Towards Scalable Resource Management for Supercomputers

Yiqin Dai*, Yong Dong*[1], Kai Lu*[1], Ruibo Wang*, Wei Zhang*, Juan Chen*, Mingtian Shao*, and Zheng Wang†
* National University of Defense Technology, Changsha, China
Email: {daiyq, yongdong, kailu, ruibo, weizhang, juanchen, shaomt}@nudt.edu.cn
†University of Leeds, Leeds, United Kingdom
Email: z.wang5@leeds.ac.uk

***Abstract*—Today's supercomputers offer massive computation resources to execute a large number of user jobs. Effectively managing such large-scale hardware parallelism and workloads is essential for supercomputers. However, existing HPC resource management (RM) systems fail to capitalize on the hardware parallelism by following a centralized design used decades ago. They give poor scalability and inefficient performance on today's supercomputers, which will worsen in exascale computing. We present ESLURM, a better RM for supercomputers. As a departure from existing HPC RMs, ESLURM implements a distributed communication structure. It employs a new communication tree strategy and uses job runtime estimation to improve communications and job scheduling efficiency. ESlurm is deployed into production in a real supercomputer. We evaluate ESLURM on up to 20K nodes. Compared to state-of-the-art RM solutions, ESLURM exhibits better scalability, significantly reducing the resource usage of master nodes and improving data transfer and job scheduling efficiency by a large margin.**

***Index Terms*—Resource management, Exascale computing, Scheduling**

I. Introduction

Modern high-performance computing (HPC) systems are integrated with an ever-growing number of tightly coupled computing nodes. It is typical for a supercomputer today to have millions of parallel processes running on hundreds of thousands of nodes at any time [1]–[3]. Under such settings, efficiently managing the massive computing resources and jobs is vital for the success of any HPC system.

Hardware resource management is a heavily studied field in HPC [4], [5]. However, as being highlighted in Table I, many of today's top-ranked supercomputers still rely on a centralized resource manager (RM) like Slurm [6] and IBM LSF [7], where the RM runs on a master node to manage hardware resource allocation and batched job scheduling for the entire HPC system. Since an RM is responsible for key optimization metrics like server utilization, system throughput, job turnaround time, and fairness, it is important to make sure the RM system can efficiently scale to a large number of computing nodes and jobs.

While being widely deployed, a centralized RM is ill-suited for next-generation supercomputers and can quickly become a bottleneck on a large-scale HPC system. For example, deploying Slurm - a widely used open-source supercomputing RM system - to manage a *small-size* computing cluster with only 500 nodes running 100K jobs daily would require the master node of RM to be equipped with at least 32 to 48 GB of RAM with a multi-core CPU running at a high clock frequency [8]. Similarly, after running the production Tianhe-2A [9] supercomputing system for several years, we also observed countless scenarios where the poor performance of the centralized Slurm leads to slow job response or even system-wide crashes.

We present ESLURM[2], a distributed RM designed for large-scale HPC systems. ESLURM improves existing supercomputing RM solutions by employing a distributed RM management structure that integrates master and satellite nodes to manage resource allocation and job scheduling. This hierarchical structure minimizes the resource requirement of the master node compared to a centralized RM solution, preventing the master node from becoming a bottleneck. Such a distributed design also improves the scalability of the RM system and reduces the chance of system-wide crashes due to the failure of the RM node. To improve the robustness of distributed communication, ESLURM employs a failure prediction-based tree (FP-Tree) structure. Unlike existing tree-based communication schemes, ESLURM predicts which nodes are likely to fail in advance and uses this information to proactively adjust their positions in the communication tree to improve data transfer efficiency.

On top of the distributed design, ESLURM enhances existing HPC RMs by adopting a machine-learning-based job scheduler. Specifically, ESLURM combines unsupervised clustering and supervised learning techniques to estimate the runtime of a given job based on historic information of similar jobs. ESLURM then uses the runtime estimation to schedule user jobs. This strategy avoids the pitfall of inaccurate job completion time supplied by the user, improving the system utilization while reducing the job failure rate.

The design of ESLURM draws aspiration from distributed solutions of resource and task management developed for data centers [10]–[12]. However, these data center oriented solutions target containerized workloads and microservices with a focus on co-locating tasks to improve server utiliza-

[1] Corresponding author

[2] Code available at: https://github.com/YiqinDai/eslurm.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

TABLE I
RESOURCE MANAGERS OF TOP-10 SUPERCOMPUTERS AS OF NOV. 2021

Rank	System	RM	Rank	System	RM
1	Fugaku	Fujitsu	6	Selene	Slurm
2	Summit	LSF	7	Tianhe-2A	Slurm
3	Sierra	LSF	8	JUWELS	Slurm
4	Sunway Taihulight	LSF	9	HPC5	*unknown*
5	Perlmutter	Slurm	10	Frontera	Slurm

tion [13]–[15]. They are not designed for distributed, batch-oriented HPC workloads where jobs typically run in isolation on individual nodes (but use e.g. the message-passing interface (MPI) for communication and synchronization) and are sensitive to timeliness and affinity. Due to the distinct characteristics and requirements of workloads, existing data center RMs are infeasible for supercomputing.

We have implemented ESLURM and deployed it to the production environment of the Next Generation Tianhe Supercomputer. We compare ESLURM against five mainstream HPC RMs: Slurm [6], LSF [7], SGE [16], Torque [17], and OpenPBS [18] using 16,384 nodes on the top-ranked Tianhe-2A supercomputer and 20K+ nodes on the Next Generation Tianhe Supercomputer (NG-Tianhe). Experimental results show that ESLURM incurs lower CPU load, memory footprint, and network bandwidth compared to alternative schemes. Compared to the widely used Slurm RM, ESLURM significantly improves system utilization and reduces the average job waiting time, exhibiting better scalability as the number of computing nodes to be managed increases.

This paper makes the following contributions:

- It presents a distributed RM implementation for supercomputers and HPC systems (Section III);
- It shows, for the first time, how node failure prediction can be employed to improve distributed communications (Section IV);
- It proposes a novel machine-learning-based runtime job scheduler for HPC job scheduling (Section V);
- It shares the experience of developing a decentralized RM in the production of the NG-Tianhe Supercomputer.

II. BACKGROUND AND MOTIVATION

A. HPC Resource Managers

The majority of supercomputer RMs follow a centralized master-slave structure. For example, Slurm runs a control (`slurmctld`) daemon on a master node to manage and schedule hardware resources among parallel jobs, and a lightweight service daemon (`slurmd`) on each computing node for tasks like launching and terminating processes and redirecting I/O requests. A similar master-slave architecture is also be used by RMs within the PBS family [17], [18], LSF [7], SGE [16] and Condor [19].

The centralized RM design is widely used by today's supercomputers for resource management and job scheduling. Table I lists the RMs used by the top 10 supercomputers in the TOP500 list (published in November 2021) [20]. All but HPC5 (whose RM was not disclosed) of the top-10 supercomputers use a master-slave architecture built upon Slurm or LSF. While HPC systems are designed to provide hardware parallelism, ironically, the current mainstream HPC RMs do not capitalize on hardware parallelism.

B. Observations in a Production Environment

Our trial deployment of Slurm (v20.11.7) on the Next Generation Tianhe Supercomputer with 20K+ nodes shows that Slurm can not effectively manage a cluster of this size. For example, the RAM usage of the main scheduling daemon (i.e., `slurmctld`) on the master node quickly increased to 70 GB in a week, which continued to grow with a longer running time. We also observed that the CPU of the master node was fully loaded most of the time, and the number of concurrent TCP connections could reach hundreds of thousands. Due to the excessive resource usage and communication links on the control node, the RM cannot timely respond to user job requests, giving an average response time of more than 27 seconds for a user request, with around 38% of user requests failing to connect to the master node at a given time. We also observed the Slurm crashed numerous times in scenarios with bursts of communication, when many jobs were submitted and terminated at the same time, or when a large number of computing nodes failed. The average time between two Slurm RM crashes on our system is around 42 hours, but an RM reboot takes more than 90 minutes. Worse still, the system-wide resource utilization of our system is under 30%.

As can be seen from the observations, a single control node does not fit for managing a large HPC cluster. With the explosive growth of HPC system resources, a centralized RM is likely to become a major performance bottleneck. It is, therefore, a massive missed opportunity to not utilize the hardware parallelism provided by supercomputers. ESLURM is designed to avoid the pitfall of a centralized RM.

C. Design Choices

There are two main approaches to address the scalability issue. The first is to develop a fully decentralized system. In the context of HPC RMs, this can be achieved by replicating multiple master nodes with similar functionalities, where each master node manages a subset of jobs and computing nodes. This strategy is shown to be useful in decentralized or distributed networks [21]–[23] and certain HPC job scheduling scenarios [24]. The second approach is to introduce intermediate control layers to build a hierarchical system to avoid a single node becoming the bottleneck [25], [26]. ESLURM falls into this category.

In a fully distributed architecture, the functionalities of the traditional master node are partitioned across distributed nodes. Doing so requires a redesign of resource allocation and job scheduling policies across multiple control nodes. In addition, the synchronization overhead of multiple fully decentralized control nodes can incur significant overhead in a fully distributed architecture. In contrast, ESLURM's hierarchical design retains a master node and only offloads large-scale communication from the master node to a layer

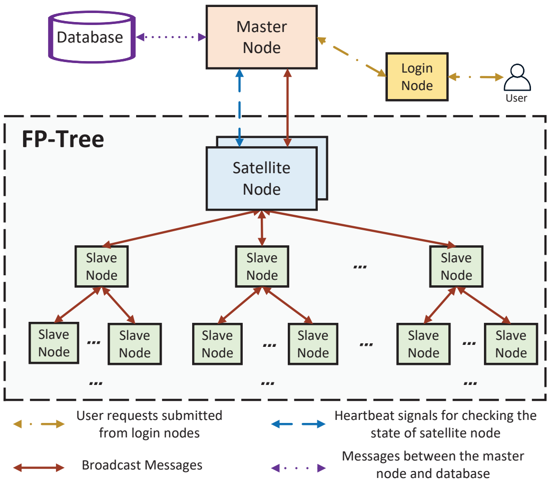
Database
Master Node
Login Node
User
FP-Tree
Satellite Node
Slave Node
...
User requests submitted from login nodes
Heartbeat signals for checking the state of satellite node
Broadcast Messages
Messages between the master node and database

Fig. 1. Overview of the ESLURM's distributed RM architecture.

of auxiliary (or satellite) nodes. This approach preserves the master node's global view of resources and jobs as well as the original efficient resource allocation and job scheduling logic, ensuring the global optimality of system scheduling. We choose a hierarchical design to implement ESLURM, because it provides an easy way to upgrade existing supercomputer RMs, supporting practices that are widely adopted in supercomputer job management today.

III. Distributed RM Architecture of ESLURM

A. Overview of ESLURM

Fig. 1 gives a high-level overview of ESLURM that extends the classical master-slave architecture by introducing the intermediate satellite nodes. The master node coordinates the system-wide resource and job scheduling by only interacting with the satellite nodes. The key is to distribute the extensive communications of computing nodes (i.e., slave nodes) to the satellite nodes to reduce the communication traffic of the master node. The satellite nodes do not participate in computing tasks and do not retain any system state. They act as bidirectional communication buffers with initial data aggregation and processing capabilities between the master node and the computing nodes. This design principle is based on the observation that communication and synchronization between the control node and the computing nodes are responsible for poor scalability (see Section VII) in large-scale HPC systems.

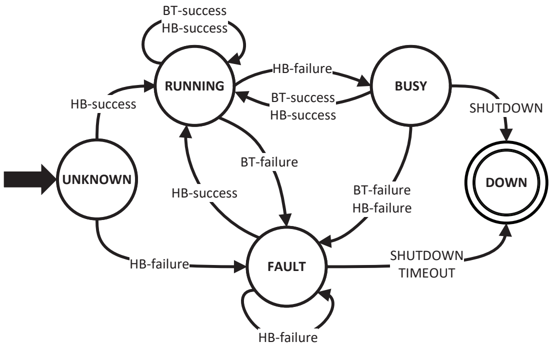

Fig. 2. The state transition of a ESLURM satellite node.

TABLE II
ESLURM SATELLITE NODE STATUS

State/Event	Description
UNKNOWN	Satellite node state remains unknown
RUNNING	Satellite node is operating as expected
BUSY	Satellite node is processing broadcast tasks
FAULT	Satellite node has failed.
DOWN	Satellite node is shut down.
BT-success	Satellite node successfully processed a broadcast task.
BT-failure	Satellite node failed to process a broadcast task.
HB-success	Satellite node is health.
HB-failure	Satellite node is abnormal.
SHUTDOWN	A shutdown command is sent to the satellite node
TIMEOUT	Satellite node is in the FAULT state for a long time (e.g., $\geq$20 min)

In a centralized RM, the master node uses broadcast messages to communicate directly with all computing nodes for tasks like launching or terminating jobs and sending heartbeat signals to slave nodes for fault detection. After the broadcast, the responses from slave nodes will typically need to be collected and aggregated for the master node. ESLURM takes a different approach. Within ESLURM, the master node only needs to communicate with a smaller number of satellite nodes through broadcasting. The satellite nodes then relay the messages to slave nodes. This is achieved by partitioning and organizing the slave nodes as an FP-tree to be managed by the satellite node. Similarly, the satellite node can aggregate messages from slave nodes within its FP-tree and send the aggregated message to the master node. We note that the FP-tree is dynamically constructed based on the communications and the prediction of failed computing nodes. This is discussed in more details in Section IV.

By distributing the communications across multiple satellite nodes, ESLURM reduces the chance for the master node to become a communication bottleneck when managing a large number of computing nodes. This design, in turn, also reduces the computation resource requirements of the master node, increasing the scalability of the RM. A key challenge of ESLURM is ensuring load balancing and fault tolerance of satellite nodes. We achieve this by employing the dynamic task allocation and failure detection mechanisms described in the next two subsections.

B. Dynamic Satellite Node Allocation

When the master node issues a broadcast message to s participating (or slave) nodes, ESLURM dynamically splits the participation list into N sub-lists handled by N satellite nodes (with the same message content). By doing so, the master node only needs to interact with N satellite nodes (where $N \ll s$), reducing the master node communication traffic.

ESLURM uses the following analytical model to determine the number of (N) satellite nodes used to relay a broadcasting message to s slave nodes:

$$N = \begin{cases} 1, & s <= w \\ s/w, & w < s < m * w \\ m, & s >= m * w \end{cases} \tag{1}$$

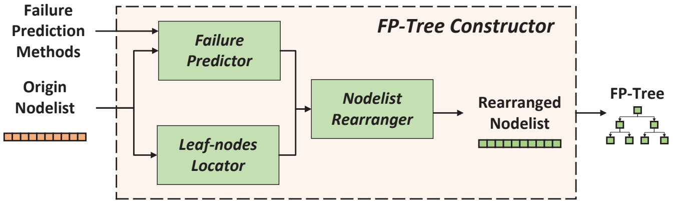

Fig. 3. Workflow of the ESLURM *FP-Tree Constructor*.

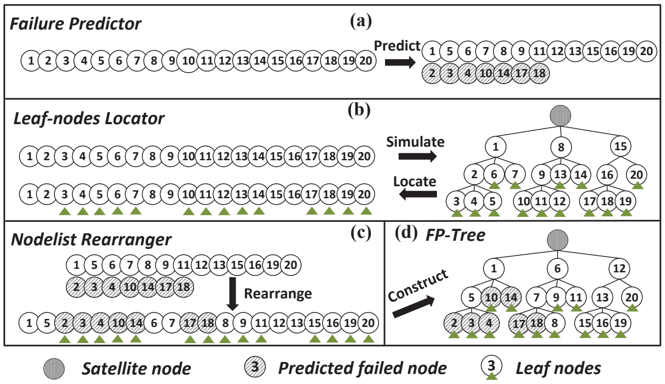

Fig. 4. Workflow of each component in the ESLURM *FP-Tree Constructor*.

where w is the width of the FP communication tree, and m is the number of all satellite nodes configured in the cluster (see Section VII-C for our default setting). Essentially, this formula tries to avoid using all satellite nodes unless a large number of slave nodes is involved. After obtaining N, ESLURM equally divides the list of participating nodes across N satellite nodes to create N sub communication tasks. We use round-robin [27] to map nodes from the satellite node pool to the slave node partition. The master node performs this mapping to assign satellite nodes to slave nodes.

C. Failure Detection of Satellite Nodes

In a large computing cluster, some of the nodes may fail from time to time. Such failure can happen to satellite nodes too. Fig. 2 and Table II describe how ESLURM detects and recovers the failures of satellite nodes. ESLURM checks and updates the status of each satellite node of the satellite node pool. Failures can be detected by either checking a satellite node has processed a broadcast task successfully (e.g., BT-success and BT-failure in Table II) or sending heartbeat signals at regular intervals (HB-success and HB-failure in Table II). Only satellite nodes at the RUNNING state will be chosen to participate in message broadcasting, and satellite nodes with the DOWN state will require administrator intervention.

Although only satellite nodes in the RUNNING state are allowed to participate in message broadcasting, it is still possible for satellite nodes to fail during data broadcasting. At this time, ESLURM reallocates the broadcast to the next satellite nodes in the round-robin and sets the failed satellite node to the FAULT state. However, if the number of reallocation trails for the same task exceeds a threshold (default to 2 in ESLURM), the master node will take over the broadcast task, ensuring that the task is processed correctly and promptly.

IV. FAILURE PREDICTION BASED TREE STRUCTURE

Like other HPC RMs [28]–[30], ESLURM uses a logical tree to disseminate messages across computing nodes. Prior works in the area focus on tuning the width and depth of the tree but largely ignore the impact of node failures on the communication latency.

A node failure can be caused by a power outage, network disconnection, network congestion, and memory shortages, all of which lead to communication failures between nodes. Failed computing nodes of the communication tree incur communication latency. This is partly because the parent node must wait for a timeout threshold before taking action, and in particular, failures on non-leaf nodes cause communication latency in all their descendant nodes. In addition, once a non-leaf node fails, the parent node also needs to redesign its communication according to a fault tolerance mechanism to ensure that the descendant nodes of the failed node are reachable, a process that is also time-consuming. Therefore, the more descendant nodes of a failed node have, the higher the delay will be caused by a single point of failure. In the production environment of Tianhe-2A, we track the communication processes (e.g., broadcasting and heartbeats) that use trees. We found that for every 1% increase in the number of failed nodes, the latency of a tree-based communication structure will increase by 3-28 seconds. These problems will manifest more frequently when future HPC systems integrate more computing nodes where the frequency of node failures [31] and the probability of having a failure node in the communication tree will increase [32], [33]. To reduce the impact of a failed node, we would like to move a node that is like to fail further down to the communication tree. The ESLURM communication tree is designed for this purpose.

A. The ESLURM Communication Tree

Fig. 1 shows the failure prediction based tree structure (FP-Tree) of ESLURM. Upon receiving a communication task, the satellite node first constructs an FP-Tree (via the *FP-Tree Constructor* shown in Fig. 3) containing the satellite node and all its slave nodes specified within the broadcast task. The key of the FP-Tree is to estimate which nodes are most likely to fail so that these nodes can be placed as leaf nodes to reduce the impact of failures on the communication latency, improving the response time and system throughput.

B. Communication Tree Construction

To construct a communication tree, each satellite node breaks the list of participating slave nodes into small groups. Note that the number of groups determines the width of the tree. Each satellite node uses the first node in the partitioned list as the first-layer node. Then, the satellite node sends each remaining list with the broadcast message to its corresponding first-layer node. Each first-layer node groups the remaining node list into several small node lists before selecting the second-layer nodes and so on. If all nodes use the same grouping method in the above process, the node's location in the initial node list received by the satellite node corresponds to its location in the tree. Therefore, rearranging the node list before constructing a tree can change the location of nodes in the tree. An effective rearranging strategy has the opportunity to deliver significant optimization results for the tree-like communication mode. Fig. 4 depicts the workflow of FP-Tree construction. Given that tree construction occurs frequently and that the size of the node list is usually large in large-scale systems, we expect the additional time cost of constructing an FP-Tree to be $O(n)$, where n is the number of nodes in the node list.

C. Failure Node Prediction

ESLURM leverages monitoring infrastructures commonly available in most HPC systems to identify nodes with abnormal behaviors as possible failures. On the Tianhe HPC systems, we adopt the principle of over-prediction. This is because the prediction result only changes the node's position in the communication tree and does not affect the state and performance of the node. The corresponding node is predicted as a failed node once an alert is received from the monitoring and diagnostic subsystem. The monitoring and diagnostic subsystem of Tianhe HPC systems consist of three layers of management units, the Board management Unit (BMU), including the Chassis Management Unit (CMU), and the System Management Unit (SMU), which are connected in a unified way through a dedicated monitoring and diagnostic network [34]. The subsystem has more than 200 hardware monitoring indicators, covering voltage, current, temperature, humidity, liquid cooling system, air cooling system, self-developed high-speed network card, and many other aspects.As the failure node prediction mechanism is implemented as a plugin, more advanced techniques can be easily integrated with ESLURM [31], [35], [36].

D. Leaf-nodes Location

ESLURM first simulates the entire construction process of the communication tree and then locates the corresponding locations of the leaf nodes in the nodelist (Fig. 4 (b)). The most important and complex step in the process is to simulate the grouping process of each node recursively from top to bottom by the divide and conquer method. For each node, if the number of nodes received in the node list is greater than the treewidth, it is first divided into w groups, and then the recursion continues for each group. If the number of nodes received is less than the treewidth, it is directly divided into n groups, and the complexity of the process is $O(1)$. The following equation can describe the complexity of the recursive process:

$$T(n) = \begin{cases} O(1), & n < w \\ w * T(n/w) + w, & n \geq w \end{cases} \qquad (2)$$

where w is the width of the tree. Using the master theorem [37], we can quickly get the time complexity of the recursive formula as $T(n) = \Theta(n)$.

TABLE III
WORKLOAD TRACES

Traces	#Jobs	Time Period
Tianhe-2A	154,081	June/2021-Sep/2021
NG-Tianhe	52,162	Oct/2021-Mar/2022

E. Nodelist Rearranging

The ESLURM nodelist rearranger uses the results of the other two components to rearrange the original input nodelist. This tool traverses each location on the original nodelist and selects a proper node for filling the location. If a location corresponds to a leaf node, the tool prioritizes the selection of a node from the set of predicted failed nodes; otherwise, it preferentially selects a node from the complement of the failed nodes set. The time complexity of this step is $O(n)$ for n computing nodes. The rearranged nodelist will be used to construct an FP-Tree. Fig. 4 (c) visually presents this process, and Fig. 4 (d) shows the corresponding FP-Tree, in which the predicted failed nodes are placed on the leaf nodes.

When there are few failed nodes, the FP-Tree makes few changes to the communication structure. In practical deployments, we found that in most cases, failed nodes account for less than 2% of the total number of nodes (see Section VII-A). Therefore, for systems that use topological information to optimize communication, the communication tree can be constructed first using topology-aware techniques and then fine-tuned using the FP-Tree constructor. This approach can reduce the impact of failed nodes while preserving the topology-aware properties of the tree.

V. JOB RUNTIME ESTIMATE FRAMEWORK

Having a good estimation of the job runtime is critical for effective job scheduling. Existing HPC RM schedulers rely on runtime estimation given by the users. However, studies have shown that users tend to overestimate their job running time [38]–[40]. Our analysis on the workload trace of over 200K jobs (Table III) from two production HPC systems suggests that runtime overestimation is a common issue. For example, Fig. 5 (a) gives the cumulative distribution of the runtime estimated accuracy (P) computed from these real-life work traces, where $P > 1.0$ suggests an overestimation. As can be seen from the diagram, around 80-90% of the job runtime were overestimated by users. ESLURM is designed to improve the efficiency of job scheduling planning by utilizing more accurate job runtime estimation.

Fig. 6 depicts the job runtime estimation framework of ESLURM, which consists of three components: an estimation model generator, a real-time estimation module, and a record module, described in the following subsections.

A. Estimation Model Generator

Our estimation model generator periodically selects historical jobs within a configurable interest window from the historical job queue. Next, it applies unsupervised clustering to the selected jobs to reduce the required training dataset size

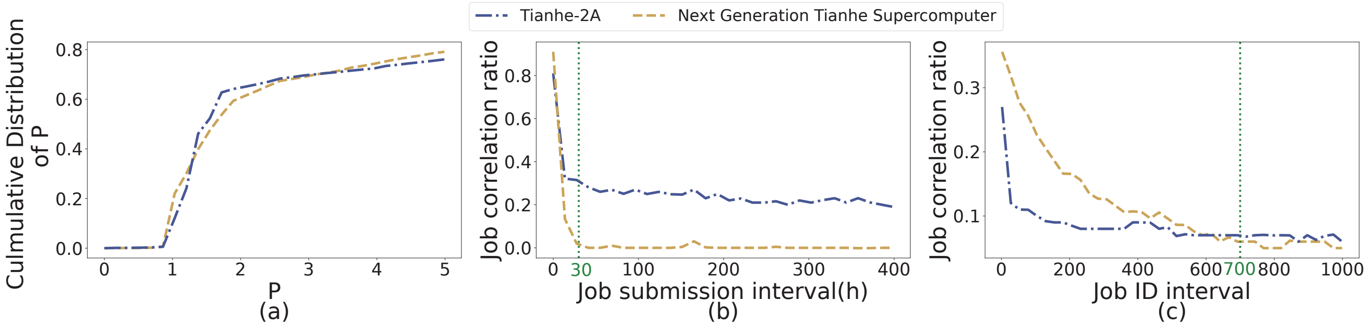

Fig. 5. Figure a shows the cumulative distribution of the runtime estimation accuracy, P, computed as t_s/t_r, where t_s is the user-submitted job runtime estimation and t_r is the actual job runtime. Note that $P > 1.0$ suggests an overestimation of job runtime. Figures b and c show the variation of job correlation ratio with the job submission interval and the difference of job ID.

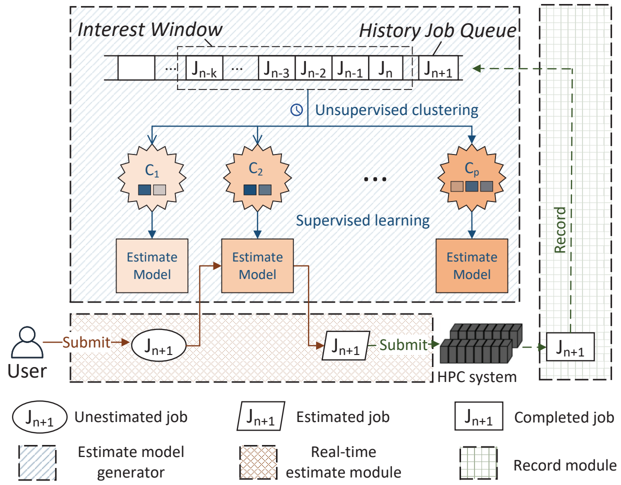

Fig. 6. Workflow of the ESLURM job runtime estimate framework.

while improving the accuracy of the estimation model. It then creates a job estimation model for each cluster. Specifically, we use K-means++ [41], for clustering and a support vector machine (SVM) model for regression (SVR) for runtime estimation.

Observations and design choices. We use the workload traces presented in Table III to study the locality of job runtime by considering two metrics: *job correlation* and *the job correlation ratio*. Here, two jobs (referred to as a job pair) are considered to be correlated when they have similar job names, required resources, and job runtime. The job correlation ratio is the proportion of correlated job pairs among all job pairs that satisfy certain conditions. Fig. 5 (b) shows how job correlation varies with the job submission interval on two production HPC systems: NG-Tianhe and Tianhe-2A. As the interval increases, job correlation decreases significantly. When the interval reaches 30 hours, the job correlation ratio of NG-Tianhe gradually stabilizes at 0, while that of Tianhe-2A stabilizes at 0.3. This difference is because the Tianhe-2A has been in production for many years with more stable users and applications than the Next Generation Tianhe Supercomputer. Hence, the update frequency of the estimation model should not be longer than every 30 hours. Fig. 5 (c) shows job correlation changes as the job ID gap increases. As can be seen from the figure, the job correlation gradually decreases as the ID gap increases and stabilizes at about 0.08 after the ID gap is greater than 700. Based on this profiling information, we set the estimation model generation module to run every 15 hours by default, and the size of the interest window defaults to 700 jobs. We also provide a configuration interface for two parameters, allowing a system administrator to reconfigure the parameters.

TABLE IV
JOB FEATURES USED IN JOB RUNTIME ESTIMATION.

	Features	Type
1	Job name	String
2	User name	String
3	Required nodes	Integer
4	Required cores	Integer
5	Submission time (hours only)	Integer

Job features. For each incoming job, we use the set of quantifiable features given in Table IV to capture the job characteristics. These features are directly available from most RMs. Among the chosen features, the job name, required nodes, and cores are directly related to the job runtime. We also consider the user name and job submission time to capture the job characteristics based on our observations from real-life traces. According to the workload traces in Table III, 71.4% of jobs requiring a runtime longer than six hours were submitted between 6 pm and 12 am, and HPC users often submit the same job repeatedly. Statistically, there is an average 89.2% probability for a user to submit the same job that the user has submitted in the past 24 hours. Therefore, the user name and job submission time can also be essential for clustering the training data and for runtime estimation.

Predictive modeling. To generate training data clusters, we apply K-means++ to group the data samples into clusters in the feature space. To determine how many clusters to use (i.e., K), we use the classical elbow method [42], [43] to calculate the optimal value of K ($K = 15$ in our case), which can also be configured by a system administrator. We then train an SVR model for each job sample cluster, using the data samples within the cluster. The trained SVR model can then be applied to an incoming job by taking the feature values of the job (Table IV), which are available after a user submitted a job, as input to predict the job runtime.

B. Real-time Estimation Module

The ESLURM real-time estimation module is driven by events. It extracts the features of each newly submitted job and matches the closest cluster. It then uses the estimation model created for the cluster to estimate the job runtime of the incoming job. We note that the estimation model is trained by the estimation model generator, running asynchronously with the real-time estimation module. We multiply the runtime estimation with a weight to penalize underestimation for avoiding job failure and rescheduling:

$$t_{pi} = t_{pi} * \alpha. \tag{3}$$

where α is the slack variable and t_{pi} is the runtime estimation given by the estimation model. By default, α is set to 1.05 (see also Section VII-E). When the user does not submit a runtime estimate, we directly adopt the runtime estimation given by the estimation model. When the user gives a runtime estimation, we use the runtime estimate given by the estimation model only when the average estimation accuracy (AEA) of the estimation model is greater than 90%. Therefore, while trying to improve the estimation accuracy, encouraging users to give accurate runtime estimation is also an effective means to improve resource scheduling efficiency.

C. Record Module

The record module is also event-driven. The module adds the job to the historical job queue when the job is completed. Then, the module calculates the accuracy of the runtime estimation provided by the estimation model and updates the average estimation accuracy of the cluster to which the job belongs. We give the metric formula for the estimation accuracy of a single job and a formula calculating the average estimation accuracy within a job cluster:

$$EA_i = \begin{cases} t_{pi}/t_{ri}, & t_{pi} < t_{ri} \\ t_{ri}/t_{pi}, & t_{ri} \leq t_{pi} \end{cases} \tag{4}$$

$$AEA = 1/n * \sum_{i=1}^{n} EA_i \tag{5}$$

where t_{pi} is the runtime estimation of the i-th job, t_{ri} is the actual runtime of the job, and EA_i is the estimation accuracy of the job, which takes values in the range of 0 and 1, with values closer to 1 indicating higher estimation accuracy. AEA_i is the average estimate accuracy.

VI. Evaluation Setup

A. Hardware Platforms

We evaluate our approach on two HPC systems. The first platform is the Tianhe-2A supercomputer consisting of 16K computing nodes, which ranked in 7th place in the TOP500 list as of November 2021. The second platform is the Next Generation Tianhe Supercomputer (NG-Tianhe) consisting of 20K+ computing nodes. Each node on Tianhe-2A has 64GB of RAM with a 12-core 2.2 GHz Intel Xeon processor and a Matrix-2000 accelerator. Each computing node on the NG-Tianhe has a heterogeneous many-core MT processor. The master node has 196GB of RAM with a 10-core 2.4GHz Intel Xeon Sliver 4210R processor. Both the two HPC systems are equipped with a proprietary designed interconnection network with a network interface chip designed to provide high-speed network interconnection. On both HPC systems, a single network port uses a four-lane high-speed serial transmission link, with a communication rate up to 25 Gbps. The external one-way bandwidth of a single computing node is 400 Gbps in total.

B. Application Workloads

For job workloads, we use real-life historical workload traces collected from the production environment of Tianhe-2A and NG-Tianhe. These workloads are mainly MPI parallel applications for representative HPC applications like simulations for computational fluid dynamics, large-scale equipment electromagnetic, engine combustions, nonlinear flows, and analytic workloads like bio-informatics and mechanical strengthen analyses.

C. Experimental Roadmap

To evaluate ESLURM, we conduct five rounds of experiments. The first round of experiments compares ESLURM against five mainstream HPC RMs on the 4K nodes of the Tianhe-2A (Section VII-A). The second round of experiments compares ESLURM with Slurm on full-scale Tianhe-2A (Section VII-B). The third round of experiments is to deploy ESLURM on the full-scale NG-Tianhe to evaluate the scalability of ESLURM (Section VII-C). The fourth round of experiments is to build multiple independent clusters of different scales on Tianhe-2A and NG-Tianhe to evaluate the performance of ESLURM from multiple perspectives (Section VII-D). The last round of experiments evaluates the job runtime estimation model of the ESLURM (Section VII-E). Throughout the evaluation, the configuration of both Slurm and ESLURM follows the Large Cluster Administration Guide [44]. For Slurm and ESLURM, the optional database daemon (slurmdbd) and the master daemon (slurmctld) are placed on the master node. Throughout our evaluation, we compute the master node resource usage by measuring the resource usage of the master daemon.

VII. Experimental Results

A. Evaluation on 4K Nodes of Tianhe-2A

Using 4K nodes of Tianhe-2A, we compare ESLURM against six mainstream RMs: SGE (version 8.1.9) [45], Torque (version 6.13) [17], OpenPBS (version 20.0.1) [46], LSF (version 10.0.1) [7], and Slurm (version 20.11.7) [47]. During the experiment, we ensure there was no background workload runs on the master node.

Master node resource demands. Subgraphs (a) to (e) in Fig. 7 show the hardware resource usage of the master node for a 24 hours period after launching the RM, where a good RM would have low resource usage. The CPU load is

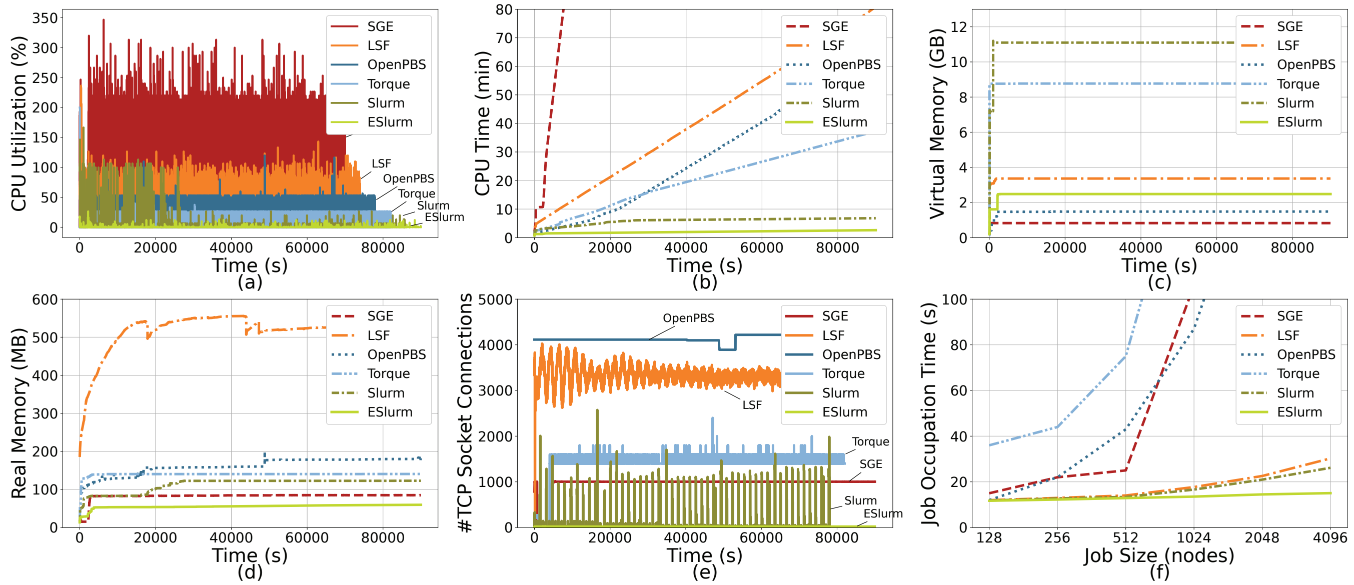

Fig. 7. Experimental results on the 4K nodes of Tianhe-2A. Figures a-e show the resource usage of the master node within 24 hours since RMs are started, with a sampling frequency of once per second. Figures a and b show the CPU usage, Figures c and d show the virtual memory and real memory usage, and Figure e uses the number of real-time sockets to measure the network resource usage. Figure f shows the job occupation time at different job size.

measured by the CPU utilization and the CPU time. The data is collected through probing */proc/stat* in Linux. Slurm and ESLURM exhibits significantly lower CPU load than other approaches, where ESLURM incurs the lowest CPU load. However, Slurm incurs the highest memory footprint, requiring 10 GB of virtual memory. ESLURM significantly reduces the memory footprint by using less than 2GB of virtual memory. If we now look at Fig. 7 (d), ESLURM has the lowest real memory usage of around 60MB at any given time. Fig. 7 (e) quantifies the network load of the master node by measuring the number of concurrent TCP socket connections. OpenPBS and SGE require frequent network communications and have to maintain a large number of concurrent TCP connections. This can put high pressure on the network switcher of the master node. Furthermore, LSF and Slurm can have burst network traffic with frequent high network traffic and a high number of concurrent TCP socket connections ($>= 1000$). By contrast, the master node of ESLURM incurs the lowest network traffic, using less than 100 concurrent TCP connections at any time. By distributing the network traffic to satellite nodes, ESLURM prevents the master node from becoming the bottleneck. Overall, ESLURM incurs the lowest demand on the CPU load, real memory footprint, and network, allowing the RM to manage a larger number of computing nodes within the same computation resources compared to alternative schemes.

Satellite node resource demands. In this experiment, we use two satellite nodes. After running for 24 hours, the average CPU time of each satellite node is about 6 minutes, the average virtual memory usage is 1.2GB, and the average real memory usage is 42.6MB.

Resource management and job scheduling. To evaluate RM's resource management and job scheduling capabilities, we use different RMs to load parallel jobs of different sizes but with a fixed runtime of 10s. We consider *the job occupation time*, that is, the time from job submission to the complete release of system resources occupied by the job when resources are sufficient. Job occupation time includes the time it takes for an RM to allocate computing resources to a job, to spawn job processes on multiple nodes, for the job to run, and for the job to reclaim its occupied resources when the job completes. Fig. 7 (f) shows the results. With the increase of job size, the job occupation time of SGE, Torque, and OpenPBS has increased to a level that is typically unacceptable in practice. Such an rapid increase in time greatly reduces the system's resource utilization, suggesting that the three RMs would struggle to scale to a 1K nodes cluster and beyond. In contrast, LSF, Slurm, and ESLURM have small increases in time as the job size grows. Specifically, with ESLURM, the job occupation time is always less than 15 seconds in our evaluation across jobs with different running times.

Message broadcasting. ESLURM reduces job occupation time by improving the message broadcast efficiency of the job loading and termination message. The design of satellite nodes and FP-Tree in ESLURM both contribute to improving the efficiency of message broadcasting. Broadcasting messages from multiple satellite nodes improves the parallelism of message broadcasting, while FP-Tree reduces the impact of failed nodes during message broadcasting. Fig. 8 (a) shows the message broadcast time of ESLURM and Slurm. When managing a large-scale job running with 4K nodes, ESLURM can reduce the average broadcast time by 63.7% and 73.6% for two types of messages, in which FP-Tree reduces the average broadcast time by 36.3% and 54.9% respectively. Extrapolating from the average data in Fig. 8 (a), ESLURM can save 25K core hours per day on a cluster with 4K nodes, 64K cores, and running about 1K jobs per day.

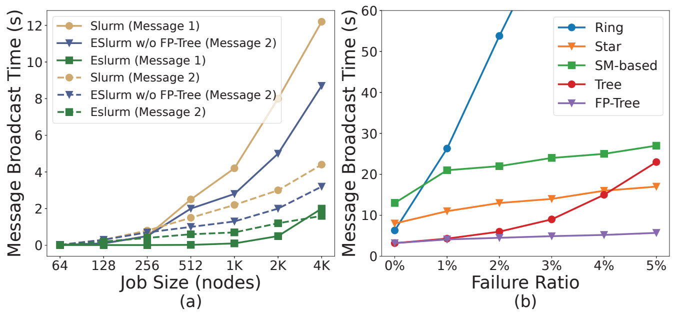

Fig. 8. Figure a depicts the average message broadcast time. Message 1 and 2 is job loading message and job termination message, respectively. Figure b shows the message broadcast time of job loading message under different failure ratio.

FP-tree node placement. In this experiment, we deployed ESLURM on 4K nodes to manage jobs for ten days. On average, each satellite node constructs 3828 FP-Trees daily, and each FP-Tree consists of 1511 nodes. During the evaluation period, a total of 28 small-scale failure events occurred, involving 103 single-node failures on 62 nodes. In addition, there was a large-scale node failure involving more than 600 nodes caused by hardware replacement and upgrade on the sixth day. A total of 1423 failed nodes are encountered when constructing FP-Tree, of which 81.7% of the failed nodes are placed on leaf nodes by FP-Tree, suggesting that our FP-Tree is highly effective in proactive identifying the failed nodes (see also Section IV).

FP-tree performance. In the same 4K-node evaluation setup, we also compare our FP-Tree with four widely used communication structures (ring, star, shared memory-based, and tree structures) at different failure ratios. Messages in the ring structure are transmitted in succession in the order of nodes. The star structure broadcasts messages directly from one location to multiple nodes. The shared memory-based structure caches the message to shared memory, and computing nodes retrieve the message from the cache. We separate the communication structure from RM and reproduce various structures using the same techniques. For example, all communication is based on socket connections, and the number of retries for connection failure is set to three. Without loss of generality, we simulate the failure of a node by powering it down. As shown in Fig. 8 (b), the communication time of the ring, star, and tree structures increases significantly with increasing failure ratio, while the shared memory-based structure does not change much. The communication time of the FP-Tree is rarely affected by failures and always maintains a minimum communication time. In some severe environments, such as when the failure ratio reaches 30%, our FP-Tree still manages to keep the communication time below 10 seconds, while other structures result in a delay of minutes.

B. Evaluation on Full-scale Tianhe-2A

Master node resource demands. Fig. 9 (a)-(c) shows the resource usage of the master nodes when applying Slurm and ESLURM to manage 16K computing nodes on Tianhe-2A.

TABLE V
RESOURCE USAGE OF THE MASTER NODE

	SE_1	SE_2	SE_3	SE_4	SE_5
CPU Time (min)	332.9	336.3	342.3	339.4	355.2
Virtual memory usage (GB)	10.7	10.8	10.8	10.8	10.9
Real memory usage (MB)	361.8	373.3	392.5	421.6	458.6
Average concurrent sockets	8.5	14.2	28.4	24.4	30.2

TABLE VI
AVERAGE OPERATIONAL DATA FOR SATELLITE NODES

	SE_1	SE_2	SE_3	SE_4	SE_5
Numbers of received tasks	6380	6398	6348	6234	6206
Average nodes in each task	6076.1	3271.2	2442.4	1411.0	1267.6
Virtual memory usage (GB)	10.8	10.6	10.5	10.3	10.3
Real Memory usage (MB)	270.5	196.2	187.6	175.9	169.0
Average concurrent sockets	118.1	94.5	90.4	89.3	70.2

While only using two satellite nodes, ESLURM significantly reduces the CPU time used by Slurm, using less than 40% of the CPU time required by Slurm. ESLURM also significantly reduces the virtual and real memory footprint, saving over 80% of the memory consumption compared to Slurm. The advantage of ESLURM can also be observed from the low number of concurrent TCP socket connections because ESLURM only needs to communicate with satellite nodes.

Satellite node resource demands. To observe how satellite nodes share the load of the master node, we monitor the resource usage of the two satellite nodes in ESLURM. As shown in Fig. 9 (d)-(f), the two satellite nodes' usage of the three types of resources is similar, reflecting a good load balance among them. The total CPU time taken by the two satellite nodes reaches about 100 minutes and the real memory usage stabilizes to about 80MB at 6000s. The number of concurrent sockets shows fluctuations, indicating that the distributed structure makes the communication load fluctuations originally appearing on the master node to be transferred to the satellite nodes. The maximum number of concurrent sockets does not exceed 80, representing an over 10x reduction compared to Slurm which can use over 1000 concurrent sockets.

C. Evaluation on Full-scale NG-Tianhe

Using 20K+ nodes of NG-Tianhe, we deploy ESLURM with different numbers of satellite nodes and run each set of experiments for ten days.

Table V compares the resource usage of the master node five ESLURM setups, SE_1 to SE_5, where the number of satellite nodes varies from 10 to 50 with a step size of 10. As the number of satellite nodes increases, the master node needs to communicate directly with more satellite nodes, and the resource usage of the master node will increase accordingly. Table VI compares the average data from satellite nodes in each ESLURM. As the number of satellite nodes increases, the number of broadcast tasks received by satellite nodes does not change much, but the number of target slave nodes in each broadcast task gradually decreases, making the satellite nodes less resource-intensive for memory and network.

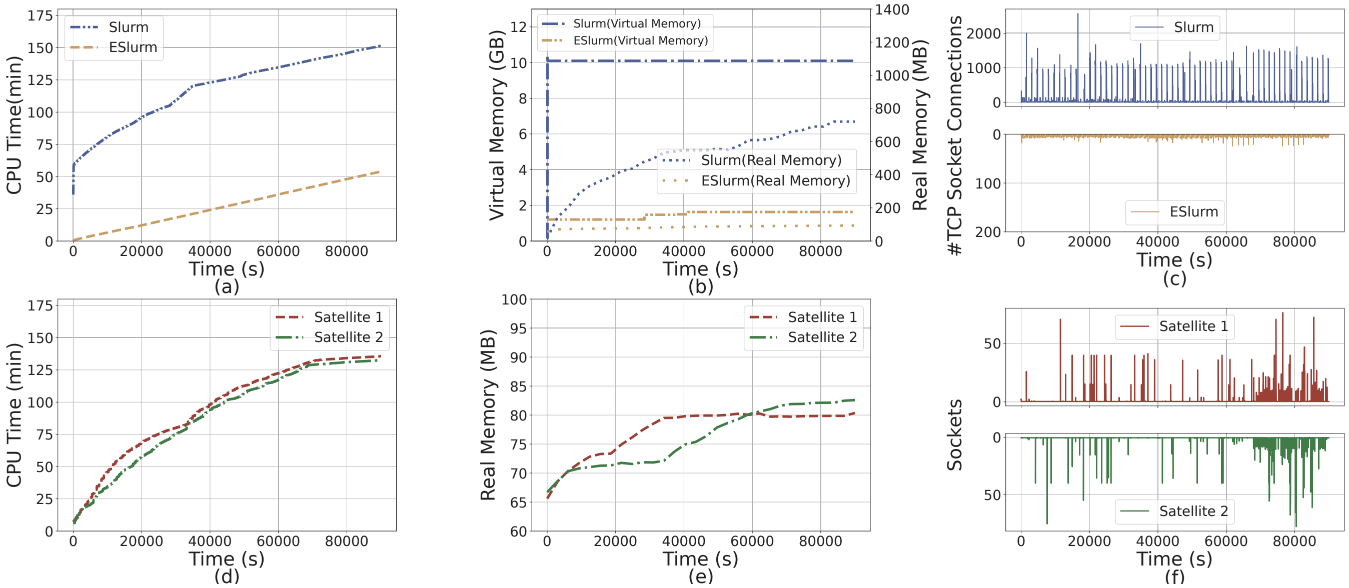

Fig. 9. Experimental results on the 16K nodes of Tianhe-2A. Figures a, b, and c show the CPU, memory, and network resource usage of the master node within 24 hours after startup when Slurm and ESLURM are deployed on 16K nodes, respectively. ESLURM uses two satellite nodes. Figures d, e, and f compare the CPU, memory, and network resource usage of two satellite nodes in ESLURM. In each figure, the data is sampled once a second.

TABLE VII
BASIC INFORMATION FOR FOUR CLUSTERS.

	#Nodes	Deployed Resource Managers	Load sources
1	1,024	SGE, Torque, OpenPBS, LSF, Slurm, ESLURM	Tianhe-2A
2	4,096	OpenPBS, LSF, Slurm, ESLURM	Tianhe-2A
3	16,384	Slurm, ESLURM	Tianhe-2A
4	20K+	Slurm, ESLURM	NG-Tianhe

As discussed in Section III, the number of satellite nodes not only affects the resource usage of the master node and satellite nodes but also directly affects the efficiency of data transfer. We track the broadcast times of the heartbeat message of each computing node to find the optimal configuration for the number of satellite nodes. As shown in Fig. 11 (a), using 20 satellite nodes for the full-scale NG-Tianhe gives the highest data transfer efficiency. Based on the results of our long-term experiments and deployments, we believe that using one satellite node for every 5K slave nodes is appropriate.

The production deployment experience on the NG-Tianhe since March 2021 shows that ESLURM can fully adapt to large-scale HPC clusters with 20K+ nodes. The average response time for user requests is less than 1s, and there are almost no crashes of the ESLURM except for hardware failures. Over 1.2 million jobs for hundreds of users have been served so far. Performance data on resource utilization and job scheduling is available in Section VII-D.

D. Evaluation on Clusters of Different Scales

In this evaluation, we set up four clusters of different scales to evaluate the resource utilization and job scheduling efficiency of different RMs. The specific configurations and workloads of the clusters are given in Table VII. In the experiment, we deployed six RMs on 1024 nodes. Since SGE and Torque cannot scale to 4,096 nodes, only four RM are deployed on the 4,096-node cluster. Finally, only Slurm and ESLURM, are deployed on the full-scale Tianhe-2A (16,384 nodes) and full-scale NG-Tianhe (20K+ nodes). The workload trace is obtained from the historical load on the real cluster during a week. In this evaluation, we use the backfill scheduling algorithm [28], [48], [49] for all RMs. We consider three metrics: (1) *system utilization* - the node-hours used for running jobs to the total elapsed node-hours of a system; (2) *average waiting time* - the gap between a job submitted and it gets executed; and (3) *average bounded slowdown* - the ratio of job response time to its actual runtime [50]:

$$slowdown = max((t_w + t_r)/max(t_r, \tau), 1) \qquad (6)$$

where t_w and t_r are the job waiting time and runtime. τ is a constant for preventing the impact of extremely short jobs, and we set it to 10.

As shown in Fig. 10, the system utilization decreases as the cluster size grows. There are two main reasons for this trend. The first is that the cost of RM to manage a larger cluster increases. Second, there are not enough small jobs in large-scale systems to backfill the resource gap created by many large-scale, long-running jobs. Nonetheless, ESLURM outperforms alternative schemes in all three metrics across all clusters. On the full-scale NG-Tianhe, ESLURM's performance advantage is more pronounced. Specifically, ESLURM improves the system utilization by 47.2% compared to Slurm. Among the multifaceted optimizations of ESLURM, the job runtime estimation framework contributes 8.7% in resource utilization while the FP-Tree contributes 6.2%. In addition, ESLURM reduces the average job waiting time by 60.5% and the average slowdown by 75.8%.

E. Performance of Job Runtime Estimation

In this experiment, we evaluate the performance of the ESLURM's job runtime estimation framework using historical

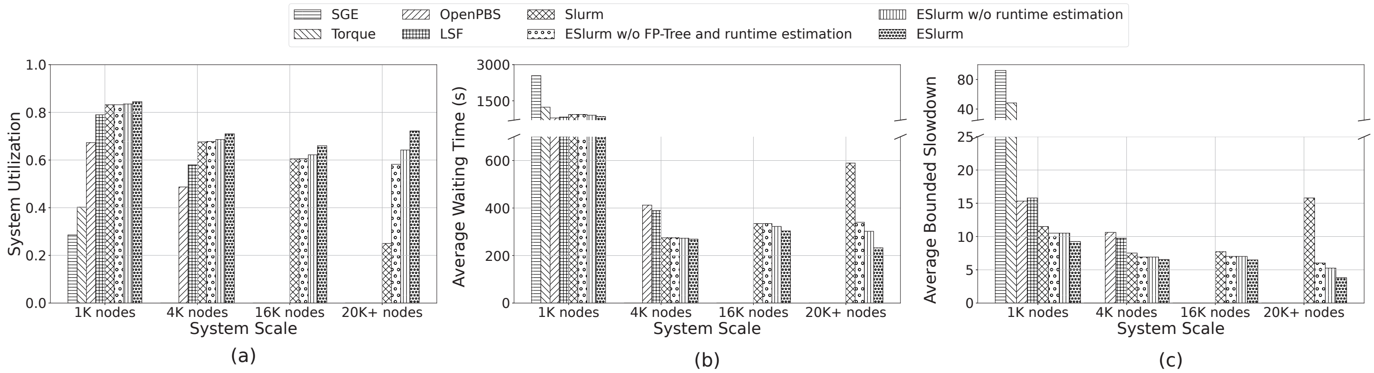

Fig. 10. Job scheduling efficiency of different RMs on Tianhe-2A. ESLURM gives the best overall performance on system utilization (*higher is better*) (a) and scheduling efficiency for two *lower-is-better* metrics (b) and (c).

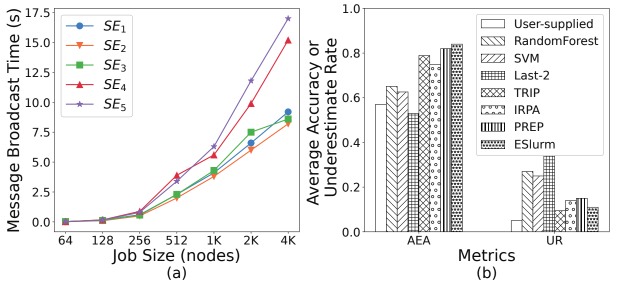

Fig. 11. Figure a depicts the message broadcast time under different satellite-node configurations. Figure b shows the performance of different runtime estimate prediction model.

TABLE VIII
IMPACT OF THE SLACK VARIABLE PRESENTED IN SECTION V-B

α	**1.00**	**1.01**	**1.02**	**1.03**	**1.04**	**1.05**	**1.06**	**1.07**	**1.08**
AEA	0.87	0.87	0.86	0.85	0.85	0.84	0.82	0.82	0.80
UR	0.54	0.31	0.26	0.18	0.14	0.12	0.12	0.12	0.11

workloads on the NG-Tianhe over the past year. Table VIII shows the impact of slack variables on the performance of the ESLURM job runtime estimation model. With the increase of the slack variable, the average estimation accuracy (AEA) and the underestimated rate (UR) gradually decreases. When the slack variable is greater than 1.05, the reduction rate of AEA becomes slower while the reduction rate of UR increases, so we set the slack variable to 1.05 by default on this cluster.

We compare ESLURM with a variety of runtime prediction models. Among them, Last-2 [40], IRPA [51], TRIP [52] and PREP [53] are the latest related work. As shown in Fig. 11 (b), the user-provided runtime predictions are less accurate and always overestimated. SVM, RandomForest, and Last-2 all have an average accuracy below 70% and an underestimation rate above 25%. In contrast, IRPA, TRIP, and PREP have higher average accuracies and lower underestimates. ESLURM performs the best with an average accuracy of 84% while keeping the underestimation around 10%.

VIII. RELATED WORK

Efforts have been devoted to design and implement RMs for supercomputers. BPROC [30] provides a single system image and process migration facilities for processes running in HPC systems. ALPS [29] is the RM of Cray systems with a single-server architecture. A similar centralized structure can be seen in LIBI/LaunchMON [54], Cplant [55], STORM [56], and ORCM [57]. These jobs are still centralized and have flaws in scalability. [24] proposes a method named Slum++, which employs multiple master nodes so that each one manages a partition of computing nodes and participates in resource allocation through resource balancing techniques.

Some attempts at distributed HPC RMs are worth noting. Flux [58] provides a fully hierarchical software framework architecture that allows scalable and customizable resource management and scheduling service modules to be dynamically loaded into Flux instances. Flux instances can spawn one or more sub-instances that can manage subsets of parent instance resources and jobs to increase parallelism, and such nesting can be further recursive. Flux completely abandons the design on the master node, implementing resource management and job scheduling as hierarchical functions. In contrast, ESLURM retains the simplified master node, offloading the large-scale communication process to the satellite nodes. However, Flux is still beta software and primarily used in single-user mode, and it has a long way to go before it can achieve the same functionality as system-level RMs such as Slurm [59]. Argo [60], [61] builds a software stack for providing increased functionality to exascale applications and runtime systems. However, the Argo project did not optimize RM and still used a centralized Slurm.

Large-scale systems usually use a dedicated resource management solution with multiple software collaborations. Fugaku's solution consists of three customized software [62], including a system management software, a job management software, and a user management software. The system management software manages the hardware and software units, the job management software provides job scheduling and management services, and the user management software provides resource management about users. BlueGene/Q [63], with 98,304 nodes, uses Slurm with custom plug-ins and dedicated BlueGene Software to complete system-wide resource management [64]. The Slurm daemon runs on only a few front-end nodes and is responsible for prioritizing work queues and deciding when and where to start or terminate jobs. The BlueGene Software allocates and releases resources for jobs based on SLURM input, launches tasks, and monitors node health. In contrast to these solutions, ESLURM is fully open source and has been proven to provide efficient resource management services for large-scale clusters. Since ESLURM does not require specialized software or plug-in support, it can be easily ported across different HPC systems.

Resource managers for data centers have a similar evolution. Early data centers used centralized master-slave resource managers, such as Borg [65] and Hadoop [66]. As the scale of the data center increases, the bottleneck of a single master node gradually emerges, and the centralized master-slave structure gradually develops into a multi-master (e.g., kubernetes [10]) or two-layer structure (e.g.,YARN [11], Mesos [12]).

In the relevant field of job runtime estimation, the Last-2 [40] method uses the average of the actual runtimes of the last two job submissions by the same user as the predicted time for new job submissions. Wu et al. [51] proposes an integrated learning model IRPA with random forest regression, support vector regression, and Bayesian ridge regression for job time estimation. [52] proposes an online adjustment framework, TRIP, which utilizes the data truncation capability of Tobit regression to obtain accurate runtime estimates. PREP [53] is a runtime prediction framework which groups jobs into several clusters according to their running paths and trains a runtime prediction model for each job cluster.

IX. Conclusion

We have presented ESLURM, a new resource management (RM) system for HPC systems. ESLURM is designed to overcome the limitations of the current HPC RM systems, which follow a centralized architecture that no longer fits next-generation HPC and supercomputer systems. ESLURM follows a distributed communication structure and uses node failure prediction and job runtime estimation to improve the efficiency of job and resource scheduling. We evaluate ESLURM in two production HPC systems using over 16K and 20K+ physical computing nodes. Experimental results show that ESLURM delivers better scalability and stronger performance for data communications and job scheduling than existing RM solutions. ESLURM has been deployed in production on the Next Generation Tianhe Supercomputer since March 2021. We hope the experience shared in this paper and the open-source release of ESLURM can provide valuable insights for designing RM for next-generation HPC systems and exascale supercomputers.

Acknowledgement

This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant agreements 61902405 and 61872294, the Natural Science Foundation of Hunan Province of China under Grant No. 2021JJ40692, and a UK Royal Society International Collaboration grant.

References

[1] W. Shin, V. Oles, A. M. Karimi, J. A. Ellis, and F. Wang, "Revealing power, energy and thermal dynamics of a 200pf pre-exascale supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476188

[2] K. Yoshikawa, S. Tanaka, and N. Yoshida, "A 400 trillion-grid vlasov simulation on fugaku supercomputer: large-scale distribution of cosmic relic neutrinos in a six-dimensional phase space," in *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds. ACM, 2021, pp. 5:1–5:11. [Online]. Available: https://doi.org/10.1145/3458817.3487401

[3] Q. Zhu, H. Luo, C. Yang, M. Ding, W. Yin, and X. Yuan, "Enabling and scaling the HPCG benchmark on the newest generation sunway supercomputer with 42 million heterogeneous cores," in *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds. ACM, 2021, pp. 57:1–57:13. [Online]. Available: https://doi.org/10.1145/3458817.3476158

[4] M. Taufer, "AI4IO: A suite of ai-based tools for io-aware HPC resource management," in *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021.* IEEE, 2021, p. 1. [Online]. Available: https://doi.org/10.1109/HiPC53243.2021.00012

[5] A. Souza, M. Rezaei, E. Laure, and J. Tordsson, "Hybrid resource management for HPC and data intensive workloads," in *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2019, Larnaca, Cyprus, May 14-17, 2019.* IEEE, 2019, pp. 399–409. [Online]. Available: https://doi.org/10.1109/CCGRID.2019.00054

[6] A. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *JSSPP*, 2003.

[7] IBM, "Lsf," 2022. [Online]. Available: https://www.ibm.com/products/hpc-workload-management

[8] SchedMD, "Slug21," 2021. [Online]. Available: https://slurm.schedmd.com/SLUG21/Field_Notes_5.pdf

[9] X. LIAO, L. XIAO, C. YANG, and Y. LU, "Milkyway-2 supercomputer: system and application," *Frontiers of Computer Science*, vol. 8, no. 3, p. 345, 2014. [Online]. Available: https://journal.hep.com.cn/fcs/EN/abstract/article_10719.shtml

[10] "kubernetes," 2022. [Online]. Available: https://kubernetes.io/

[11] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop YARN: yet another resource negotiator," in *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, G. M. Lohman, Ed. ACM, 2013, pp. 5:1–5:16. [Online]. Available: https://doi.org/10.1145/2523616.2523633

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, D. G. Andersen and S. Ratnasamy, Eds. USENIX Association, 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[13] Y. S. Patel, A. Baheti, and R. Misra, "Interval graph multi-coloring-based resource reservation for energy-efficient containerized cloud data centers," *J. Supercomput.*, vol. 77, no. 5, pp. 4484–4532, 2021. [Online]. Available: https://doi.org/10.1007/s11227-020-03439-z

[14] X. Zhang, Z. Shen, B. Xia, Z. Liu, and Y. Li, "Estimating power consumption of containers and virtual machines in data centers," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 288–293.

[15] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan, "Goldilocks: Adaptive resource provisioning in containerized data centers," in *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019.* IEEE, 2019, pp. 666–677. [Online]. Available: https://doi.org/10.1109/ICDCS.2019.00072

[16] W. Gentzsch, “Sun grid engine: towards creating a compute power grid,” in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 35–36.

[17] Torque, 2021. [Online]. Available: https://adaptivecomputing.com/cherry-services/torque-resource-manager/

[18] OpenPBS, 2021. [Online]. Available: https://openpbs.org/

[19] D. Thain, T. Tannenbaum, and M. Livny, “Distributed computing in practice: the condor experience: Research articles,” *Concurrency - Practice and Experience*, vol. 17, pp. 323–356, 02 2005.

[20] TOP500, 2021. [Online]. Available: https://www.top500.org/lists/top500/list/2021/06/

[21] A. Cilfone, L. Davoli, L. Belli, and G. Ferrari, “Wireless mesh networking: An iot-oriented perspective survey on relevant technologies,” *Future Internet*, vol. 11, no. 4, p. 99, 2019. [Online]. Available: https://doi.org/10.3390/fi11040099

[22] Q.-V. Pham, D. C. Nguyen, S. Mirjalili, D. T. Hoang, D. N. Nguyen, P. N. Pathirana, and W.-J. Hwang, “Swarm intelligence for next-generation networks: Recent advances and applications,” *Journal of Network and Computer Applications*, vol. 191, p. 103141, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521001582

[23] D.-S. Mirashe and N. Kalyankar, ““peer-to-peer network protocols”,” 02 2010.

[24] K. Wang, X. Zhou, K. Qiao, M. Lang, B. Mcclelland, and I. Raicu, “Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing,” in *International Symposium on High-performance Parallel & Distributed Computing*, 2015.

[25] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, “The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications,” in *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[26] “Icinga 2,” 2022. [Online]. Available: https://icinga.com/docs/icinga-2/latest/doc/01-about/

[27] M. Shreedhar and G. Varghese, “Efficient fair queueing using deficit round robin,” in *Proceedings of the ACM SIGCOMM 1995 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Cambridge, MA, USA, August 28 - September 1, 1995*, S. Wecker and D. Oran, Eds. ACM, 1995, pp. 231–242. [Online]. Available: https://doi.org/10.1145/217382.217453

[28] ScedMD, “Backfill,” 2022. [Online]. Available: https://slurm.schedmd.com/sched_config.html

[29] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing, “The application level placement scheduler,” 2006.

[30] E. A. Hendriks, “Bproc: the beowulf distributed process space,” in *Proceedings of the 16th international conference on Supercomputing, ICS 2002, New York City, NY, USA, June 22-26, 2002*, K. Ebcioglu, K. Pingali, and A. Nicolau, Eds. ACM, 2002, pp. 129–136. [Online]. Available: https://doi.org/10.1145/514191.514212

[31] A. Das, F. Mueller, P. Hargrove, E. Roman, and S. B. Baden, “Doomsday: predicting which node will fail when on supercomputers,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 2018, pp. 9:1–9:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291668

[32] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, “Toward exascale resilience,” *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009. [Online]. Available: https://doi.org/10.1177/1094342009347767

[33] S. Hukerikar and C. Engelmann, “Resilience design patterns: A structured approach to resilience at extreme scale,” *CoRR*, vol. abs/1708.07422, 2017. [Online]. Available: http://arxiv.org/abs/1708.07422

[34] R. Wang, K. Lu, J. Chen, W. Zhang, J. Li, Y. Yuan, P. Lu, L. Huang, S. Li, and X. Fan, “Brief introduction of tianhe exascale prototype system,” *Tsinghua Science and Technology*, vol. 26, pp. 361–269, 06 2021.

[35] E. Berrocal, L. Yu, S. Wallace, M. E. Papka, and Z. Lan, “Exploring void search for fault detection on extreme scale systems,” in *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*. IEEE Computer Society, 2014, pp. 1–9. [Online]. Available: https://doi.org/10.1109/CLUSTER.2014.6968757

[36] L. Yu and Z. Lan, “A scalable, non-parametric method for detecting performance anomaly in large scale computing,” *IEEE Trans. Parallel Distributed Syst.*, vol. 27, no. 7, pp. 1902–1914, 2016. [Online]. Available: https://doi.org/10.1109/TPDS.2015.2475741

[37] T. Hcormen, C. Eleiserson, R. Lrivest, and C. Stein, “Introduction to algorithms(third edition),” *Computer Education*, 2013.

[38] W. Tang, Z. Lan, N. Desai, and D. Buettner, “Fault-aware, utility-based job scheduling on blue, gene/p systems,” 10 2009, pp. 1 – 10.

[39] W. Cirne and F. Berman, “A comprehensive model of the supercomputer workload,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 140–148.

[40] D. Tsafrir, Y. Etsion, and D. G. Feitelson, “Backfilling using system-generated predictions rather than user runtime estimates,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007.

[41] D. Arthur and S. Vassilvitskii, “k-means++: the advantages of careful seeding,” in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, N. Bansal, K. Pruhs, and C. Stein, Eds. SIAM, 2007, pp. 1027–1035. [Online]. Available: http://dl.acm.org/citation.cfm?id=1283383.1283494

[42] BDACVID, J., KETCHEN, CHRISTOPHER, L., and SHOOK, “The application of cluster analysis in strategic management research: An analysis and critique,” *Strategic Management Journal*, 1996.

[43] P. V. Kolesnichenko, Q. Zhang, C. Zheng, M. S. Fuhrer, and J. A. Davis, “Multidimensional analysis of excitonic spectra of monolayers of tungsten disulphide: toward computer-aided identification of structural and environmental perturbations of 2d materials,” *Mach. Learn. Sci. Technol.*, vol. 2, no. 2, p. 25021, 2021. [Online]. Available: https://doi.org/10.1088/2632-2153/abd87c

[44] SchedMD, “Large cluster administration guide,” 2020. [Online]. Available: https://slurm.schedmd.com/big_sys.html?msclkid=162adcaad06511eca741b075370eff3f

[45] “Sge,” 2018. [Online]. Available: https://sourceforge.net/p/gridengine/activity/?page=0&limit=100#59bee64e5fcbc97fb3e55d96

[46] “Openpbs-github,” 2022. [Online]. Available: https://github.com/openpbs

[47] “Slurm-github,” 2021. [Online]. Available: https://github.com/SchedMD/slurm/releases/tag/slurm-20-11-7-1

[48] Altair, “Backfill,” 2021. [Online]. Available: https://2021.help.altair.com/2021.1.0/control/Chunk1924291020.html

[49] IBM, “Backfill,” 2022. [Online]. Available: https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=jobs-backfill-scheduling

[50] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, “Theory and practice in parallel job scheduling,” in *Job Scheduling Strategies for Parallel Processing, IPPS’97 Workshop, Geneva, Switzerland, April 5, 1997, Proceedings*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds., vol. 1291. Springer, 1997, pp. 1–34. [Online]. Available: https://doi.org/10.1007/3-540-63574-2_14

[51] W. U. Gui-Bao, Y. Shen, W. S. Zhang, S. S. Liao, Q. Q. Wang, and L. I. Jing, “Runtime prediction of jobs for backfilling optimization,” *Journal of Chinese Computer Systems*, 2019.

[52] Y. Fan, P. Rich, W. E. Allcock, M. E. Papka, and Z. Lan, “Trade-off between prediction accuracy and underestimation rate in job runtime estimates,” in *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 2017, pp. 530–540. [Online]. Available: https://doi.org/10.1109/CLUSTER.2017.11

[53] L. Zhou, X. Zhang, W. Yang, Y. Han, F. Wang, Y. Wu, and J. Yu, “Prep: Predicting job runtime with job running path on supercomputers,” in *50th International Conference on Parallel Processing*, ser. ICPP 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3473521

[54] J. D. Goehner, D. C. Arnold, D. H. Ahn, G. L. Lee, B. Supinski, M. P. Legendre, B. P. Miller, and M. Schulz, “Libi: A framework for bootstrapping extreme scale software systems,” *Parallel Computing*, vol. 39, no. 3, pp. 167–176, 2013.

[55] R. Brightwell and L. A. Fisk, “Scalable parallel application launch on cplant™,” in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC ’01. New York, NY, USA:

Association for Computing Machinery, 2001, p. 40. [Online]. Available: https://doi.org/10.1145/582034.582074

[56] E. Frachtenberg, F. Petrini, J. Fernández, and S. Pakin, "STORM: scalable resource management for large-scale parallel computers," *IEEE Trans. Computers*, vol. 55, no. 12, pp. 1572–1587, 2006. [Online]. Available: https://doi.org/10.1109/TC.2006.206

[57] Intel, "Orcm," 2015. [Online]. Available: https://github.com/open-mpi/orcm

[58] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. W. Scogland, B. Springmeyer, and M. Taufer, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Gener. Comput. Syst.*, vol. 110, pp. 202–213, 2020. [Online]. Available: https://doi.org/10.1016/j.future.2020.04.006

[59] "Flux administrator's guide," 2021. [Online]. Available: https://flux-framework.readthedocs.io/en/latest/adminguide.html#overview

[60] Argo, 2022. [Online]. Available: https://web.cels.anl.gov/projects/argo/overview/#resource-management

[61] R. Sakamoto, T. Cao, M. Kondo, K. Inoue, M. Ueda, T. Patki, D. A. Ellsworth, B. Rountree, and M. Schulz, "Production hardware overprovisioning: Real-world performance optimization using an extensible power-aware resource management framework," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, pp. 957–966. [Online]. Available: https://doi.org/10.1109/IPDPS.2017.107

[62] Fujitsu, "Operations management software of supercomputer fugaku," 2020. [Online]. Available: https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article10.html?msclkid=006d48e5d01911ec9dd8511f3e43b57f#cap-02

[63] IBM100, 2021. [Online]. Available: https://www.ibm.com/ibm/history/ibm100/us/en/icons/bluegene/

[64] "Slurm operation ibm bluegene/q," 2021. [Online]. Available: https://slurm.schedmd.com/slurm_ug_2011/SLURM.BGQ.pdf

[65] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, L. Réveillère, T. Harris, and M. Herlihy, Eds. ACM, 2015, pp. 18:1–18:17. [Online]. Available: https://doi.org/10.1145/2741948.2741964

[66] Apache, "hadoop," 2022. [Online]. Available: https://hadoop.apache.org/