Towards Scalable Distributed Workload Manager with Monitoring-Based Weakly Consistent Resource Stealing

Ke Wang
Illinois Institute of Technology
kwang22@hawk.iit.edu

Xiaobing Zhou
Hortonworks Inc.
xzhou@hortonworks.com

Kan Qiao
Google
qiaokan.buaa@gmail.com

Michael Lang
Los Alamos National Laboratory
mlang@lanl.gov

Benjamin McClelland
Intel
benjamin.m.mcclelland@intel.com

Ioan Raicu
Illinois Institute of Technology
iraicu@cs.iit.edu

ABSTRACT

One way to efficiently utilize the coming exascale machines is to support a mixture of applications in various domains, such as traditional large-scale HPC, the ensemble runs, and the fine-grained many-task computing (MTC). Delivering high performance in resource allocation, scheduling and launching for all types of jobs has driven us to develop Slurm++, a distributed workload manager directly extended from the Slurm centralized production system. Slurm++ employs multiple controllers with each one managing a partition of compute nodes and participating in resource allocation through resource balancing techniques. In this paper, we propose a monitoring-based weakly consistent resource stealing technique to achieve resource balancing in distributed HPC job launch, and implement the technique in Slurm++. We compare Slurm++ with Slurm using micro-benchmark workloads with different job sizes. Slurm++ showed 10X faster than Slurm in allocating resources and launching jobs – we expect the performance gap to grow as the job sizes and system scales increase in future high-end computing systems.

Categories and Subject Descriptors

C.5.1 [**Computer System Implementation**]: Large and Medium ("Mainframe") Computers – *Super (very large) computers.*

General Terms

Design, Algorithm, Performance.

Keywords

Workload manager, job launch, scheduling, resource balancing.

1. INTRODUCTION

With the predication that exascale supercomputers will have billion-way parallelism [1], one way of efficiently utilizing the whole machine is to support a mixture of applications in different domains that include traditional large-scale high performance computing (HPC), HPC ensemble runs, and fine-grained loosely coupled many-task computing (MTC) [2].

HPDC'15, June 15 - 19, 2015, Portland, OR, USA
© 2015 ACM. ISBN 978-1-4503-3550-8/15/06$15.00
DOI: http://dx.doi.org/10.1145/2749246.2749249.

Traditional **large-scale HPC applications** typically require many computing processors (e.g. half or full-size of the whole machine) for a long time (e.g. days or weeks). The jobs are tightly coupled, and use the message-passing interface (MPI) programming model [3] for communication and synchronization among all the processors. Although it is necessary to support HPC applications that demand the computing capacity of an exascale machine, it is also important to enable **ensemble runs** of applications that have uncertainty in high-dimension parameter space. Ensemble runs [4] decompose applications into many small-scale and short-duration coordinated jobs with each one doing a parameter sweep in a much lower-resolution parameter space using MPI in parallel, thus enabling a higher system utilization of exascale machines. Another domain of applications involves the **many-task computing (MTC)** [5] paradigm. MTC decomposes applications as orders of magnitude larger number (e.g. millions to billions) of embarrassingly parallel tasks with data-dependencies [6]. Tasks are fine-grained in both size (e.g. per-core) and duration (e.g. sub-second), and are represented as Direct Acyclic Graph (DAG) where vertices are tasks and the edges denote the data flows.

Running a mixture of applications in all domains on large-scale systems poses significant scalability challenges (e.g. grids [7][8], storage [9][10]) on workload managers (e.g. Slurm [11], PBS [12], and SGE [13]), which, up to date, have a centralized architecture where a controller manages all the compute daemons and are in charge of the activities, such as node partitioning, resource allocation, job scheduling and launching. This architecture cannot scale to exascale machines that will have system sizes one or two orders of magnitude larger for thousands of times more jobs with much wider distributions in both sizes and durations. Future exascale machines, along with a miscellaneous collection of applications, will demand orders of magnitudes higher job-delivering rates to make full utilization of the machine. The scalability challenge has driven us to develop the next generation distributed workload managers [14][15].

We have developed the Slurm++ [16] workload manager targeting all the applications at exascale. Slurm++ extends Slurm by applying multiple controllers with each one managing a partition of compute daemons and balancing resources among all the partitions through resource stealing techniques. Slurm++ utilizes ZHT [17], a distributed key-value store (KVS) [18][19], to keep the resource state information. In this paper, we propose a monitoring-based weakly consistent resource stealing technique to achieve dynamic resource balancing. We implement the technique in Slurm++, which shows 10X faster than Slurm in launching jobs. We expect the performance gap to grow as the job sizes and system scales increase in future high-end computing systems.

2. Slurm++ WORKLOAD MANAGER

Slurm++ is a distributed workload manager with a partition-based architecture, as shown in Figure 1. Slurm++ employs multiple controllers with each one managing a partition of compute daemons (*cd*). The partition size (the number of *cd* a controller managers) is configurable according to the application domains. We can also configure heterogeneous partition sizes to support workloads with a wide distribution of job sizes and with special requirements (e.g. only run on the partitions that have GPUs, InfiniBand, or SSDs). The users can submit jobs to arbitrary controller. Slurm++ deploys ZHT, a distributed key-value store (KVS), to manage the entire resource metadata and system state. One typical configuration is to co-locate a ZHT server with a controller forming 1:1 mapping, such as shown in Figure 1. Each controller is a ZHT client and uses the client APIs (e.g. *lookup*, *insert*) to communicate with ZHT servers to store, query and modify the metadata represented as (*key*, *value*) pair, for example, (*controllerId*, *free_node_list*), of resources of all the partitions.

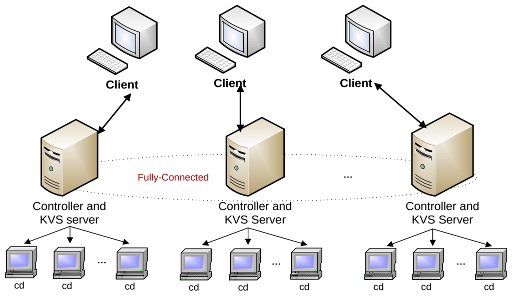
Client
Client
Client
Fully-Connected
...
Controller and KVS server
Controller and KVS Server
Controller and KVS Server
cd
cd
...
cd
cd
cd
...
cd
cd
cd
...
cd

Figure 1: Slurm++ distributed workload manager

Note that Slurm also divides the system into multiple partitions. However, this is different from the partition management of Slurm++. Firstly, Slurm layers the partitions hierarchically up to the centralized controller, while Slurm++ is distributed by employing one dedicated controller to manage a partition independently from other partitions. The hierarchical layout leads to longer latency as a job may need to go through multiple hops. What's more, the root controller is still a central piece with limited capacity. Secondly, in Slurm, when a job is scheduled on one partition, it can only get allocation within that partition. This results in long job queueing time in over-loaded partitions, and poor utilization if loads are not balanced among the partitions.

3. RESOURCE STEALING TECHNIQUE

Resource balancing means to find the required number of free nodes in all the partitions as fast as possible to satisfy concurrent job submissions. It is trivial in the centralized architecture, as the controller has a global view of the system state. However, for the distributed architecture, resource balancing is a critical goal, and should be achieved dynamically in a distributed fashion by all the controllers, in order to maintain an overall high system utilization.

Inspired by the work stealing technique [20] that achieves distributed dynamic load balancing, we introduce the resource stealing concept to achieve distributed dynamic resource balancing. Resource stealing refers to a set of techniques of stealing free nodes from other partitions if the local one cannot satisfy a job in terms of job size. When a controller allocates nodes for a job, it first checks the local free nodes. If there are enough free nodes, then the controller directly allocates the nodes; otherwise, it allocates whatever resources the partition has, and queries ZHT for other partitions to steal resources from them. Our previous work [16] proposed a straightforward random resource stealing technique, which has poor performance for big jobs. This section proposes a ***monitoring-based weakly consistent resource stealing technique*** that has better scalability than the random one. The proposed technique relies on a centralized monitoring service and each controller conducts a two-phase tuning procedure.

3.1 Monitoring Service

One of the reasons that the random technique is not scalable is because the controllers have no global view (even weakly consistent) of the system resource state. One alternative to enable all of the controllers to have global view is to alter the partition-based architecture to that the controllers know all the compute daemons. Then, there will be merely one (*key*, *value*) record of the global resource stored in a specific ZHT server. This method of strongly consistent global view is not scalable because all the frequent KVS operations on the resources are processed by a single ZHT server that stores the global resource (*key*, *value*) pair.

In order to keep a weakly consistent view of the global resources in each controller, we apply a monitoring service (MS) to query the free resources of all the partitions periodically. In each round, the MS looks up the free resource of each partition in sequence, and then gathers them together as global resource information and puts the global information as one (*key*, *value*) record in a ZHT server. This (*key*, *value*) record offers a global view of resource states for all the controllers.

This is different from the alternative mentioned above in that the frequency of querying this global (*key*, *value*) record is much less. Although the MS is centralized and queries all the partitions sequentially, we believe that it should not be a bottleneck. Because the number of partitions for large-scale HPC applications is not that many (e.g. 1K), and with the right granularity of frequency of updating and gathering the global resource information, the MS should be scalable. The MS could be implemented either as a standalone process on one compute node or as a separate thread in a controller. In Slurm++, the MS is implemented as the latter case.

3.2 Two-Phase Tuning

Each controller will conduct a two-phase tuning procedure of updating resources in the aid of allocating resources to jobs.

3.2.1 Phase 1: Pulling-based Macro-Tuning

As the MS offers a global view of the system free nodes (being kept in one ZHT server), each controller will periodically pull the global resource information by a ZHT *lookup* operation. In each round when the controller gets the global resources, it organizes the resources of different partitions as a binary-search tree (BST) data structure. Each data item of the BST contains the controller id (**char***) and the number of free nodes (**int**) of a partition. The data items are organized as a BST based on the number of free nodes of all partitions. The BST guides a controller to steal resources from the most suitable partitions.

We call this phase macro-tuning as it evicts the cached free resource information in BST, and updates the BST with the new information in each periodic query. This update is consistent for all the controllers as the resource information is pulled from a single place by all the controllers. Each controller pulls the information before it is too obsolete to offer valuable guidance.

3.2.2 Phase 2: Weakly Consistent Micro-Tuning

The controller uses the BST as a guide to choose the most suitable partitions to steal resources. The operations of the BST structure we implement to best serve the job resource allocation are:

BST_insert(BST*, char*, int): insert a data item to the BST data structure specifying the number of free nodes of a partition.

BST_delete(BST*, char*, int): delete a data item from the BST data structure for a partition.

BST_delete_all(BST*): evict all the data items from the BST data structure for all the partitions.

BST_search_best(BST*, int): for a given number of required compute nodes of a job, this operation searches for the most suitable partition to steal free nodes. There are 3 cases: (1) multiple partitions have enough free nodes; (2) only one partition has enough free nodes; (3) none of the partitions have enough free nodes. For case (1), it will choose the partition that has the minimum number of free nodes among all the partitions that have enough free nodes. For case (2), it will choose the exact partition that has enough free nodes. For case (3), it will choose the partition that has the maximum number of free nodes.

BST_search_exact(BST*, char*): given a specific controller id, this operation searches the resource information of that partition.

The complete resource allocation procedure is described as follows. When a job is submitted to a controller, the controller first tries to allocate free nodes in local partition. As long as the allocation is not satisfied, the controller searches for the most suitable partition to steal resources from the BST. The controller then queries the actual free resource of that partition via a ZHT *lookup* operation. After that, the controller issues a ZHT *compare and swap* atomic operation to allocate resources. If the allocation succeeds, the controller will insert the updated free node list of that partition to the local BST. Otherwise, if the controller experiences several failures in a row, it releases the allocated free nodes for the job, waits some time, and tries this procedure again.

We call this the micro-tuning phase because only the data of the resource of one partition is changed during one attempted stealing. Every controller updates its BST individually. As time increases, the controllers would have inconsistent view of the free resources of all the partitions. In the meantime, the controller is updating the whole BST with the most current resources of all the partitions (macro-tuning phase). With both macro-tuning and micro-tuning, the resource stealing technique has the ability to balance the resources among all the partitions dynamically, to aggressively allocate the most suitable resources for big jobs, and to find the free resources quickly under high system utilization.

4. EVALUATION

We implement the monitoring-based weakly consistent resource stealing technique in the Slurm++ workload manager. The implementation source code is made open source on GitHub repository: https://github.com/kwangiit/SLURMPP_V2. The dependencies are Slurm [11], ZHT [17], and Google Protocol Buffer [21]. We evaluate Slurm++ by comparing it with Slurm using the micro-benchmark workloads up to 500 nodes.

4.1 Comparison between Slurm++ and Slurm

We run all the experiments on the Kodiak cluster from the PROBE environment at LANL [22]. Kodiak has 500 nodes, and each node has two 64-bit AMD Opteron processors at 2.6GHz and 8GB memory. For Slurm++, the partition size is set to be 50. At the largest scale of 500 nodes, there are 10 controllers. The workloads include the simplest possible NOOP "sleep 0" jobs that require various numbers of nodes per job with 3 different distributions: each controller runs 50 **one-node jobs**; each controller runs 50 jobs with sizes having uniform distribution that has an average of half partition – 25 (1 to 50), referred to **half-partition jobs**; and each controller runs 20 jobs with sizes having uniform distribution that has an average of full partition – 50 (25 to 75), referred to **full-partition jobs**.

Figure 2 shows throughput speedups between Slurm++ and Slurm with the three workloads. We see that for all the workloads, Slurm++ is able to launch jobs faster than Slurm. The performance slowdown (9.3X from one-node to full-partition jobs) of Slurm due to increasingly large jobs is much more severe than that (2.3X from one-node to full-partition jobs) of Slurm++. This highlights the better scalability of Slurm++. In addition, the speedup is increasing as the scale increases for all the workloads, indicating that at larger scales, Slurm++ would outperform Slurm even more. Another important fact is that as the job size increases, the speedup also increases (2.61X for one-node, 8.5X for half-partition, and 10.2X for full-partition). This trend proves that the proposed technique has great scalability for big jobs.

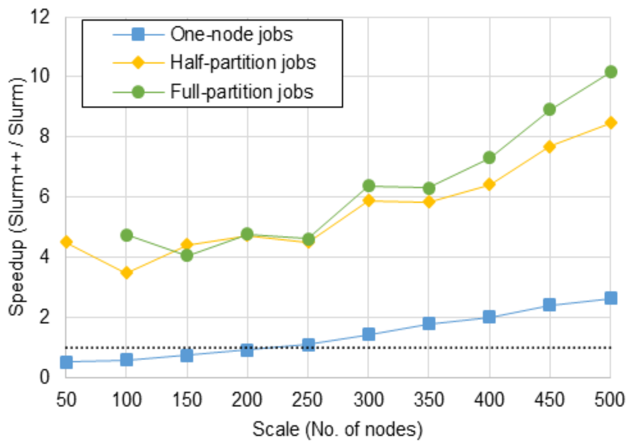

Figure 2: Speedup summary between Slurm++ and Slurm

5. RELATED WORK

Slurm [11] does scalable job launch via a tree based overlay network. But as we have evaluated, as scales grow, the scheduling cost per node increases, requiring coarser granularity workloads to maintain efficiency. BPROC [23] was a single system image and single process space clustering environment where all process ids were managed and spawned from the head node. BPROC moved virtual process spaces from the head node to the compute nodes via a tree spawn mechanism. However, BPROC was a centralized server with no failover mechanism. ALPS [24] is Cray's resource manager that constructs a management tree for job launch, and controls separate daemon with each one having a specific purpose. It is multiple single-server architecture, with many single-point of failures. ORCM [25] is an Open Resilient Cluster Manger originated from the Cisco runtime system for monitoring enterprise-class routers, and is under development in Intel to do resource monitoring and scalable job launching. Currently, the state management of ORCM is centralized in the top layer aggregator, which is not scalable. The use of KVS to manage the state similar to Slurm++ is an alternative for ORCM.

6. CONCLUSIONS AND FUTURE WORK

Exascale machine require next generation workload managers to deliver jobs with much higher throughput and lower latency for a mixture of applications. With the proposed resource stealing technique, Slurm++ showed performance 10X better than the Slurm production system, and the performance gap is expected to grow as the jobs and system scales increase. In the future, we will explore elastic resource allocation that could dynamically expand and shrink the allocated resources for the composed applications in Slurm++. This will help Slurm++ maintain high system utilization for a broader category of ensemble applications. Additions to this work would also include the investigations of distributed power-aware scheduling at the core level. Currently, Slurm++ allocates the whole node to a job, and doesn't consider the power effect. We will over-decompose a node, and launch jobs at the core level in order to save power. Another extension is to integrate Slurm++ with the MTC task execution fabric, MATRIX [26][27][28] (and the SimMatrix simulator [29][30][31]) and CloudKon [32], and study the scheduling of data-intensive HPC applications [33][10][34].

7. ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy contract AC52-06NA25396, and in part by the National Science Foundation under award CNS-1042543 (PRObE). We thank Ralph Castain from Intel for his thoughtful comments.

8. REFERENCES

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

[2] Michael Wilde, et al. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", SciDAC 2009.

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

[4] A. Basermann and K. Solchenbach. "Ensemble Simulations on highly Scaling HPC Systems (EnSIM)." CiHPC - Competence in High Performance Computing, June 2010.

[5] I. Raicu, Y. Zhao, et al. "Many-Task Computing for Grids and Supercomputers." IEEE MTAGS 2008.

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

[7] Catalin Dumitrescu, Ioan Raicu, Ian Foster. "Experiences in Running Workloads over Grid3", GCC 2005

[8] Catalin Dumitrescu, Ioan Raicu, Ian Foster. "The Design, Usage, and Performance of GRUBER: A Grid uSLA-based Brokering Infrastructure", JGC 2007.

[9] Dongfang Zhao, Ioan Raicu. "Distributed File Systems for Exascale Computing", Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012

[10] D. Zhao, Z. Zhang, et al. "FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems." IEEE International Conference on Big Data 2014.

[11] M. Jette, A. Yoo, M. Grondona. "SLURM: Simple Linux utility for resource management." JSSPP 2003.

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

[13] W. Gentzsch, et al. "Sun Grid Engine: Towards Creating a Compute Power Grid." CCGrid 2001.

[14] K. Wang, I. Raicu. "Towards Next Generation Resource Management at Extreme-Scales." IIT, PhD Proposal, 2014.

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

[16] K. Wang, X. Zhou, et al. "Next generation job management systems for extreme-scale ensemble computing." ACM HPDC 2014.

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

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

[19] A. Kulkami, K. Wang, et al. "Exploring Design Tradeoffs for Exascale System Services through Simulation." Tech Report, Los Alamos National Laboratory, 2013.

[20] J. Dinan, D.B. Larkins, et al. "Scalable work stealing." IEEE/ACM Supercomputing/SC, 2009.

[21] Google. "Google Protocol Buffers," available at https://github.com/google/protobuf/, 2015.

[22] G. Gibson, G. Grider, et al. "Probe: A thousand node experimental cluster for computer systems research." 2013.

[23] E. Hendriks. "BProc: The Beowulf distributed process space." ACM Proceedings of ICS, 2002.

[24] M. Karo, R. Lagerstrom, et al. "The application level placement scheduler." Cray User Group, pp. 1-7, 2006.

[25] Intel, ORCM, https://github.com/open-mpi/orcm, 2015.

[26] K. Wang, A. Rajendran, et al. "MATRIX: MAny-Task computing execution fabRIc at eXascale." Tech Report, IIT, 2013.

[27] K. Wang, I. Raicu. "Scheduling Data-intensive Many-task Computing Applications in the Cloud." NSFCloud Workshop, 2014.

[28] K. Wang, A. Rajendran, et al. "Paving the Road to Exascale with Many-Task Computing." Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012.

[29] K. Wang, K. Brandstatter, I. Raicu, "SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascale." ACM HPC, 2013.

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

[31] D. Zhao, D. Zhang, et al. "Exploring Reliability of Exascale Systems through Simulations." ACM HPC 2013.

[32] I. Sadooghi, S. Palur, 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] K. Wang, X. Zhou, et al. "Optimizing Load Balancing and Data-Locality with Data-aware Scheduling." IEEE International Conference on Big Data 2014.

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