The Application Level Placement Scheduler

Michael Karo[1], Richard Lagerstrom[1],
Marlys Kohnke[1], Carl Albing[1]

Cray User Group
May 8, 2006

Abstract

Cray platforms present unique resource and workload management challenges due to their scale and complexity. The Application Level Placement Scheduler (ALPS) is a software suite designed to address these challenges. ALPS provides uniform access to computational resources by masking many of the architecture specific characteristics of the system from the user. This paper provides an overview of the ALPS software together with the methodologies used during its design.

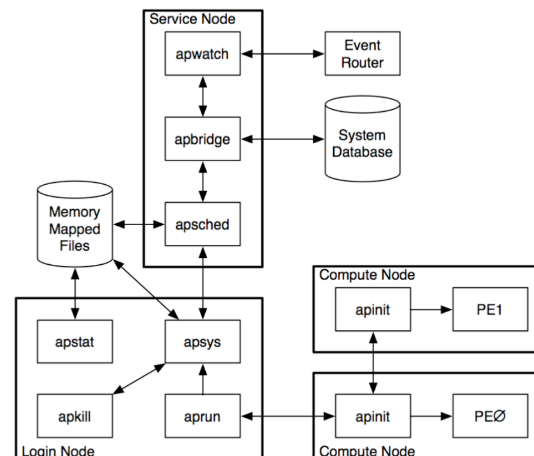
1.0 Introduction

Current and future Cray platforms will consist of large numbers of heterogeneous computational resources simultaneously running many independent operating system instances. The existing resource management infrastructure on legacy Cray platforms was not designed to operate in this type of environment. These circumstances have necessitated the design and implementation of a new software component called ALPS. The ALPS software design is intended to address both the requirements of future Cray platforms and the limitations inherent to legacy software components. The ALPS infrastructure incorporates a robust modular design to ensure extensibility and maintain a level of abstraction between the resource management model and the underlying hardware and operating system architectures. Emphasis has been placed on the separation of policy and mechanism to more clearly identify the functional requirements of the software.

It is important to note the target platform for ALPS is limited to systems running Compute Node Linux (CNL). There are no plans to replace yod/CPA on systems running the Catamount operating system.

2.0 ALPS Architecture

The ALPS architecture is divided into several components, each responsible for fulfilling a specific set of functional requirements. This model ensures a modular design that will remain maintainable and encourage code reuse. The ALPS components that run on each node of a system vary depending on the type of service the node is intended to provide. The following diagram illustrates several of the ALPS components together with their interactions:



The ALPS software components communicate using the XML-RPC protocol. The protocol provides an extensible language that may easily be enhanced in future revisions of the software to support additional message types and services. In addition, ALPS makes use of memory mapped files to consolidate and distribute data efficiently. This reduces the demand on the daemons that maintain these files by allowing clients and other daemons direct

[1] Cray Inc., Mendota Heights, MN, USA, [mek|rnl|kohnke|albing]@cray.com

access to data they require.

3.0 The Application Lifecycle

The aprun client represents the primary interface between the user and their application. Upon invocation, the user specifies command line arguments that convey the resource requirements for the application together with the location of the executable binary file. The aprun client parses the command line arguments and contacts the local apsys daemon running on the login node. The apsys daemon then forks an apsys agent to handle the remainder of the request.

Once in contact with the apsys agent, the aprun client relays the information it has collected from the user. The apsys agent then forwards this request to the apsched daemon to obtain a placement list that is consistent with the resource requirements the user specified. If a suitable set of available resources cannot be identified, this process will be repeated periodically until one exists. Once the target resources have been identified, apsched generates a placement list, registers a reservation, and relays it to the aprun client. The reservation ensures the resources assigned to the user may no longer be committed to another application.

The aprun client then contacts the apinit daemon running on the first node assigned to the application (PEØ). The apinit daemon forks an application shepherd to manage the process(es) that will execute on the node. The aprun client then transmits the placement list for the application and (optionally) the executable binary data to the shepherd. The shepherd may contact additional apinit daemons running on other nodes assigned to the application, relaying the placement list to each. Each apinit daemon contacted forks an application shepherd to manage the process(es) local to the node. This process continues until an application control tree is established between the shepherds running on each node assigned to the application. The radix of the control tree is set to eight by default, but may be configured by the administrator to a value between two and thirty-two.

Once the control tree has been established and the placement list communicated to each shepherd, the application initialization process begins. The initialization process recreates the user's environment on the login node for each of the PEs of the application. Depending on the architecture of the nodes, initialization may also entail allocation of distributed memory for the application. Once initialization is complete, control is passed to the application.

While the application is running, each shepherd monitors the PEs of the application. If the aprun process or one of the PEs catches a signal, it will be propagated to each PE through the ALPS control network for the application. In addition, the aprun client manages the standard input, standard output, and standard error streams for the application. Any characters received on the standard input stream are forwarded to PEØ of the application. Any characters generated by the PEs of the application are sent to the aprun client and returned on the appropriate stream to the user.

When an application exits, either normally or due to error, ALPS must ensure that all resources allocated for the application are surrendered. This may involve deallocation of distributed memory and the forceful removal of stray processes. Once cleanup is complete, the aprun client exits.

4.0 ALPS Clients

The ALPS clients provide the user interface to ALPS and application management. They are separated into four distinct areas of functionality:

1. Application submission (aprun)
2. Application monitoring (apstat)
3. Application signaling (apkill)
4. Batch system integration (apbasil)

Each client is described in more detail in the following subsections.

4.1 The aprun Client

As previously stated, the aprun client represents the primary interface between the user and their application. Its primary function is to submit applications to the ALPS system for placement and execution. The aprun client is responsible for parsing command line arguments, forwarding the user's environment, forwarding signals, and management of the standard input, output and error streams. It is also capable of supporting both automatic and manual node selection.

4.2 The apstat Client

The apstat client relays status information from ALPS to the user. The information may include data describing current resource availability, reserved resources, and running applications. It is not necessary for apstat to generate a request that is passed to a daemon for processing. Instead, apstat is able to utilize the memory mapped files that the daemons maintain to acquire the data it needs to generate a report for the user. This capability reduces the demands on the ALPS daemons, allowing them to more effectively service applications

4.3 The apkill Client

The apkill client is responsible for delivering signals to applications. When apkill is called, it parses the supplied command line arguments including the signal type and application ID. The client contacts the local apsys daemon, which generates an apsys agent to manage the remainder of the transaction. The agent locates the login node on which the aprun for the target application ID resides by consulting the memory mapped files. If the aprun is running on the local node, the apsys agent will deliver the signal itself. If not, the apsys agent contacts the apsys daemon on the target node to proxy the request. Once delivered, a response is returned to the apkill client indicating the result of the operation.

4.4 The apbasil Client

The apbasil client represents the interface between ALPS and the batch system. This client implements the Batch and Application Scheduler Interface Layer (BASIL). The BASIL protocol will be discussed in more detail in a later section. The client acts as the gateway between ALPS and external resource managers that implement scheduling policies for the system.

5.0 ALPS Daemons

The ALPS daemons implement a variety of services required to support application submission, placement, execution, and cleanup on the system. Each daemon addresses a specific service. The daemons work in concert with the ALPS clients and system components to manage applications and computational resources.

5.1 The apbridge Daemon

In order for the apsched daemon to make sensible placement decisions, apsched must have a description of the hardware on which it will place jobs. This includes the quantity and speed of specific processors as well as some elementary topological information. Furthermore it requires on-going status information, to know which PEs are available and which are down (or otherwise unavailable).

The apbridge daemon is the *bridge* from the architecture *independent* ALPS software to the architecture *dependent* specifics of the underlying system. It is the layer between apsched and the specifics of a platform. The apbridge daemon queries the system database (SDB) to collect data on the hardware configuration and topology and supplies it to the apsched daemon.

Ongoing status information is supplied to apsched from apbridge, though apbridge gets the status changes via the apwatch daemon (described below).

This modular breakdown has allowed for development and testing to occur on systems distinct from the final hardware architecture through the use of a simple test harness at a few key points. Furthermore, we have run apbrige in conjunction with synthetic hardware databases with configurations from a few dozen to over 15,000 processing elements.

5.2 The apwatch Daemon

Ongoing status information is supplied to apsched from apbridge, but apbridge gets its event information from apwatch. The apwatch daemon registers with the host-specific mechanism for receiving events. Those events, when received, are translated into an architecture-neutral format and passed to apbridge for further processing and eventual delivery to apsched.

For development and testing purposes we have been able to substitute a script-based event replay mechanism and a random event generator to simulate individual node up/down events, clusters of such events on adjacent nodes, and larger numbers of events on unrelated nodes.

5.3 The apsys Daemon

The apsys daemon is a local privileged program that provides access to apsched from ALPS client programs. There is one apsys daemon per login node. The apsys daemon writes pending application status information into a file for display by apstat. During ALPS startup, the apsys daemon attempts to recover connections to aprun clients that had previously been running applications. The apsys daemon is responsible for notifying apsched about resource reservations to be freed.

The apsys daemon forks an apsys agent child to process incoming requests from ALPS client programs. The apsys agent child retains a persistent socket connection to aprun for the lifetime of the aprun program. All other socket connections are transitory. All apkill signal requests are either handled locally or forwarded to the appropriate remote apsys daemon for processing. The signal is delivered to aprun who then forwards the signal over the ALPS control network for delivery to the application. Resource reservation messages from apbasil are also forwarded to apsched for processing.

5.4 The apinit Daemon

An important design principle of the control mechanism used to manage the compute nodes is that the daemon on every node is independent and every daemon behaves in the same way. With very few exceptions we have been able to apply this principle to apinit.

Every compute node has an apinit master daemon started as part of the boot procedure. A known privileged listening port is opened and the master waits for a connection to be requested. The master daemon initiates all new activity on a compute node, typically by forking a child process and transferring responsibility to that child. The major functions are launching applications and creating the tool helper environment.

An application first comes into existence on a compute node when aprun connects to the apinit daemon on the first node of an application's allocated node set. Aprun sends a launch message containing all of the information the compute nodes need to launch and manage the new application to the daemon. The master daemon extracts just enough information from this message to construct a control structure that it will use to maintain knowledge of the application on its compute node. The master daemon forks a child (called apshepherd here to distinguish it from the master daemon) dedicated to managing the specific application on that compute node. The message and the socket connection from aprun are transferred to apshepherd. The master daemon continues to listen for new messages and monitor all of the apshepherd instances on its compute node.

Apshepherd takes a role on the compute node that is very similar in many respects to that of the login shell on support nodes. Each PE of the application assigned to its node runs as a child of apshepherd. Apshepherd establishes the user's identity and provides the stdin (PEØ only), stdout and stderr connection to the remote aprun. Apshepherd initiates the application after performing architecture specific setup functions to prepare the environment for the application to run.

Before the application can be brought into execution all of the compute nodes assigned to the application must be contacted. To scale to large numbers of nodes, apshepherd divides the list of un-contacted destination nodes into two to thirty-two parts depending on the fanout factor included in the message. Each destination is contacted by apshepherd with the original launch message altered to tell the destination node which portion of the placement list it is required to launch and forward. Every apshepherd does the same, which generates the control network tree for the application.

Aprun is the root of the control network and the apshepherd instances running on each assigned compute node make up the branches. The control tree exists for the life of the application. Synchronization, signal, stdout, stderr and low-level interface traffic flow on the control network. The control network is heavily used during application setup and teardown to orchestrate and synchronize these operations. While an application is running only signals, stdout, stderr and low-level interface messages use the control network. The low-level interface provides a way for a PE to send requests to the local apshepherd. Some of those requests make it possible for one PE to send a signal to another PE or group of PEs within its application for application management purposes.

An application is brought into execution by each apshepherd once the control network is established and all required synchronization is complete. The details of inter-node synchronization are architecture dependent and

depend upon the amount of initialization that is required to establish the proper application execution environment.

When the application is in execution apshepherd is mostly asleep waiting for events on the control sockets and pipes or for signals caused by the termination of parts of the application running on its node. The primary responsibility of apshepherd when the application is in execution is to manage exit events. There are two ways an exit event can be handled. The controlled exit happens when a PE tells apshepherd it is about to terminate and then does so. That is considered normal application termination and does not result in any preemptive action. An uncontrolled exit happens when a PE exist without telling apshepherd it intended to do so. This is considered a fatal application error and will result in killing all of the application.

An entire application can be killed only by a signal message from aprun. In the uncontrolled exit case, the apshepherd noticing the event composes a signal message and sends it to aprun. Aprun receives and retransmits the kill signal from the root of the tree. Each apshepherd in the control tree acts on the signal message and forwards it to the apshepherds it controls. In this way the message reaches all of the compute nodes assigned to the application.

The other major control event is when one of the network connections drops. Each apshepherd is sensitive to its control network connections. If any of them close, apshepherd will kill all local PEs and then terminate. In this way it closes all of the portions of the control tree it controls. Each apshepherd reacts similarly so the control network is dissolved. Aprun detects this when its connection to the apshepherd on PEØ closes. This is considered a fatal error, which occurs only if a compute node crashes or an apshepherd aborts.

5.5 The apsched Daemon

Apsched manages memory and processor resources associated with applications running on compute nodes. Some compute node architectures have additional requirements typically involving distributed memory and the interconnect between compute nodes.

Apsched does not enforce policy. Policy enforcement belongs to the workload manager or other system components. Apsched only guarantees the correctness of application placement and may optimize placement for performance to the extent possible. The output of a reservation or placement request is a specific resource list. There is no assumption of resource uniformity over the compute nodes. Each compute node must be treated independently since there is no guarantee that any two nodes have exactly the same set of available resources at a given time. It is vital that memory usage on the compute nodes be strictly enforced and managed. There is no virtual memory or swapping generally available on compute nodes, as one would expect in a virtual memory system. Memory oversubscription is therefore typically fatal to either the offending application or some other application running on the compute node.

Although the process of placement can be complex, apsched has a simple interface to its clients. A privileged socket is opened when apsched starts to be used for configuration. The apbridge daemon will send configuration information to apsched initially. The socket connection to apbridge is maintained for the lifetime of apsched. Event messages from apbridge indicating changes to the state of compute nodes keep apsched informed of the compute node resources available at all times.

When initial configuration is complete apsched performs any required recovery and identification tasks, creates its shared memory files, and populates them with the initial system state. These files are windows into internal scheduler information needed by apstat and other clients for display or decision making purposes. After this is complete, apsched opens its scheduling listening port and begins to accept placement, reservation and other messages.

In the simple case of an interactive user typing an aprun command a placement message originating from aprun will arrive at the scheduler. A preliminary check for obvious errors or impossible requirements is made. If the request seems possible to handle, apsched begins a sequence of placement operations whose goal is to deliver a placement list to aprun that will be forwarded to the compute nodes. This process will result in a placement list or a message indicating the request could not be placed.

If the request was satisfied, apsched updates its internal and shared information so the resource usage is retained and becomes visible. If the resources could not be allocated, a "try again" message is returned with information about why the request failed. Apsched has no queues or memory of waiting requests. Other players in the ALPS complex deal with pending

requests and other things that require long-term memory. When an application terminates, an exit message is sent to apsched so it will release the resources reserved for that application. The simple case is called *atomic placement*. Interactive aprun commands will use this form of placement.

Workload managers can use the confirmation/claim form of placement. In this case the workload manager will confirm the application resources before the batch job is initiated. If the confirmation is successful, the job will start; if not, it will be requeued by the workload manager and tried later. With resources confirmed prior to the batch job being initiated, aprun placement requests within the batch job will operate in the *claim* mode. When the placement request arrives at apsched from aprun it is matched with the prior confirmation and a placement list is delivered immediately. Multiple aprun requests may claim the confirmed resources during the life of the batch job.

In confirmation/claim mode the workload manager must cancel the resource confirmation when the batch job completes for the resources to become generally available.

An important implementation rule in apsched is that the scheduler cannot stall. If there is any activity that might take an arbitrary length of time, that work is delegated to a forked child so that apsched can continue without delay. This is important because apsched is the single place that all placement requests must pass through. If it stalls, new applications cannot be launched.

6.0 Application Helpers

ALPS provides compute node launch assistance to login node application tools which require a helper program to run on the same set of compute nodes as an application. Examples of these application tools are debuggers and performance analysis programs.

These login node tools and their compute node helper programs link with an ALPS library. The library provides interfaces for the helper program to launch on the application compute nodes. It also provides interfaces to obtain application placement list information, gather ALPS control network tree information, perform startup and exit synchronization between the tool and the application, and determine application PE information per compute node.

7.0 Batch System Integration

Historically, integration between batch systems and lower level system resource managers has been an afterthought in the design of both components. With the design of ALPS, this is not the case. The apbasil client acts as the gateway between ALPS and third party batch systems. The communication between apbasil and the batch system uses an interface called the Batch and Application Scheduler Interface Layer (BASIL). The BASIL protocol is implemented in XML-RPC to maintain extensibility and backward compatibility in future revisions. The initial BASIL protocol implements three primary functions:

1. Inventory
2. Reservation creation
3. Reservation cancellation

When a user submits a job to the batch system, the batch scheduler must determine whether sufficient resources exist to run the job. To accomplish this, it must obtain a current picture of available and assigned resources. BASIL provides this capability through its inventory interface, providing detailed information in XML format that may be quickly parsed by the batch system. Once the data is parsed, the batch scheduler may use the data to schedule one or more batch jobs for execution.

Once a batch job has been scheduled, the batch system must initialize the job on one or more login nodes of the Cray system. During initialization, the batch system must create an ALPS reservation for the job to ensure resources will remain available throughout its lifetime. During execution of the batch job there may be times when the resources it has been assigned are not being fully utilized. The reservation prevents ALPS from creating conflicting resource assignments.

During execution of the batch job, there may be several calls to aprun to launch applications on the reserved set of resources. ALPS recognizes when an application launch originates from a batch job and assigns resources from the pool that had been previously reserved.

Upon completion of the batch job, the batch system must make a final BASIL request to cancel the reservation for the job. This frees the reserved resources, making them available for reassignment.

8.0 Summary

The ALPS software suite is comprised of multiple clients and servers, each intended to fulfill a specific set of responsibilities as they relate to application and system resource management. The design of the ALPS system is intended to address the needs of both current and future Cray platforms. Special attention has been paid to ensure the design remains modular, extensible, scalable, and maintainable. Features of ALPS including apbridge, the tool helper interface, and the BASIL protocol help to maintain a separation between ALPS and external components. The implementation of a reservation mechanism ensures availability of resources to both batch and interactive users. The elegance of the ALPS design is due in large part to its simplicity.