Exploring Distributed Resource Allocation Techniques in the SLURM Job Management System

Xiaobing Zhou[*], Hao Chen[*], Ke Wang[*], Michael Lang[†], Ioan Raicu[*] [‡]
[*]Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA
[†]Ultra-Scale Research Center, Los Alamos National Laboratory, Los Alamos NM, USA
[‡]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL, USA
xzhou40@hawk.iit.edu, hchen71@hawk.iit.edu, kwang22@hawk.iit.edu, mlang@lanl.gov, iraicu@cs.iit.edu

***Abstract*— With the exponentially growth of distributed computing systems in both flops and cores, scientific applications are growing more diverse with a variety of workloads. These workloads include traditional large-scale High Performance Computing MPI jobs, and ensemble workloads, such as Many-Task Computing workloads comprised of extremely large number of tasks of finer granularity, where tasks are defined on a per-core or per-node level, and often execute in milliseconds to seconds. Delivering high throughput and low latency for these heterogeneous workloads requires developing distributed job management system that is magnitudes more scalable and available than today's centralized batch-scheduled job management systems. In this paper, we present a distributed job launch prototype SLURM++, which extends the SLURM resource manager by integrating the ZHT zero-hop distributed key-value store for distributed state management. SLURM++ is comprised of multiple controllers with each one managing several SLURM daemons, while ZHT is used to store all the job metadata and the SLURM daemons' state. We compared SLURM with our SLURM++ prototype with a variety of micro-benchmarks of different job sizes (small, medium, and large) at modest scales (500-nodes) with excellent results (10X higher job throughput). Scalability trends shows expected performance to be many orders of magnitude higher at tomorrow's extreme scale systems.**

Keywords-job management systems; job launch; distributed scheduling; key-value stores

I. INTRODUCTION

A. Background

Exascale supercomputers (10^18 ops/sec) will have millions of nodes and billions of concurrent threads of execution [1]. With this extreme magnitude of component count and concurrency, ensemble computing is one way to efficiently use the exascale machines without requiring full- scale jobs. Given the significant decrease of Mean-Time-To-Failure (MTTF) [2] at exascale levels, these ensemble workloads should be more resilient by definition given that failures will affect a smaller part of the machines. Ensemble computing would combine the traditional High Performance Computing (HPC) workloads that are large-scale applications with message-passing interface (MPI) [3] as the method for communication, with the ensemble workloads that would support the investigation of parameter sweeps using many more but smaller-scale coordinated jobs [4].

One example of ensemble workloads comes from the Many-Task Computing (MTC) [5] paradigm, which has several orders of magnitude larger number of jobs (e.g. billions) of finer granular tasks in both size (e.g. per-core, per-node) and duration (from sub-second to hours) that do not require strict coordination of processes at job launch as the traditional HPC workloads do. David Keyes identified reasons why today's computational scientists want performance: resolution, fidelity, dimension, artificial boundaries, parameter inversion, optimal control, uncertainty quantification, and the statistics of ensembles [6]; the last four can be addressed by MTC. A decade ago or earlier, it was recognized that applications composed of large numbers of tasks may be used as an driver for numerical experiments that may be combined into an aggregate method [7]. In particular, the algorithm paradigms well suited for MTC are Optimization, Data Analysis, Monte Carlo and Uncertainty Quantification. Various applications that demonstrate characteristics of MTC cover a wide range of domains, from astronomy, physics, astrophysics, pharmaceuticals, bioinformatics, biometrics, neuroscience, medical imaging, chemistry, climate modeling, and economics [8].

The job scheduling/management systems (JMS) for exascale ensemble computing will need to be versatile, scalable and available enough, in order to deliver extremely high throughput for both traditional HPC and ensemble workloads. However, today's batch schedulers (e.g. SLURM [9], Condor [10], PBS [11], SGE [12]) have centralized master/slaves architecture where a server/controller is handling all the activities, such as metadata management, resource provisioning, and job execution. This centralized architecture is not well suited for the demands of exascale computing, due to both poor scalability and single-point-of-failure.

One of the more popular and light-weight JMS, SLURM, reported maximum throughput of 500 jobs/sec [13]; however, we will need many orders of magnitude higher job submission and execution rates (e.g. millions jobs/sec) for next-generation JMS, considering the significant increase of scheduling size (10X higher node counts, 100X higher core counts, and significantly higher job counts), along with the much finer granularity of job durations (milliseconds/minutes, as compared to hours/days).

B. Motivation

The given distributed job launch prototype has poor performance in allocating resources when there are many launching threads that require resource concurrently. In this work, we aim to improve the distributed job launch prototype that is built on top of SLURM and ZHT [14, 28] systems. Resource contention will be much more severe if the workloads are big HPC jobs (e.g. 100-node job).

This paper proposes a distributed architecture for job management systems, and identifies the challenges and solutions towards supporting job management system at extreme-scales. We developed a distributed job launch prototype (SLURM++) with multiple servers (controllers) participating in allocating resources and launching jobs – an extension to the open source batch scheduler SLURM [9]. In order to hide the communication/coordination messages involved in maintaining distributed service architectures, such as those related to failure/recovery, replication and consistency protocols, we utilized distributed key-value stores (DKVS) to simplify the design and implementation. The general use of DKVS in building distributed system services was proposed, and evaluated through simulation in our previous work [15]. This work validates some of our prior simulation results that DKVS are a viable building block towards the development of more complex distributed services. We replaced the centralized controller (slurmctld) with many distributed controllers with each one managing a partition of compute resources. ZHT [14, 28] is the DKVS used in our prototype to store all the information related to the resources and jobs in a scalable, distributed, and fault tolerant way.

The key contributions of this paper are:

1. *Proposed a distributed architecture for job management systems, and identified the challenges and solutions towards supporting job management system at extreme-scales*
2. *Designed and developed a distributed resource stealing algorithm for efficient HPC job launch*
3. *Designed and implemented a distributed job launch prototype SLURM++ for extreme scales by leveraging SLURM and ZHT*
4. *Evaluate the centralized batch scheduler SLURM and our distributed SLRUM++ up to 500-nodes with various micro-benchmarks of different job sizes (small, medium, and large) with excellent results up to 10X higher throughput*
5. *Analyzed the evaluation results of SLURM++ and SLURM to point out what parts could be optimized to improve the overall throughput of the system*

The rest of this paper is organized as follows. Section II proposes the architecture, the problem set and the solutions, along with the design and implementation details of our distributed job launch prototype, SLURM++. Section III presents SLURM++ evaluation as it is compared to SLURM. Section IV analyzes the evaluation results of SLURM++ and SLURM. Section V presents the related work. We discuss future works and draw conclusions in Section VI and Section VII.

II. PROPOSED SOLUTION

The overall architecture of SLURM++ is envisioned and shown in Figure 1. Instead of using one centralized server/controller to manage all the compute daemons (cd), there will be multiple distributed controllers with each one managing a partition of cd. The controllers are fully-connected meaning that each controller is aware of all others. In addition, a distributed data storage system is demanded to be deployed on the machine to manage the entire job and resource metadata, as well as the state information in a scalable, reliable and fault tolerant way. The data storage system should also be fully-connected, and one configuration is to co-locate a data server with a controller on the same node forming one to one mapping, such as shown in Figure 1.

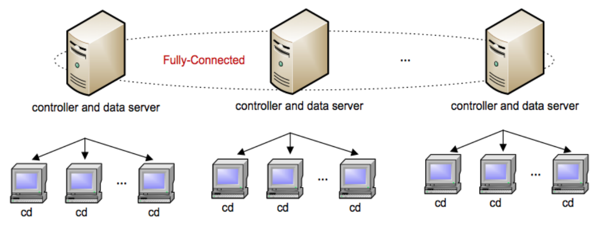
Fully-Connected
...
controller and data server
controller and data server
controller and data server
cd
cd
...
cd
cd
cd
...
cd
cd
cd
...
cd

Figure 1: Architecture for SLURM++; "cd" refers to compute daemon

The partition size (the ratio of the number of controllers to the number of compute daemons) is configurable, and can vary according to the application domain. For example, for a large-scale HPC workload where jobs usually require a large number of nodes to run, we can have each controller manage thousands of cd, so that the jobs are likely to be managed and launched locally (within the controller); for MTC jobs/tasks where a task usually requires small amount of nodes, or even a small number of cores within one node, we can push the controller down to the compute node to have the one-to-one mapping (millions of controllers and compute daemons at exascale). We can even have heterogeneous partition sizes for each controller to support different ensemble workloads.

The fully-connected topology is scalable under failure/recovery, replication and consistency protocols when the numbers of controllers and data servers are not many, such as in the HPC environment, both of them can be 1K at exascale with each one managing 1K cd. Besides, our previous simulation work showed that when the client messages that relate to process requests dominate all the messages in the system, the fully-connected topology has great scalability up to exascale [15]. When comes to the 1:1 mapping ideal for MTC workloads, we expect some distributed monitoring service to manage and maintain the fully-connected topology.

The distributed storage system could be a distributed file system, or a light-weight distributed key-value store (DKVS). For example, if we apply DKVS as the storage system, each controller would be initialized as a DKVS client, which then uses the simple DKVS client API interfaces (e.g. lookup, insert, remove) to communicate with the DKVS servers to query and modify the job and resource information, and the system state information transparently. In this way, the controllers don’t need to communicate explicitly with each other to query resources and jobs. Another benefit of using DKVS is that the DKVS could take over all the communications among controllers needed for maintaining distributed features, such as failure/recovery, replication, and consistency protocols.

A. Resource Contention vs. Compare and Swap

Resource contention happens when different controllers try to allocate the same resources. By querying the data storage system, different controllers may have the same view of the specific resources. For example, if we use DKVS to store the resource information of all the controllers, and controller 1 has 1000 nodes available currently. If controller 1 and controller 2 both need to allocate 100 nodes from controller 1, after they query DKVS, they would both get 1000 available nodes. Therefore, they will both update the number of free nodes of controller 1 to be 900. However, in reality, controller 1 needs to give out 200 nodes, which would lead to 800 free nodes.

One naive way to solve this resource contention problem is to add a global lock for each queried “key” in the DKVS. This approach is apparently not scalable considering the tremendously large volume of data stored. Another scalable approach is to implement some atomic operation in the DKVS that can tell the controllers whether the resource allocation succeeds or not. Learned from the traditional compare and swap atomic instruction [16], we come up with a specific compare and swap atomic algorithm that could address resource contention problem. The compare and swap procedure is given in the pseudo-code in ALGORITHM 1.

ALGORITHM 1. *Compare and Swap*

```
Input: key (key), value seen before (seen_value), new value
intended to insert (new_value), and the storage hash map (map).
Output: A Boolean value indicates success (TRUE) or failure
(FALSE), and the current actual value (current_value).
current_value = map.get(key);
if (!strcmp(current_value, seen_value)) then
    map.put(key, new_value);
    return TRUE;
else
    return FALSE;
end
```

Specifically, when a controller allocates and releases resources, the *compare* and *swap* operation will be used. Before a controller calls *compare* and *swap*, it queries the *seen_value* of the supplied key. Then, the controller updates the *seen_value* to get a *new_value*, and calls the *compare* and *swap*. After the DKVS server receives the *compare* and *swap* request, it executes the *compare* and *swap* operation, and returns the status to the controller. If the status is TRUE, then the request has been served successfully; otherwise, the client would use the returned the slurmctld first looks up the global resource data structure to allocate resources for the job. Once a job gets its allocation, it can be launched via a tree-based network rooted at rank-0 slurmd. When a job is finished, the rank-0 slurmd returns the result to slurmctld. The input to SLURM is one configuration file that is read by slurmctld and all slurmds. The configuration file specifies the identities of slurmctld and all the slurmds so that they can communicate *current_value* as the next *seen_value* and retry the procedure until getting success.

We implemented this compare and swap as extension to ZHT. The API looks like *compare_swap(key, seen_val, new_val, current_val)*. For the standard compare and swap, there are only two parameters, *seen_val* and *new_val*. It is like *compare_swap(seen_val, new_val)*, but in order to be compliant with ZHT key/value semantics, the key parameter in added, and moreover, one more parameter, *current_val*, is added as augment to keep the most recent value queried by that key in terms of ZHT server clock, even if this *compare* and *swap* probably is not satisfied. This additional parameter *current_val* saves one lookup network round trip that is required to get the most recent value by the later *compare* and *swap*.

B. Remote Polling vs. State Change Callback

SLURM does tree-based job launch, in a few words, the slurmctld talks to the first slurmd(with rank 0) in the list of slurmd that are assigned to jobs to be run. When job is done, control flows from the first slurmd to slurmctld. This design is not good enough for multiple controllers that are fully connected, for example, controller A steals resource from B, slurmd in B helps run jobs for A and return control to only B's slurmctld, in order to avoid coupled communication between controllers such as A and B, B has to write when job is done to ZHT, and A keeps polling ZHT to know when job is done by lookup. This polling contributes 90% of total messages involved.

One operation we implemented in ZHT is the state change callback operation that is specific to the job returning procedure in SLURM++. Without this new operation, the controller would have to poll the ZHT server on a regular basis on a regular basis incuring intensive network communication overhead between the controller and ZHT. This new operation implemented a blocking "state change callback" operation that keeps polling in the server side (with local messages) until the correct state is found, and a callback over the network is performed to complete the call. In order to make the "state change callback" operation more adaptive, we set adjustable timeout to the blocking period. After the timeout, if the state hasn't changed, the quering controller will do state change callback operation again. By moving the polling messages to local messages, this operation helps improve performance under intense resource stealing by reducing the number of remote message.

C. Operation Level Thread Safe vs. Socket Level Thread Safe

ZHT server is highly scalable to support concurrent incoming requests, so it would be nice to make client highly scalable in terms of concurrency in multi-threaded context. The ZHT server is thread-safe, while ZHT client is not. The naive way to do that is to create a shared mutex to protect any shared data access by operations in client, that's so called operation level thread-safe, but benchmark shows that the single shared mutex is performance bottleneck, because all operations, e.g. *insert*, *delete*, *append*, *lookup* and *compare_swap* are scheduled totally sequentially for synchronization. As we dig deeper, we found that any network related concurrency issues come from shared socket over which send/receive overlapped. We propose making ZHT client as thread safe not in operation level but in socket as well as MPI rank level, when there are many ZHT servers, the mutex contention due to concurrency caused by SLURM++ controller's multithreaded job launch could be distributed to many sockets or MPI ranks, which is a promising improvement.

D. Random Stealing vs. Resource Status Caching

Instead of doing totally random selection of controllers to steal resource from, we propose that a better solution would be caching how much resource available in certain controller, after it is contacted and stolen resource from. Based on this knowledge, when resource is needed for next time job scheduling, the controller would select the cached controllers with maximum free resources as candidates to steal from. In order to make this mechanism adaptive to frequent resource changes, several parameters are designed to be tuned, for example, the interval of cleaning total cache, eviction polices, cache pool size, and so on.

E. Random Stealing vs. Consulting Resource Monitor

A distributed monitor is designed to memorize resource status of controllers. When controller starts, it registers itself to distributed monitor. Controller reports its resource status to monitor when it changed. So when controller needs to steal resources, it consulted monitor that has global knowledge of free resource.

Most state-of-art distributed monitors are a cluster of nodes that replicate states reported for durability, and process incoming query requests concurrently by multiple nodes, but they are not totally distributed so horizontal scalability is hard to be achieved. In order to make distributed monitor dynamically expands and shrinks, we propose to augment ZHT to implement a distributed Bi-directional sorted map as a distributed monitor. The keys are sorted, and so are values. For example, key is controller id, value is number of free nodes along with free nodes list, where number of free nodes is used as index for sorting. Whenever a query request for free nodes is issued by SLURM++ scheduler, distributed monitor responds by returning the first N lightly loaded controllers with most free nodes, simply by lookuping the distributed Bi-directional workload sorted map maintained. Since this distributed monitor is on top of ZHT, it inherits horizontal scalability due to peer-to-peer symmetric feature.

Another approach is to adopt AMQP protocol based system like Apache Qpid. We prefer all to all mapping, that is, creating a distributed queue for every SLURM++ controller in initialization, all controllers register themselves as resource-state-change subscriber of all other controllers' distributed queues. Whenever one certain controller's resource state changed, it publishes this event to get all interested subscribers notified of. To make this approach ideally scalable horizontally and system dynamically expanding and shrinking, the brokers that take care these distributed queues have to be fully connected and equipped with bi-directional routes as double of all.

F. Implementation Details

We developed SLURM++ in the C programming language. We implemented the controller code, re-wrote part of the "srun" code of SLURM inside the controller, which summed to around 3K lines of code; this is in addition to the SLURM codebase of approximately 50K-lines of code (which was left mostly unmodified) and the ZHT codebase of 8K lines of code (of which 2K lines of code were added to implement compare and swap, as well as the state change callback). We put the controller and ZHT directly in the SLURM source file, and name the whole prototype SLURM++. SLURM++ has dependencies on Google Protocol Buffer [17], ZHT [14], and SLURM.

III. EVALUATION

With the solution that we proposed in last section, we evaluate the SLURM++ prototype by comparing it with the SLURM job launch with three different micro-benchmarks (small jobs, medium jobs and big jobs) on a Linux cluster up to 500 nodes. We configure SLURM++ with both HPC and MTC architectures to evaluate the general use case of SLURM++. This section presents the experiment environment, evaluation metrics, as well as architecture configuration and evaluation results.

A. Experiment Environment

We conduct all the experiments on the Kodiak cluster from the Parallel Reconfigurable Observational Environment (PROBE) at Los Alamos National Laboratory [18]. Kodiak has 1028 nodes, and each node has two 64-bit AMD Opteron processors at 2.6GHz and 8GB memory. The network supports both Ethernet and InfiniBand. Among the 1028 nodes, there were 500+ nodes available for our experiments (some nodes are pre-occupied, and some were down). We conducted experiments up to 500 nodes. The version of SLURM we use is version 2.5.3, the latest stable version when we started this work.

B. Metrics

The metrics used to evaluate the performance are throughput (jobs/sec), and ZHT message count. Throughput is calculated as the number of jobs finished dividing by the total launch time. For SLURM, the total launch time is the time difference between the earliest starting time of launching individual jobs, and the latest ending time of launching individual jobs. For SLURM++, the throughput of each controller is calculated, and then all the throughputs are summed up as the final total throughput. The ZHT message count metric just applies to our SLURM++.

C. Partition Size

The partition size (number of slurmds a controller manages) is configurable. In our experiment sets, for HPC configuration, we set the partition size to 50; each controller is responsible for 50 slurmds, and at the largest scale (500 nodes), the number of controllers is 10. We choose a moderate partition size insure a sufficient number of controllers to compare and contrast the performance of SLURM job launch with that of our distributed job launch; for MTC configuration, the partition size is 1 and the controller and slurmds are collocated at the same compute nodes, therefore, at the largest scale, we will have 500 controllers and 500 slurmds with one-to-one mapping.

D. Small-Job Workload (job size is 1 node)

In our evaluation, we use micro-benchmark workloads. Since our goal is to study the ability of SLURM and SLURM++ to handle job launch efficiently, we designed the simplest possible workload – many independent NOOP HPC jobs (e.g. sleep 0) that require different number of compute nodes per job.

The first workload we used just includes one-node small jobs (essentially MTC jobs), and does not require any resource stealing because all the jobs could be satisfied locally. Specifically, each controller launches 50 jobs, with each job requiring just 1 node. The format of the jobs is "srun –N1 /bin/sleep 0". Therefore, when the number of controller is n (number of compute nodes is 50*n), the total number of jobs is 50*n. The same workload is applied to SLURM job launch – 50*n nodes will have 50*n "srun –N1 /bin/sleep 0" jobs. This workload is used to test the pure job launching speed in the best case scenario from a performance perspective. The throughput comparison results are shown in Figure 2, and Figure 3 shows the ZHT message count (both summation and average) of SLURM++.

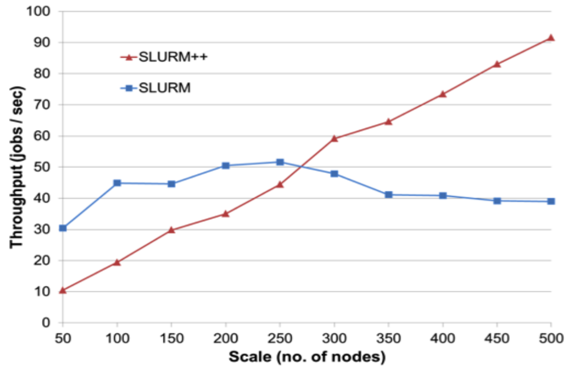

Figure 2: Small-Job; throughput comparison between SLURM and SLURM++

From Figure 2, we see that for SLURM job launch, the throughput first increases to a saturation point, and then has a decreasing trend as the number of nodes scales up (51.6 jobs / sec at 250 nodes, down to 39 jobs / sec at 500 nodes). This is because the processing capacity of the centralized “slurmctld” is limited. Even though all jobs can be satisfied in terms of job size, it takes longer and longer time for the slurmctld to launch jobs as the job count and system scale increase. For our SLURM++, the throughput increases almost linearly with respect to the scale, and this linear speedup trend is likely to continue at larger scales. At scale of 500 nodes, SLURM++ can launch jobs at 2.34X faster rates than SLURM job launch (91.5 jobs/sec vs. 39 jobs/sec). In addition, given the fact that the throughput of SLURM++ is increasing linearly while SLURM has a decreasing trend, we believe that the gap between SLURM++ and SLURM will only grow as the scale is increased.

Figure 3 shows the individual and overall message counts going to the ZHT servers from all the controllers. The overall message count experience perfectly linear increase with respect to the scale, while the average message count remains almost constant. These trends show great scalability of our SLURM++ for this small-job workload. In prior work on evaluating ZHT [14], micro-benchmarks showed ZHT achieving more than 1M ops/sec at 1024K node scales. We see that at the largest scale, the number of all messages is about 6K for 500 jobs (or about 12 messages / job). Even with our NOOP workloads achieving 91.5 jobs per second at 500 node scales, it generates 1098 messages / second. ZHT was far from being a bottleneck for the workload and scale tested.

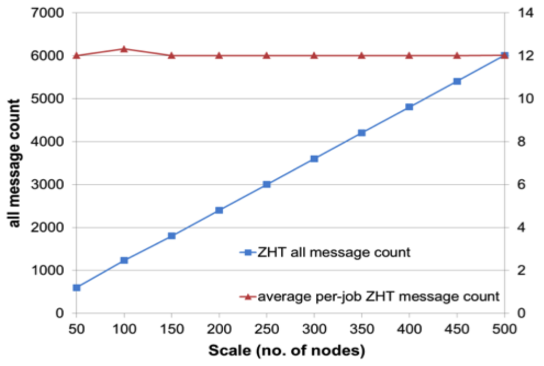

Figure 3: Small-Job; ZHT message count of SLURM++

We also tuned the configuration to best support MTC workloads (e.g. one-to-one mapping of controllers to slurmd, both running on the same compute node). Each controller launches just one job requiring just one node. We ran experiment up to 200 nodes; there would be 200 controllers and 200 slurmds with one-to-one mapping. The throughput and all ZHT message count results are shown in Figure 4.

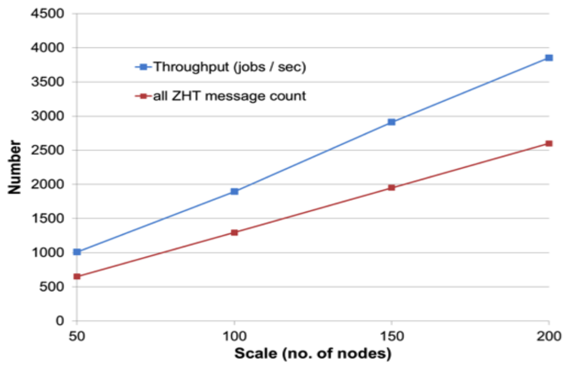

Figure 4: Small-Job; throughput and message count of MTC configuration

Both the throughput and all ZHT message count increase linearly with respect to the scale perfectly. In the 1:1 MTC configuration, based on these trends, ideally, we can achieve 20K jobs / sec at 1K-node scale, which will need to process 650K ZHT messages; ZHT can support more than 1M ops/sec at 1K-nodes, which is about twice as high as the expected number of messages generated by the 20K jobs/sec expected from SLURM++. Besides, SLURM++ could be configured less aggressively for larger partition sizes, which effectively would reduce the traffic load to ZHT due to smaller number of controllers.

E. Medium-Job Workload (job size is 1-50 nodes)

The second experiment tests how SLURM and SLURM++ behave under moderate job sizes that will result in moderate resource stealing. The workload is that each controller launches 50 jobs, with each job requiring a random number of nodes ranging from 1 to 50. So, at the largest scale, the total number of jobs is 500, and each job requires 1-50 nodes. The throughput and the ZHT message count results are shown in Figure 5 and Figure 6, respectively.

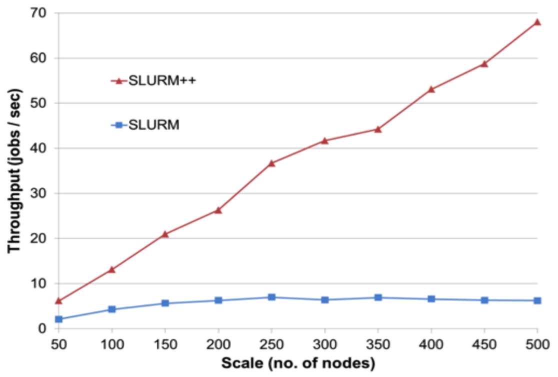

Figure 5: Medium-Job; throughput comparison between SLURM and SLURM++

Figure 5 shows that for SLURM, as the number of nodes scales up, the throughput increases a little bit (from 2.1 jobs / sec at 50 nodes to 7 jobs/sec at 250 nodes), and then keeps almost constant or with a slow decrease. For SLURM++, the throughput increases approximately linearly with respect to the scale (from 6.2 jobs / sec at 50 nodes to 68 jobs / sec at 500 nodes). Our SLURM++ prototype can launch jobs faster than SLURM at any scale we evaluated, and the gap is getting larger as the scale increases. At the largest scale, SLURM++ can launch jobs 11X (68 / 6.2) faster than SLURM; and the trends show that this speedup would continue at larger scales.

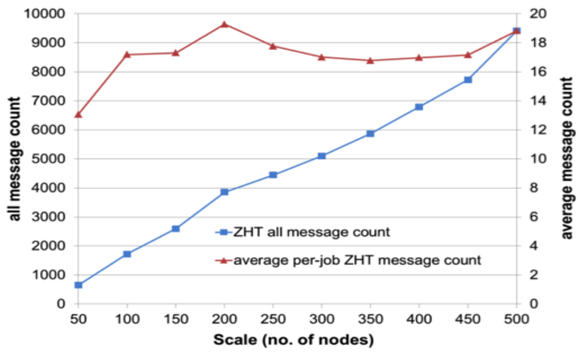

Figure 6: Medium-Job; ZHT message count of SLURM++

From Figure 6, we see that the overall ZHT message count increases somehow linearly with respect to the scale; the average per-job ZHT message count first increases slightly (from 13 messages / job at 50 nodes to 19 messages / job at 200 nodes), and then experience perturbations after that. The average per-job ZHT message count will likely keep within a range (17-20), and might be increasing slightly at large scales. This extra number of messages comes from the involved resource stealing operations.

F. Big-Job Workload (job size is 25 – 75 nodes)

The third experiment sets test the ability of both SLURM and SLURM++ to launch jobs under a serious resource stealing case. In this case, each controller launches 20 jobs, where each job requires a random number of nodes ranging from 25 to 75. At the largest scale, the total number of jobs is 20 * 10 = 200, and each job requires 25-75 nodes. The throughput and the ZHT message count results are shown in Figure 7 and Figure 8.

In Figure 7, SLURM shows a throughput increasing trend up to 500 nodes (from 1.2 jobs / sec at 100 nodes to 4.3 jobs / sec at 500 nodes), and the throughput is about to saturate after 400 nodes (from 3.8 jobs / sec at 400 nodes to 4.3 jobs / sec at 500 nodes). On the other hand, the throughput of SLURM++ keeps increasing almost linearly up to 500 nodes, and will likely keep the trend at larger scales. Like the mid-job case, SLURM++ can launch jobs faster than SLURM at any scale we evaluated, and the gap is getting larger as the scale increases. At the largest scale, SLURM++ can launch jobs about 4.5X (19.3 / 4.3) times faster than SLURM; and again, the trends show that this speedup would continue at larger scales.

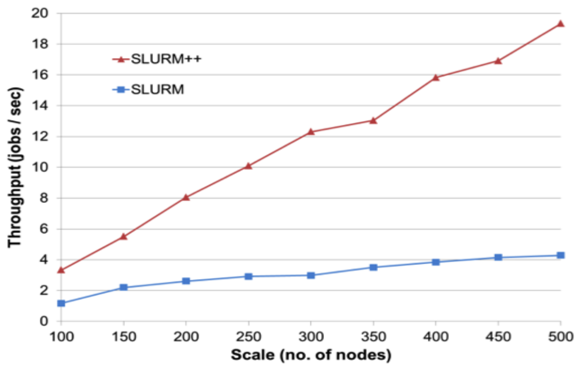

Figure 7: Large-Job; throughput comparison between SLURM and SLURM++

Figure 8 shows that the overall ZHT message count is increasing sub-linearly and the average per-job ZHT message count shows decreasing trend (from 30.1 messages / job at 50 nodes to 24.7 messages at 500 nodes) with respect to the scale. This is likely because when adding more partitions, each job that needs to steal resource would have higher chance to get resource as there are more options. This gives us intuition about how promising the resource stealing and compare and swap algorithms would solve the resource allocation and contention problems of distributed job management system towards exascale ensemble computing.

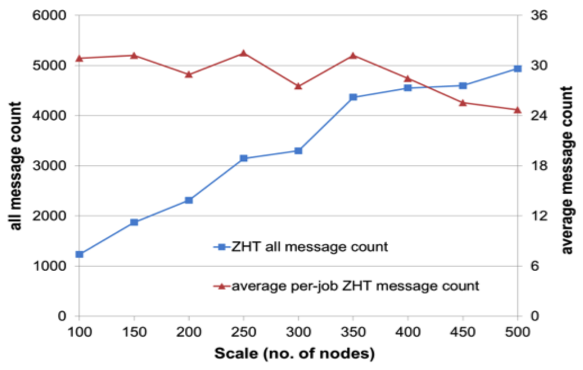

Figure 8: Large-Job; ZHT message count of SLURM++

G. Discussion

The conclusions we can draw up to here are that SLURM++ with multiple distributed controllers and ZHT servers have great scalability, and is likely to deliver orders of magnitude higher throughput than SLURM at extreme scales for traditional HPC workloads, as well as ensemble workloads. In order to understand the impact of workload on the performance of job launch, we show the result of comparing the throughputs of the three workloads with different resource stealing intensities (small-job/medium-job/big-job) in Figure 9.

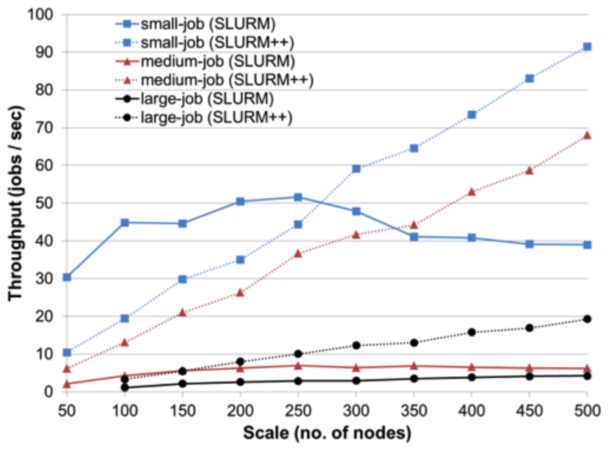

Figure 9: Throughput comparison with different workloads

For SLURM (the solid lines), we see that from small-job to large-job, the throughput decreases from 39 jobs/sec to 4.3 jobs / sec at 500 node scales (a slowdown of 9X). For our distributed job launch (the dotted lines), we observe that from small-job to large-job, the throughput decreases from 91.5 jobs / sec to 19.3 jobs / sec at the largest scale (a slowdown of 4.7X). We point out that not only does SLURM++ outperform SLURM in nearly all cases (except the small scale), but the performance slowdown due to increasingly larger jobs at large scale is better for SLURM++ by 2X, highlighting the better scalability of SLURM++.

As the result shown above, it is safe to draw the conclusion that SLURM++ surpasses SLURM in efficiently handling job launch at three different kinds of workloads. For the small one, SLURM++ exceeds SLURM at a certain scale (250 nodes) and still increasing, but SLURM has a decreasing trend after that point. It is due to the single centralized server/controller (slurmctld) issue in SLURM; the job launching time becomes longer when to system scale goes up. In contrast, SLURM++ uses multiple distributed controllers with each one control a partition of computer daemons. For the medium one, SLURM++ launches jobs faster than SLURM at any scale, and the gap between them is becoming larger as the scale increases since the throughput of SLURM keeps almost constant even with a certain level of decreasing while it increases linearly on the side of SLURM++. For the big-job case, SLURM++ also launches jobs faster than SLURM at any scale that we evaluated with the gap becomes large and large.

IV. PERFORMANCE ANALYSIS

The current resource stealing approach works as following, when a controller allocates nodes for the job, it first checks the local free nodes by querying the data storage system. If there are enough available nodes, then the controller directly allocates the nodes; otherwise, it will query for other partitions to steal resources from them. The resource stealing procedure is given in the pseudo-code in ALGORITHM 2.

As long as there are not enough nodes to satisfy the allocation, the resource stealing algorithm will randomly selects a controller and tries to steal nodes from it. Every time when the selected controller has no available nodes, the launching controller sleeps some time (*sleep_length*) and retries. If the launching controller experiences several failures (*num_retry*) in a row because the selected controller has no free nodes, it will release the resources it has already stolen, and then tries the resource stealing algorithm again. The number of retries and the sleep length after stealing failure are critical to the performance of the algorithm, especially for many big jobs, where all the launching controllers try to allocate nodes and steal resources from each other.

ALGORITHM 2. Resource Stealing

```
Input: number of nodes required (num_node_req), number of controllers (num_ctl), controller membership list (ctl_id[num_ctl]), sleep length (sleep_length), number of reties (num_retry).
Output: list of involved controller ids (ctl_id_inv), participated nodes (par_node[]).
num_node_allocated = 0; num_try = 0; num_ctl_inv = 0;
while num_node_allocated < num_node_req do
    remote_ctl_idx = Random(num_ctl);
    remote_ctl_id = ctl_id[remote_ctl_idx];
    again:
    remote_free_resource = lookup(remote_ctl_id);
    if (remote_free_reource == NULL) then
        continue;
    else
        remote_num_free_node = strtok(remote_free_source);
        if (remote_num_free_node > 0) then
            num_try = 0;
            remote_num_node_allocated =
                remote_num_free_node > (num_node_req –
                num_node_allocated) ? (num_node_req –
                num_node_allocated) : remote_num_free_node;
            if (allocate nodes succeeds) then
                num_node_allocated +=
                    remote_num_node_allocated;
                par_node[num_ctl_inv++] = allocated node list
                strcat(ctl_id_inv, remote_ctl_id);
            else
                goto again;
            end
        else
            sleep(sleep_length);
            num_try++;
            if (num_try > num_retry) do
                release all the allocated nodes;
                Resource Stealing again;
            end
        end
    end
end
return ctl_id_inv, par_node;
```

A. Per-Job Sampling vs. Batch Sampling

The resource stealing cited above is pretty naive. Simple random selection is not good because multiple random selections of controllers shared nothing between. The current approach is not batch sampling but nothing more than per-job sampling. We can view one random selection as one probe. In per-job sampling, number of free nodes on controller is the only basis for consideration, for example, chances are that controller with few free nodes will release nodes more quickly than that with more free nodes but release nodes slower. In this case, the former is a choice in priority rather than the latter to steal resource from because these will result in higher throughput as overall. Batch sampling adopts the power of two techniques, that is, it suggests that, each time, the SLURM++ controller is to randomly probe N*2 or N*2*2 controllers for N jobs to be scheduled, this will overcome per-job's shortcomings due to shared information between probes.

B. Eagerly Resource Stealing vs. Late Binding

In current approach, if there are free nodes in controllers probed, scheduler eagerly steals the resources, but this doesn't necessarily guarantee higher throughput as overall. A better solution would be late binding, for example, the scheduler send resource reservation to the controllers probed if they have free nodes. If the controllers promised the reservation, and the initial scheduler will wait for notification when the reservation is really satisfied, and then schedule running the job. Even if the reservation is partially satisfied at some point by a certain controller, the initial scheduler is left freedom and enough information to decide where to steal resource from since it is informed of many potential candidates with free nodes that promised reservation. This will surprisingly improve overall throughput.

C. Sleep-and-wait vs. Nonsleep-and-notify

In current approach, scheduler randomly selects a controller and tries to steal nodes from it. Every time when the selected controller has no available nodes, the launching controller sleeps some time (*sleep_length*) and retries. If the launching controller experiences several failures (*num_retry*) in a row because the selected controller has no free nodes, it will release the resources it has already stolen, and then tries the resource stealing algorithm again. During the sleeping time, controller occupies nodes stolen but does nothing; this could lead to accumulatively and causally related lower utilization. Scheduling proficiency heavily relies on parameters tuning. We prefer nonsleep-and-notify method as described in B.

D. Distributed Job Submission

So far, all job submissions take place on a single node for both SLURM baseline and SLURM++ benchmark. If the submissions are distributed many nodes, the throughput will be higher.

V. RELATED WORK

SLURM [9] is one of the most popular traditional batch schedulers, which uses a centralized controller (slurmctld) to manage compute nodes that run daemons (slurmd). SLURM does have scalable job launch via a tree based overlay network rooted at rank-0, but as we will show in our evaluation, the performance of SLURM remains relatively constant as more nodes are added. This implies that as scales grow, the scheduling cost per node increases, requiring coarser granular workloads to maintain efficiency. SLURM also has the ability to configure a "fail-over" server for resilience, but this doesn't participate unless the main server fails. There are other JMSs that have been deployed on clusters and supercomputers as resource managers for years, such as Condor [10], PBS [11], and SGE [12]. All of them have a similar centralized architecture as SLURM. We choose SLURM as the basis of our work instead of others, because SLURM is open source and well supported.

There have also been several other projects that have addressed efficient job launch mechanisms. In STORM [20], the researchers leveraged the hardware collective available in the hardware of the Quadrics QSNET interconnect. They then used the hardware broadcast to send out the binaries to the compute nodes. Though this work is as scalable as the interconnect, the server itself is still a single point of failure. BPROC [21] was a single system image and single process space clustering environment where all process id were managed and spawned from the head node, and then distributed to the compute nodes. BPROC transparently 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. LIBI/LaunchMON [22] is a scalable lightweight bootstrapping service specifically to disseminate configuration information, ports, addresses, etc. for a service. A tree is used to establish a single process on each compute node, this process then launches any subsequent processes on the node. The tree is configurable to various topologies. 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 [23] is the process management layer in MPICH2. It is close to our work in that it uses a KVS to store job and system information. But the KVS is a single server design rather than distributed and therefore has scalability as well as resilience concerns. 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.

There are also light-weight task execution frameworks that are developed specifically for ensemble workloads. In the MTC domains, Falkon [25] is a centralized task execution fabric with the support of hierarchical scheduling, while MATRIX [26] is the distributed task execution framework that uses work stealing [19] to achieve distributed load balancing. Though Falkon can deliver tasks at thousands of tasks/sec for MTC workloads, it is not sufficient for exascale systems and it lacks support for HPC workloads. Another fine granular framework that schedules sub-second tasks for data centers is Sparrow [27]. Though MATRIX and Sparrow have shown great scalability for MTC workloads, neither of them currently supports HPC workloads. The next-generation JMS should be able to schedule HPC and MTC, as well as ensemble workloads in an efficient, scalable and fault tolerant way.

VI. FUTURE WORK

We are still working on Resource Caching and Distributed Monitoring. Resource caching implementation is based on our CS525 (Advanced Database Organization) buffer manager prototype, which realizes customization of cache eviction policies (e.g. LRU, FIFO, CLOCK), pool size, frame and entity mapping, dirty marking, pin and unpin, and so on.

We implemented the distributed monitor proposed as part of our CS550 final project (Diskon: Distributed tasK executiON framework). It is based on AMQP based Apche Qpid distributed message queue system. So far, it's running as only one instance and showing throughput flattening or degrading when it scales up to hundreds of nodes. We will try that the brokers that take care these distributed queues have to be fully connected and equipped with bi-directional routes as double of all, as mentioned in section II.

Distributed Bi-directional sorted map based on ZHT is what we are also working on. We will compare it to AMQP based distributed monitor from scalability and throughput perspective.

VII. CONCLUSION

Extreme-scale supercomputers require next-generation job management systems to be more scalable, available while delivering jobs with much higher throughput. We have shown that DKVS is a valuable building block to allow scalable job launch and control. The performance is more preferable than the production job launch software – SLURM, and is better for both traditional HPC workloads and ensemble MTC workloads at modest scales of 500 nodes. Furthermore, the distributed job launch prototype proved to have better scalability trends, and showed its potential to scaling to extreme scales towards supporting both MTC and HPC workloads.

REFERENCES

[1] 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.

[2] 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.

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

[4] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu. “Swift: Fast, Reliable, Loosely Coupled Parallel Computation,” IEEE Workshop on Scientific Workflows 2007

[5] 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.

[6] D. Keyes. “Exaflop/s, seriously!,” Keynote lecture for Pan-American Advanced Studies Institutes Program (PASI), Boston University, 2010.

[7] D. Abramson, J. Giddy, L. Kotler. “High performance parametric modeling with Nimrod/G: Killer application for the global grid,” In Proc. International Parallel and Distributed Processing Symposium, 2000.

[8] I. Raicu, I. Foster, M. Wilde, Z. Zhang, A. Szalay, K. Iskra, P. Beckman, Y. Zhao, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, D. Thain. “Middleware Support for Many-Task Computing,” Cluster Computing Journal, 2010.

[9] M. A. Jette, A. B. Yoo, M. Grondona. “SLURM: Simple Linux utility for resource management.” 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003), pages 44–60, Seattle,Washington, USA, June 24, 2003.

[10] D. Thain, T. Tannenbaum, M. Livny, “Distributed Computing in Practice: The Condor Experience” Concurrency and Computation: Practice and Experience 17 (2-4), pp. 323-356, 2005.

[11] 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.

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

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

[14] 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 International Parallel & Distributed Processing Symposium (IPDPS) 2013.

[15] K. Wang, A. Kulkarni, M. Lang, D. Arnold, I. Raicu. "Using Simulation to Explore Distributed Key-Value Stores for Extreme-Scale Systems Services," IEEE/ACM Supercomputing/SC 2013.

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

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

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

[19] 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.

[20] Frachtenberg, E., Petrini, F., Fernández, J., & Pakin, S. (2006). Storm: Scalable resource management for large-scale parallel computers. Computers, IEEE Transactions on, 55(12), 1572-1587.

[21] Hendriks, E. (2002, June). BProc: The Beowulf distributed process space. In Proceedings of the 16th international conference on Supercomputing (pp. 129-136). ACM.

[22] Goehner, J. D., Arnold, D. C., Ahn, D. H., Lee, G. L., de Supinski, B. R., LeGendre, M. P., ... & Schulz, M. (2012). LIBI: A Framework for Bootstrapping Extreme Scale Software Systems. Parallel Computing.

[23] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Krishna, J., Lusk, E., & Thakur, R. (2010). 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.

[24] Karo, M., Lagerstrom, R., Kohnke, M., & Albing, C. (2006). The application level placement scheduler. Cray User Group, 1-7.

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

[26] K. Wang, A. Rajendran, I. Raicu. "MATRIX: MAny-Task computing execution fabRIc at eXascale," available at: http://datasys.cs.iit.edu/projects/MATRIX/index.html, 2013.

[27] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica. "Sparrow: Distributed, Low Latency Scheduling," SOSP '13, Farmington, Pennsylvania, USA.

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