Open Job Management Architecture for the Blue Gene/L Supercomputer

Yariv Aridor, Tamar Domany, Oleg Goldshmidt,
Yevgeny Kliteynik, Jose Moreira*, Edi Shmueli

{yariv,tamar,olegg,kliteyn,edi}@il.ibm.com

IBM Haifa Research Laboratory
University Campus, Haifa 31905, Israel

*jmoreira@us.ibm.com

IBM Systems and Technology Group
3605 Highway 52 North, MC 030-2/D202
Rochester MN 55901, USA

Abstract

We describe an open job management architecture of the Blue Gene/L supercomputer. The architecture allows integration of virtually any job management system with Blue Gene/L with minimal effort. The architecture has several "openness" characteristics. First, any job management system runs outside the Blue Gene/L core (i.e. no part of the job management system runs on Blue Gene/L resources). Second, the logic of the scheduling cycle (i.e. when to match jobs with resources) can be retained without modifications. Third, job management systems can use different scheduling and resources allocation models and algorithms.

We describe the architecture, its main components, and its operation. We discuss in detail two job management systems, one based on LoadLeveler, the other — on SLURM, that have been successfully integrated with Blue Gene/L, independently of each other. Even though the two systems are very different, Blue Gene/L's open job management architecture naturally accommodated both.

1 Introduction

Blue Gene/L is a highly scalable parallel supercomputer developed by IBM Research for the Lawrence Livermore National Laboratory [1]. The supercomputer is intended to run highly parallel computational jobs developed using the popular MPI programming model [2, 3]. The computational core of the full Blue Gene/L consists of $64 \times 32 \times 32 = 65536 = 2^{16}$ nodes connected by a multi-toroidal interconnect [5]. A Blue Gene/L prototype of a quarter of the full size ($16 \times 32 \times 32 = 16384 = 2^{14}$ nodes) is currently rated the fastest supercomputer in the world [4]. The computational core is augmented by an I/O subsystem that is comprised of additional nodes and an internal control subsystem.

An essential component of parallel computers is a job management system that schedules submitted jobs for execution, allocates computational and communications resources for each job, launches the jobs, tracks their progress, provides infrastructure for run-time job control (signaling, user interaction, debugging, etc.), and handles job termination and resource release. A typical job management system consists of a master scheduling daemon and slave daemons that launch, monitor, and control the running parallel jobs upon instructions from the master and periodically report the jobs' state. Quite a few such systems are available: the Portable Batch System (PBS, see [6]), LoadLeveler [7], Condor, [8], and SLURM [9], are but a few examples.

Job management systems are usually tightly integrated with the multicomputers they run on. Often the architectural details (such as the hardware and the OS, the interconnect type and topology, etc.) of the machine are exposed to — and hardwired into — the job management system. Even if this is not the case, there is still the problem that the slave daemons run on the same nodes that execute the user jobs. This means that the machine architecture and the requirements of the applications put restrictions on the job management system that can be used on the particular machine: a job management system that has not been ported to the particular architecture cannot be used. On the other hand, an application that needs, for example, a particular version of an operating system can only be run if there is a corresponding port of the slave job management daemons available.

This paper describes the *open job management architecture* we developed for Blue Gene/L. By "open" we mean that virtually any job management system, be it a third party product or an in-house development, can be integrated with Blue Gene/L without architectural changes or a major porting effort. In other words, our architecture decouples the job management system from Blue Gene/L's core, thus removing the restrictions mentioned above. The particular "openness characteristics" of Blue Gene/L's job management architecture are:

1. The whole job management system runs outside of Blue Gene/L's core, keeping Blue Gene/L clean of any external software and removing the traditional dependency of the job management system on the core architecture.

2. The job management system logic can be retained without modifications. For instance, while one job management system may search for available resources each time it schedules a job, another may partition the machine in advance and match the static partitions with submitted jobs. Our architecture allows both schemes, as well as many others.

3. The job management system can access and manipulate Blue Gene/L's resources in a well-defined manner that allows using different scheduling and resource allocation schemes. For instance, a job management system can use any scheduling policy (first come first served, backfilling, etc.),[1] and any algorithm for matching resources to the job (first fit, best fit, and so on), since Blue Gene/L presents raw information

[1]At present, Blue Gene/L does not support checkpoint/restart, and therefore cannot preempt running jobs. This is not a limitation of our job management architecture: once the core support is added it will be simple to add the corresponding functionality to the job management system described here.

on its components without imposing a particular model of allocation of available resources.

In this paper we present the basic structure of Blue Gene/L's open job management architecture below, discuss its main features, and describe an implementation of a job management system based on IBM's LoadLeveler [7] for Blue Gene/L. We also describe a very different job management system based on SLURM [9] that has recently been integrated successfully with Blue Gene/L. The sequence of job and resource management operations in the two systems differs very significantly. Nevertheless the architecture is capable of naturally accommodating both schemes.

The rest of the paper is organized as follows. Section 2 describes Blue Gene/L's open job management architecture in detail. Section 3 describes our implementation of LoadLeveler for Blue Gene/L. Section 4 shows how a very different SLURM-based job management system can be integrated into the same framework. Section 5 presents some experiments and performance measurements we conducted, and Section 6 concludes the paper.

2 The Open Job Management Architecture

The core of the job management infrastructure of Blue Gene/L consists of two main components. One is a portable API we developed to interact with the Blue Gene/L's internal control system. The control system is responsible for everything that happens inside the Blue Gene/L core, from boot to shutdown. In particular, it implements all the low level operations that are necessary for managing the Blue Gene/L's resources and the parallel jobs that run in the core. The API provides an abstraction for the job management functionality, namely access to information on the state of Blue Gene/L (which computational and communications resources are busy or free, which components are faulty or operational, what jobs are running, what resources are allocated to them, etc.) and a set of job and resource management primitives (such as allocation of resources, booting the nodes, launching, signaling, and terminating jobs, and so on), while hiding the internals of Blue Gene/L from the job management system. We called the API the *Job Management Bridge API*, or *Bridge API* for short, since it provides a "bridge" between the job management system and the internal Blue Gene/L control system.

The Bridge API is essential for providing Blue Gene/L with the openness characteristics 1 and 3 described above. Moreover, the implementation of the API does not impose any restrictions on the order in which the primitives can be invoked, which is what provides property 2 (see also Section 2.2.4 below).

The second component is a special program called `mpirun`, instances of which run on a cluster of designated machines outside of the Blue Gene/L core. Each instance of `mpirun` is a "proxy" of the real parallel job running on the Blue Gene/L core, and communicates with it using the Bridge API. The slave daemons of the job management system run on the same dedicated cluster and interact only with the `mpiruns` (see Figure 1). All the job control operations such as launching, signals, and termination are passed from a daemon to the corresponding `mpirun`, and from the `mpirun` to the real parallel job it controls. All the feedback from the running parallel job is delivered to the `mpirun` and passed to the job management system. Thus, the traditional job management architecture is preserved: the job management system sees Blue Gene/L as a cluster running `mpiruns`. This provides a clean separation between the Blue Gene/L core and the job management system satisfying the openness property 1 above.

From the point of view of the slave daemons the `mpiruns` represent the real jobs. The `mpiruns` communicate with Blue Gene/L's internal control system via the Bridge API, which provides the necessary abstraction. The master daemon that manages a queue of submitted jobs[2], schedules jobs, and allocates resources, communicates with the slaves and uses the

[2]There may be more than one such queue, as is the case in Condor [8]. This is internal to the job management system and is transparently supported by our architecture.

Bridge API to query the machine state and determine which resources are available (cf. Figure 1).

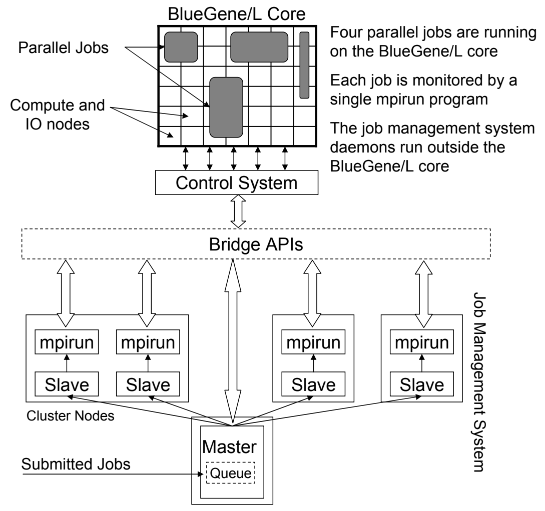

BlueGene/L Core
Parallel Jobs
Compute and IO nodes
Four parallel jobs are running on the BlueGene/L core
Each job is monitored by a single mpirun program
The job management system daemons run outside the BlueGene/L core
Control System
Bridge APIs
mpirun
Slave
Cluster Nodes
Master
Queue
Submitted Jobs
Job Management System

Figure 1: Blue Gene/L's open job management architecture

2.1 Resource Management on Blue Gene/L

A central part of the job management system is allocating resources for each scheduled job. On Blue Gene/L each job requests — and is allocated — several classes of resources, namely computational nodes, interconnect resources such as network switches and links, and I/O resources. For reasons of protection and security Blue Gene/L does not allow messages belonging to one job pass through hardware allocated to another job (this also helps avoiding contention on network resources and simplifies routing). Thus, all the resources allocated to a job are dedicated.

The set of resources that belong to a job is called a *partition*. A partition consists of a set of computational nodes[3] connected according to the job's requirements and the corresponding network and I/O resources. Before a job can run its partition must be "booted." By booting a partition we mean the process of booting and configuring all the nodes that belong to the partition, wiring all the network switches as required, and performing all the initialization required to start a job. This process is not instantaneous in general, and after issuing the appropriate command the system should monitor the partition's status until the partition is ready (cf. Section 2.2.2 below). Destroying a partition is the reverse process.

In what follows we will treat Blue Gene/L partition management as an integral part of the job management system.

2.2 The Bridge API

The Bridge API is logically divided into several major areas: the data retrieval part, the partition management part, and the job management part. The API can be called directly from C or C++ code. All the Bridge API functions return a status code that indicates success or failure of each operation.

2.2.1 Machine State Query

To allocate resources for a job, the scheduler must have access to the current state of Blue Gene/L. There is a number of accessor functions that will fetch the information on the machine as a whole or parts thereof from Blue Gene/L itself. Each accessor will allocate memory for the structure retrieved and fill it with data. This arrangement allows using the structures in name only, rather than in size, keeping the details hidden from the client. This means, however, that client code must free the memory when a data structure is no longer needed, and the API provides the corresponding functions.

[3]We assume below that a partition is a logical rectangle of nodes with dimensions that are specified by the job to which the partition is allocated. There may be other policies that specify the partitions' shapes: the jobs may specify the overall number of nodes only, letting the job management system determine the dimensions; the partition sizes may be predetermined and not related to job sizes at all (cf. Section 4 below); in general, partitions don't even have to be rectangular. All these options are supported by our architecture.

The highest-level function that provides access to the machine state is `get_BGL()` that brings the full snapshot of the current state of all the Blue Gene/L resources into the scheduler. Once the snapshot is available, a generic data retrieval function, `get_data()` can be called to retrieve various components, properties, and fields: it accepts a pointer to the queried structure, a field specification, and a pointer to memory where the query result is written. This is the only mechanism to access the machine resources, and the implementation details are not exposed to the client.

2.2.2 Partition Management

The partition management API facilitates adding and removing partitions, boot and shutdown of partition components, and queries of the partition state. The basic functions are:

1. `add_partition()` — aggregate some of the Blue Gene/L resources into a partition and add the partition to the system. This operation, as well as the complementary `remove_partition()` (cf. item 2 below) does not cause any physical side effects in the Blue Gene/L core but only creates a logical association of resources. Each resource, for instance a compute node or a network link, can belong to more than one partition as long as no more than one of the partitions is active (cf. item 3 below). Partitions may be added without reference to a particular job. This facilitates such operations as partition reservation. A job management system can limit the number of partitions that a resource can belong to, and, once `add_partition()` is called, consider the partition components "allocated" and unavailable until the partition is removed, but this is not mandatory.
2. `remove_partition()` — remove a partition from the system. This does not necessarily mean that the resources that belonged to that partition are free — some or all of them may belong to other partitions, one of which may be active. Just like `add_partition()`, `remove_partition()` does not have any physical consequences, it only removes a logical association of resources.
3. `create_partition()` — activate a partition, i.e. boot all the nodes that belong to it, connect all the switch ports according to the partition topology specification, and prepare the partition to run a job. This operation, together with the complementary `destroy_partition()` (cf. item 4 below) cause real changes in the Blue Gene/L core. In particular, any core resource, e.g. a node or a network link, can belong only to one active partition at a time, and no other partition that shares one or more resources with an active partition can be activated until the active partition is destroyed (cf. item 4 below). Executing `create_partition()` does not necessarily mean that the partition is allocated to a job — it is a policy decision left to the job management system. A partition that is already active but not allocated to a particular job can be used to run a job from the submit queue, if it fits the requirements.
4. `destroy_partition()` — deactivate, (i.e. shut down) a partition, usually after the job running in the partition terminates. This does not destroy logical associations between Blue Gene/L resources and partitions — the partition still exists and its resources remain appropriately marked.
5. `get_partition()` — retrieves the full information about a partition. This function is useful for various queries, the most important of which is checking the partition's state, for instance whether the partition is active or not.

2.2.3 Job Management

The job management API facilitates control of the jobs running on Blue Gene/L. The basic job management functions are:

1. `add_job()` — adds a job to Blue Gene/L. This is a purely logical operation that does not mean

that the job starts to run, or has been scheduled, or has been allocated resources. It is, however, a necessary step before a job can run on Blue Gene/L.

2. `remove_job()` — removes a job from Blue Gene/L. This normally happens after the job terminates, whether normally or abnormally.

3. `start_job()` — launches the job. This does not necessarily mean that the job starts running immediately, nor is it implied that the job's partition is active: the job can remain "pending," i.e. waiting for its partition to boot, and it will start when the boot is completed.

4. `signal_job()` — send a signal to a job. The job does not have to be running. However, there is not much sense in sending a signal to a job that is not running: the signal will either be ignored or an error code will be returned that can be handled by the caller.

5. `cancel_job()` — cancel a job. This is a special case of `signal_job()` used to terminate a running job. Again, there is no sense is canceling a job that is not yet running — one should use `remove_job()` (item 2 above).

6. `get_job()` — retrieves the full information about the job. The most important use is to query the job's status, for instance whether or not it has terminated.

2.2.4 Order of Operations

It is important to note that very few restrictions are placed on the order of the above Bridge API functions. Consider the following scenarios that show how few dependencies there are:

- Partitions can be created in advance, i.e. not as a result of a request from a particular job. Thus, some or all of `add_partition()`, `create_partition()`, and `get_partition()` (items 1, 3, and 5 from Section 2.2.2) may precede the scheduling cycle that will consist of job management operations (Section 2.2.3) only. For instance, the SLURM-based job management system described in Section 4 below takes advantage of this.

- A job can be started before a partition is ready and will wait for the partition to boot, providing a "launch and forget" functionality. In principle there is nothing that prevents monitoring of the status of such "pending" job, or even terminating it before it starts executing — thus calls to `add_job()`, `start_job()`, and `get_job()` (items 1, 3, and 6 from Section 2.2.3) can precede calls to functions `create_partition()` and `get_partition()` (items 3 and 5 from Section 2.2.2).

- In off-line (batch) scheduling systems, when the job list is known in advance, one can populate the system with all the needed partitions and jobs. The scheduling system will just need to start each job in its designated partition when the time comes. Thus `add_partition()` and `add_job()` can be called in the preparation phase and only `create_partition()` and `start_job()` will be called for each job.

There are some trivial exceptions, of course, e.g., a job must be added to the system with `add_job()`) before it can start via `start_job()`, a partition must be added with `add_partition()` before it can be queried with `get_partition()`, and calling `destroy_partition()` will have no effect unless `create_partition()` has been called.

In general, there are purely logical operations, such as

- `add_partition()`,
- `remove_partition()`,
- `add_job()`,
- `remove_job()`,

physical operations like

- `create_partition()`,
- `destroy_partition()`,
- `start_job()`,
- `signal_job()`,
- `cancel_job()`,

and query operations like

- `get_partition()`,
- `get_job()`.

The logical operations have no real side effects (apart from storing or erasing information), and accordingly `add_partition()` and `add_job()` can be called virtually any time. The physical and query operations can be performed only on objects that logically exist, and one cannot destroy an inactive partition of signal or cancel a job that is not running.

These restrictions will be satisfied by any sane job management system. No other restrictions are imposed on the logic of the job management system, and any job cycle model can be used.

2.3 mpirun

A typical job management system consists of a master scheduling daemon running on the central management node and slave daemons that execute on the cluster or multicomputer nodes. The master daemon accepts the submitted jobs and places them in a queue. When appropriate, it chooses the next job to execute from that queue and the nodes where that job will execute, and instructs the slave daemon on that machine to launch the job. The slave daemon forks and executes the job, which can be serial or parallel. It continuously monitors the running job and periodically reports to the master that the job is alive. Eventually, when the job terminates, the slave reports the termination event to the master, signaling the completion of the job's lifecycle.

The slave daemons are not allowed to run inside the Blue Gene/L core, any action they perform has a corresponding Bridge API function that delegates the action to Blue Gene/L's internal control system. The slave daemons run on designated machines outside the Blue Gene/L core, and instead of forking the real user job they execute `mpirun`, which starts the real job in Blue Gene/L's core via the Bridge API (as shown in Figure 1). A slave daemon needs monitor only the state of the corresponding `mpirun`, not the state of the real parallel job, while `mpirun` queries the state of the real job via the Bridge API and communicates the result to the slave daemon.

When `mpirun` detects that the job has terminated — normally or abnormally — it exits with the return code of the job. The slave reports the termination event and the exit code to the master, signaling the completion of that job's lifecycle.

On the other hand, a failure of `mpirun` is noticed by Blue Gene/L's internal control system via the usual socket control mechanisms. The parallel job the `mpirun` controlled is orphaned, loses its communication channel with the job management system, and, in general, dies. Thus, a parallel job and the corresponding `mpirun` do indeed form a single logical entity with the same lifespan.

The role of `mpirun` is not limited to just querying the state of the parallel job. It can actively perform other actions such as allocating and booting a partition for the job, as well as cleaning and halting the partition when the job terminates. Obviously, in this case the master daemon should not concern itself with these additional tasks.

This flexibility allows different job management system to optimize their performance by designating a different set of responsibilities to `mpirun`. For example, in our LoadLeveler port to Blue Gene/L (see Section 3 below), the booting of the partition is initiated by the LoadLeveler master daemon, but it is the `mpirun` that waits for the partition to boot and launches the job on it. This allows the single scheduling thread of LoadLeveler to consider the next jobs in the queue, even if the partition for the previous job is not yet ready.

Redirection of standard input and output to and from the parallel job is an additional important role of mpirun. Any input that is received on mpirun's standard input, is forwarded to the parallel job running on the Blue Gene/L core. When the parallel job writes to standard output or error, its output is forwarded back to mpirun's standard output or error respectively.

This redirection is important because it is similar to the way job management systems handle their jobs' I/O. When the slave daemons fork and execute jobs, they redirect the jobs standard input and output to files. For a Blue Gene/L job, this means that the files used for mpirun's standard output and error will actually contain the parallel job's standard output and error, and the file used as an input for mpirun will be forwarded to the parallel job.

2.4 Related Work

There are other efforts to provide open architectures for resource and job management, notably in the realm of grid computing, e.g. GRAM [10], DRMAA [11]. The main focus in those efforts lies in management of heterogeneous resources and providing consistent, standardized APIs in heterogeneous systems. While there are similarities with our Bridge API, our focus is quite different. We have a homogeneous machine, and we are interested in allowing any job management system to be integrated with it, while GRAM and DRMAA aim to allow a single job management system to operate on heterogeneous resources, or to facilitate co-operation between local schedulers and global meta-schedulers.

3 LoadLeveler for Blue Gene/L

To validate our design we implemented our own job management system on the basis of LoadLeveler, a job scheduling system developed by IBM [7], and integrated it with Blue Gene/L. LoadLeveler is a classical scheduling system that consists of a master scheduling daemon and slave daemons that launch and monitor jobs on cluster nodes. We successfully deployed LoadLeveler on a $16 \times 32 \times 32 = 2^{14}$ node Blue Gene/L prototype [4]. LoadLeveler launches and controls mpiruns on the job management cluster of 4 nodes, and the mpiruns, in turn, control the real parallel jobs running on the Blue Gene/L core.

Adapting LoadLeveler to work with Blue Gene/L included development of a partition allocator that used the Bridge API and was called by the LoadLeveler scheduler, and making the slave daemons call mpirun with the proper arguments. The difficulty of creating a partition allocator depends primarily on the sophistication of the associated algorithms, which may vary according to the needs of the customer. The actual integration with the Bridge API and mpirun is simple.

3.1 LoadLeveler Job Cycle Model

LoadLeveler does not make any assumptions regarding Blue Gene/L's workload. It is an on-line system where jobs of any size and priority can arrive at any time. The lifecycle of a job on Blue Gene/L starts when a user submits the job to the job management system. The job is placed on a queue, and the scheduler picks a job from the queue periodically and attempts to schedule it.

The scheduler's task is to allocate a partition to the chosen job and launch the job on the partition. The job carries a set of requirements that the partition must satisfy. In particular, the job may specify its total size or a particular shape (a three-dimensional rectangle of specified size $x \times y \times z$), and the required partition topology — a mesh or a torus (cf. [5]). It may happen that no partition that can accommodate the job can be found, then the scheduler may choose another job from the queue, according to some algorithm (e.g. backfilling, cf. [12]). The details of possible scheduling algorithms are beyond the scope of this paper — we focus on the general architecture of the job management system that can accommodate different schedulers.

In general, the input for the scheduler of the job management system consists of the job requirements and the current state of Blue Gene/L's resources. A partition that contains the necessary resources (computational nodes, communication, and I/O) is created, and the job is launched. For each terminating job the corresponding partition is destroyed, thus returning the resources to the system for further reuse.

Down to a finer level of details, the typical job lifecycle in Blue Gene/L looks as follows:

1. The scheduler obtains the full information on the machine state to make a decision whether to launch or defer the job and how to allocate resources for it. The information is obtained via the `get_BGL()` function of the Bridge API (cf. Section 2.2.1).
2. If suitable resources cannot be found another job may be picked from the submissions queue. Once resources are found they are aggregated into a partition, and the Blue Gene/L's control system is informed via the `add_partition()` function of the Bridge API (item 1 in Section 2.2.2).
3. The new partition (i.e. all the computational and I/O nodes that belong to it) is booted next. This is accomplished via the `create_partition()` function of the Bridge API (item 3 in Section 2.2.2). From this moment on the resources included in the partition are considered effectively "allocated" by LoadLeveler.
4. The boot is not instantaneous, so the job management system will monitor the partition's state until the partition is ready to run the job. The probing functionality is provided by the `get_partition()` function of the Bridge API (item 5 in Section 2.2.2).
5. Once the partition is up and ready, the scheduled job is added to Blue Gene/L's control system. The `add_job()` of the Bridge API (item 1 in Section 2.2.3) is used for this task.
6. The next logical task is launching the job on the prepared partition. This is achieved via the `start_job()` function of the Bridge API (item 3 in Section 2.2.3).
7. The system then keeps monitoring the running job, checking its status periodically using the `get_job()` function (item 6 in Section 2.2.3) until the job terminates, normally or abnormally.
8. Once the job terminates, it is removed from the system via the `remove_job()` (item 2 in Section 2.2.3) and steps can be taken to release the resources.
9. The partition is shut down — a task performed using the `destroy_partition()` of the Bridge API (item 4 in Section 2.2.2).
10. The now idle partition can be removed — the `remove_partition()` (item 2 in Section 2.2.2) provides the necessary facilities — and the corresponding resources can be reused for other jobs from this moment on.

Note that if a job fails the system releases the resources in exactly the same way as in the case of normal termination, except that the failure is handled as appropriate by `mpirun`. Note also that Blue Gene/L kills a parallel job if any of its processes fails for any reason. This is a characteristic of the Blue Gene/L itself, and not a limitation of the job management architecture.

The job cycle described above is depicted as a flow chart in Figure 2. The chart shows how the machine state query, the partition management, and the job management components of the Bridge API are used.

3.2 `mpirun` for LoadLeveler

In our architecture, the actual communication with Blue Gene/L's internal control system is a function of the `mpirun` proxy job. In our implementation `mpirun` handles stages 4 through 8 (cf. Section 3.1 above). In other words, LoadLeveler handles job scheduling and creation and destruction of partitions, while `mpirun` is responsible for monitoring the partitions' boot, adding, starting, monitoring jobs, and removing terminated jobs from the system.

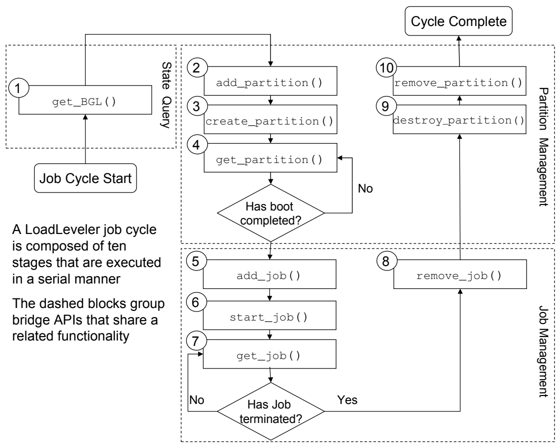


Figure 2: LoadLeveler job cycle

Figure 3 shows the same job cycle flow chart as Figure 2, showing the separation of concerns between LoadLeveler and mpirun.

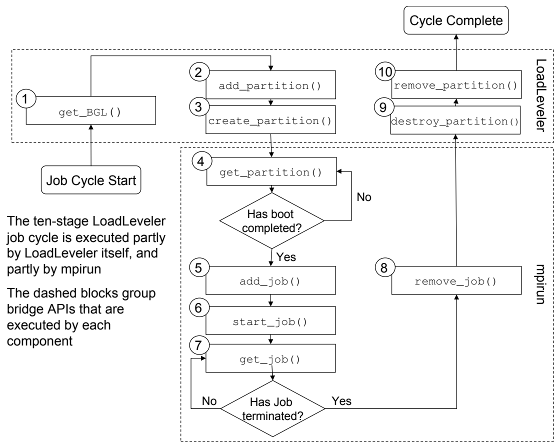


Figure 3: mpirun for LoadLeveler

This division of labor between LoadLeveler and mpirun is by no means the only one possible. For instance an implementation of mpirun may be passed the job and partition structures (or references thereof) and handle all the stages starting from 2 to the final 10 (cf. Section 3.1). Alternatively, the role of mpirun may be reduced to stages 6 and 7 only: if the scheduler is invoked each time a job terminates it can handle stages 8 through 10 before picking another job from the queue.

Our LoadLeveler scheduler is single-threaded, so any additional tasks, e.g. synchronous waiting for a partition to boot (stage 4 in Section 3.1), will prevent it from scheduling another job from the queue while it is busy. A multi-threaded scheduler will be free of this disadvantage, at the expense of added complexity. Our design lets the scheduler process a job and allocate resources to it, and delegates all the tasks performed on a job and its allocated partition, including get_partition(), to mpirun.

4 SLURM and Blue Gene/L

In this section we describe another — very different — implementation of a job management system for Blue Gene/L based on SLURM (Simple Linux Utility for Resource Management, [9]). SLURM is an open source resource manager designed for Linux clusters. Like other classic job management systems, it consists of a master daemon on one central machine and slave daemons on all the cluster nodes. SLURM for Blue Gene/L was successfully deployed and tested on a 32K node Blue Gene/L machine [14]: like LoadLeveler, it launches and controls mpirun instances that in turn control parallel jobs running on Blue Gene/L. SLURM's job cycle, however, is very different from that of LoadLeveler (cf. Section 3.1).

4.1 Partitioning and Job Cycle in SLURM

SLURM operates under a number of assumptions regarding the expected workload that lead to a very different job cycle model and hence a different resource management scheme from LoadLeveler. In particular, SLURM assumes that

- most jobs are short enough for the partition boot time overhead to be substantial (see Section 5 for relevant performance measurements);
- all job sizes are powers of 2;

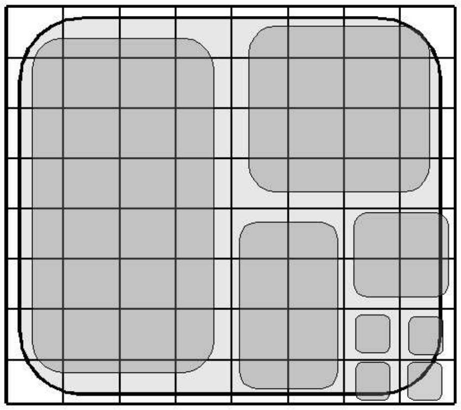
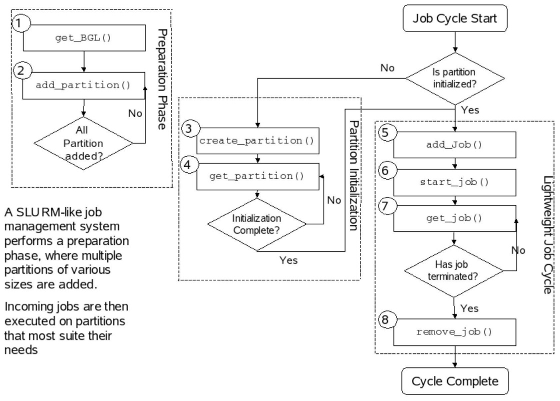


Figure 4: Possible SLURM partitions

- jobs do not request a specific shape, specifying the total size only; the system is free to choose a partition of arbitrary shape and does so, picking a partition from the prepared set;
- jobs that require the full machine have low priority and thus can be delayed (e.g. until the next weekend).



A SLURM-like job management system performs a preparation phase, where multiple partitions of various sizes are added.

Incoming jobs are then executed on partitions that most suite their needs

Figure 5: Job cycle in SLURM

SLURM takes advantage of the fact that the Bridge API allows resources belonging to different inactive partitions to overlap. Each resource can belong to a set of partitions. The system will not consider the resource allocated until a job starts in one of the partitions the resource belongs to. Accordingly, SLURM adds a set of partitions of various sizes to the system (using `add_partition()`, cf. Section 2.2.2) in advance. The partition set covers all resources and the partitions may overlap. For example, SLURM may divide the machine into partitions of size $1/2^k$, i.e. there will be partitions containing halves, quarters, eighths of the machine, etc., as well as a partition that contains the entire machine (cf. Figure 4).

SLURM maintains two job queues: one for jobs of size up to half the system and one for jobs larger than half the machine. The latter jobs are only run during specific time (e.g. on weekends). On job arrival, SLURM picks a partition for it from the existing set of partitions and boots it, if needed. Once booted, a partition will remain booted until its resources are required for another partition (i.e. the full machine partition). This scheme is designed to minimize the number of partition boots.

From a more general point of view we can say that the entire system is divided into a set of already booted partitions, and there is a job queue for each job size. Since the partitioning is done in advance the job cycle can be summarized as finding a suitable partition, starting the job, and waiting for the job to terminate, which corresponds to stages 5 through 8 in Section 3.1 above (cf. Figure 5). The lightweight job cycle is controlled by `mpirun`, while the preparation phase is performed by SLURM proper.

SLURM maintains the partition information and the job queues. For each job it chooses a partition from the existing set and starts `mpirun`. The `mpirun` in turn adds the job, starts it, monitors it, and removes it upon termination.

Note how different this picture is from the LoadLeveler model of Section 3. Note also that partition management is not a part of the normal job cycle. Only when jobs requiring the whole machine are run over weekends need partitions be rebooted. During normal operation they remain pre-allocated and pre-initialized.

size (nodes)	stages $1-3$	stage 4	stages $5-8$	stages $9-10$	total (sec)
512	1	26	14	10	51
1024	1	36	15	11	63
2048	2	38	19	11	70

Table 1: Times (in seconds) taken by different job cycle stages of Section 3.1.

Nonetheless, both SLURM and LoadLeveler are accommodated equally well by the open job management architecture of Blue Gene/L.

5 Performance

We performed some experiments on a 4096-node Blue Gene/L prototype in order to assess the performance of our architecture. We intentionally ran jobs that do nothing to isolate the overhead of adding, creating, and destroying partitions, and launching jobs. Some representative results are shown in Table 1 that lists times (in seconds) taken by different stages of the LoadLeveler job cycle (cf. Section 3.1). All the times are averages of several experiments.

These results are not final. The system is still under development, and the performance is continuously being improved.

One can note that the dependency of partition boot times (stage 4 of Section 3.1) and the rest of the operations on the size of the partition is rather weak. This is expected since the boot process and the job loading are parallelized. Resource management proper (stages $1-3$ and $9-10$) takes even less time than booting a partition. Overall, we can conclude that for jobs longer than a few minutes the resource and job management overhead (of the order of a minute, according to Table 1) is small.

6 Conclusions

We presented Blue Gene/L's open job management architecture that allows integration of virtually any job management system with Blue Gene/L. The job management system runs outside of Blue Gene/L core, and the integration does not involve any architectural changes in the system, or affect its logic.

The two main components of the architecture are the Bridge API that provides an abstraction on top of Blue Gene/L's internal control system, and a proxy `mpirun` program that represents the real parallel job to the job management system. The Bridge API imposes only the most trivial restrictions on the job lifecycle model and logic used by the job management system, and the division of labor between `mpirun` and the job management system is also very flexible.

There are two implementations of job management system that have been successfully integrated with Blue Gene/L, one based on LoadLeveler, the other — on SLURM. Blue Gene/L's open job management architecture accommodates both system equally well, despite very significant differences. Work is now under way to integrate yet another job management system, PBS.

Acknowledgments

We are grateful to Morris Jette and Danny Auble for discussions of their work on SLURM for Blue Gene/L.

References

[1] N. R. Adiga et al. "An Overview of the Blue Gene/L Supercomputer," Supercomputing 2002.

[2] `http://www.mpi-forum.org/docs/\`
`mpi-20-html/mpi2-report.html`

[3] W. Gropp, E. Lusk, & A. Skjellum. "Using MPI: Portable Parallel Programming with the Message Passing Interface," 2nd edition, MIT Press, Cambridge, MA, 1999.

[4] Top 500,
`http://www.top500.org/lists/2004/11/`

[5] Y. Aridor et al. "Multi-Toroidal Interconnects: Using Additional Communication Links to Improve Utilization of Parallel Computers" In: 10th Workshop on Job Scheduling Strategies for Parallel Processing, New York, NY, 2004.

[6] `http://www.openpbs.org`

[7] A. Prenneis, Jr. "LoadLeveler: Workload Management for Parallel and Distributed Computing Environments." In Proceedings of Supercomputing Europe (SUPEUR), 1996.

[8] `http://www.cs.wisc.edu/condor/`

[9] M. A. Jette, A. B. Yoo, and M. Grondona "SLURM: Simple Linux Utility for Resource Management." In D. G. Feitelson and L. Rudolph, editors, Job Scheduling Strategies for Parallel Processing, pp. 37 51. Springer-Verlag, 2003.

[10] `http://www-unix.globus.org/toolkit/\`
`docs/3.2/gram/ws/index.html`

[11] `http://www.drmaa.org/`

[12] A. W. Mualem and D. G. Feitelson "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling." In IEEE Transactions on Parallel and Distributed Computing, v. 12, pp. 529—543, 2001.

[13] G. Almasi et al. "System Management in the BlueGene/L Supercomputer," 3rd Workshop on Massively Parallel Processing, Nice, France, 2003.

[14] M. Jette, D. Auble, private communication, 2005. See also "SLURM Blue Gene User and Administrator Guide",
`http://www.llnl.gov/linux/slurm/\`
`bluegene.html`