

Slurm++: a Distributed Workload Manager for Extreme-Scale High-Performance Computing Systems

Ke Wang

Data-Intensive Distributed Systems Laboratory
Computer Science Department
Illinois Institute of Technology

CS554: Data-Intensive Computing, IIT
February 9th, 2015

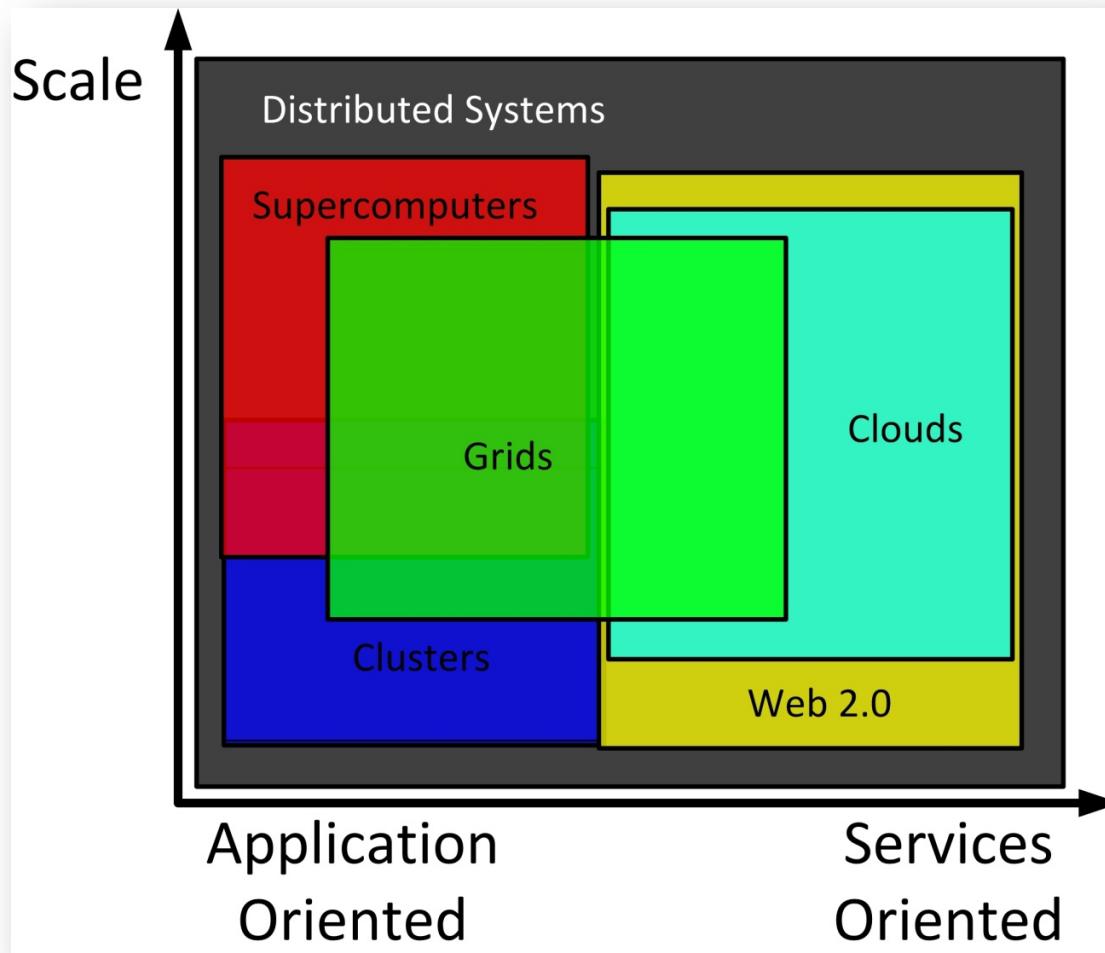
Outline

- **Introduction & Motivation**
- **Problem Statement**
- **Proposed Work**
- **Evaluation**
- **Conclusions**
- **Future Work**

Outline

- **Introduction & Motivation**
- Problem Statement
- Proposed Work
- Evaluation
- Conclusions
- Future Work

Distributed Systems



Exascale Computing

PERFORMANCE DEVELOPMENT

PROJECTED



2019

http://s.top500.org/static/lists/2014/06/TOP500_201406_Poster.pdf

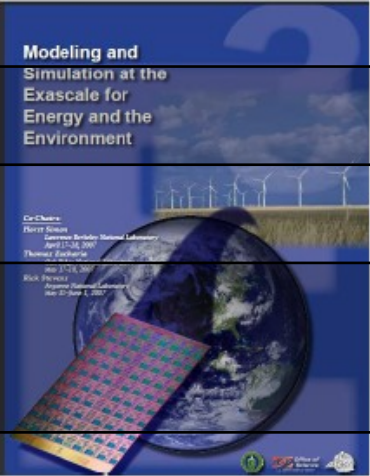

<http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf>

Slurm++: a Distributed Workload Manager for Extreme-Scale High-Performance Computing Systems

Resource Manager

- Manages resources
 - compute
 - storage
 - network
- Job scheduling
 - resource allocation
 - job launch
- Data management
 - data movement
 - caching

Motivation

1970s	<p>Medical Image Processing: Functional Magnetic Resonance</p> <p>Ensemble simulations are important for future exascale platforms</p>	
1980s	<p>Chemistry Domain: MolDyn</p> <p>Molecular Dynamics: DOCK</p> <p><i>One approach to dealing with uncertainty is to perform multiple ensemble runs (parameter sweeps) with various combinations of the Run in parallel. In the space of parameters will be of high dimension, we will have to address the challenges of designing efficient parameter sweep methods for high-dimensional spaces. Recent advances in approximation theory and data mining methods, such as sparse grids, offer new approaches to this problem. Furthermore, recent results in approximation theory can be used to guide us in using exascale computing power to search for efficient methods.</i></p>	
1990s	<p>Production Runs in Drug Design</p> <p>Economic Modeling: MARS</p>	
2000s	<p>Large-scale Astronomy Application Evaluation</p> <p>Astronomy Domain: Montage</p>	
2010s	<p>Data Analytics: Sort and WordCount</p> <p><i>(Source: Horst, Simon et al. Modeling and Simulation at the Exascale for Energy and the Environment)</i></p>	

Outline

- Introduction & Motivation
- **Problem Statement**
- Proposed Work
- Evaluation
- Conclusions
- Future Work

Problem Statement

Job Scheduling System Challenges

- **Scalability**
 - System scale is increasing
 - Workload size is increasing
 - Processing capacity needs to increase
- **Efficiency**
 - Allocating resources fast
 - Making fast enough scheduling decisions
 - Maintaining a high system utilization
- **Reliability**
 - Still functioning well under failures

Outline

- Introduction & Motivation
- Problem Statement
- **Proposed Work**
- Evaluation
- Conclusions
- Future Work

Job Scheduling Systems

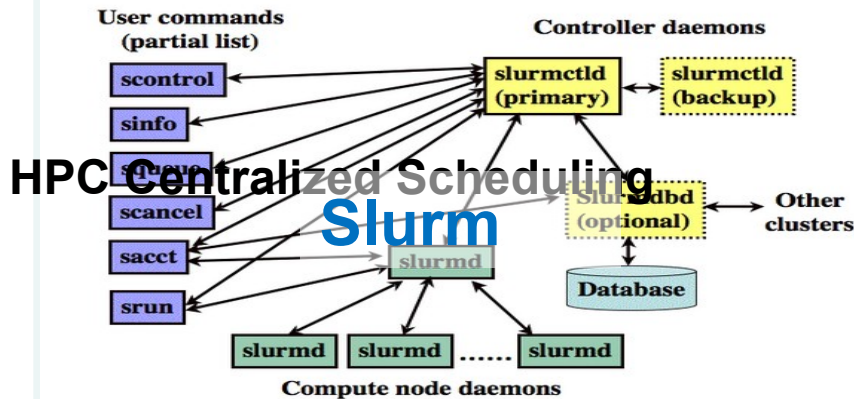
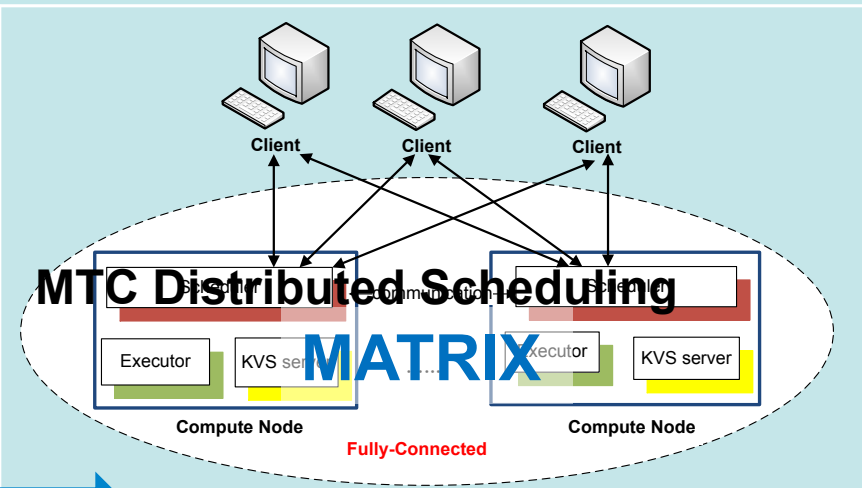
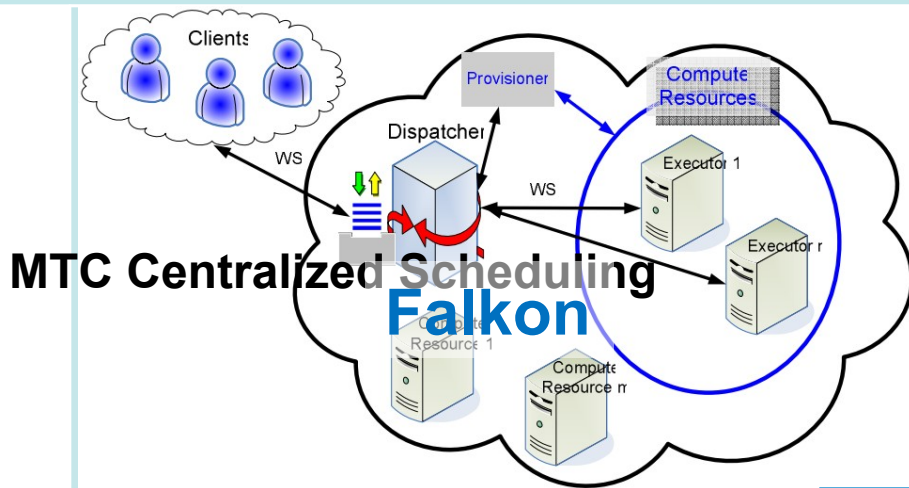
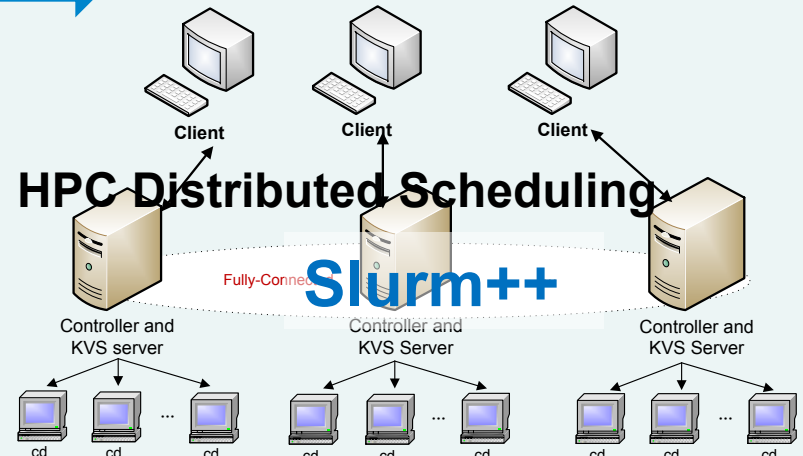


Figure 1. SLURM components



Slurm++: a Distributed Workload Manager for Extreme-Scale High-Performance Computing Systems

Slurm++ Architecture

Slurm workload manager

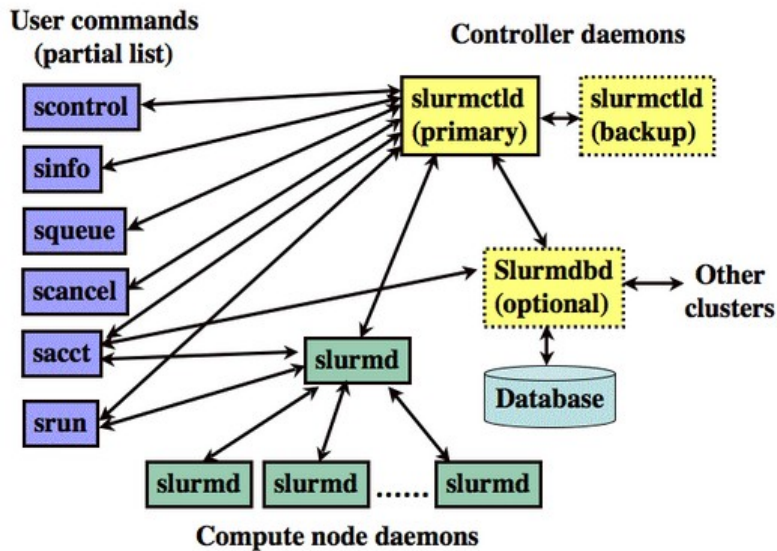
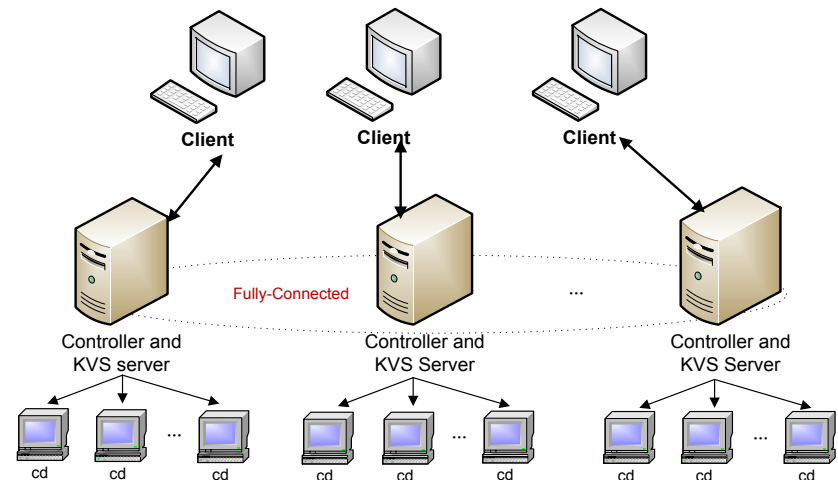


Figure 1. SLURM components

Slurm++: extension to Slurm



- Distributed
- Partition-based
- Resource sharing
- Distributed metadata management

Job Launching Procedure

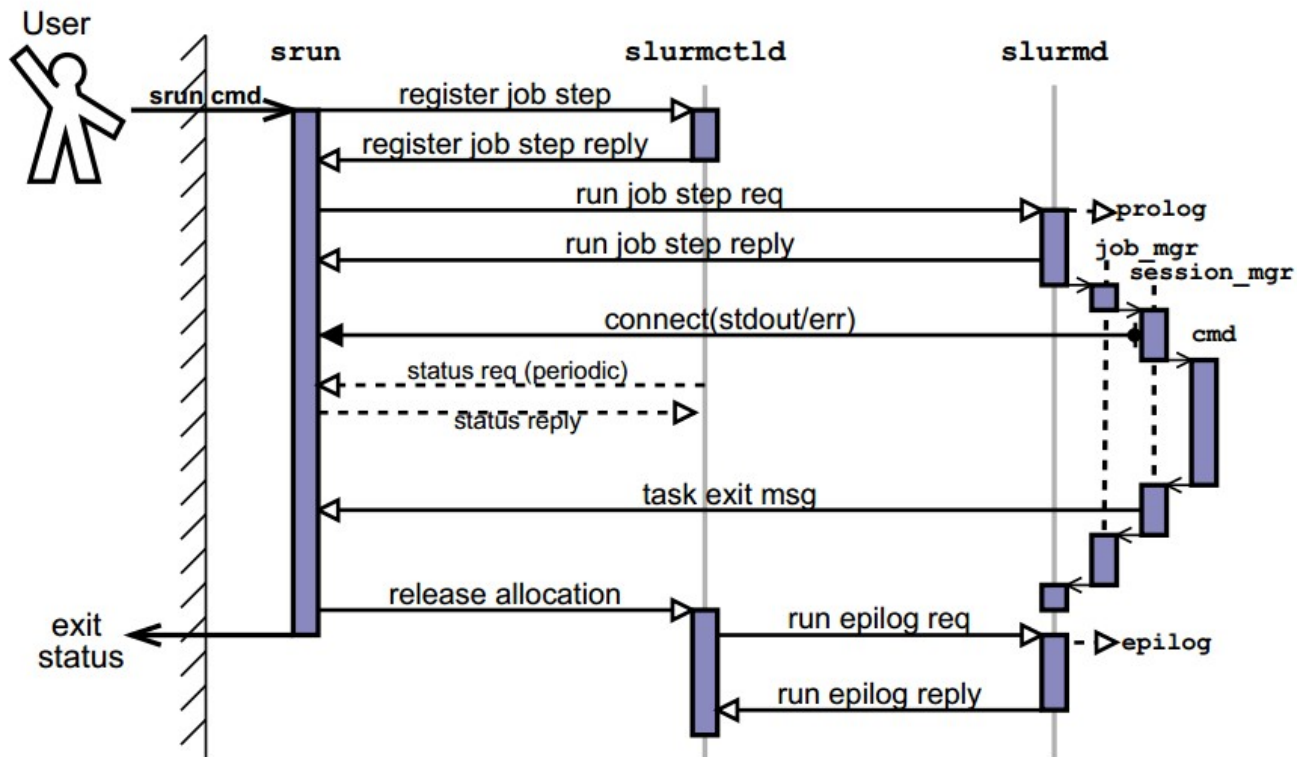


Fig. 5. Interactive job initiation. `srun` simultaneously allocates nodes and a job step from `slurmctld` then sends a run request to all `slurmds` in job. Dashed arrows indicate a periodic request that may or may not occur during the lifetime of the job

Distributed Metadata Management

Key	Value	Description
controller id	free node list	Free nodes in a partition
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 each partition that are involved in launching a job

Dynamic Resource Balancing

- **Balancing resources among all the partitions**
- **Trivial for centralized architecture**
- **Challenging for distributed architecture**
 - No global state information
 - Need to be achieved dynamically
 - Resource conflict may happen
 - Distributed resource stealing

Resource Conflict

- **Different controllers may modify the same resource concurrently**
- **Resolved through atomic operation**

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*) of a ZHT server.

Output: A Boolean indicates success (*TRUE*) or failure (*FALSE*), and the current actual value (*current_value*).

```
1  current_value = map.get(key);
2  if (current_value == seen_value) then
3      map.put(key, new_value);
4      return TRUE;
5  else
6      return FALSE, current_value
7  end
```

Random Resource Stealing

ALGORITHM 2: RANDOM RESOURCE STEALING

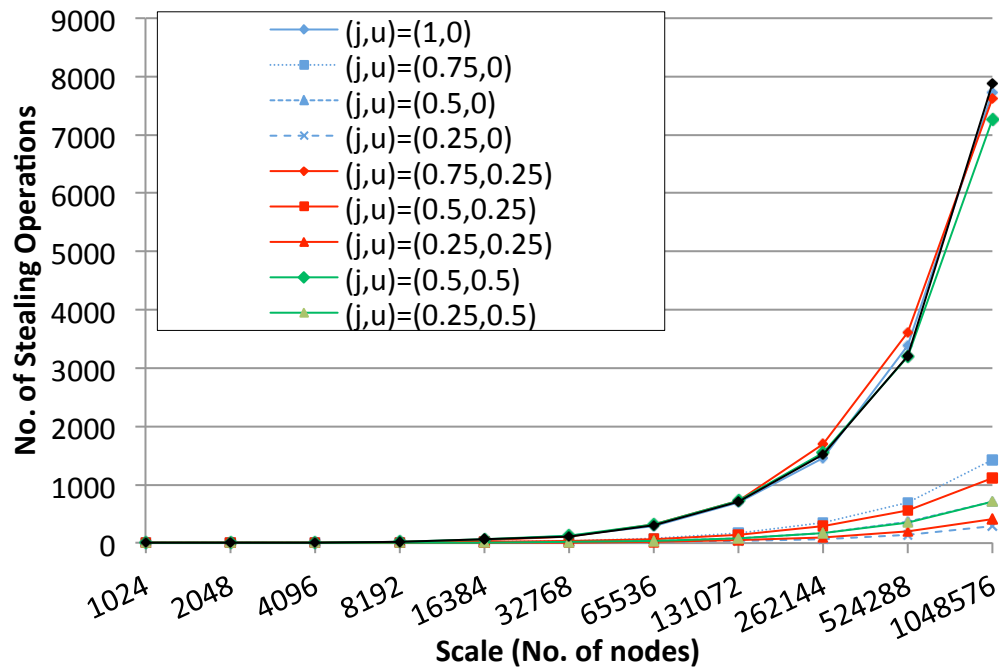
Input: number of nodes required (*num_node_req*), number of nodes allocated (**num_node_alloc*), number of controllers/partitions (*num_ctl*), controller membership list (*ctl_id[num_ctl]*), sleep length (*sleep_length*), number of retries (*num_retry*).

Output: list of involved controller ids (*list_ctl_id_inv*), participated nodes of each involved controller (*part_node[]*).

```
1  num_try = 0; num_ctl_inv = 0; default_sleep_length = sleep_length;
2  while *num_node_alloc < num_node_req do
3      sel_ctl_idx = ctl_id[Random(num_ctl)]; sel_ctl_node = zht_lookup(sel_ctl_idx);
4      if (sel_ctl_node != NULL) then
5          num_more_node = num_node_req - *num_node_alloc;
6  again:  num_free_node = sel_ctl_node.num_free_node;
7          if (num_free_node > 0) then
8              num_try = 0; sleep_length = default_sleep_length;
9              num_node_try = num_free_node > num_more_node ? num_more_node : num_free_node;
10             list_node = allocate(sel_ctl_node, num_node_try);
11             if (list_node != NULL) then
12                 *num_node_alloc += num_node_try; part_node[num_ctl_inv++] = list_node; list_ctl_id_inv.add(remote_ctl_idx);
13             else
14                 goto again;
15             end
16         else if (++num_try > num_retry) do
17             release(list_ctl_id_inv, part_node); *num_node_alloc = 0; sleep_length = default_sleep_length;
18         else
19             sleep(sleep_length); sleep_length *= 2;
20         end
21     end
22 end
23 return list_ctl_id_inv, part_node;
```

Limitations

- **Poor performance (resource deadlock)**
 - For big jobs (e.g. full-scale, half-scale)
 - Under high system utilization



- **Linearly increasing as the system scale linearly with the system scales**
- **Exponentially increasing with both the job size and utilization**
- **At 1M-node scale, a full-scale job needs about 8000 stealing operations**

Weakly-Consistent Monitoring based Resource Stealing Technique

- **Monitoring-based weakly consistent**
- **A centralized monitoring service**
 - collect all the resources of all partitions periodically (global view)
- **Controllers: Two phase tuning**
 - pull all the resources periodically (macro-tuning)
 - store the resources in a binary-search-tree (BST) (local view)
 - find the most suitable partition according to BST
 - lookup the resource of that partition
 - update the BST (micro-tuning, not consistent)

Monitoring Service

ALGORITHM 3. MONITORING SERVICE LOGIC

Input: number of controllers or partitions (*num_ctl*), controller membership list (*ctl_id[num_ctl]*), update frequency (*sleep_length*).

Output: void.

```
1  global_key = "global resource specification"; global_value = "";
2  while TRUE do
3      global_value = "";
4      for each i in 0 to num_ctl - 1; do
5          cur_ctl_res = zht_lookup(ctl_id[i]);
6          if (cur_ctl_res == NULL) then
7              exit(1);
8          else
9              global_value += cur_ctl_res;
10         end
11     end
12     zht_insert(global_key, global_value); sleep(sleep_length);
13 end
```

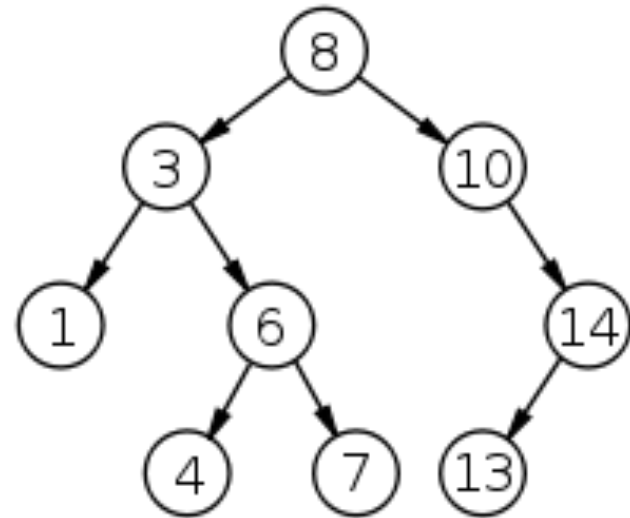
Binary-Search Tree (BST)

- **Property**

- left child \leq root \leq right child

- **Operations**

- *BST_insert*(BST*, char*, int)
- *BST_delete*(BST*, char*, int)
- *BST_delete_all*(BST*)
- *BST_search_best*(BST*, int)



Two-phase Tuning

PHASE 1: PULL-BASED MACRO-TUNING

Input: number of controllers or partitions (*num_ctl*), controller membership list (*ctl_id[num_ctl]*), update frequency (*sleep_length*), binary search tree of the number of free nodes of each partition (**bst*)

Output: void.

```
1  global_key = "global resource specification"; global_value = "";  
2  while TRUE do  
3      global_value = zht_lookup(global_key);  
4      global_res[num_ctl] = split(global_value);  
5      pthread_mutex_lock(&bst_lock);  
6      BST_delete_all(bst);  
7      for each i in 0 to num_ctl - 1; do  
8          BST_insert(bst, ctl_id[i], global_res[i].num_free_node);  
9      end  
10     pthread_mutex_unlock(&bst_lock);  
11     sleep(sleep_length);  
12 end
```

Two-phase Tuning

PHASE 2: MICRO-TUNING RESOURCE ALLOCATION

Input: number of nodes required (*num_node_req*), number of nodes allocated (**num_node_alloc*), self id (*self_id*), sleep length (*sleep_length*), number of retries (*num_retry*), binary search tree of the number of free nodes of each partition (**BST*)

Output: list of nodes allocated to the job

```
1  *num_node_alloc = 0; num_try = 0; alloc_node_list = NULL; target_ctl = self_id;
2  while *num_node_alloc < num_node_req do
3      resource = zht_lookup(target_ctl); nodelist = allocate_resource(target_ctl, resource, num_node_req - *num_node_alloc);
4      if (nodelist != NULL) then
5          num_try = 0; alloc_node_list += nodelist; *num_node_alloc += nodelist.size(); new_resource = resource - nodelist;
6          pthread_mutex_lock(&bst_lock);
7          old_resource = BST_search_exact(bst, target_ctl);
8          if (old_resource != NULL) then
9              BST_delete(bst, target_ctl, old_resource.size());
10         end
11         BST_insert(bst, target_ctl, updated_resource.size());
12         pthread_mutex_unlock(&bst_lock);
13     else if (num_try++ > num_retry) then
14         release_nodes(nodelist); alloc_node_list = NULL; *num_node_alloc = 0; num_try = 0; sleep(sleep_length);
15     end
16     if (*num_node_alloc < num_node_req) then
17         pthread_mutex_lock(&bst_lock);
18         if ((data_item = BST_search_best(bst, num_node_req - *num_node_alloc) != NULL) then
19             target_ctl = data_item.ctl_id; BST_delete(bst, target_ctl, data_item.size());
20         else
21             target_ctl = self_id;
22         end
23     end
24 end
25 return alloc_node_list;
```

Implementation Details

- **Code Available (developed in c)**
 - https://github.com/kwangiit/SLURMPP_V2
- **Code Complexity**
 - 5K lines of new code
 - Plus 500K lines of SLURM code
 - 8K lines of ZHT code
 - 1K lines of auto-generated code by Google Protocol Buffer
- **Dependencies**
 - Google Protocol Buffer
 - ZHT key value store
 - Slurm resource manager

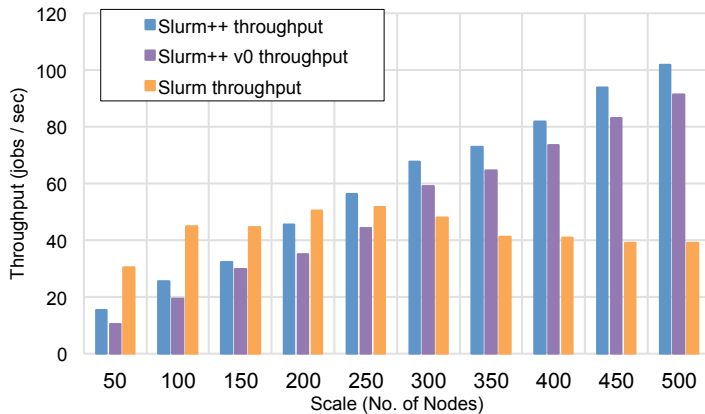
Outline

- Introduction & Motivation
- Problem Statement
- Proposed Work
- **Evaluation**
- Conclusions
- Future Work

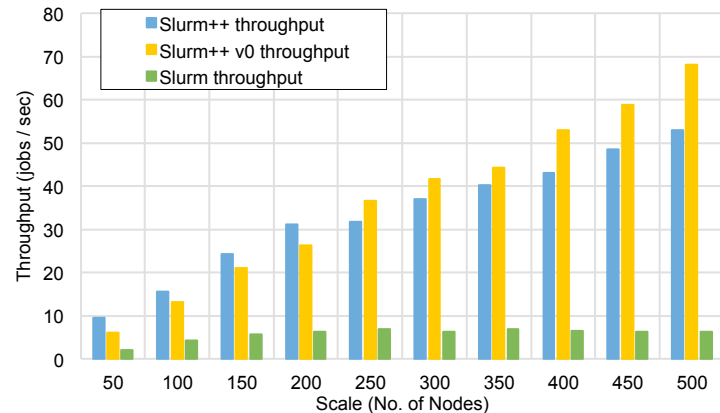
Evaluation

- **Test-bed**
 - The Probe 500-node Kodiak Cluster at LANL
 - Each node has two AMD Opteron (tm) processors with 252 (2.6GHZ), and has 8GB memory
 - The network supports both Ethernet and Infini-Band.
 - Our experiments use 10Gbits Ethernet (default configuration of Kodiak).
 - Experiments up to 500 nodes
- **Workloads**
 - Micro-benchmarking NOOP sleep jobs
 - Application traces from IBM Blue Gene/P supercomputers
 - Scientific numeric MPI applications from the PETSc tool package

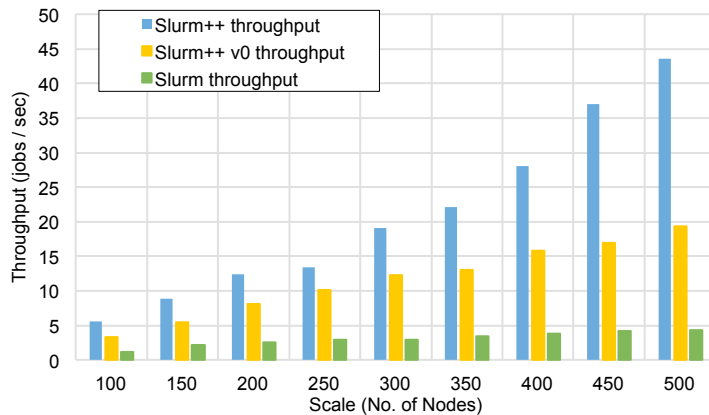
Micro-Benchmarking Workloads



One-node jobs



Half-partition jobs

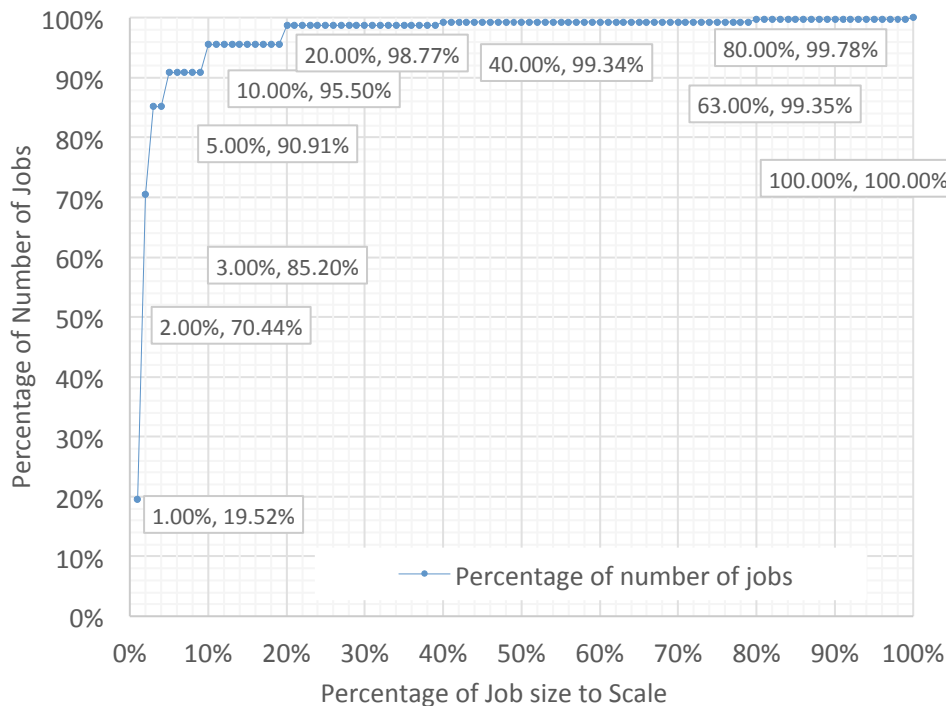


Full-partition jobs

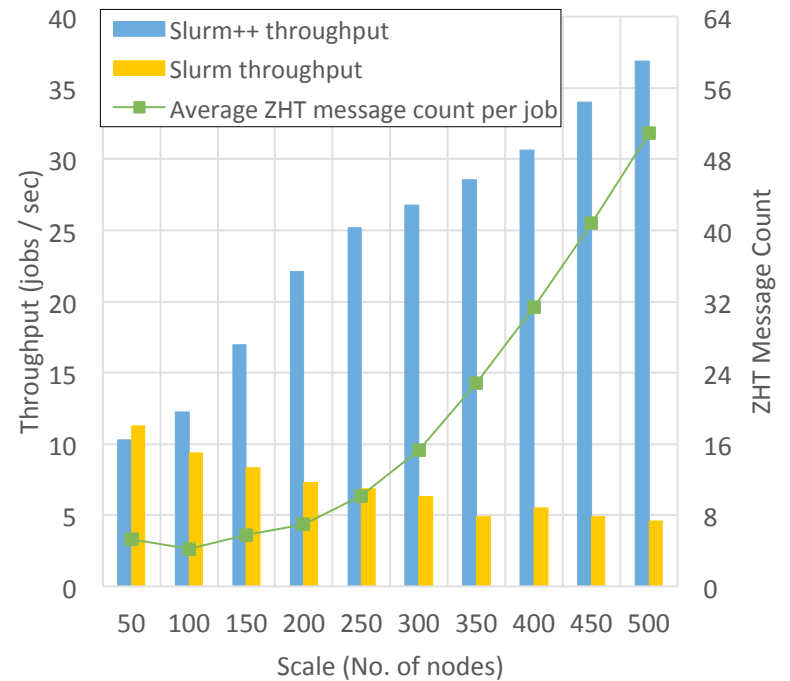


Speedup Summary

Application Traces

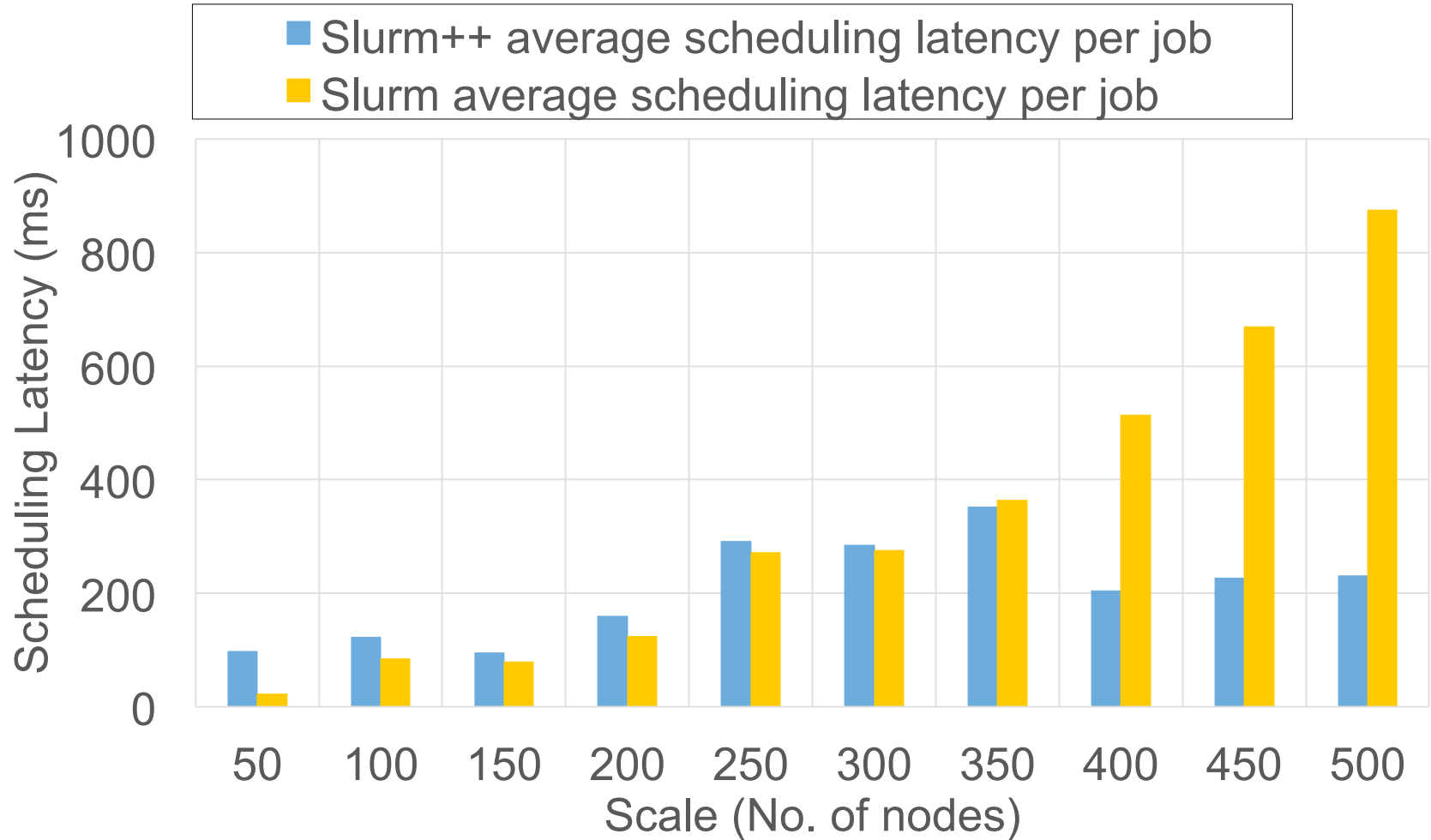


Workload Specification

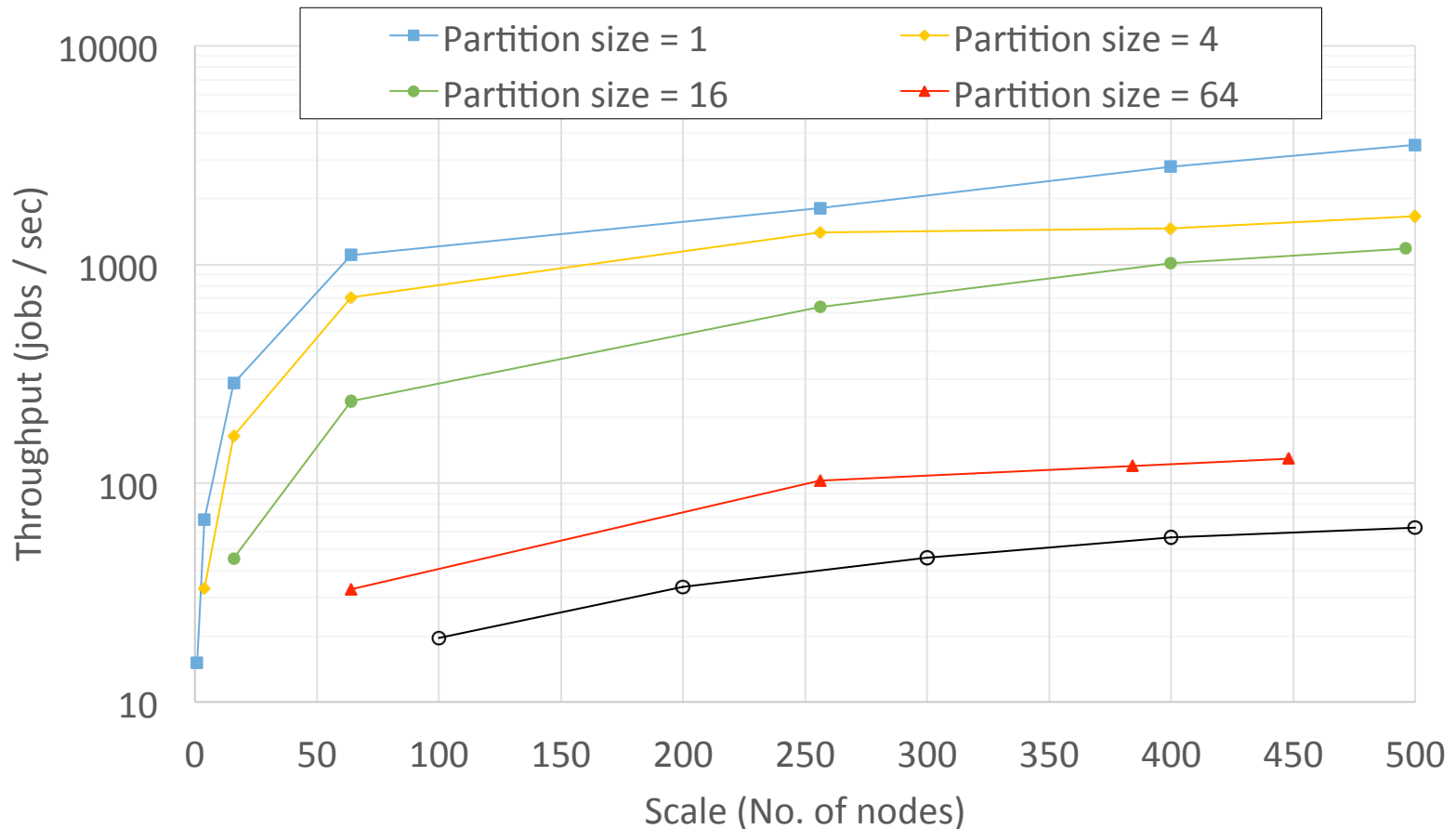


Slurm++ vs. Slurm

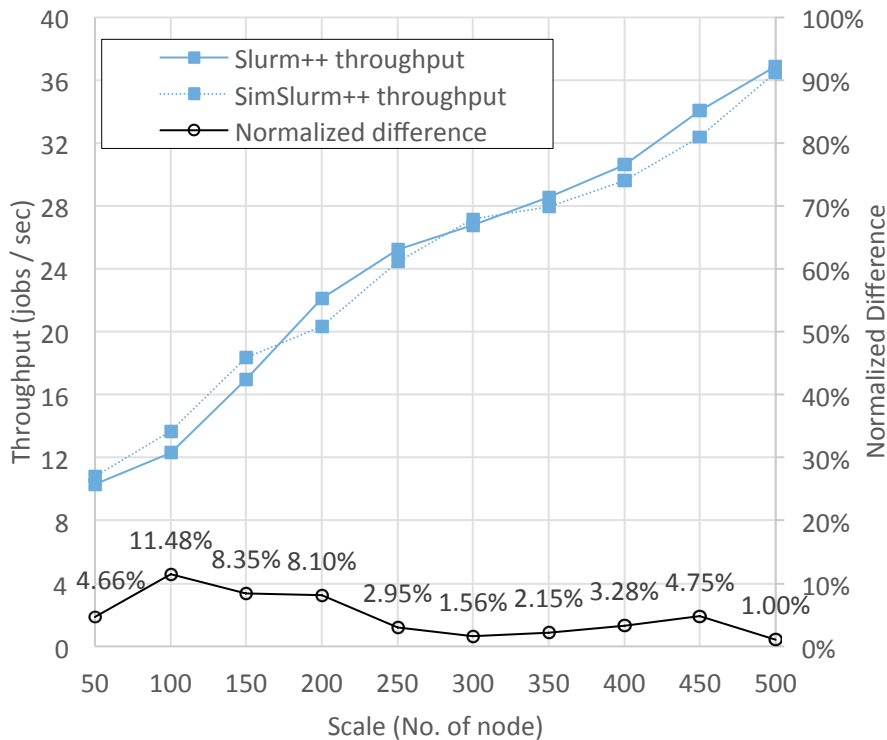
MPI Applications



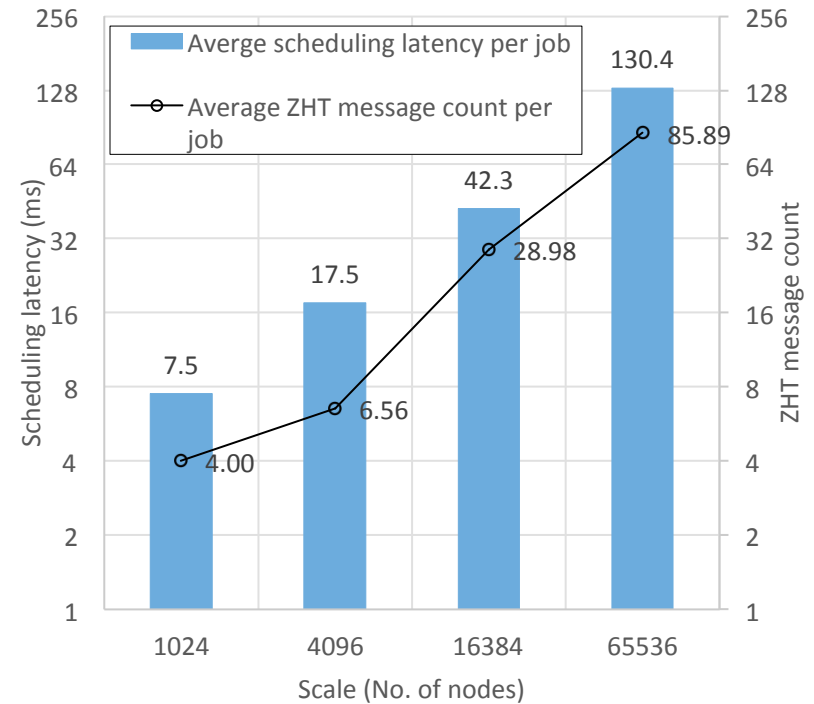
Different Partition Sizes



Exploring Scalability via Simulations



Validation



Scalability

Outline

- Introduction & Motivation
- Problem Statement
- Proposed Work
- Evaluation
- **Conclusions**
- Future Work

Conclusions

- **Workloads are becoming finer-grained**
- **Key-value store is a viable building block**
- **Distributed resource manager is demanded**
- **Fully distributed architecture is scalable**
- **Slurm++ is 10X faster in making scheduling decisions**
- **Simulation results show the technique is scalable**

Outline

- Introduction & Motivation
- Problem Statement
- Proposed Work
- Evaluation
- Conclusions
- **Future Work**

Future Work

- **Contribute to the Intel ORCM cluster manager**
 - Built on top of open-mpi
 - Make key-value store as a new Plugin
- **Distributed data-aware scheduling for HPC applications**
- **Elastic resource allocation**

More Information

- More information:
 - <http://datasys.cs.iit.edu/~kewang/>
- Contact:
 - kwang22@hawk.iit.edu
- Questions?