management systems to handle billions of jobs/tasks [3].

With exascale computing, we expect that job management systems (JMS) will have to be much more scalable and flexible to handle both HPC and MTC applications in order to achieve the highest job throughput, system utilization and load balancing. Scalability of JMS refers to the increasing of processing capacity (measured by throughput) as the workload (number of tasks) and computing resources scale. Research about real JMSs is impossible at exascale, because not only we lack the exascale computers, but the experimental results obtained from the real-world platforms are often irreproducible due to resource dynamics [7]. Therefore, we fall back to simulations to study various JMS architectures and algorithms. Simulations have been used extensively as an efficient method to achieve reliable results in several areas of computer science for decades, such as microprocessor design, network protocol design, and scheduling. Discrete event simulation (DES) [8] utilizes a mathematical model of a physical system to portray state changes at precise simulated time. In DES, the operations of a system are represented as a chronological sequence of events. A variation of DES is parallel DES (PDES) [9], which takes advantage of the many-core architecture to access larger amount of memory and processor capacities, and to be able to handle even more complex systems in less end-to-end time. However, PDES adds significant complexity to the simulations, adds consistency challenges, requires more expensive hardware, and often does not have linear scalability as resources are increased.

This paper proposes a light-weight and scalable discrete event simulator, SimMatrix, which simulates job scheduling system comprising of millions of nodes and billions of cores/tasks (tasks and jobs are used interchangeably throughout the paper). Careful consideration was given to the SimMatrix architecture, to ensure that it would scale to exascale levels on modest resources in a single node shared memory system. We compare SimMatrix with two existing simulators, SimGrid [10] and GridSim [11] in terms of resource (time and memory) consumption with scales. We design, architect and implement SimMatrix. It supports both centralized (e.g. first-in-first-out or FIFO) and distributed (e.g. work stealing) scheduling. **The main contributions of this paper are:**

1. *Design and implementation of the SimMatrix simulator*
2. *Performance evaluation between SimMatrix, SimGrid and GridSim; evaluation done up to millions of nodes, billions of cores, and tens of billions of tasks*
3. *Supports of homogenous/heterogeneous systems, various programming models (HPC/MTC/HTC), and scheduling strategies (centralized/distributed/hierarchical)*

The rest of the paper is organized as follows. Section 2 gives the related work. Section 3 presents the SimMatrix architectures, the design and implementation details. Section 4 shows the evaluation and experimental results of SimMatrix, and the comparison with SimGrid and GridSim. Section 5 covers the conclusions and future work.

2. RELATED WORK

A lot of real JMSs have been developed. Condor [12] was developed as one of the earliest JMSs, to harness the unused CPU cycles on workstations for long-running batch jobs. Portable Batch System (PBS) [13] 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 [14] is the load-sharing and batch-queuing component of a set of workload-management tools. All of these systems are designed for either HPC or HTC workloads, and generally have high scheduling overheads. Other JMSs, such as Cobalt [15], typically used on supercomputers (e.g. IBM Blue Gene systems [16]), lack the granularity of scheduling jobs at node/core level. Falkon [17], a light-weight task execution framework, was developed specifically for MTC applications. Falkon also has a centralized architecture, and although it scaled and performed orders of magnitude better than the state-of-the-art JMS, it did not even scale to petascale systems. A na ïve hierarchical implementation of Falkon was shown to scale to a petascale system in [5], however, the approach taken by Falkon suffered from poor load balancing under failures, high variance and unpredictability of task execution times.

Simulators for distributed systems have been developed over past decades, such as SimGrid [10], GridSim [11], SimJava [18]. SimGrid is a joint project staring from 1999, which now uses PDES and claims to have 2M nodes' scalability. However, it has consistency challenge and is unpredictable. It is neither suitable to run exascale MTC applications, due to the complex parallelism. GridSim is developed based on SimJava, which use multi-threading with one thread per simulated element (cluster), making them impossible to reach extreme scales of millions nodes or billions of cores on a single shared-memory system.

3. SIMMATRIX SIMULATOR

This section describes the SimMatrix architectures (Figure 1), and the design and implementation details. The software is released as open source software [19]. For simplicity, we assign consecutive integer numbers as the node ids, ranging from 0 to the number of nodes N-1.

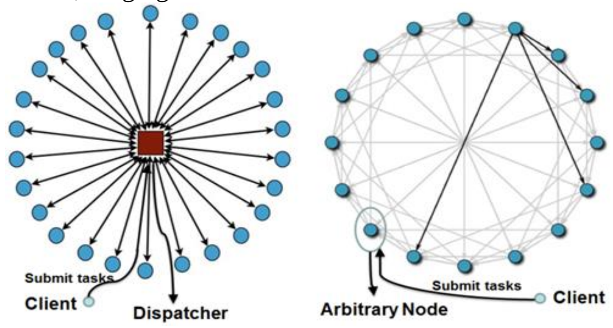

Figure 1: SimMatrix architectures for both centralized (left) and distributed (right) scheduling

SimMatrix supports the granularity of scheduling at the node/core level at extreme scales. The system could be centralized (Figure 1 left part), where a single dispatcher maintains a task queue and manages the task submission, task assignment, and task execution state updates. It could also be distributed (Figure 1 right part), where each computing node maintains a task execution framework, and they cooperate with each other to achieve load balancing. The centralized approach suffers scalability, due to its limited processing capacity at a single node (typically called the head node). We believe that distributed scheduling with innovative load balancing techniques (e.g. work stealing) is the approach to exascale. Another one is the hierarchical architecture, where several dispatchers are organized in a tree-based topology. SimMatrix could be easily extended to support hierarchical scheduling.

3.1. Centralized Scheduler

In centralized scheduler, a dispatcher maintains a task queue and manages the task submission, task assignment, and task execution state updates. All tasks are submitted to the dispatcher, which then assigns tasks to the first available node using FIFO policy [20]. None of the compute nodes have task waiting queues. If all cores are occupied, the dispatcher will wait until some tasks are finished. Then it sends tasks again to the nodes that have idle cores. This procedure continues until all the tasks have been finished.

3.1.1. Task Description

Each task can be described with various attributes, such as task length (the time taken to complete the task), task cores (the number of cores required to execute the task), task size (data size required by the task), and task timestamps (submission time, start time, end time). We expect that some other higher level system is managing all the task dependencies, such as some parallel programming system (e.g. Swift [5][21]**[**37**]**, Charm++ [22]). Future work will also investigate the support of task dependency to evaluate the feasibility of distributed workflow engine approaches, as it applies to grids [34][36], clouds [40], and supercomputers [5].

3.1.2. Global Event Queue

Before settling on SimMatrix being a DES, we explored how many threads could be supported under Java, and found on our 48-core system with 256GB of memory, 32K threads is the upper bound. Since it was not feasible for us to run 1M threads in Java (or C/C++ which we also explored), we decided on creating an object per simulated node. Any behavior is converted to an event, and all events are put in a global event queue, and sorted based on the occurrence time. We advance the simulation time to the occurrence time of the first event removed from the queue. The events are:

a) TaskEnd: Signals a task completion event (frees a processing core). The scheduler advances to the next task to schedule. The compute node (with the available core) will wait for the dispatcher to assign more tasks.

b) Submission: Client submits tasks to the dispatcher, triggered when the waiting queue length in the dispatcher is below the threshold.

c) 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, instant throughput, etc.

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 frequent updates, which re-order the queue. We use the TreeSet [23] data structure in Java. It is a set of elements ordered using their natural ordering, or by a comparator provided at set creation time. In SimMatrix, it is ordered by a comparator based on the event occurrence time, along with the event Id (if events have the same occurrence time). The TreeSet is implemented based on Red-Black tree [24], which guarantees $\Theta(\log n)$ time for removing and inserting, and $\Theta(1)$ time for getting the first event.

3.1.3. Node Load Information

The load of a node 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 waiting tasks. If we were to naively go through all the nodes to get the load information, the simulator would be inefficient when the number of nodes is large (e.g. 1 million).

We implement the load information using a Hash Map [25]. The "Keys" are the node loads (from 0 to number of cores), while the "Value" is a hash set, which contains the node ids whose loads are all equal to the "Key". This means that nodes in the simulator are grouped together in containers that have similar load. Each time when the dispatcher wants to assign some tasks to a node, it goes through all the node load containers sequentially, finds the first set of nodes, which have idle cores (load is less than the number of cores), and then assigns tasks to all the nodes in a FIFO pattern. As the number of cores per node is relatively small (e.g. 1000 cores), we consider this lookup operation taking $\Theta(c)$ time, where c is the number of cores of a node, and $c<=1000$. Once the right load level is identified, inserting, getting or removing an element in the nested hash set only takes $\Theta(1)$ time. This nested data-structure helped reduce the time complexity by orders of magnitude, from a $\Theta(n)$ (n is the number of nodes) to $\Theta(c*1)$ for one dispatching, and allowed the simulator to run orders of magnitude faster at exascale.

3.1.4. Dynamic Task Submission

Although SimMatrix supports the submission of a static set of tasks (predefined in some workload file, or by some workload generator), SimMatrix also supports dynamic task submission which allows task submission throttling to limit the memory foot print of the simulator to only the active tasks. Essentially, the simulator can limit the number of submitted tasks based on the number of waiting tasks and some predefined threshold.

3.2. Distributed Scheduler

One of the major motivations to architect and implement SimMatrix was to study different distributed scheduling algorithms and techniques at extremely large scales, assuming that centralized schedulers would not scale to exascale levels. This section describes the distributed scheduler, which uses a distributed load balancing approach called Work Stealing [26] (in which, processors needing work steal computational tasks from other processors). Work stealing is a distributed load balancing technique that has been used successfully in parallel languages such as Cilk [27], to load balance threads on shared memory parallel machines. With work stealing, each node has task waiting queue and could steal/dispatch tasks from/to its neighbors. The work stealing algorithm and how to choose the optimized work stealing parameters are out of the scope of this paper and the subject of future work. This paper focuses on the design and implementation of the simulator SimMatrix. The distributed scheduler share common features with the centralized one, such as task description and dynamic task submission.

Tasks are submitted to any arbitrary node. For simplicity, we let the clients submit tasks to the first node (id = 0). This is the worst scenario from a load balancing perspective. The best case would be if the original clients submitted tasks randomly over all compute nodes in a load balancing fashion (e.g. uniform random, modular). Every node has a global knowledge of all other nodes in the system (membership list), a dedicated task waiting queue, and several neighbors to communicate with. Figure 1 (right part) shows a fully connected homogeneous topology. All nodes have the same amount of neighbors and cores; in this example, the neighbors of a node are just its several left and right nodes with consecutive ids, we call this schema as the static neighbor selection. Also, our simulator allows dynamic random neighbor selection, which means every time when doing work stealing, a node selects several neighbors randomly from the membership list.

When a node has no waiting tasks, it will ask the load information (the number of waiting tasks minus the number of idle cores) of all the neighbors in turn, and try to steal tasks from the heaviest loaded one. When a node receives a load information request, it will send its load information to the calling neighbor. If a node receives work stealing request, it then checks its task waiting queue, if which is not empty, the node will send some tasks to the neighbor, or it will send information to signal a steal failure. When a node fails to steal tasks, it will wait some time, referred to as the poll interval, and then try again. The termination condition is that all the tasks submitted by client are finished. We do this by setting a global counter, which can be read by all simulated nodes to signal the termination of the simulation.

3.2.1. Global Event Queue

Our distributed scheduler also has a global event queue, which has the same implementation as that of the centralized one. This global event queue allows the simulator to be implemented in a relatively straightforward manner, easing the implementation, tuning, and debugging. The trade-off is perhaps the limited concurrency. However, as we will show in section 4, even with this design architecture, we have been able to significantly outperform several other simulators. The events are:

a) TaskEnd: Signals a task completion event. The compute node starts to execute another task (if its task waiting queue isn't empty) by inserting another 'TaskEnd' event, or steal tasks from its neighbors. Or, if it is the first node and its waiting queue length is below the threshold, a 'TaskReception' event will be triggered on the client's side.

b) Log: The same as the centralized scheduler.

c) Steal: Signals the work stealing algorithm to invoke the steal operation. First, the node asks for the load information of its neighbors in turn, and then selects the most loaded one to steal tasks by inserting a 'TaskReception' event. If all neighbors have no tasks, the node will wait for some time to 'Steal' again.

d) TaskDispatch: Dispatch tasks to a neighbor. If at the current time, the node happens to have no tasks, it will inform the neighbor to steal tasks again, by inserting a 'Steal' event from the neighbor. Else, the node dispatches a part (e.g. half) of its waiting tasks to the neighbor by inserting a 'TaskReception' event from that neighbor.

e) TaskReception: Signals the receiving node to increase the length of task waiting queue. The task received could be from the client, or from a neighbor.

f) Visualization: It is used as an event to visualize the load information of all nodes.

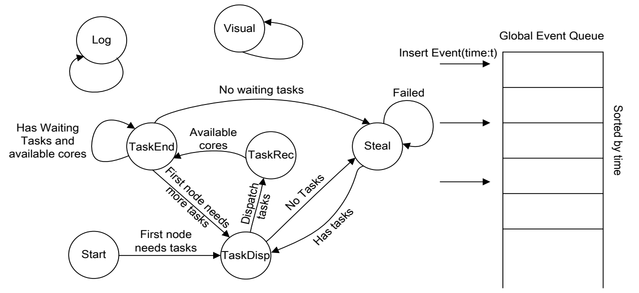

Figure 2: Event State Transition Digram

The state transition diagram of all the events are shown in Figure 2, where each state is an event that is executed, the next state is the event to be inserted in the event queue signaled after finishing the current event. For example, if the current event is "TaskEnd", meaning that a node finishes a task and has one more available core. If the node has waiting tasks, it will insert another "TaskEnd" event for the available core; otherwise, it will steal tasks from neighbors. Or if current node is the first node, and needs more tasks, it will ask the clients to submit more tasks.

3.2.2. Work Stealing Poll Interval

We implement a dynamic poll interval policy in order to achieve reasonable simulation performance while still keeping the work stealing algorithm responsive. Without this policy, we observed that under idle conditions, many nodes would poll neighbors to do work stealing, which would ultimately fail leading to more work stealing requests. If the polling interval was set large enough to limit the number of work steal events, work stealing would not respond quickly to change conditions. Therefore, we change the poll interval of an idle node dynamically by doubling it when all of the neighbors have no tasks, and setting it back to the default small value whenever it steals some tasks successfully. This algorithm is similar to the exponential backoff approach in the TCP networking protocol [28]. We set the default poll interval to be small value (e.g. 1 sec).

4. EVALUATION

This section presents the validation of SimMatrix against Falkon [17] (a centralized light-weight job management system), and MATRIX [19] (a distributed scheduler built on top of a distributed hash table ZHT [43]), the experimental results showing the resource requirement of SimMatrix with scales, plus the comparisons between SimMatrix, SimGrid and GridSim. All experiments are performed on fusion.cs.iit.edu, which boasts 48 AMD Opteron cores at 800MHz, 256GB RAM, and a 64-bit Linux kernel 2.6.31.5. SimMatrix is developed in JAVA and has no other dependencies. The metrics we use to evaluate the performance of SimMatrix are throughput (number of tasks finished per second) and 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). We have two workloads used in this paper:

a) AVE_5K: The average task length is 5000 seconds (0 - 10000), with uniform distribution

b) ALL_1: All tasks have 1-second length

4.1. Validation

We validated SimMatrix against the state-of-the-art MTC systems (e.g. Falkon [17] and MATRIX [19]). The validation results are shown in Figure 3 (with Falkon), and Figure 4 (with MATRIX).

We set the number of cores per node to be 4, and the network bandwidth and latency the same as the case of Blue Gene/P machine. The number of tasks is 10 task/core and 100 task/core for Falkon and MATRIX respectively. We measured SimMatrix (dotted lines) has an average 2.8% higher efficiency than Falkon (solid lines) for several sleep tasks (sleep 1, 2 and 4) in Figure 3. Figure 4 shows the validation results comparing SimMatrix and MATRIX for raw throughput on a "sleep 0" workload. The simulation matched the real performance data with average 5.85% normalized difference (abs(SimMatrix - MATRIX) / SimMatrix), a relatively small amount of error.

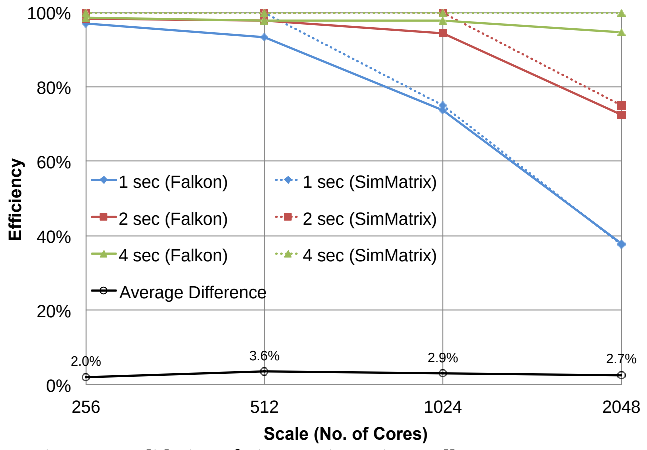

Figure 3: Validation of SimMatrix against Falkon up to 2K cores

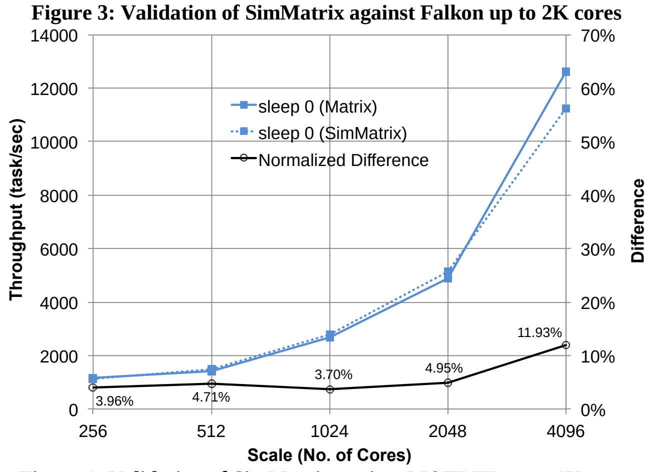

Figure 4: Validation of SimMatrix against MATRIX up to 4K cores

The reasons for these differences are twofold. Falkon and MATRIX are real complex systems deployed on a real supercomputer. Our simulator makes simplifying assumptions, such as the network; for example, we increase the communication overhead linearly with the system scale. It is also difficult to model communication congestion, resource sharing and the effects on performance, and the variability that comes with real systems. We believe the relatively small differences (2.8% and 5.85%) demonstrate that SimMatrix is accurate enough to produce convincible results (at least at modest scales).

4.2. Resource Requirement of SimMatrix

In this section, we show the resource requirement (time and memory consumption) of SimMatrix with scales for both centralized and distributed simulators in Figure 5. The AVE_5K workload is used. We set the number of cores per node to be 1000, and the network bandwidth and latency the same as the case of Blue Gene/P machine. The number of tasks is 10 tasks/core. From this point, all experiments have the same configuration.

Figure 5 shows that both the time and memory consumptions increase slowly than the system scale (less than double when the system scale doubles), which means that our simulations are resource efficient. At exascale with 1M nodes, 1 billion cores and 10 billion tasks, the centralized simulator consumes just 14.1GB memory, 17.4 hours, and the distributed simulator needs about 142.1GB memory, 162.8 hours (still moderate considering the extreme scale). These relatively low costs at exascale levels of concurrency will lead to innovative studies in scheduling algorithms at unprecedented scales.

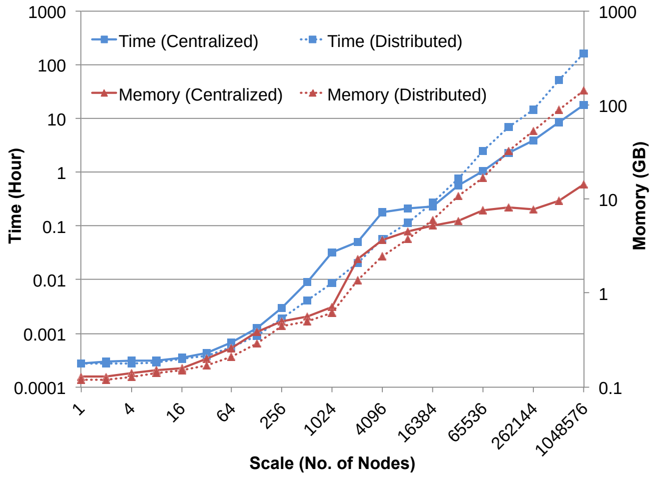

Figure 5: Time and memory of SimMatrix up to 1M nodes

4.3. Centralized vs. Distributed Scheduling

We compare the centralized and distributed schedulers, in terms of system efficiency and throughput. We do two groups of experiments. The first uses the AVE_5K workload, and the second uses ALL_1, for both schedulers. The results are shown in Figure 6 and Figure 7.

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

For ALL_1, the centralized scheduling saturates at about 8 nodes with upper bound throughput of about 1000 tasks/sec, while the distributed one slows down the increasing speed after 128K nodes with throughput of about 60M tasks/sec; it finally reaches 1M nodes with a throughput of 75M tasks/sec. The reason that the distributed scheduler begins to saturate at 128K nodes is because at the final stage when there is not much tasks, work stealing requires too many messages (because almost all nodes are out of tasks leading to more work staling events) as the system scales up, to the point where the number of messages is saturating either the network and/or processing capacity. After 128K nodes, the number of messages per task increases exponentially. One way to address this message chocking at large scales is to set an upper bound of the poll interval. When a node reaches the upper bound, it would not do work stealing anymore. In addition, we believe that having sufficiently long tasks to amortize the cost of this many messages would be critical to achieve good efficiency at exascale. With an upper bound of 75M tasks/sec, the distributed scheduler could handle workloads that have an average length of at least 14 seconds with 90%+ efficiency. It is worth noting that the largest trace of MTC workloads [29][30] has shown MTC tasks to be on average 64 sec average length.

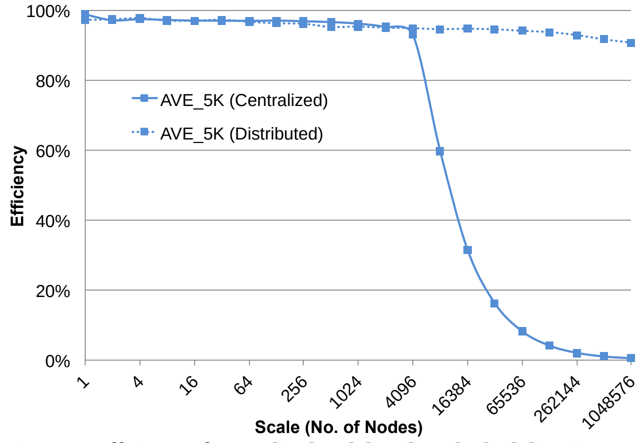

Figure 6: Efficiency of centralized and distributed scheduling (AV_5K)

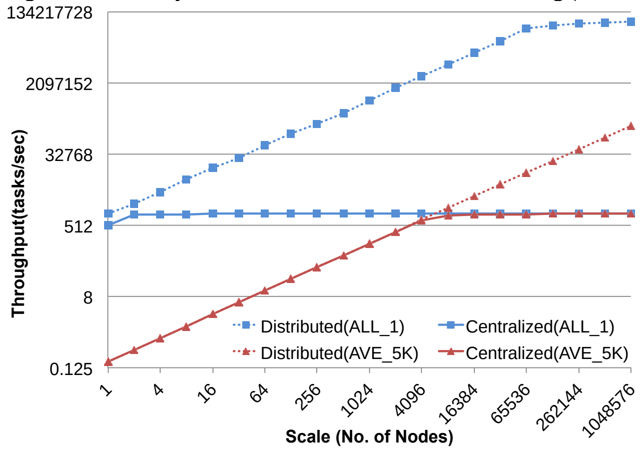

Figure 7: Throughput of centralized and distributed scheduling

4.4. SimMatrix vs. SimGrid and GridSim

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

SimGrid provides functionalities for the simulation of distributed applications in heterogeneous distributed environments. It is a PDES, being claimed the scalability of 2 million nodes [31]. We examined SimGrid, went for the MSG interface, and used the basic Master/Slaves application. We used the AVE_5K workload, and converted the task length to the value of million instructions (MI), as the computing power is represented as MIPS. Each slave has 1000 cores, with each core 4000MIPS (about 1GFlops as 1 CPU cycle usually has 4 instructions), so the computing power of 1 million nodes is 1GFlops×1M×1K=1EFlop, achieving the exascale computing.

GridSim allows simulation of entities in parallel and distributed computing systems, such as users, resources, and resource brokers (schedulers). A resource can be a single processor or multi-processor with shared or distributed memory and managed by time or space shared schedulers. It is a multi-threaded simulator, where each entity is a thread. We developed an application on top of GridSim, which consists of one user (has tasks) and one broker (centralized scheduler) and several resources (computing nodes). Each resource is configured having just one node (Machine), which then has 1000 cores (PEs).

As the saturated throughput of SimGrid is about 2000, in order to make fair comparison, we configured SimMatrix having exactly the same throughput upper bound by setting the processing time per task to be 0.0005 sec (which is 0.001 sec before and achieved the 1000 upper bound). The comparison results are shown in Figure 8 and Figure 9.

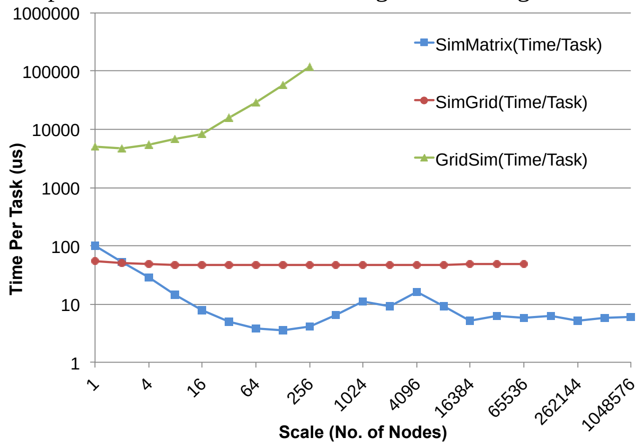

Figure 8: Comparison of time per task

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

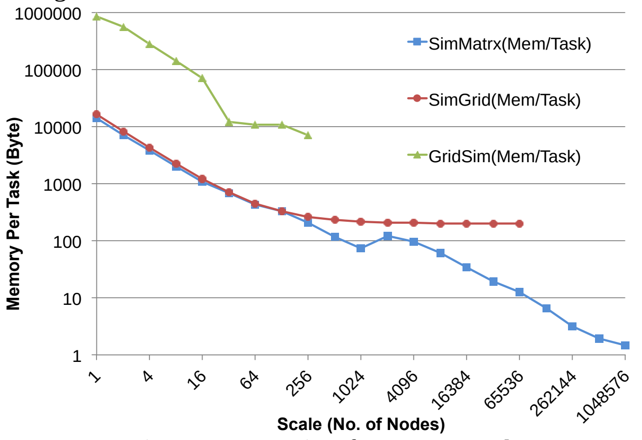

Figure 9: Comparsion of memory per task

4.5. Application Domains of SimMatrix

SimMatrix could be potentially used in several application domains:

Data Centers: large-scale data centers (e.g. Google, Amazon) are composed of thousands of (10 to 100× in near future) servers geographically distributed around the world. Load balancing among all the servers with data-intensive workloads is very important, yet non-trivial. SimMatrix is able to study different network topologies connecting all the servers and data-aware scheduling, which could be applied in scheduling of data centers.

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

Workflow System: although SimMatrix relies on high level workflow systems (Swift, Charm++) to manage the data-flow and task dependency now, we could develop SimMatrix to simulate workflow system with dependent tasks. We have already run SimMatrix with MTC workload achieved from Swift workflow system up to exascale, and achieved ~87% efficiency [32] (Figure 10). We use coefficient variance of the number of tasks finished by each node as a measurement of the load balancing. The closer the value is to 0, the better the load balancing would be.

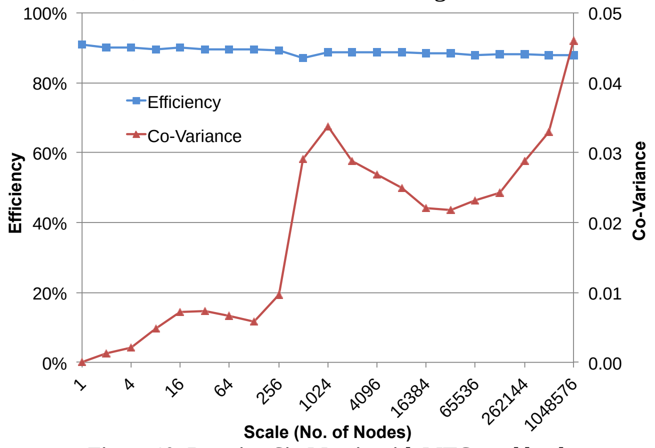

Figure 10: Running SimMatrix with MTC workload

Many-core Simulation: instead of configuring SimMatrix as an exascale system, we also configured it as a single many-core chip node up to thousands of cores with 2D/3D mesh topology. We applied work-stealing at the core level within one many-core node, and found that up to thousand cores level, 2D mesh topology needs at least 13 hops of neighbors, while 3D mesh needs at least 5 (Figure 11), in order to achieve high system efficiency.

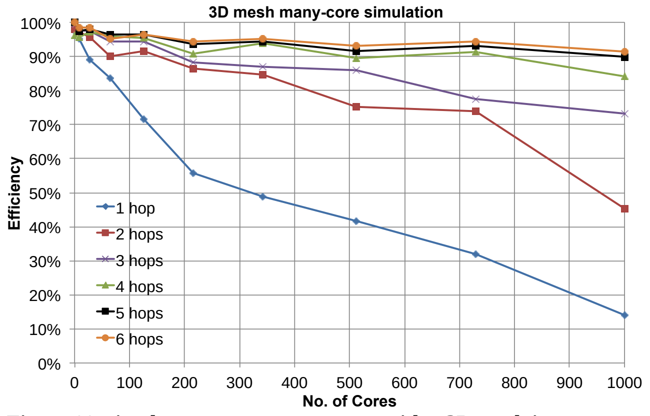

Figure 11: simulate many-core processor with a 3D-mesh interconnect

5. CONCLUSION AND FUTURE WORK

Exascale computing will bring several challenges, which need to be solved by new programming models. We believe that MTC could offer many advantages over HPC. However, efficient JMSs are needed to manage the system resource allocation and job submission, in order to maximize the job throughput and system utilization. We developed a light-weight and scalable DES of JMS, SimMatrix, at exascale. We validated SimMatrix against Falkon and MATRIX, and performed scalability evaluations up to exascale. We also compared SimMatrix with SimGrid and GridSim. The scalability and resource consumption of SimMatrix are significantly better.

In the future, we plan to explore more complex network topologies for exascale systems, such as Fat Tree, 3D/4D/nD Torus, and InfiniBand. We believe SimMatrix could also be developed to simulate workflow systems, and it would allow us to study job dependency and data aware scheduling with more realistic constraints. It is critical to develop scalable simulators to explore challenges at exascale now, so that by the time when exascale computer comes, we thoroughly understand what techniques, algorithms, and programming models would likely work best to ensure the success of exascale computing.

REFERENCES

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

[2] M. Snir, S.W. Otto, et al. "MPI: The Complete Reference", MIT Press, 1995.

[3] I. Raicu, P. Beckman, I. Foster. "Making a Case for Distributed File Systems at Exascale", ACM Workshop on LSAP, 2011.

[4] 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.

[5] I. Raicu, Z. Zhang, et al. "Toward Loosely Coupled Programming on Petascale Systems," IEEE SC 2008.

[6] Y. Zhao, I. Raicu, et al. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments", book chapter in Grid Computing Research Progress, ISBN: 978-1-60456-404-4, Nova Publisher 2008.

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

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

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

[10] M. Quinson, C. Rosa, and C. Thiéry. "Parallel Simulation of Peer-to-Peer Systems." In Proceedings of the 12th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'12), May 2012. IEEE Computer Society Press.

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

[12] T. Tannenbaum, D. Wright, et. al. "Condor - A Distributed Job Scheduler", in Thomas Sterling, editor, Beowulf Cluster Computing with Linux, The MIT Press, 2002. ISBN: 0-262-69274-0.

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

[14] LSF: http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/index.html, 2013.

[15] Cobalt: http://trac.mcs.anl.gov/projects/cobalt, 2013.

[16] The Blue Gene/P Team. "Overview of the ibm blue gene/p project," IBM Journal of Research and Development, vol. 52, no. 1.2, pp. 199–220, jan. 2008.

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

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

[19] MATRIX: MAny-Task computing execution fabRIc at eXascales. http://datasys.cs.iit.edu/projects/MATRIX/index.html, 2013.

[20] J. L. Stone, K. Beck, et al. New Jersey 07632: Prentice-Hall, Inc. div. of Simon & Schuster. pp.150. ISBN 0-13-195884-4.

[21] 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.

[22] L. Kale, A. Arya, et al. “Charm++ for Productivity and Performance: A Submission to the 2011 HPC Class II Challenge.” 2011 November. 11-49. Parallel Programming Laboratory.

[23] TreeSet:http://download.oracle.com/javase/6/docs/api/java/util/TreeSet.html, 2013.

[24] T. H. Cormen, C. E. Leiserson, et. al. Introduction To Algorithms, Third Edition, The MIT Press, 2009.

[25] HashMap:http://download.oracle.com/javase/1.4.2/docs/api/java/util/HashMap.html, 2013.

[26] R. D. Blumofe and C. Leiserson. “Scheduling multithreaded computations by work stealing”, In Proc. 35th Symposium on FOCS, pages 356–368, Nov. 1994.

[27] M. Frigo, C. E. Leiserson, et al. “The implementation of the Cilk-5 multithreaded language”, In Proc. Conf. on PLDI, pages 212–223. ACM SIGPLAN, 1998.

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

[29] I. Raicu, I. Foster, et al. “Middleware Support for Many-Task Computing”, Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010.

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

[31] M. Quinson, C. Rosa, et al. “Parallel Simulation of Peer-to-Peer Systems, ” inria-00602216, version 2-6 Dec. 2011.

[32] K. Wang, I. Raicu. “Paving the Road to Exascale with Many-Task Computing”, Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012.

[33] A. Szalay, J. Bunn, J. Gray, I. Foster, I. Raicu. “The Importance of Data Locality in Distributed Computing Applications”, NSF Workflow Workshop 2006.

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

[35] I. Raicu, I. Foster, A. Szalay, G. Turcu. “AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis”, TeraGrid Conference 2006, June 2006

[36] I. Raicu, C. Dumitrescu, M. Ripeanu, I. Foster. “The Design, Performance, and Use of DiPerF: An automated DIstributed PERformance testing Framework”, International Journal of Grid Computing, Special Issue on Global and Peer-to-Peer Computing, 2006.

[37] 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.

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

[39] I. Raicu, et al. “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.

[40] Y. Zhao, I. Raicu, S. Lu, X. Fei. "Opportunities and Challenges in Running Scientific Workflows on the Cloud", IEEE International Conference on Network-based Distributed Computing and Knowledge Discovery (CyberC) 2011.

[41] K. Wang, Z. Ma, I. Raicu. “Modelling Many-Task Computing Workloads on a Petaflop IBM BlueGene/P Supercomputer”, IEEE CloudFlow 2013.

[42] D. Zhao, D. Zhang, K. Wang, I. Raicu. “RXSim: Exploring Reliability of Exascale Systems through Simulations”, ACM HPC 2013.

[43] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu. “ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, IEEE IPDPS 2013.