Next Generation Job Management Systems for Extreme-Scale Ensemble Computing

Ke Wang
Illinois Institute of Technology
Chicago IL, 60616, USA
kwang22@hawk.iit.edu

Xiaobing Zhou
Illinois Institute of Technology
Chicago IL, 60616, USA
xzhou40@hawk.iit.edu

Hao Chen
Illinois Institute of Technology
Chicago IL, 60616, USA
hchen71@hawk.iit.edu

Michael Lang
Los Alamos National Laboratory
Los Alamos UM, 87544, USA
mlang@lanl.org

Ioan Raicu
Illinois Institute of Technology
Chicago IL, 60616, USA
iraicu@cs.iit.edu

ABSTRACT
With the exponential growth of supercomputers in parallelism, applications are growing more diverse, including traditional large-scale HPC MPI jobs, and ensemble workloads such as finer-grained many-task computing (MTC) applications. Delivering high throughput and low latency for both workloads requires developing a distributed job management system that is magnitudes more scalable than today's centralized ones. In this paper, we present a distributed job launch prototype, SLURM++, which is comprised of multiple controllers with each one managing a partition of SLURM daemons, while ZHT (a distributed key-value store) is used to store the job and resource metadata. We compared SLURM++ with SLURM using micro-benchmarks of different job sizes up to 500 nodes, with excellent results showing 10X higher throughput. We also studied the potential of distributed scheduling through simulations up to millions of nodes.

Categories and Subject Descriptors
D.4.7 [**System Design**]: Organization and Design – *distributed systems.*

General Terms
Performance, Design, Algorithm.

Keywords
Job management systems, job launch, scheduling, key-value store.

1. INTRODUCTION
Exascale machines will have billions of concurrent threads of execution [1]. With this extreme magnitude of parallelism, ensemble computing is one way to efficiently use the machines without requiring full-scale jobs. Ensemble computing would combine the traditional HPC workloads that are large-scale applications using MPI [2] as the communication method, with the ensemble workloads that support the investigation of parameter sweeps using many more but smaller-scale coordinated jobs [3]. Given the significant decrease of Mean-Time-To-Failure [4][5] at exascale, ensemble workloads should be resilient because failures affect a smaller part of the machines.

One example of ensemble workloads comes from the MTC [6][7] paradigm. MTC applications have orders of magnitude larger number of jobs/tasks (e.g. billions) with finer granularity in both size (e.g. per-core) and duration (e.g. sub-second to hours) [8] . The tasks do not require strict coordination of processes at job launch as the HPC workloads do. Furthermore, these applications could be data-intensive in nature [9]. Applications that demonstrate characteristics of MTC cover various domains, such as astronomy, bioinformatics, medical imaging and climate modeling [10], and have been run in clusters, grids, supercomputers, and clouds [11].

The job management systems (JMS) for extreme-scale ensemble computing will need to be available and scalable in order to deliver the extremely high throughput and low latency. However, today's batch schedulers (e.g. SLURM [12], Condor [13], PBS [14], SGE [15]) have centralized architecture that is not well suited for the demands, due to both bounded scalability and single-point-of-failure. A popular JMS, SLURM, reported maximum throughput of 500 jobs/sec [16]; however, we will need much higher job scheduling rates (e.g. millions jobs/sec) for next-generation JMS, considering the significant increase of scheduling size and the much finer job granularity. This paper proposes a distributed architecture that supports JMS at extreme-scales.

We implemented a distributed job launch prototype (SLURM++) with multiple controllers participating in allocating resources and launching jobs – an extension to the open source batch scheduler SLURM [12]. We utilized distributed key-value stores (DKVS), specifically ZHT [17], to keep the job and resource metadata. The general use of DKVS in building distributed system services was proposed, and evaluated through simulation in our previous work [18]. We compared SLURM++ with SLURM using micro-benchmarks of different job sizes up to 500 nodes, with excellent results showing 10X higher throughput. In addition, we developed a simulator of SLURM, SimSLURM++, which enables us to study the performance towards exascale with millions of nodes.

2. DISTRIBUTED ARCHITECTURE
The architecture of the next-generation JMS is shown in Figure 1. There will be multiple controllers with each one managing a partition of compute daemons (cd). The controllers are **fully-connected**. In addition, a distributed data storage system is deployed to manage the entire job and resource metadata in a scalable and reliable way.

HPDC'14, June 23 - 27 2014, Vancouver, BC, Canada

http://dx.doi.org/10.1145/2600212.2600703

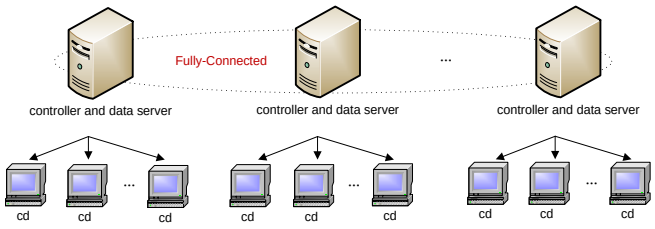

Figure 1: Architecture for distributed JMS; "cd" refers to compute daemon

The **partition size** (number of cd a controller manages) is configurable. For a large-scale HPC workload, the partition size could be thousands; for MTC tasks, the partition size could be one which has the 1:1 mapping (millions of controllers and cds at exascale). We can also have heterogeneous partition sizes.

The distributed storage system could be a DKVS. Each controller would be initialized as a DKVS client, which then uses the simple client APIs (e.g. "lookup", "insert", "remove") to communicate with the servers to query and modify the job and resource information, and the system state information transparently.

We propose the **Resource Stealing** technique to balance free "cd" in all the partitions. We implemented a simple **random** resource stealing algorithm. A controller first checks the local free nodes when launching a job. If there are enough available nodes, the controller directly allocates the nodes; otherwise, it allocates whatever resources the partition has, and randomly queries for other partitions (through a "lookup" operation) to steal resources. If the launching controller experiences several failures in a row due to the selected victims have no free nodes, it will release the resources it has already allocated.

One problem of resource stealing technique is the **Resource Conflict** that happens when different controllers try to modify the same resource. We implement the traditional compare and swap atomic instruction [19] as a normal operation in the ZHT. As ZHT serializes requests at one server, this operation guarantees that only one controller could modify a specific resource at one time.

2.1 SLURM++ PROTOTYPE

We developed a distributed job launch prototype, SLURM++, which serves as a core part for JMS. We adopted the open source SLURM [12], and extended it with multiple controllers participating in allocating resources and launching jobs. We used the ZHT DKVS to keep the job and resource metadata.

SLURM++ is directly extended from SLURM. SLURM has a centralized controller (slurmctld) manage all the cds (slurmd). SLURM keeps all the metadata in a centralized local file system. Upon receiving a job, the slurmctld first looks up the global file system to allocate resource. Once a job gets its allocation, it can be launched via a tree-based network rooted at rank-0 slurmd.

In SLURM++, we developed a light-weight distributed controller that can directly talk with slurmds. We utilize the whole slurmd, preserve the hierarchical job launching part unchanged. In addition, each controller is initialized as a ZHT client, and can call the ZHT client APIs to query and modify the job and resource information. Upon receiving all the slurmds' registration messages within a partition, the controller inserts the available nodes to ZHT server. Then, the controllers randomly steal resources from each other when needed.

We developed SLURM++ in C. We implemented the controller code, which summed to around 5K lines of code; we put the controller and ZHT directly in the SLURM source file, and named the whole prototype SLURM++. The source code is available at the GitHub website: https://github.com/kwangiit/SLURMPP. SLURM++ has dependencies on Google Protocol Buffer [20], ZHT [17], and SLURM [12].

2.2 SimSLURM++ SIMULATOR

In order to study the scalability of the proposed architecture, we developed a simulator of SLURM++, SimSLURM++, which consists of multiple nodes, and each node has different roles to play (controller, ZHT server, compute daemon). There are two parallel queues in each simulated node: a communication queue for sending and receiving messages, and a processing queue for handling requests locally. The two queues operate in parallel, while within one queue, the requests are processed sequentially.

We followed the simulation work we did before [18][21][22] to build SimSLURM++. SimSLURM++ is a discrete event simulator [23] that was built on top of peersim, a scalable peer-to-peer simulator that offers the framework and functionality of simulating distributed systems. SimSLURM++ is developed in Java, and has about 1500 lines of code, along with the peersim 1.5.0 codebase package. The source code is available at the GitHub website: https://github.com/kwangiit/SimDJL. There are no other dependencies.

3. EVALUATION

We evaluate SLURM++ by comparing it with SLURM using micro-benchmarks containing "sleep 0" jobs on the Kodiak cluster from the Parallel Reconfigurable Observational Environment at Los Alamos National Laboratory [24] up to 500 nodes. We used SLURM version 2.6.5, the latest version when we ran experiments. We also run SimSLURM++ up to exascale with millions of nodes using real application traces with different configurations on the machine fusion.cs.iit.edu at IIT [18].

3.1 SLURM++ vs SLURM

The micro-benchmark contains independent "sleep 0" HPC jobs that require different number of compute nodes per job. The partition size is configured as 50; at the largest scale (500 nodes), the number of controllers is 10. We will use SLURM++ (M:N) to specify the ratio of the number of controller to the number of slurmds, where M is the number of slurmds and N is the number of controllers, such as SLURM++ (50:1), SLURM++ (1:1).

We conducted experiments with three workloads: small-job workloads (50 jobs per controller, and job size is 1 node), medium-job workloads (50 jobs per controller, and job size is 1-50 nodes), and big-job workloads (20 jobs per controller, and job size is 25-75 nodes). Figure 2 shows that not only does SLURM++ outperform SLURM in nearly all cases, but the performance slowdown due to increasingly larger jobs at large scale is better for SLURM++ by 2X to 11X depending on the job size. The reason that large jobs perform worse than medium jobs is because the larger the jobs are, the more extensively they compete resources. For small jobs, SLURM++'s performance is bounded by SLURM job launching procedure leading to the smallest improvement. Another fact is that as the scale increases, the throughput speedup is also increasing. This indicates that at larger scales, SLURM++ would outperform SLURM even more.

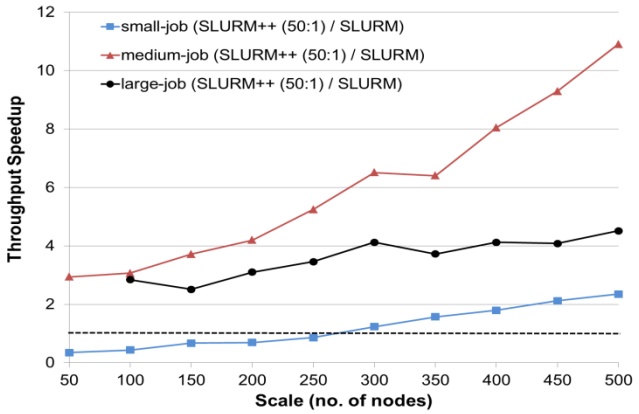

Figure 2: Throughput comparison with different workloads

3.2 Evaluation through SimSLURM++

This section presents the evaluation of the scalability of our proposed work through SimSLURM++ towards millions of nodes.

3.2.1 SimSLURM++ HPC Configuration (1024:1)

We configured SLURM++ with 1024:1 mapping. The workload comes from real applications run on the ANL Blue Gene/P machine, during an 8-month period [25]. There are 68,936 jobs. At each scale, we generated a workload with all jobs that preserve the job size distribution of the original workload by applying the job size percentage of the machine size. In addition, we reduced the job duration by 1M times to reduce the job duration granularity to pose significant challenge on launching jobs.

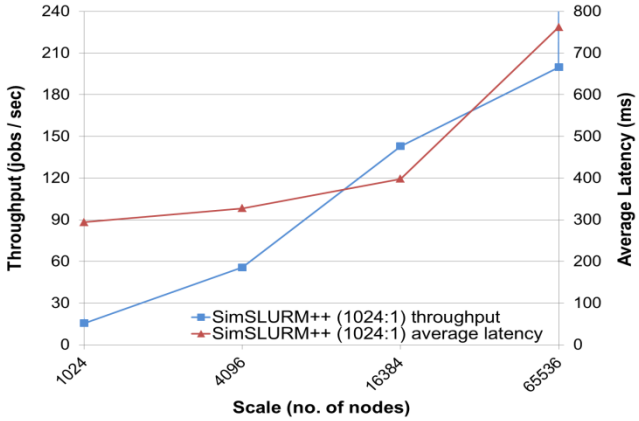

Figure 3: SimSLURM++ (1024:1) throughput and latency

Figure 3 shows the throughput and per-job average latency of SimSLURM++ with HPC configuration. We see that the throughput is increasing with the system scale. This shows that the proposed architecture is scalable. At the meanwhile, the per-job average latency increases moderately from 1024-node to 65536-node.

3.2.2 SimSLURM++ MTC Configuration (1:1)

We also evaluate SimSLURM++ up to millions of nodes with MTC orientation. The workload is micro-benchmark: each controller handles 10 "sleep 0" jobs, and each job requires 1 or 2 nodes. Figure 4 shows the throughput and per-job average latency of SimSLURM++ with MTC configuration. We see that the throughput is increasing perfectly with the system scale. At 1M-node scale, SimSLURM++ achieves throughput as high as 1.75M jobs/sec, which is very promising. At the same time, the per-job average latency increases trivially from 4-node to 1M-node. These results satisfy the requirements of high throughput and low latency of next-generation JMS for exascale ensemble computing.

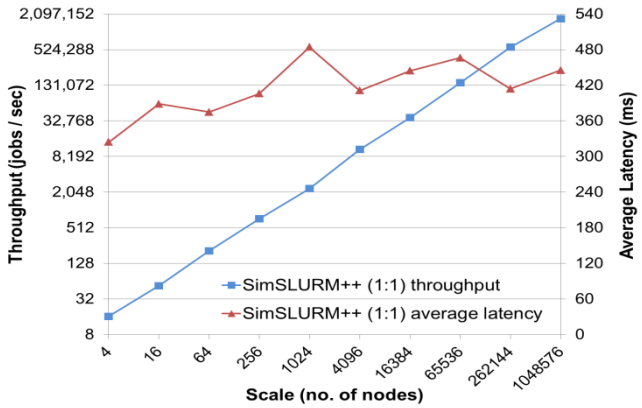

Figure 4: SimSLURM++ (1:1) and latency

4. RELATED WORK

There are other projects that have explored efficient job launch mechanisms. STORM [26] leveraged the hardware collective available in the Quadrics QSNET interconnect to broadcast the binaries to the compute nodes. However, the server is a single-point-of-failure. LIBI/LaunchMON [27] is a scalable bootstrapping service where a tree is used to establish a single process on each compute node. This is a centralized service with no failover or no persistent daemons or state, therefore if a failure occurs they can just re-launch. PMI [28] is the process management layer in MPICH2. It uses a KVS to store job and system information. But the KVS is centralized.

The light-weight task execution frameworks that are developed specifically for ensemble MTC workloads are Falkon [29], a centralized task execution fabric with the support of hierarchical scheduling, and MATRIX [30][31][32], a distributed task execution framework that uses work stealing [33] for load balancing. Though Falkon can deliver tasks at thousands of task/sec for MTC workloads, it is not sufficient for exascale systems and it lacks support for HPC workloads. Another fine grained framework that schedules sub-second tasks for data centers is Sparrow [34]. Though MATRIX and Sparrow have shown great scalability for MTC workloads, neither of them supports HPC workloads.

5. CONCLUSIONS AND FUTURE WORK

Extreme-scale supercomputers require next-generation JMS to be fully distributed that can be much more scalable to deliver jobs with much higher throughput. We have shown that DKVS is a valuable building block to allow scalable job launch. The performance is more preferable (10X) than the centralized production system. Furthermore, our simulation results showed that the distributed architecture resulted in great scalability trends towards extreme-scales supporting both MTC and HPC workloads.

In future work, we will explore several techniques, such as caching and distributed monitoring, and MPI applications in both SLURM++ and SimSLURM++ to improve our work. Additions to this work would also include the investigations of distributed power-aware job launch at the core level. Currently, SLURM++ allocates the whole node to a job. In the future, we will over-decompose a node, and launch jobs at the core level in order to save power. Another extension would be to integrate SLURM++ with the MTC task execution fabric, MATRIX [30] (and/or the

SimMatrix simulator [35]), and study different job scheduling algorithms for both MTC and HPC workloads [36][37].

6. ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy (DOE) contract AC52-06NA25396, and in part by the National Science Foundation (NSF) under award CNS-1042543 (PRObE). We thank Morris Jette and Danny Auble from SchedMD for their help and suggestions about SLURM, and Tonglin Li for his help with ZHT.

7. REFERENCES

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

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

[3] Y. Zhao et al. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," IEEE Workshop on Scientific Workflows 2007.

[4] D. Zhao et al. "Exploring reliability of exascale systems through simulations." ACM HPC 2013.

[5] I. Raicu et al. "Making a Case for Distributed File Systems at Exascale," ACM Workshop on LSAP, 2011.

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

[7] 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

[8] K. Wang et al. "Modeling Many-Task Computing Workloads on a Petaflop IBM Blue Gene/P Supercomputer." IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) 2013.

[9] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C.M. Moretti, A. Chaudhary, D. Thain. "Towards Data Intensive Many-Task Computing", book chapter in "Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management", IGI Global Publishers, 2009

[10] I. Raicu et al. "Middleware Support for Many-Task Computing," Cluster Computing Journal, 2010.

[11] 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

[12] M. A. Jette et al. "SLURM: Simple Linux utility for resource management." JSSPP 2003, pages 44–60, June 24, 2003.

[13] D. Thain et al. "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience 17 (2-4), pp. 323-356, 2005.

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

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

[16] M. Jette and Danny Auble, "SLURM: Resource Management from the Simple to the Sophisticated", Lawrence Livermore National Laboratory, SLURM User Group Meeting, October 2010.

[17] T. Li et al. "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", IEEE IPDPS, 2013.

[18] K. Wang et al. "Using Simulation to Explore Distributed Key-Value Stores for Extreme-Scale Systems Services," IEEE/ACM Supercomputing/SC 2013.

[19] T. L. Harris et al. "A Practical Multi-Word Compare-and-Swap Operation," In Proceedings of the 16th International Symposium on Distributed Computing, pp 265-279, Springer-Verlag. 2002.

[20] Google. "Google Protocol Buffers," available at http://code.google.com/apis/protocolbuffers/, 2013.

[21] K. Wang et al. "Exploring Design Tradeoffs for Exascale System Services through Simulation." Tech Report, LANL 2013.

[22] K Wang et al. "Centralized and Distributed Job Scheduling System Simulation at Exascale." Tech Report, IIT, 2011.

[23] J. Banks et al. "Discrete-event system simulation - fourth edition." Pearson 2005.

[24] G. Grider. "Parallel Reconfigurable Observational Environment (PRObE)," available from http://www.nmc-probe.org, October 2012.

[25] Available online: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[26] E. Frachtenberg et al. "Storm: Scalable resource management for large-scale parallel computers." IEEE Transactions on Computers, 55(12), 1572-1587, 2006.

[27] J. D. Goehner et al. "LIBI: A Framework for Bootstrapping Extreme Scale Software Systems". Parallel Computing, 2012.

[28] P. Balaji et al. "PMI: A scalable parallel process-management interface for extreme-scale systems". In Recent Advances in the Message Passing Interface (pp. 31-41). Springer Berlin Heidelberg, 2010.

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

[30] K. Wang et al. "MATRIX: MAny-Task computing execution fabRIc at eXascale," tech report, IIT, 2013.

[31] K. Wang et al. "Paving the Road to Exascale with Many-Task Computing." Doctor Showcase, IEEE/ACM SC 2012.

[32] I. Sadooghi et al. "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing." IEEE/ACM CCGrid, 2014.

[33] J. Dinan et al. "Scalable work stealing", IEEE/ACM SC 2009.

[34] K. Ousterhout et al. "Sparrow: Distributed, Low Latency Scheduling," SOSP '13, Farmington, Pennsylvania, USA.

[35] K. Wang et al. "SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascales," ACM HPC 2013.

[36] K. Ramamurthy et al. "Exploring Distributed HPC Scheduling in MATRIX." Tech Report, IIT, 2013.

[37] X. Zhou et al. "Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System." Tech Report, IIT, 2013.