MATRIX:DJLSYS

EXPLORING RESOURCE ALLOCATION TECHNIQUES FOR DISTRIBUTED JOB LAUNCH UNDER HIGH SYSTEM UTILIZATION

XIAOBING ZHOU (xzhou40@hawk.iit.edu)
HAO CHEN (hchen71@hawk.iit.edu)
Contents

- Introduction
- ZHT Enhancement for SLURM++
  - Compare and Swap
  - Resource State Change Callback
  - Thread Safe
    - Operation Level
    - Socket Level
  - ZHT Client Lock Exception Safe
Contents

- Related Work
- Benchmark
  - SLURM Baseline Benchmark
  - SLURM vs. SLURM++
- Working-on
  - Distributed Monitoring
  - Cache
  - Libnap standalone library
Proposal

- Resource State Change Callback
- Compare and Swap
- Socket Level Thread Safe
- Distributed Monitoring
- Cache and Buffer Management
Introduction

- SLURM++: A distributed job launch prototype for extreme-scale ensemble computing (IPDPS14 submission)
Job Management Systems for Exascale Computing

- Ensemble Computing
- Over-decomposition
- Many-Task Computing
- Jobs/Tasks are finer-grained

Requirements
- High availability
- Extreme high throughput (1M tasks/sec)
- Low latency
Current Job Management Systems

- Batch scheduled HPC workloads
- Lack the support of ensemble workloads
- Centralized Design
  - Poor Scalability
  - Single-point-of-failure
- SLURM maximum throughput of 500 jobs/sec
- Decentralized design is demanded
Goal

- Architect, and design job management systems for exascale ensemble computing
- Identifies the challenges and solutions towards supporting job management systems at extreme scales
- Evaluate and compare different design choices at large scale
Contributions

- Proposed a distributed architecture for job management systems, and identified the challenges and solutions towards supporting job management system at extreme-scales
- Designed and developed a novel distributed resource stealing algorithm for efficient HPC job launch
- Designed and implemented a distributed job launch prototype SLURM++ for extreme scales by leveraging SLURM and ZHT
- Evaluated SLURM and SLRUM++ up to 500-nodes with various micro-benchmarks of different job sizes with excellent results up to 10X higher throughput
Controllers are fully connected

Ratio and Partition Size are configurable for HPC and MTC

Data servers are also fully connected
# Job and Resource Metadata

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controller id</td>
<td>number of free node, free node list</td>
<td>The free (available) nodes in a partition managed by the corresponding controller</td>
</tr>
<tr>
<td>job id</td>
<td>original controller id</td>
<td>The original controller that is responsible for a submitted job</td>
</tr>
<tr>
<td>job id + original controller id</td>
<td>involved controller list</td>
<td>The controllers that participate in launching a job</td>
</tr>
<tr>
<td>job id + original controller id + involved controller id</td>
<td>participated node list</td>
<td>The nodes in a partition that are involved in launching a job</td>
</tr>
</tbody>
</table>
SLURM++ Design and Implementation

- SLURM description
- Light-weight controller as ZHT client
- Job launching as a separate thread
- Implement the resource stealing algorithm
- Developed in C
- 3K lines of code + SLURM 50K lines of code + ZHT 8K lines of code
Compare and Swap

- **Use case**
  - When different controllers try to allocate the same resources
  - Naive way to solve the problem is to add a global lock for each queried key in the DKVS
  - Atomic compare and swap operation in the DKVS that can tell the controllers whether the resource allocation succeeds
  - SLURM++ uses it to contend nodes resources

- **Standard compare-and-swap:**
  - `compare_swap(key, seen_val, new_val)`

- **Augument standard compare-and-swap**
  - `compare_swap(key, seen_val, new_val, queried_val)`
  - queried_val saves one lookup

- **Problem!**
  - Not atomic: lookup, compare, insert, lookup
  - Need NOVOHT supports atomicity
Compare and Swap Workflow
compare_swap API reference

- `int c_zht_compare_swap(const char *key, const char *seen_value, const char *new_value, char *value_queried)`, in C
- `int compare_swap(const string &key, const string &seen_val, const string &new_val, string &result)`
  - Return 0(zero), if `SEEN_VALUE` equals to value lookuped by the key, and set the value to `NEW_VALUE` returned
  - Return non-zero, if the above doesn’t meet, and `VALUE_QUERIED`
  - `SEEN_VALUE`: value expected to be equal to that lookuped by the key
  - `NEW_VALUE`: if equal, set value to `NEW_VALUE`
  - `VALUE_QUERIED`: if equal or not equal, get new value queried
Resource State Change Callback

- Use case
  - A controller needs to wait on specific state change before moving on
  - Inefficient when client keeps polling from the server
  - The server has a blocking state change callback operation
  - SLURM++ uses it to monitor if job's finished when job's stolen and run by other controller since there are no direct communication between controllers

- Idea: if key's value changed, notify change of client
Resource State Change Callback

- **Implementation**
  - For every call, launch *worker thread* in server
  - Block client
  - Notify client when states changed
  - *Lease-based* approach to deal with states-never-changed
  - User-defined interval to poll states
    - `SCCB_POLL_INTERVAL`
state_change_callback API reference

- int c_state_change_callback(const char *key, const char *expeded_val, int lease), in C
- int state_change_callback(const string &key, const string &expected_val, int lease), in C++

- monitor the value change of the key, block or unblock ZHT client
- EXPECDED_VAL: the value expected to be equal to what is lookuped by the key, if equal, return 0(zero), or keep polling in server-side and block ZHT client
- LEASE: the lease in milliseconds after which ZHT client will be unblocked.
Thread Safe

- **Operation Level**
  - Insert, lookup, append, remove, compare_swap, state_change_callback, all *shared a single mutex*
  - Performance killer

- **Socket Level**
  - *Distinct mutex* attached to every socket connection
  - Network related concurrency issues come from shared socket over which send/receive overlapped
lock_guard class

Constructor

lock_guard(pthread_mutex_t *mutex) { lock(mutex); }

Destructor

~lock_guard() { unlock(mutex); }

Even if ZHT client crashed, Destructor will always be called, and release the lock
**Small-Job Workload**

- For N nodes, submit N jobs, e.g., 50 jobs submitted for 50 nodes scale
- Each job requiring just 1 node, MTC job
- Each job runs 1 task (sleep 0)
Medium-Job Workload

- For N nodes, submit N jobs, e.g., 50 jobs submitted for 50 nodes scale
- Each job requiring a random (1~50) number of nodes, HPC job
- Each job runs 1 task (sleep 0)
**Large-Job Workload**

- For every scale (100, 150, 200, 250, 300, 350), submit (#scale * 20) jobs, e.g., 20 jobs submitted for 100 nodes scale; 40 jobs submitted for 150 nodes scale; 60 jobs submitted for 200 nodes scale;
- Each job requiring a random (25~75) number of nodes, HPC job
- Each job runs 1 task (sleep 0)
SLURM vs. SLURM++

Small-Job Workload

Each controller manages 50 nodes
Each controller launches 50 jobs,
MTC job
Each job requiring 1 node
Each job runs 1 task (sleep 0)

#nodes/50 = #controllers
SLURM vs. SLURM++

Each controller manages 50 nodes
Each controller launches 50 jobs,
HPC job
Each job requiring a random
(1~50) number of nodes
Each job runs 1 task (sleep 0)

#nodes/50 = #controller
SLURM vs. SLURM++

Each controller manages 50 nodes
Each controller launches 20 jobs,
HPC job
Each job requiring a random
(25~75) number of nodes
Each job runs 1 task (sleep 0)

#nodes/50 = #controller

Large-Job Workload
SLURM vs. SLURM++

Small-Job; ZHT message count of SLURM++
SLURM vs. SLURM++

Medium-Job; ZHT message count of SLURM++
SLURM vs. SLURM++

Large-Job; ZHT message count of SLURM++
SLURM vs. SLURM++

Throughput comparison with different workloads
Distributed Monitoring – ZHT Approach
Distributed Monitoring – AMQP Approach
Distributed Monitoring – AMQP Approach

- Federation is used to provide geographical distribution of brokers. A number of individual brokers, or clusters of brokers, can be federated together. This allows client machines to see and interact with the federation as though it were a single broker. Federation can also be used where client machines need to remain on a local network, even though their messages have to be routed out.
Cache
Libnap Standalone Library

- Libnap: Library for Network Abstracted Protocols
- For new version MATRIX development
Thank you!

Q&A