

Planning Considerations for Job Scheduling in HPC Clusters

As cluster installations continue growing to satisfy ever-increasing computing demands, advanced schedulers can help improve resource utilization and quality of service. This article discusses issues related to job scheduling on clusters and introduces scheduling algorithms to help administrators select a suitable job scheduler.

BY SAEED IQBAL, PH.D.; RINKU GUPTA; AND YUNG-CHIN FANG

Cluster installations primarily comprise two types of standards-based hardware components—servers and networking interconnects. Clusters are divided into two major classes: high-throughput computing clusters and high-performance computing clusters. High-throughput computing clusters usually connect a large number of nodes using low-end interconnects. In contrast, high-performance computing clusters connect more powerful compute nodes using faster interconnects than high-throughput computing clusters. Fast interconnects are designed to provide lower latency and higher bandwidth than low-end interconnects.

These two classes of clusters have different scheduling requirements. In high-throughput computing clusters, the main goal is to maximize throughput—that is, jobs completed per unit of time—by reducing load imbalance among compute nodes in the cluster. Load balancing is particularly important if the cluster has heterogeneous compute nodes. In high-performance computing clusters, an additional consideration arises: the need to minimize communication overhead by mapping applications appropriately to the available compute nodes. High-throughput computing clusters are suitable for executing loosely coupled parallel or distributed applications, because such applications do not have high communication requirements among compute nodes during execution time. High-performance computing clusters are more suitable for tightly coupled parallel

applications, which have substantial communication and synchronization requirements.

A resource management system manages the processing load by preventing jobs from competing with each other for limited compute resources. Typically, a resource management system comprises a resource manager and a job scheduler (see Figure 1). Most resource managers have an internal, built-in job scheduler, but system administrators can usually substitute an external scheduler for the internal scheduler to enhance functionality. In either case, the scheduler communicates with the resource manager to obtain information about queues, loads on compute nodes, and resource availability to make scheduling decisions.

Usually, the resource manager runs several daemons on the master node and compute nodes including a scheduler daemon, which typically runs on the master node. The resource manager also sets up a queuing system for users to submit jobs—and users can query the resource manager to determine the status of their jobs. In addition, a resource manager maintains a list of available compute resources and reports the status of previously submitted jobs to the user. The resource manager helps organize submitted jobs based on priority, resources requested, and availability.

As shown in Figure 1, the scheduler receives periodic input from the resource manager regarding job queues and available resources, and makes a schedule that determines the order in which jobs will be executed. This is done while

maintaining job priority in accordance with the site policy that the administrator has established for the amount and timing of resources used to execute jobs. Based on that information, the scheduler decides which job will execute on which compute node and when.

Understanding job scheduling in clusters

When a job is submitted to a resource manager, the job waits in a queue until it is scheduled and executed. The time spent in the queue, or *wait time*, depends on several factors including job priority, load on the system, and availability of requested resources. *Turnaround time* represents the elapsed time between when the job is submitted and when the job is completed; turnaround time includes the wait time as well as the job's actual execution time. *Response time* represents how fast a user receives a response from the system after the job is submitted.

Resource utilization during the lifetime of the job represents the actual useful work that has been performed. *System throughput* is defined as the number of jobs completed per unit of time. *Mean response time* is an important performance metric for users, who expect minimal response time. In contrast, system administrators are concerned with overall resource utilization because they want to maximize system throughput and return on investment (ROI), especially in high-throughput computing clusters.

In a typical production environment, many different jobs are submitted to clusters. These jobs can be characterized by factors such as the number of processors requested (also known as job size, or *job width*), estimated runtime, priority level, parallel or distributed execution, and specific I/O requirements. During execution, large jobs can occupy significant portions of a cluster's processing and memory resources.

System administrators can create several types of queues, each with a different priority level and quality of service (QoS). To make intelligent schedules, however, schedulers need information regarding job size, priority, expected execution time (indicated by the user), resource access permission (established by the administrator), and resource availability (automatically obtained by the scheduler).

In high-performance computing clusters, the scheduling of parallel jobs requires special attention because parallel jobs comprise several subtasks. Each subtask is assigned to a unique compute node during execution and nodes constantly communicate among

themselves during execution. The manner in which the subtasks are assigned to processors is called *mapping*. Because mapping affects execution time, the scheduler must map subtasks carefully. The scheduler needs to ensure that nodes scheduled to execute parallel jobs are connected by fast interconnects to minimize the associated communication overhead. For parallel jobs, the job efficiency also affects resource utilization. To achieve high resource utilization for parallel jobs, both job efficiency and advanced scheduling are required. Efficient job processing depends on effective application design.

Under heavy load conditions, the capability to provide a fair portion of the cluster's resources to each user is important. This capability can be provided by using the *fair-share* strategy, in which the scheduler collects historical data from previously executed jobs and uses the historical data to dynamically adjust the priority of the jobs in the queue. The capability to dynamically make priority changes helps ensure that resources are fairly distributed among users.

Most job schedulers have several parameters that can be adjusted to control job queues and scheduling algorithms, thus providing different response times and utilization percentages. Usually, high system utilization also means high average response time for jobs—and as system utilization climbs, the average response time tends to increase sharply beyond a certain threshold. This threshold depends on the job-processing algorithms and job profiles. In most cases, improving resource utilization and decreasing job turnaround time are conflicting considerations. The challenge for IT organizations is to maximize resource utilization while maintaining acceptable average response times for users.

Figure 2 summarizes the desirable features of job schedulers. These features can serve as guidelines for system administrators as they select job schedulers.

Using job scheduling algorithms

The parallel and distributed computing community has put substantial research effort into developing and understanding job scheduling algorithms. Today, several of these algorithms have been implemented in both commercial and open source job schedulers. Scheduling algorithms can be broadly divided into two classes: time-sharing and space-sharing. *Time-sharing* algorithms divide time on a processor into several discrete intervals, or *slots*. These slots are then assigned to unique jobs. Hence, several jobs at any given time can share the same compute resource. Conversely, *space-sharing* algorithms give the requested resources to a single job until the job completes execution. Most cluster schedulers operate in space-sharing mode.

Common, simple space-sharing algorithms are first come, first served (FCFS); first in, first out (FIFO); round robin (RR); shortest job first (SJF); and longest job first (LJF). As the names suggest, FCFS and FIFO execute jobs in the order in which they enter the queue. This is a very simple strategy to implement, and works acceptably well with a low job load. RR assigns jobs to nodes as they arrive in

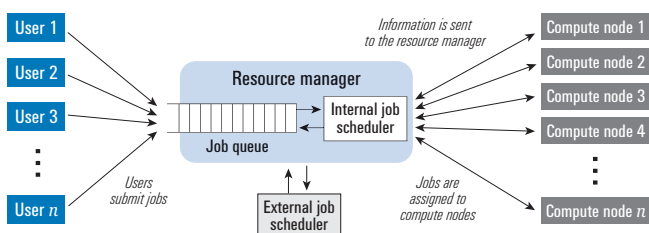


Figure 1. Typical resource management system

Feature	Comments
Broad scope	The nature of jobs submitted to a cluster can vary, so the scheduler must support batch, parallel, sequential, distributed, interactive, and noninteractive jobs with similar efficiency.
Support for algorithms	The scheduler should support numerous job-processing algorithms—including FCFS, FIFO, SJF, LJF, advance reservation, and backfill. In addition, the scheduler should be able to switch between algorithms and apply different algorithms at different times—or apply different algorithms to different queues, or both.
Capability to integrate with standard resource managers	The scheduler should be able to interface with the resource manager in use, including common resource managers such as Platform LSF, Sun Grid Engine, and OpenPBS (the original, open source version of Portable Batch System).
Sensitivity to compute node and interconnect architecture	The scheduler should match the appropriate compute node architecture to the job profile—for example, by using compute nodes that have more than one processor to provide optimal performance for applications that can use the second processor effectively.
Scalability	The scheduler should be capable of scaling to thousands of nodes and processing thousands of jobs simultaneously.
Fair-share capability	The scheduler should distribute resources fairly under heavy conditions and at different times.
Efficiency	The overhead associated with scheduling should be minimal and within acceptable limits. Advanced scheduling algorithms can take time to run. To be efficient, the scheduling algorithm itself must spend less time running than the expected saving in application execution time from improved scheduling.
Dynamic capability	The scheduler should be able to add or remove compute resources to a job on the fly—assuming that the job can adjust and utilize the extra compute capacity.
Support for preemption	Preemption can occur at various levels; for example, jobs may be suspended while running. Checkpointing—that is, the capability to stop a running job, save the intermediate results, and restart the job later—can help ensure that results are not lost for very long jobs.

Figure 2. Features of job schedulers

the queue in a cyclical, round-robin manner. SJF periodically sorts the incoming jobs and executes the shortest job first, allowing short jobs to get a good turnaround time. However, this strategy may cause delays for the execution of long (large) jobs. In contrast, LJF commits resources to longest jobs first. The LJF approach tends to maximize system utilization at the cost of turnaround time.

Basic scheduling algorithms such as these can be enhanced by combining them with the use of advance reservation and backfill techniques. *Advance reservation* uses execution time predictions provided by the users to reserve resources (such as CPUs and memory) and to generate a schedule. The *backfill* technique improves space-sharing scheduling. Given a schedule with advance-reserved, high-priority jobs and a list of low-priority jobs, a backfill algorithm tries to fit the small jobs into scheduling gaps. This allocation does not alter the sequence of jobs previously scheduled, but improves system utilization by running low-priority jobs in between high-priority jobs. To use backfill, the scheduler requires a runtime estimate of the small jobs, which is supplied by the user when jobs are submitted.

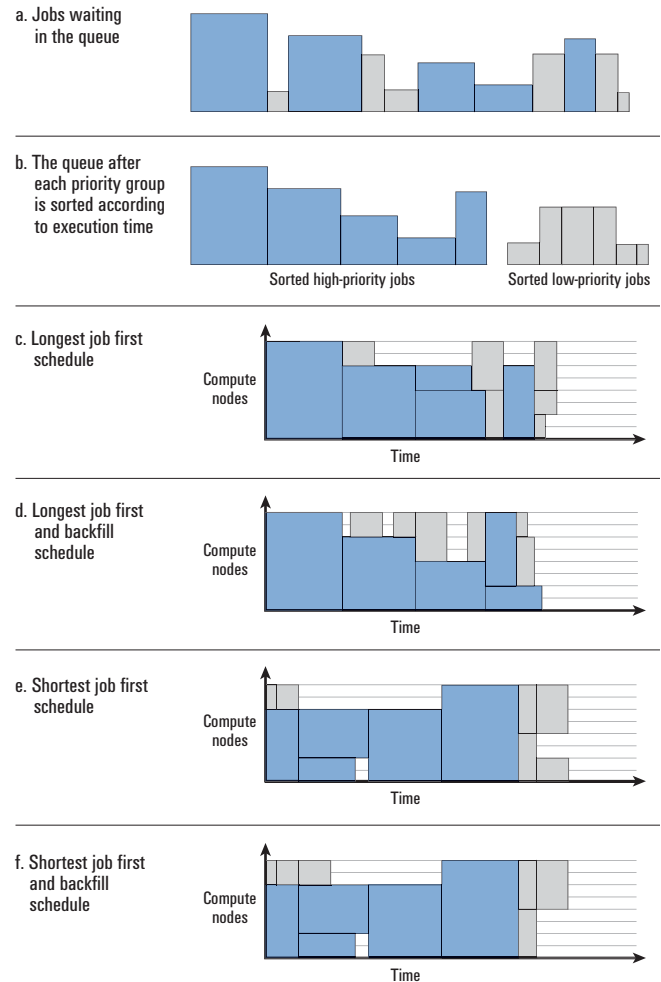


Figure 3. Job scheduling algorithms

Figure 3 illustrates the use of the basic algorithms and the enhancements discussed in this article. Figure 3a shows a queue with 11 jobs waiting; the queue has both high-priority and low-priority jobs. Figure 3b shows these jobs sorted according to their estimated execution time.

The example in Figure 3 assumes an eight-processor cluster and considers only two parameters: the number of processors and the estimated execution time. This figure shows the effects of generating schedules using the LJF and SJF algorithms with and without backfill techniques. Sections c through f of Figure 3 indicate that backfill can improve schedules generated by LJF and SJF, either by increasing utilization, decreasing response time, or both. To generate the schedules shown, the low- and high-priority jobs are sorted separately.

Examining a commercial resource manager and an external job scheduler

This section introduces scheduling features of a commercial resource manager, Load Sharing Facility (LSF) from Platform Computing, and an open source job scheduler, Maui.

Platform Load Sharing Facility resource manager

Platform LSF is a popular resource manager for clusters. Its focus is to maximize resource utilization within the constraints of local administration policies. Platform Computing offers two products: Platform LSF and Platform LSF HPC. LSF is designed to handle a broad range of job types such as batch, parallel, distributed, and interactive. LSF HPC is optimized for HPC parallel applications by providing additional facilities for intelligent scheduling, which enables different QoS in different queues. LSF also implements a hierarchical fair-share scheduling algorithm to balance resources among users under all load conditions.

Platform LSF has built-in schedulers that implement advanced scheduling algorithms to provide easy configurability and high reliability for users. In addition to basic scheduling algorithms, Platform LSF uses advanced techniques like advance reservation and backfill.

Platform LSF and Platform LSF HPC both have a dynamic scheduling decision mechanism. The scheduling decisions under this mechanism are based on processing load. Based on these decisions, jobs can be migrated among compute nodes or rescheduled. Loads can also be balanced among compute nodes in heterogeneous environments. These features make Platform LSF suitable for a broad range of HPC applications. In addition, Platform LSF can dynamically migrate jobs among compute nodes. Platform LSF can also have multiple scheduling algorithms applied to different queues simultaneously. Platform LSF HPC can make intelligent scheduling decisions based on the features of advanced interconnect networks, thus enhancing process mapping for parallel applications.

The term *resource* has a broad definition in Platform LSF and Platform LSF HPC. Resources can be CPUs, memory, storage space, or software licenses. (In some sectors, software licenses are expensive and are considered a valuable resource.)

Platform LSF and Platform LSF HPC each have an extensive advance reservation system that can reserve different kinds of resources. In some distributed applications, many instances of the same application are required to perform parametric studies. Platform LSF and Platform LSF HPC allow users to submit a job group that can contain a large number of jobs, making parametric studies much easier to manage.

Platform LSF and Platform LSF HPC can interface with external schedulers such as Maui. External schedulers can complement features of the resource manager and enable sophisticated scheduling. For example, using Platform LSF HPC, the hierarchical fair-share algorithm can dynamically adjust priorities and feed these priorities to Maui for use in scheduling decisions.

Maui job scheduler

Maui is an advanced open source job scheduler that is specifically designed to optimize system utilization in policy-driven, heterogeneous HPC environments. Its focus is on fast turnaround of large parallel jobs, making the Maui scheduler highly suitable for HPC. Maui can work

with several common resource managers including Platform LSF and Platform LSF HPC, and potentially improve scheduling performance compared to built-in schedulers.


Maui has a two-phase scheduling algorithm. During the first phase, the high-priority jobs are scheduled using advance reservation. In the second phase, a backfill algorithm is used to schedule low-priority jobs between previously scheduled jobs. Maui uses the fair-share technique when making scheduling decisions based on job history. *Note:* Maui's internal behavior is based on a single, unified queue. This maximizes the opportunity to utilize resources.

Typically, users are guaranteed certain QoS, but Maui gives a significant amount of control to administrators—allowing local policies to control access to resources, especially for scheduling. For example, administrators can enable different QoS and access levels to users and jobs, which can be preemptively identified. Maui uses a tool called QBank for allocation management. QBank allows multisite control over the use of resources. Another Maui feature allows charge rates (the amount users pay for compute resources) to be based on QoS, resources, and time of day. Maui is scalable to thousands of jobs, despite its nondistributed scheduler daemon, which is centralized and runs on a single node.

Maui supports job preemption, which can occur under several conditions. High-priority jobs can preempt lower-priority or backfill jobs if resources to run the high-priority jobs are not available. In some cases, resources reserved for high-priority jobs can be used to run low-priority jobs when no high-priority jobs are in the queue. However, when high-priority jobs are submitted, these low-priority jobs can be preempted to reclaim resources for high-priority jobs.

Maui has a simulation mode that can be used to evaluate the effect of queuing parameters on the scheduler performance. Because each HPC environment has a unique job profile, the parameters of the queues and scheduler can be tuned based on historical logs to maximize scheduler performance.

Satisfying ever-increasing computing demands

As cluster sizes scale to satisfy growing computing needs in various industries as well as in academia, advanced schedulers can help maximize resource utilization and QoS. The profile of jobs, the nature of computation performed by the jobs, and the number of jobs submitted can help determine the benefits of using advanced schedulers. 

Saeed Iqbal, Ph.D., is a systems engineer and advisor in the Scalable Systems Group at Dell. He has a Ph.D. in Computer Engineering from The University of Texas at Austin.

Rinku Gupta is a systems engineer and advisor in the Scalable Systems Group at Dell. She has a B.E. in Computer Engineering from Mumbai University in India and an M.S. in Computer Information Science from The Ohio State University.

Yung-Chin Fang is a senior consultant in the Scalable Systems Group at Dell. He specializes in cyberinfrastructure management and high-performance computing.