Towards Next-Generation Distributed Job Management Systems for Extreme Scales

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

***Abstract*—Today's computing systems are growing in both flops and cores at an exponential rate. Petascale systems are becoming common with the top 33 of the top 500 systems having peak petaflop capabilities; the fastest machines have millions of cores and tens of petaflops ratings. Yet many of the system services used in these state-of-the-art systems still have a predominantly centralized architecture that was conceived decades ago when systems had thousands of cores and teraflop performance. These systems have inherent scalability issues at tomorrow's extreme scales, and are vulnerable to single point of failure. In this work, we address the increasing pressure on centralized services used in High Performance Computing (HPC) systems by adopting some features of distributed system services that are successful in commercial Internet and Cloud space. Specifically, we developed a distributed job launch prototype based on the popular open source SLURM job management system, as well as the ZHT, a zero-hop distributed key-value store (DKVS). The prototype is comprised of multiple controllers with each one managing several SLURM daemons. ZHT is used to store all the job metadata and is used to resolve any contention for the resources. By using ZHT to hide the complexity involved in creating distributed services, we show that a DKVS can be used as a building block for distributed services and can speed their development and deployment. We compared SLURM with our prototype with various workloads. Results show that our prototype outperforms SLURM at scales up to 900 nodes even with the added complexity of a distributed service, and speculate on the potential impact such distributed job launch architecture could have at the extreme scales of tomorrow.**

Keywords—job management systems; system services; job launch; extreme scales; key-value store

I. INTRODUCTION

Exascale supercomputers (10^{18} ops/sec) will be consist of millions of nodes, and billions of concurrent threads of execution [1]. With this extreme magnitude of component count and concurrency, the mean-time-to-failure (MTTF) will be dramatically decreased to the order of several hours [2]. Therefore, system software and services at exascale will need to be fault-tolerant, self-healing and adaptive for efficient system utilization and sustained operation. However, most of today's HPC system services still have centralized Master/Slaves (Server/Clients) architecture, where a centralized server is in charge of all the activities, such as metadata management, resource provisioning, and job execution. This centralized server is inherently a bottleneck at extreme scales.

The solution to mitigate the issues arising from centralized architecture is to distribute the centralized server in either hierarchical or fully distributed architectures. In the hierarchical architecture, the servers are organized in a tree with different layers, which adds significant complexity when re-building the tree when servers fail. We propose using a fully-connected distributed server architecture, in which, each server is aware of all others. This architecture involves communication/coordination messages among all the servers.

In this work, we prototype a distributed job launch service to motivate and justify distributed key value stores (DVKS) as a general solution to HPC services, such as job and resource management, I/O forwarding, monitoring, booting, power management, fabric management, configuration management, and those run-time systems for programming models and communication libraries [3][4][5][6]. For extreme scale systems, these services all need to operate on large volume of data in a consistent, resilient and efficient manners. In order to transparently hide the complexity of servers communicating with each other, we apply the DKVS [23] in distributed system services and expect that the DKVS will take over all the communications involved in the features of distributed system services, such as recovery, replication and consistency models.

The specific DKVS we are using is ZHT [7], a zero-hop distributed key value store for high-end extreme scale systems. We list some use cases of DKVS in distributed system services here: for job management systems, DKVS can be used to keep all the resource information (busy or idle), and job execution status information (job submission, queuing, execution, and end times, job execution node list); for file systems, DKVS can be used to store all the file metadata (file creation, reading, and modifying times and frequencies); for monitoring, DKVS can be used to maintain system active logs; for job start-up, DKVS can be used to disseminate configuration and initialization data amongst composite tool or application processes, this is under development of MRNet [6]; application developers from Sandia National Laboratory are targeting DKVS to support local check-pointing restart [8]. Additionally, we have used ZHT to implement MATRIX [9][10], a many-task computing execution fabric, where ZHT is used to keep the information for task submission, dependency, and progress information; and the fusion based distributed file system, FusionFS [11], where ZHT is used in tracking the metadata.

Specifically, in this paper, we focus on developing one system service, distributed job launch. Instead of re-

implementing job management systems from scratch, we decided on starting with a popular open source centralized system SLURM [3], and prototype a distributed job launch architecture. Instead of using one centralized controller (slurmctld) to manage all the compute resources (slurmd), we distribute several controllers with each one managing a partition of compute nodes. The controllers are fully-connected, where each controller is aware of every other ones. Both the number of controllers and the partition sizes could be up to thousands or tens of thousands at extreme scales of millions of nodes (expected at exascales). ZHT is used to store all the information related to the resources and jobs in a scalable, distributed, and fault tolerant system. We use one-to-one mapping (this is configurable) of ZHT server to controller, meaning that for each controller, a ZHT server is co-located at the same node. A job can be served locally if there are enough resources in one partition; it can also be served by stealing sufficient resources from other partitions, for which, we defined a resource stealing protocol.

The key contributions of this paper are:

- *Architect, design and implement a distributed job launch prototype targeting extreme scales*
- *Apply DKVS towards the support of distributed services at extreme scales*
- *Evaluate SLURM and distributed job launch up to hundreds of nodes' scale with various workloads*

The rest of this paper is organized as follows: section II presents some related work about developing distributed job launch and other system services; section III describes the architecture, the design and implementation of our distributed job launch prototype; the prototype is evaluated and compared against SLURM in section IV; we draw conclusions and envision the future work in section V.

II. RELATED WORK

SLURM [3] uses a centralized controller (slurmctld) to manage compute nodes. 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 uses persistent daemons (slurmds) on compute nodes. 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 resource managers, such as Condor [12], PBS [13], and SGE [14], which have a similar centralized architecture as SLURM. We choose SLURM as the basis of our work instead of others, because SLURM is open source, is well supported, and it is light-weight (compared to other comparable systems).

There have also been several other projects that have addressed efficient job launch mechanisms. In STORM [15], 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 [16] 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 used "vexecmove" to transparently move virtual process spaces from the head node to the compute nodes via a tree spawn mechanism. BPROC was a centralized server with no failover mechanism. LIBI/LaunchMON [17] 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 [18] 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 [19] is Cray's resource management mechanism, it uses "aprun", which is much like SLURM that constructs a management tree for job launch, and controls separate daemon with each one having a specific purpose. It is a multiple single server architecture, with many single point of failures.

The chief difference between our prototype and the existing implementations is the use of a DKVS to allow multiple coordinated control servers. This employment of DKVS allows higher scalability, performance, and reliability.

III. DESIGN OF DISTRIBUTED JOB LAUNCH PROTOTYPE

In this section, we describe the architecture, the design and implementation of the distributed job launch prototype. We retain the original slurmd from SLURM, and write our own slim controllers (slurmcltd) which act as ZHT clients and send requests to the ZHT servers.

A. Prototype Architecture

The overall prototype architecture is shown in Figure 1, in which, we distributed several controllers with each managing a partition of compute nodes. The controllers are fully-connected, where each controller is aware of all others. ZHT is used to keep all the information related to the resources and jobs, and for each controller, there is one ZHT server co-located on the same node. This one-to-one mapping of controller and ZHT server is actually not a requirement; we can configure any ratio of controller to ZHT server. Each controller is initialized as a ZHT client, which then uses the ZHT client API interface to communicate with the ZHT servers. The SLURM daemon (slurmd) is located on every compute node, and a partition of compute nodes are managed by a dedicated controller. The controller can launch a job locally if there are enough resources, or it can steal enough resources from other partitions. One of the benefits of using ZHT is that the controllers don't need to communicate with each other to query resources and jobs, these will be conducted by querying the ZHT servers.

The ratio of the number of controllers to the number of slurmds is configurable, and depends on the application domain. For example, for an HPC workload (jobs usually

require a large number of nodes), we can have each controller manage thousands of slurmds; for Many-Task Computing (MTC) jobs/tasks (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 slurmds at exascale).

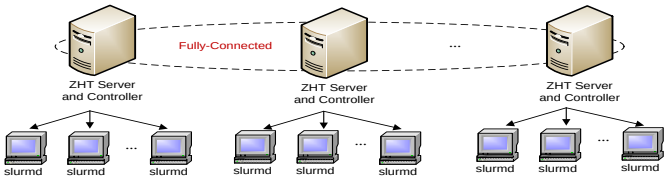

Figure 1: Distributed job launch prototype architecture

B. Job metadata

ZHT is a zero-hop persistent DKVS where each ZHT client has a global view of all the ZHT servers. For each operation of ZHT server (*insert*, *lookup*, *remove*, *append*, *compare and swap*), there is a corresponding client API. The client calls the API, which sends a request to the exact server that is responsible for the request by hashing the key. Upon receiving a request, ZHT server converts it to the corresponding operation type and executes the operation. ZHT serves as a data management building block for extreme scale system services, and has been tested with up to 8K nodes (32K cores) on a IBM Blue Gene /P supercomputer [20].

We use ZHT to keep important data related to jobs and resources of our distributed job launch prototype, part of the data stored and the description is listed in Table 1, where both key and value are strings (ZHT is not restricted to strings and supports generic data types).

Table 1: Job metadata stored in ZHT

Key	Value	Description
controller id	number of free node, free node list	The free (available) nodes in a partition managed by the corresponding controller
job id	original controller id	The original controller that is responsible for a submitted job
job id + original controller id	involved controller list	The controllers that participate in launching a job
job id + original controller id + involved controller id	participated node list	The nodes in a partition that are involved in launching a job

The controllers are initialized as ZHT clients. With these data stored in ZHT, the controllers can easily get the information about the resource utility and job execution specification by calling the ZHT client APIs (*insert*, *lookup*, *compare and swap*) without directly communicating with each other. We put the complexity (e.g. replication model, consistency model) introduced by distributed architecture into the DKVS [23], and expect that this design will accelerate the development and deployment of distributed system services.

C. Inter-Controller Communication

In order to facilitate communication between the controllers, we adopted the ZHT distributed key/value store. Three of the five operations of ZHT are used to store and manage the important data of our distributed job launch prototype. The operations used are *insert*, *lookup*, and *compare and swap*.

1) insert: An insert operation takes a (key, value) pair, puts the pair into the ZHT storage system (both the main memory and the persistant storage), and returns status.

2) lookup: A lookup operation takes a key, finds the value corresponding to the key in the ZHT storage system, and then returns the value.

3) Compare and swap: In order to resolve the resource contention problem caused by multiple threads or client processes requesting and releasing resources, ZHT provides a compare and swap operation. This allows multiple requests to concurently modify the same key/value objects. ZHT uses epoll event driven model to process all client requests, and therefore can guarantee all operations are atomic. This operation was added to ZHT specifically for the distributed job launch to support concurent access to common key/value objects, and to reduce both the number of messages, and potential race conditions. The *compare and swap* psuedo-code is given 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*).

```
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 is used. Before a client does *compare and swap*, it calls the *lookup* API to retrieve the *seen_value* of the supplied key. Then, the client updates the *seen_value* to get a *new_value*, and calls the *compare and swap* API. After the ZHT server receives the *compare and swap* request, it executes the *compare and swap* operation (Algorithm 1), and returns the status to client. If the status is *TRUE*, then the request has been served successfully; otherwise, the client would retry the procedure (first *lookup*, then *compare and swap*) until getting success.

For example, if the partition of controller 1 has 100 free nodes, and in ZHT, we store (*controller_id*, {*num_free_node*, *free_node_list*}) that is (1, {100, node-1, node-2, …, node-100}). If there are two jobs requiring 10 nodes of that partition concurrently, they both do *lookup*(1) first, and then get the *seen_value* {100, node-1, node-2, …, node-100}. After that they both try to allocate 10 nodes, and try to write the *new_value* {90, node-11, node-12, …, node-100} by doing *compare and swap* (1, {100, node-1, node-2, …, node-100}, {90, node-11, node-12, …, node-100}) . As ZHT uses epoll event driven model which serves the requests in sequential, it will process the two *compare and swap* operations (Algorithm 1) sequentially. The first *compare and swap* will succeed and update the value to {90, node-11, node-12, …, node-100}. For the second *compare and swap*, the *seen_value* {100, node-11, node-12, …, node-100} and the *curent_value* {90, node-11, node-12, …, node-100} are not equal, so the operation will fail. The job failed in the *compare and swap* operation will retry

again until it succeeds, which will eventually write the value {80, node-21, node-22, …, node-100}. This also applies when releasing resources.

D. Distributed Job Launch

The procedure to do job launch of each controller can be divided into the following sequential steps:

1) Initialize as ZHT client: A controller is initialized as a ZHT client. Before running the controllers, the ZHT servers should be started. The controllers can therefore call the ZTH client APIs (*insert, lookup, compare and swap*) as needed.

2) Read the controller membership list: The controller then reads the controller membership list from a local file. Before starting the controllers, all the controller ids would be written into a global membership list file, which is then copied locally to each controller. The controller membership list will be used for selecting the corresponding partition to steal resources if a job could not be satisfied locally (not enough free resources).

3) Read workload: The controller reads all the jobs to be launched from local worklod file. These workload files are pre-generated with each line representing a job. Each line has the format "srun –N$k /bin/sleep 0", where $k specifies the number of nodes this job requires. Currently, there is no client submitting jobs, which is future work. We consider "sleep 0" jobs to measure the overhead of our prototype in launching jobs, and the "srun" prefix comes from the SLURM.

4) Create a thread to receive messages: Next, the controller creates a receiving thread to receive messages. There are two types of messages that will be sent to controller, the *slurmd_registration* message, and the *job_step_finish* returing message.

5) Create individual thread to launch each job: The controller will be blocked until it receives the registration messages from all the slurmds (running on compute nodes). Then, for each job, the controller creates an individual thread to launch the job. We modified the SLURM "srun" program to launch jobs. The job launch includes four steps, and we use the functions offered by SLURM directly in each step. The steps are:

a) Allocate nodes: Unlike the SLURM, which allocates nodes based on global resource list, our prototype queries ZHT for avaible nodes.

b) Create job: After allocating enough nodes (specified by $k), the launching thread then creates a job structure.

c) Create job step: Next, the launching thread creates a job step structure based on the job structure. For now, each job has exact one job step.

d) Launch the job step: Finally, the launching thread launches the job step to the selected nodes.

6) Wait until all jobs are finished: The controller is then blocked until it receives all the *job_step_finish* returning messages. If all the nodes allocated to a job aren't local, then the *job_step_finish* message will go to the first involved controller instead of the original one. In this case, the first controller will insert a record (job_id+original_controller_id, "finished") to ZHT after it recevies the *job_step_finish* message, and the launching thread of the job will keep polling ZHT by doing *lookup*(job_id+original_controller_id) until it gets the "finished" value.

E. Resource Stealing

Resource stealing was inspired by a well-known technique known as work stealing [24]. When a launching thread allocates nodes for the job, it first checks the local free nodes by doing *lookup*(original_controller_id). If there are enough available nodes, then the launching thread directly allocates the nodes (using *compare and swap*); otherwise, it will query ZHT for other partitions to steal resources from them. The resource stealing pseudo-code is given in Algorithm 2.

ALGORITHM 2. Resource Stealing

```
Input: number of nodes required (num_node_req), number of controllers
(num_ctl), controller membership list (ctl_id[num_ctl]).
Output: involved controller ids (ctl_id_inv), participated nodes (par_node[]).
num_node_allocated = 0; num_try = 0; num_ctl_inv = 0;
ctl_id_inv = calloc(20 * 100, sizeof(char));
for each i in 0 to 19; do
    par_node[i] = calloc(100 * 100, sizeof(char));
end
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 = c_zht_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 //compare and swap
                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
            usleep(100000);
            num_try++;
            if (num_try > 2) do
                release all the allocated nodes;
                Resource Stealing again;
            end
        end
    end
end
return ctl_id_inv, par_node;
```

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 thread sleeps some time and retries. If the launching thread experiences several failures 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 big jobs, where all the launching threads from the same or different controllers try to allocate nodes. After tuning these parameters, we have choosen to retry 3 times, and sleep 100ms .

F. Implementation Details

We developed our prototype in C programming language. We implemented the controller code, re-wrote part of the "srun" code of SLURM inside the controller, which summed to 3K lines of code; this is in addition to the SLURM codebase of approximately 50K-lines of code and the ZHT codebase of 7K lines of code. We put the controller directly in SLURM source file. The prototype has dependencies on Google Protocol Buffer [21], ZHT [7], and SLURM [3].

IV. EVALUATION

We evaluate the distributed job launch prototype by comparing it with the SLURM job launch with three different workloads (small jobs, medium jobs and big jobs) on a Linux cluster up to 900 nodes. SLURM controller (slurmctld) is centralized, and is aware of all the slurmds. In the distributed job launch, there are multiple controllers with each one managing a partition of slurmds.

A. Experiment Environment

We conduct all the experiments on the Kodiak cluster from the Parallel Reconfigurable Observational Environment (PROBE) at Los Alamos National Laboratory [22]. Kodiak has 1028 nodes, where 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 900+ nodes available for our experiments (some nodes are pre-occupied, and some were down). We conducted experiments up to 900 nodes.

B. Metrics

The metrics used to evaluate the performance are **throughput** (jobs/sec), and **ZHT message count** (*insert* message count, *lookup* message count, *compare and swap* message count, and all 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 our distributed job launch prototype, 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 distributed job launch prototype.

C. Partition Size

The partition size (number of slurmds a controller manages) is configurable. In our experiment sets, we set the partition size to 100; each controller is responsible for 100 slurmds, and at the largest scale (900 nodes), the number of controllers is 9. The reason to choose 100 partition size is to balance the tradeoff between the configuration complexity of running the experiments, and having enough controllers to reflect the distributed system complexity.

If the partition size is too small (e.g. 10), it would be complicated to configure and run the experiments (at least in our current prototype). For example, each partition needs its own configuration file (like the SLURM configure file), and the default directory of the configuration file is predefined by "*./configure --prefix --sysconfdir*" command, after which, we do a "*make*" and a "*make install*" command. Therefore, we need to compile the SLURM program n times if we have n controllers in our current bootstrapping systems. Alternatively, if the partition size is too big (e.g. 500), as we just have 900 available nodes, we would not be able to have enough number of controllers to distinguish the centralized architecture with the distributed one. Therefore, we choose a moderate partition size (100) to ease the experiment configuration, and at the same time insure a sufficient number of controllers to compare and contrast the performance of SLURM job launch with that of our distributed job launch.

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

The first workload we used just includes one-node small jobs, and does not require resource stealing. Specifically, each controller launches 100 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 nodes is 100n), the total number of jobs is 100n. The same workload is applied to SLURM job launch – 100n nodes will have 100n "srun –N1 /bin/sleep 0" jobs. This workload is used to test the pure job launching speed at the best scenario. The performance results are shown in Figure 2 (throughput comparison), and Figure 3 (ZHT message count of our prototype).

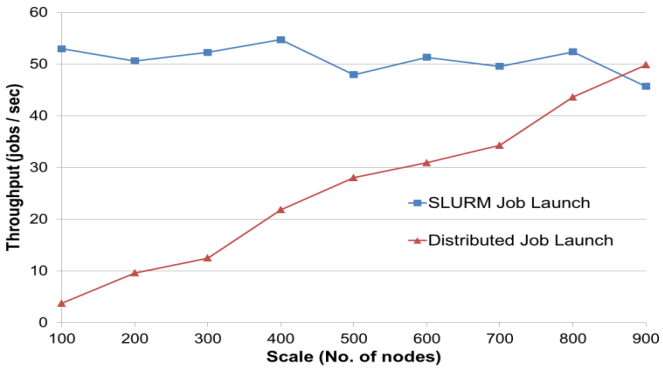

Figure 2: Small-Job; throughput comparison between SLURM and distributed job launch

From Figure 2, we see that for SLURM job launch, the throughput has a decreasing trend as the number of nodes scales up (53 jobs / sec at 100 nodes, down to 46 jobs / sec at 900 nodes). For the distributed job launch, the throughput increases linearly with respect to the scale, and this linear speedup trend is likely to continue at larger scales. By scales of 900 nodes, the distributed job launch is faster than SLURM job launch (50 jobs/sec vs. 46 jobs/sec). In addition, the throughput of our prototype is increasing linearly while SLURM has a decreasing trend, we believe that the gap between our prototype and SLURM will be bigger as the scale is increased.

Figure 3 shows the individual and overall message counts going to the ZHT servers from all the controllers. All the message counts experience linear increase with respect to the

scale. In prior work on evaluating ZHT [7], 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 less than 12K for 900 jobs (or about 13 messages per job). With experiments lasting many seconds (even tens of seconds in some cases), ZHT was far from being the bottleneck.

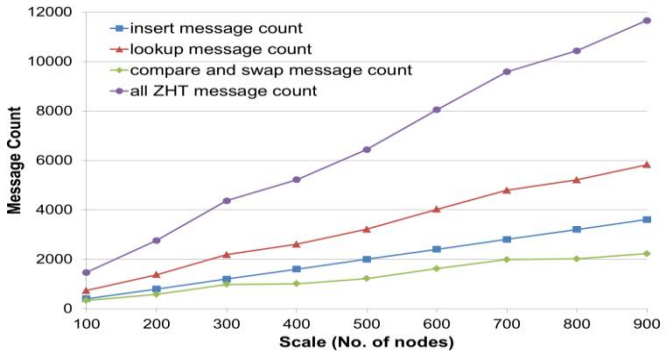

Figure 3: Small-Job; ZHT message count of distributed job launch

In the current HPC-workload configuration (1 controller to 100 slurmds), the throughput of our distributed job launch for small-job workload is low, especially at small scales (e.g. 2.8 jobs / sec at 100 nodes). We also tuned the configuration for MTC workload, that is one controller manages one slurmd, and both of the controller and the slurmd are at the same compute node. Each controller launches just one job requiring just one node. We ran experiment up to 10 nodes, and the throughput result is shown in Figure 4.

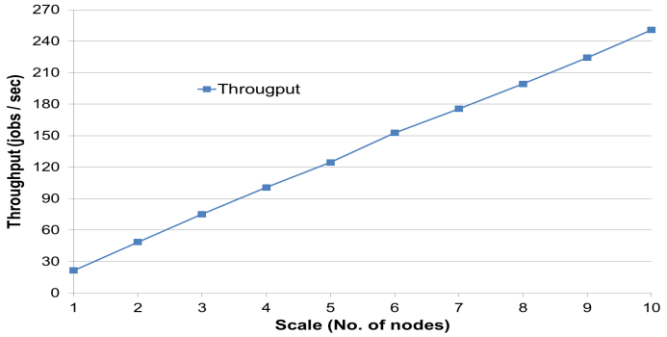

Figure 4: Small-Job: throughput of MTC configuration

The throughput depicted in Figure 4 is much higher, 22 jobs / sec at one node, and 250 jobs / sec at 10 nodes. In addition, the throughput increases linearly with respect to the scale perfectly. Within each controller, there is just one launching thread, no resource contention would happen. For the 1:100 mapping before, there are 100 launching threads querying ZHT for the same resource that leads to resource contention, which is why the throughput is as low as 2.8 jobs / sec. In the 1:1 MTC configuration, ideally, we can achieve 25K jobs / sec at 1K nodes, and 25M jobs / sec at exascale with 1M nodes; ZHT can support more than 1M ops/sec at 1K-nodes, which is one order of magnitude higher than the expected number of messages generated by the 25K jobs/sec expected from the distributed job launch. Our prototype is configurable, and it is expected that it could be configured differently for different workloads, with different mapping between the number of controllers and the number of slurmds in order to achieve the highest throughput.

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

The second experiment sets test how both job launchers will behave under a moderate job sizes (resulting 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 50 * 9 = 450, and each job requires 1-50 nodes. Statistically, the average number of required nodes of all jobs is 25, and the ratio of the number of required nodes to the number of available node is 12.5 (50 * 25 / 100), – for the small-job workload, the ratio is 1 – this ratio will lead to moderate resource stealing among the controllers. The performance results are shown in Figure 5 and Figure 6.

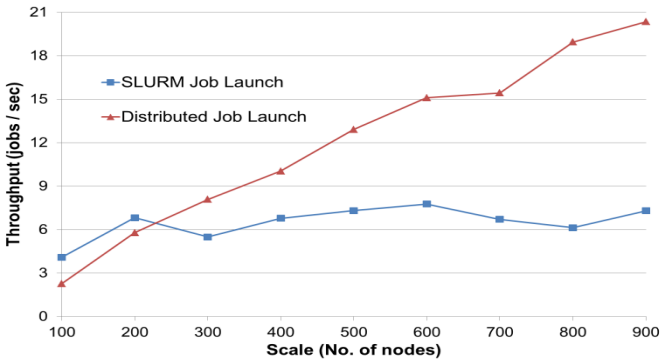

Figure 5: Medium-Job; throughput comparison between SLURM and distributed job launch

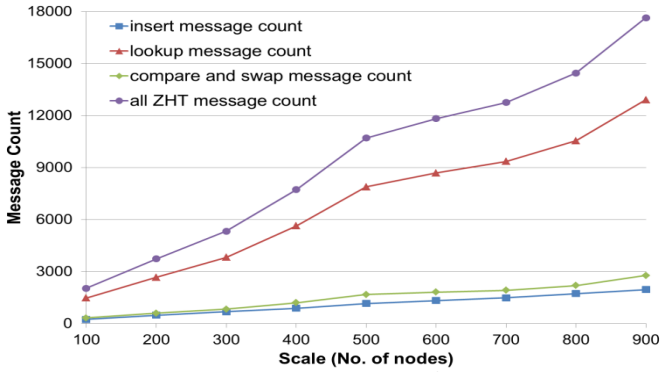

Figure 6: Medium-Job; ZHT message count of distributed job launch

Figure 5 shows that for SLURM job launch, as the number of nodes scales up, the throughput increases a little bit (from 4 jobs / sec at 100 nodes to 8 jobs/sec at 600 nodes), and then keeps almost constant or with a slow decrease. For the distributed job launch, the throughput increases approximately linearly with respect to the scale (from 2.8 jobs / sec at 100 nodes to 20.8 jobs / sec at 900 nodes). After 200 nodes, our prototype can launch jobs faster than SLURM, and the gap is getting larger as the scale increases. At the largest scale, the distributed job launch prototype can launch jobs about 2.5 (20.8 / 7.5) times faster than SLURM; and the trends show that this speedup would continue at larger scales.

From Figure 6, we see that the *insert* and *compare and swap* message counts don't increase from small-job workload to medium-job workload; however, the *lookup* message count of the medium-job case is more than 2 times of that of small-job workload, and it accounts for 75% (1300 / 17500) of all the

ZHT messages (50% for small-job case). This extra number of *lookup* messages comes from the more intense resource stealing operations.

F. Big-Job Workload (job size is 50 – 100 nodes)

The third experiment sets test the job launching capacity of both SLURM and the distributed prototype under a serious resource stealing case. In this case, each controller launches 50 jobs, with each job requiring a random number of nodes ranging from 50 to 100. So, at the largest scale, the total number of jobs is 50 * 9 = 450, and each job requires 50-100 nodes. For this workload, the average number of required nodes of all jobs is 75, and the ratio of the number of required nodes to the number of available node is 37.5 (50 * 75 / 100), – for the small-job case, the ratio is 1, and for the medium-job case, the ratio is 12.5 – this ratio will lead to serious resource stealing among the controllers. The performance results of this workload are shown in Figure 7 and Figure 8.

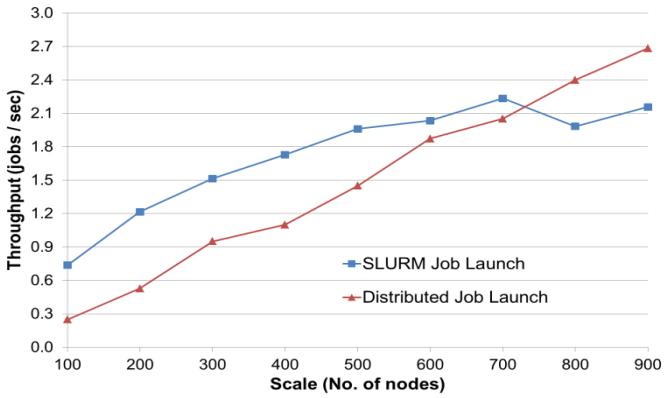

Figure 7: Big-Job; throughput comparison between SLURM and distributed job launch

In Figure 7, SLURM job launch shows a throughput increasing trend up to 700 nodes (from 0.75 jobs / sec at 100 nodes to 2.25 jobs / sec at 900 nodes), and then the throughput keeps almost constant (actually a little bit decreasing from 700 nodes to 900 nodes). However, like the previous two cases, the distributed job launch prototype experiences a linear increasing trend of the throughput with respect to scale (from 0.28 jobs / sec at 100 nodes to 2.7 jobs / sec at 900 nodes). After 700 nodes, our prototype can launch jobs faster than SLURM.

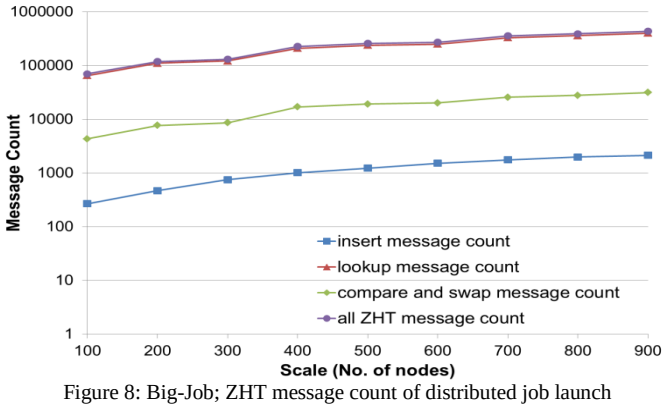

Figure 8: Big-Job; ZHT message count of distributed job launch

Figure 8 shows that both the *compare and swap* and the *lookup* message counts are much bigger than those of the previous two cases. At the largest scale, there are 30K *compare and swap* messages (10 times of that of the small-job and medium-job cases), and 400K *lookup* messages (more than 300 times of that of the medium-job case, and 600 times of that of the small-job case). The number of *lookup* messages accounts for almost 100% of the number all messages (these two lines are overlapped). This gives us intuition about how much overhead introduced by the frequent resource stealing activities among all the controllers.

We expect that the distributed job launch to perform best for many small job sizes (often found in HTC and MTC workloads). The larger the job sizes, the less of a performance gap there might be, although the gap will still exist at sufficiently large scales (e.g. 1K+ nodes). We will work towards optimizing the resource stealing algorithm in reducing the number of messages, and ultimately improving its performance for large job sizes.

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.

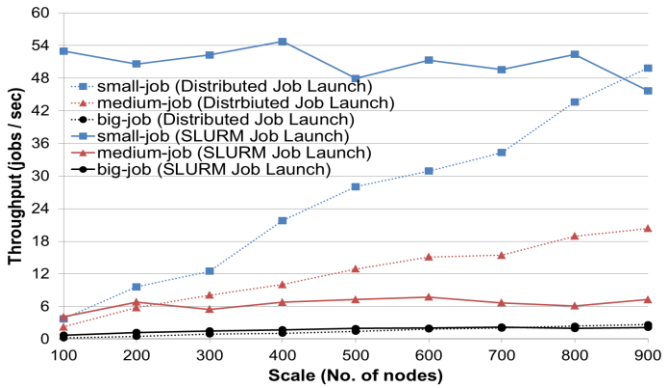

Figure 9: Throughput comparison with different workloads

For SLURM job launch (the solid lines), we see that from small-job to medium-job, the throughput decreases by about 84% at the largest scale (from about 45.7 jobs / sec to 7.3 jobs / sec); from medium-job to big-job, the throughput decreases by 70% (from 7.3 jobs / sec to 2.2 jobs / sec). For our distributed job launch (the dotted lines) , we observe that from small-job to medium-job, the throughput decreases by about 58% at the largest scale (from about 50 jobs / sec to 20.8 jobs / sec); from medium-job to big-job, the throughput decreases by almost 87% (from 20.8 jobs / sec to 2.7 jobs / sec) at largest scale. The more intensive the resource stealing is, the less throughput the prototype can achieve. In our distributed job launch, for the big-job case, there are times that every job launching thread is trying to steal resources from other controllers, while each of the launching threads holds part (but not enough) of the resources. We can experience "resource deadlock" at some point, if we don't set the relevant parameters (e.g. number of retries when failing to steal resources, sleep length after a failure of stealing resource, etc.) correctly, or if we don't release the reserved resources. Our prototype does release resources, and after tuning the parameters, we set the number of retries to 3, and the sleep length to 100ms (Algorithm 2).

Currently, the resource stealing algorithm is simple and straightforward. In future work, we will come up with better algorithms, such as asynchronous free node list table and caching previous discovered resources in order to reduce the contention and the message count in communicating with ZHT.

V. Conclusions and Future Work

We have shown that DKVS is a valuable building block to allow scalable job launch and control. The performance is comparable to production job launch software – SLURM, and is better for some workloads at modest scales of 900 nodes. Furthermore, the distributed job launch has better scalability trends, making it better suited at scaling to extreme scales. On the other hand, our distributed job launch prototype has limitations. It is very hard to preserve job order; Since we have many controllers, maintaining any strict job ordering policies would be hard, and would likely incur even more overheads. In addition, it involves laborious work to configure and run our prototype. We will address these limitations in the Future.

The prototype was developed to investigate the applicability of DKVS for exascale system services. High-priority additions to the work include the ability to launch MPI jobs and investigations of resilient job launch. MPI jobs require the additional information needed to setup the rank to processes mapping etc. The DKVS will aid in this work by allowing the mapping to be stored in the DKVS for distributed and scalable retrieval. Resilience studies of the DKVS job launch service will allow fast fault tolerant job allocation and launch, which are very fragile at current 10K node scales and are a much needed focus for future extreme-scale systems. Other future work involves investigating efficient "resource stealing" algorithms when the system is highly loaded.

Another application of this work is supporting the Many-Task Computing workload, where jobs/tasks are small and require a few nodes or cores. We will integrate our distributed job launch prototype with the many task computing execution fabric, MATRIX, and study different job scheduling algorithms.

Acknowledgment

This work was supported by the U.S. Department of Energy (DOE) contract DE-FC02-06ER25750, and in part by the National Science Foundation (NSF) under award CNS-1042543 (PRObE). We thank Tonglin Li from Illinois Institute of Technology, and Morris Jette and Danny Auble from SchedMD for their help and suggestions.

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. 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.

[4] N. Ali, P. Carns, K. Iskra, et al. "Scalable I/O Forwarding Framework for High-Performance Computing Systems".

[5] A. Vishnu, A. R.Mamidala, H. W. Jin, D. K. Panda. "PerformanceModeling of SubnetManagement on Fat Tree InfiniBand Networks using OpenSM." In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), Workshop 18, Washington, DC, USA, 2005.

[6] P. C. Roth, D. C. Arnold, B. P. Miller. "MRNet: A software-based multicast/reduction network for scalable tools." In Proc. IEEE/ACM Supercomputing '03, 2003.

[7] 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.

[8] M. A. Heroux. "Toward Resilient Algorithms and Applications," April 2013. Available from http://www.sandia.gov/~maherou/.

[9] I. Raicu, I. T. Foster, Y. Zhao. "Many-task computing for grids and supercomputers." In Many-Task Computing on Grids and Supercomputers Workshlp (MTAGS) 2008.

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

[11] D. Zhao, I. Raicu. "Distributed file systems for exascale computing. In Doctoral Showcase," SC'12: Proceedings of the 2012 ACM/IEEE Conference on Supercomputing, Salt Lake City, UT, November 2012.

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] Overview of the IBM Blue Gene/P project. "IBM Journal of Research and Development," 52(1.2):199‑220, Jan. 2008.

[21] Google Protocol Buffers: http://code.google.com/apis/protocolbuffers/, 2013.

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

[23] K. Wang, A. Kulkarni, M. Lang, D. Arnold, I. Raicu. "Using Simulation to Explore Distributed Key-Value Stores for Exascale Systems Services", under review at IEEE/ACM SC13

[24] R. D. Blumofe, et. al. "Scheduling multithreaded computations by work stealing," In Proc. 35th FOCS, pages 356–368, Nov. 1994.