# Allocation and Scheduling for a Computational Grid

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Engineering

by

Carel A. Lewis

December 2001

Advisor: Dr. Walter B. Ligon III

December 14, 2001

To the Graduate School:

This thesis entitled "Allocation and Scheduling for a Computational Grid" and written by Carel A. Lewis is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Engineering.

Walter B. Ligon III, Advisor

We have reviewed this dissertation
and recommend its acceptance:

Ron Sass

Adam Hoover

Accepted for the Graduate School:

## Abstract

Parallel computers are becoming increasingly important for modern engineering and scientific simulation. A successful type of parallel computer is the Beowulf cluster. These clusters emphasize using many commodity processors in parallel to try to achieve the performance levels of more expensive supercomputers. A growing research area is in connecting multiple Beowulfs into a Computational Grid to create a distributed system of clusters.

With multiple resources distributed across a system, an effective way to combine their processing power must be examined. A way to transfer the ownership of the remote processors on a cluster to a separate cluster on the Grid is needed to be able to combine the resources into a larger computer. This paper explores the different aspects of scheduling and allocating resources in a Grid of Beowulfs. It also describes the design of a specialized node allocation mechanism for a such a cluster. This mechanism integrates with current Beowulf software which allows the user to see a "single" computer instead of a distributed system.

The techniques used by this new mechanism to borrow and return nodes between separate clusters is discussed, as well as methods for order of contact and ownership of nodes. This paper also surveys a few different allocation and scheduling tools that are currently used in parallel computers. By using these ideas and the new mechanism, the organization and the computational power of a Grid of Beowulfs will be improved.

# Dedication

To my friends and family, for their love and support through the last five years. Without your help and patience, none of this would have been possible.

Especially to my parents, each of whom has given me love, knowledge, and a special trait that has helped me to succeed. To my mother, who has given me her strength and commitment to excellence. To my father, who has given me his adaptivity and technical skills. To my step-father, who has helped me to develop management and organizational skills. To my step-mother, who has helped me to develop persistence and patience.

## Acknowledgments

I would like to especially thank Dr. Walt Ligon for his guidance and support. Your commitment to research has helped me to expand my knowledge in many directions. I would also like to thank my committee members and Dr. Daniel Stanzione for their direction and advice.

# Table of Contents

# List of Figures

Figure Page

# Chapter 1

# Introduction

The amount of information gathered around the world, from satellites to hospital research, is increasing at a dramatic rate. This data could hold the answers to weather forecasting, environmental situations, or even the early detection and treatment of genetic diseases. Processing this data is becoming increasingly difficult, due to the enormous amounts available for different types of testing. Computers have become key in getting results quickly. Even so, many different algorithms can be performed on the same data, causing an even greater need for computational power. Demand for faster computers has driven research in processor speed and caused an almost exponential increase in performance, but this increase is still not enough. By finding ways to increase the computational power of existing machines, researchers can process the available data with much greater speed.

Several architectures exist that use computers in parallel to process information. By scheduling jobs and allocating these computers in efficient ways, several users can be running processes at the same time. The more competent the scheduling and allocation services are, the less time users must wait for answers. If these tools were capable of combining several existing parallel computers into one large computer, the available computational power for a single user would allow an increase in computational research in almost any area or field.

## 1.1 Computational Grids

High performance computing is filling the gap between the processing speed required and what is currently available. On large data sets that require certain computations, parallel machines can almost achieve a linear speedup over single processor computers. For a time, this power was only available to a limited few who could afford large, expensive supercomputers. However, in the last few years a movement has been made to use many commodity processors in parallel to try to achieve the same performance levels. The original project was named Beowulf and was started by Tom Sterling and Donald Becker in 1994 [18]. The first cluster contained sixteen processors and was created at NASA's Goddard Space Flight Center.

A Beowulf cluster is a grouping of computers, each with its own processors, memory, hard drives, and network cards. Normally these computers use the Linux operating system and Ethernet switches for communication. On each cluster there exists a "head node" that is usually connected to an external network. This head node contains connections to the "remote nodes", or the nodes that would do the actual computation of a program. Each node in the cluster is dedicated to the cluster and since the remote nodes are not subject to the external network, their performance corresponds only to the program currently executing.

Beowulfs have grown in popularity and are found in a wide range of places. Their affordability and computational power encourages experimentation. A problem occurs though, when a single user tries to manage a large number of processors. Scalability becomes an issue, because of the number of concurrent processes that are actively communicating with each other. To check the status of each computer, the user must log on to each node to find out any information. Debugging becomes especially difficult when communication between computers fails for unknown reasons. Special software has been developed to simplify this process.

The Scyld operating system is designed to allow a user to see a Beowulf as a single computer, simplifying programming and administration. The core of the Scyld system is a distributed process space created by a set of daemons, called Bproc. Processes are started on the head node and migrated out to the remote nodes, for actual computation. A "ghost" process can still be seen on the head node that mirrors all information about the actual process on the remote node, which allows simplified management. This single image view of a cluster computer and the success of this system encourages the development of other user software specifically designed for Beowulfs.

More recently researchers have been trying to find a way to combine resources that are spread out over great distances to allow for even greater computational power. These resources include a variety of hardware including extremely powerful supercomputers, databases, networks, and clusters. These groupings are known as Computational Grids and programs for these arrays usually include a great deal of data partitioned over the Grid. A Computational Grid is similar to a Power Grid, where access to the Grid is available at many points. A user can plug in anywhere on the Grid to get the necessary power, whether it be computational or electrical. Timing, security, and stability become issues when dealing with the extensive size and heterogeneous nature of such a system.

Simpler versions of Computational Grids are in development. These "Mini–Grids" try to use a localized setting as an advantage for communications, instead of relying on slow Internet connections to transfer data and process information. They also contain a homogeneous mixture of resources consisting of Scyld Beowulfs. These Mini–Grids can be illustrated as one large cluster that is broken into several smaller

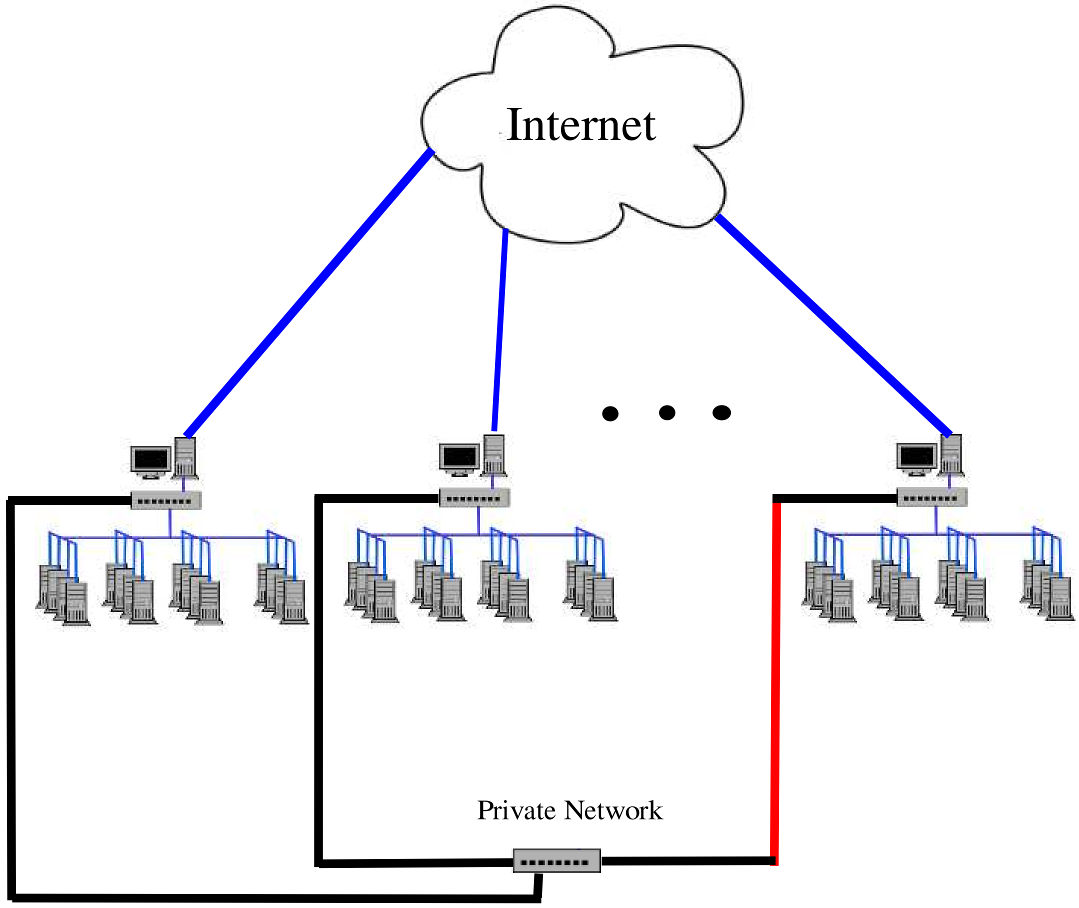


Figure 1.1: Mini–Grid Architecture

sub–clusters, which helps to simplify administration, as seen in Figure 1.1. Each sub–cluster contains its own head node, and no hierarchy exists between them, so each one can be used as a full cluster. The sub–clusters' remote nodes are connected in the background with communication devices on a private network. This allows any remote node to belong to any head node. In this way the entire Mini–Grid architecture can be combined to be one massive Beowulf computer.

## 1.2 Ideal Mini-Grid Capability

The capabilities that should be available in a Mini-Grid can be described with two example allocations, or jobs with different allotments of nodes. The first is on a single cluster and concerns the basic functionality necessary in an allocation tool. The second is an example on our target architecture the Mini-Grid with four separate clusters. Three of those clusters are owned by different groups, and the fourth cluster is a node pool that is available to all the other clusters.

### 1.2.1 One Cluster Example

The first example is the use of a single cluster. The architecture used for this demonstration contains thirty-two nodes and can be seen in Figure 1.2. This example shows multiple users accessing a single cluster at the same time.

The first job to arrive is a Shared allocation request for eight nodes. The second job that is filled requests Exclusive use, because it is time-restrictive, and requests twenty nodes. These two requests are fulfilled with out any problems.

The next request asks for sixteen nodes, but only twelve are available for shared mode, eight of which already have a user on them. At this point the allocator checks to see if the user will except fewer nodes. If not, an error occurs and a acknowledgment

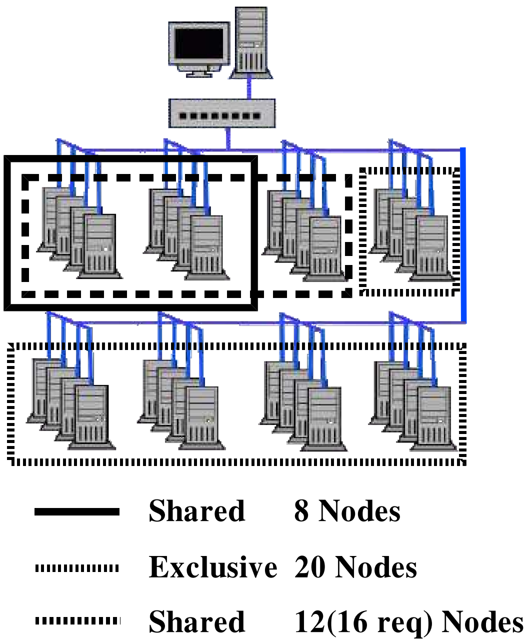


Figure 1.2: Example of allocation on one cluster.

is sent telling the user that the nodes are not available. However, for this example, the flag is set and the nodes are allocated.

Several other Shared jobs could still access twelve nodes. No Exclusive jobs could be started and no nodes could be borrowed though, until nodes became Free.

### 1.2.2 Mini-Grid Example

The second example shows several users on each each cluster of a Mini-Grid. This illustration can be seen in Figure 1.3. The architecture for this example is taken from a testbed that is discussed further in Section 4.1. All options are represented by different jobs on the cluster. There are three groups that have access to this Mini-Grid, the Center for Advanced Engineering Fibers and Films (CAEFF), the Clemson University Genomics Institute (CUGI), and the Parallel Architecture Research Lab (PARL).

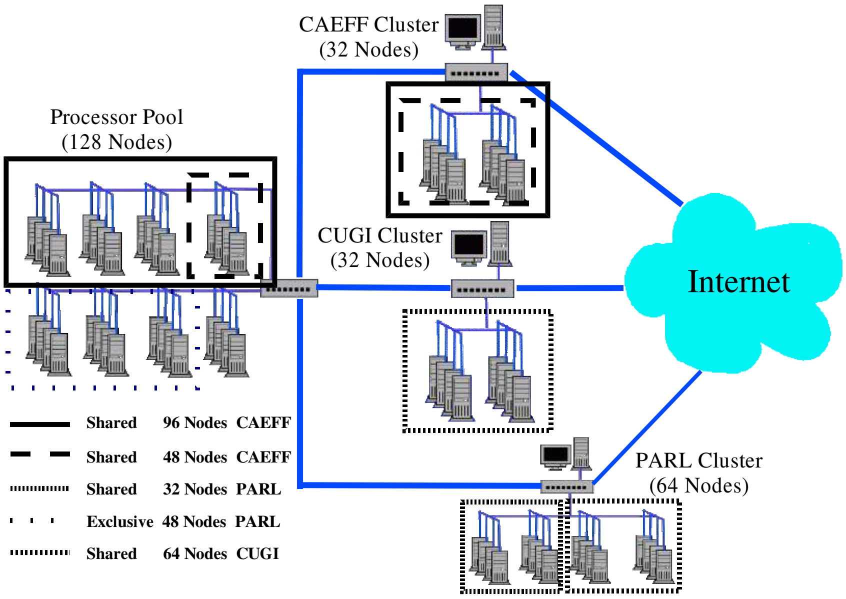


Figure 1.3: Example of use on Mini-Grid.

This example consists of five jobs. The first arrives on the CAEFF cluster requesting 96 nodes in Shared mode. This cluster only contains 32 nodes, and must borrow nodes if any are available. The first place every cluster queries for borrowed nodes is the Node Pool. This Pool is used only to provide the other clusters with spare nodes. Assuming the borrowing is allowed, the cluster accesses the other nodes, and allocates those nodes once the request can be filled.

The next job is on the same cluster. This request is for 48 Shared nodes, which can be filled on the local nodes and the currently borrowed nodes. Since the nodes have already been borrowed, the reboot delay is not encountered.

The third job is on the PARL cluster and is requesting 32 Exclusive nodes. It can be filled on the local cluster. The next job is requesting 48 Exclusive nodes, but it requests that all of the nodes are on the same cluster. This request cannot be filled on the local cluster, so the other allocators are queried and the allocation is completed on the Farm.

The final job is on the CUGI cluster and is requesting 64 Shared nodes. This request has set to take the most nodes available first, instead of by the priority of the clusters. All the clusters are queried and it is discovered that the PARL cluster can loan the entire amount needed. The CUGI cluster borrows from it before the Pool and only has nodes spread across two clusters instead of three.

These two examples show the primary implementation requirements that are necessary in the allocation tool. However, these are just two examples of the wide range of available allocations of nodes that should be available, and the efficacy of such functions should be shown.

## 1.3 Allocation and Scheduling Issues

Allocators and schedulers are incorporated into almost every computer. Schedulers organize jobs to be run on machines in many different ways, each trying to deal with separate issues such as speed of completion, priorities, and efficiency. A scheduler says when, where, and how long each job will be executed. While schedulers are trying to set the order of jobs, allocation tools are trying to set the resources being used. A couple of the main issues in allocation are locality, speed of communication, and security policies (which users are allowed which resources). These tools vary in use and are usually hardware specific.

Allocation and scheduling problems arise with each new architecture developed. Many tools dealing with particular problems in clusters are already in development. Such problems addressed are choosing which nodes to use, balancing the number of users on each node, and allowing the cluster to be reserved for large jobs. By changing the architecture of a cluster, and creating the Mini–Grid, new problems arise. One of the main aspects of the Mini–Grid is its ability to be combined into one large cluster, however no allocator currently exists that performs this function.

Using Scyld Beowulf and currently available allocation tools, the only way to combine the nodes into one cluster is manually, changing configuration files and rebooting the Bproc daemons. A new way to dynamically and transparently "borrow" nodes between the clusters needs to be developed in an allocation tool. This service must:

- Maintain the state of nodes distributed across the grid,
- Allow for multi–user and single–user access to nodes,
- Provide a mechanism for transferring nodes between clusters,
- Provide borrowing and allocation options for policy implementation in a scheduler,
- Be transparent to the user,
- Enforce usage policies,
- Have an efficient implementation, and
- Be able to integrate with existing software.

While a few tools offer some of these options, none work with the Scyld operating system and allow nodes to be transferred from one cluster to another.

## 1.4 Proposed Solution

We propose the design of a new Beowulf allocation tool (Balloc) which would allow nodes to be transparently "borrowed" between locally connected clusters and provide extensibility and simplicity of use. This tool would fulfill the previously listed requirements.

This new service would contain a structure like that in Figure 1.4. The user would have several options to access the allocator, including MPI scripts, function calls, and a control manager. The Balloc tool would consist of daemons running on each cluster listening on communication ports. These daemons would have the ability to send information between each other and transfer nodes between the clusters.

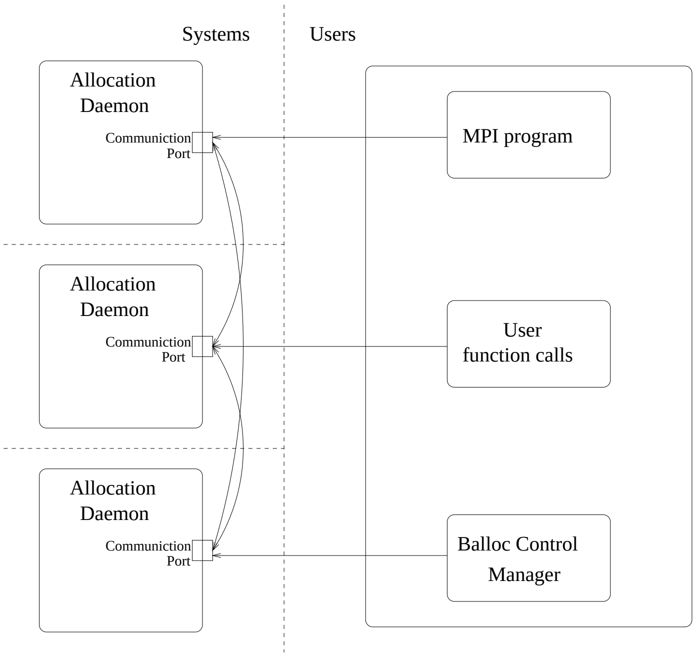


Figure 1.4: Proposed structure for a Beowulf allocation tool.

## 1.5 Outline

In this thesis, we start with background about differences between schedule and allocation services. Two examples of schedulers are discussed to give background on necessary interfaces for allocation software. We then review two previous allocators. The first is of an early supercomputer allocator for the Connection Machine CM5. The second service is the allocator used in the Globus Toolkit, the Globus Resource Allocation Manager (GRAM) in coordination with the Dynamically Updated Request Online Co–Allocator (DUROC), designed for a Computational Grid. Each of the allocation tools are analyzed for possible use in the Clemson University Mini–Grid.

Next, the development of a new allocation tool designed for our architecture and operating system is discussed. The success of this tool is based upon the requirements for our structure. This daemon reaches the listed requirements, using a separate API for the user, functions implemented in Scyld for node manipulation, and two separate databases for owned and borrowed nodes.

The Beowulf Allocator, or Balloc, is examined in two separate discussions. The first discussion is of the basic functionality of Balloc. This functionality includes an examination of the user API and describes the possible Balloc function calls available. The description of the user API also includes the incorporation of Balloc into an existing parallel programming language environment called the Message–Passing Interface, or MPI. Administrators for the Mini–Grid also need a Balloc control manager application, that would allow easy examination of the state of the Grid.

The second discussion is a description of the actual Balloc daemon implementation. The basic uses of an allocator that would exist on a single cluster are incorporated, such as multi–user or single–user access and an organized way of keeping track of set information. Also in this Section, the borrowing functions needed for the Mini–Grid are described. The implementation of borrowing and loaning nodes between clusters is examined, with the manipulation of databases and Scyld operating files.

Once the implementation is discussed, actual experiments are performed, and the success of the system is evaluated. The experiments are described using the `mpirun` function call, with some test parallel programs. Timing for these experiments is also evaluated. With file manipulations occurring whenever nodes are borrowed and reboots necessary, timing becomes a difficult issue. Evaluation of this requirement and its fulfillment are examined, along with the success of the allocator for the other Mini–Grid requirements.

Finally, we conclude whether or not Balloc meets the design goals and is capable of borrowing nodes on a Mini–Grid. We then examine future work related to Balloc.

# Chapter 2

# Related Work

In this chapter we discuss an overview of schedulers and allocators. The interfaces to schedulers are illustrated and several examples of allocators are examined.

Schedulers and allocators are used as a system to allow organized access to computer resources. This access is restricted to the programs entered into a queuing manager. The main steps of this process can be seen in Figure 2.1. First a problem solving environment or user level program starts a job on the Beowulf cluster. This job is put into a queue of waiting jobs. The queue manager contacts the scheduler to let it know there is a new process waiting to run. The scheduler looks at the system, and decides when the next job should be allowed to start. It contacts the allocator for the appropriate resources and the allocator returns which processors, networks, databases, etc. are now reserved for use by the job. The scheduler would then dequeue the appropriate job, and start the processes on the required resources.

While this scheme may vary from system to system, the main components are the same. In some cases the queue manager or allocator might take more control over the actual start time of the job. For our purposes though, the above arrangement is a good abstract representation.

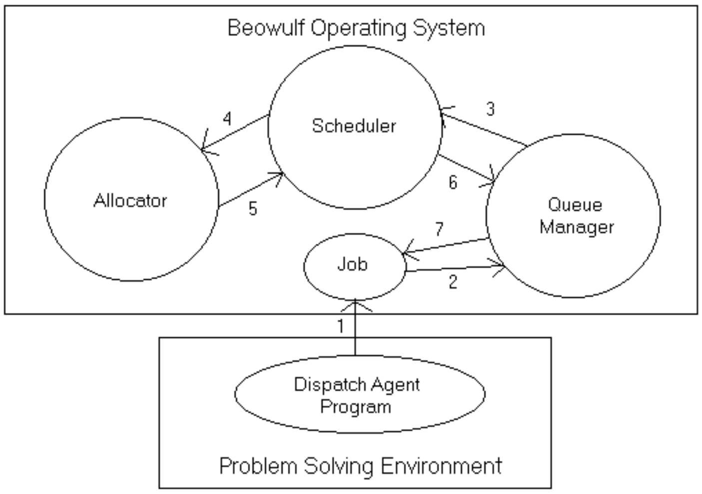


Figure 2.1: Steps taken by services to run user processes.

The function of an allocator and a scheduler should not be confused. While the scheduler decides when and how much to allocate to a given job, the allocator controls the availablity of the resources and actually makes the allocation decisions.

## 2.1 Schedulers

Since schedulers play such a large part in organizing and starting processes, it is imperative that an appropriate interface be examined for integrating any allocator into an existing scheme. Two commonly used schedulers on Beowulf clusters are the MAUI Scheduler and the Beowulf Batch Queue, or Bbq.

### 2.1.1 MAUI

Maui [8] was initially part of a Master's Thesis on Scheduling Optimizations. It was first created in 1995 at Brigham Young University(BYU). It uses its own allocation algorithms, but needs a resource manager to work appropriately. The resource managers that currently have an interface to Maui are Wikiman [9], IBM's Loadlever [1], and PBS [5]. Several other institutions are involved with the current form of the project, including, but not limited to, the University of Utah, the University of Pennsylvania, Pacific Northwest National Laboratory, Boeing, SAIC, and the Maui High Performance Computing Center (MHPCC) at the University of Hawaii.

"Maui is an advanced batch scheduler with a large feature set well suited for high performance computing(HPC) platforms including large Alpha and PC clusters...it makes decisions about where, when, and how to run jobs as specified by admin–configurable policies" [8]. It was designed to be able to be installed transparently without user knowledge. This allows users to be able to continue to submit jobs as previously, but with added scheduling algorithms taking place in the background to increase system throughput.

The capabilities of Maui include several available statistics gathering and diagnostic utilities. These utilities make this scheduler very attractive to administrators, as well as users. The statistics can be gathered per user, per node, or even per job. The diagnostics allow users to track jobs from when they are placed in the queue until completion.

Other options for administrators include the Quality of Service, or QOS, feature and the throttling policies available. The QOS allows policies to be geared toward the mission statement or purpose of a given organization. Special privileges can be provided to users or accounts that have a greater priority in the funded research by the fs.cfg file. This file would specify exemptions from policies restricting access to processing time or resources.

Throttling policies are some of the policies that QOS might exempt a user from. These policies are implemented by an administrator to resrict the flow of jobs through a system at any given moment in time. These restrictions can be for a single job, a single user, or for the entire system. Such restrictions can be on the number of jobs presented, the number of processors or nodes being accessed, a job's duration, or the amount of memory being utilized.

Capabilities for the actual scheduling of resources include options such as advance reservations, backfill, and node allocation policies. Advance reservations "guarantees the availability of a set of resources at a particular time" [8]. Reservations must include the resources required, the time–frame to reserve, and an access control list, or ACL. When the reservation is filled only users or accounts in the ACL can access those restricted resources.

Backfill is the method used by Maui to try to utilize the system as much as possible. For this algorithm to work, each job must send an estimated wall–clock runtime. If backfill scheduling is turned on some lower priority jobs might run before a higher priority job, as long as the higher priority job is not delayed. This incident might occur when a high priority process is waiting on a resource. The scheduler knows the approximate time the job currently using that resource is going to be free. Other resources needed by the high–priority job could just sit empty, but Maui tries to find other jobs that would run in the remaining time and fill those jobs early. This approach is extremely helpful if correct time estimates are used.

There are also many node allocation policies available in Maui. These algorithms would be useful in an allocation tool. A couple of these policies are termed by the `FASTEST`, `CPULOAD`, and `FIRSTAVAILABLE`. `FASTEST` allocates the fastest nodes first, while `CPULOAD` allocates the nodes with the greatest amount of CPU power still available. `FIRSTAVAILABLE` allocates nodes in the order they are reported to the scheduler by the resource manager. These are just a few of the many algorithms available. The one that might be the most important to this research, is the `LOCAL`, or user specified algorithm, that could contact an allocator for the manipulation and borrowing of nodes from other clusters.

Maui is only a scheduler, even though it has the capability to implement allocation algorithms. For this reason Maui will only run correctly if it is attached to a resource manager that is in place and operational. A resource manager is a program that keeps track of the available resources and the allocation of each to specified users. It often has information stored in a database or in configuration files which allow Maui to interface as well as gather statistical information. Certain policy issues can be implemented on these managers, such as which users are allowed on which nodes or partitions.

A partition is a division of the resources that are available, and by default jobs can not bridge these resources. Some special processes have spanning capability. It might be interesting to see if these partitions can be manipulated while Maui is running. If so "borrowing" between clusters on the Mini–Grid might be implemented by controlling this ability. However, partition implementation in Maui may be incorrect, and testing would need to be done to ensure stability.

Maui currently has three available resource manager interfaces, which would help with the development of a new one. Maui has a variable called `RMTYPE` which instructs it to connect with a particular manager. The location of the manager is specified by the `RMNAME`, `RMHOST`, and `RMPORT` parameters. Four functions are at the heart of the interactions between Maui and the manager. `GETJOBINFO` collects state information about currently running jobs. `GETNODEINFO` collects state information about connected nodes. `STARTJOB` and `CANCELJOB` tell Maui to start or stop a job, respectively, on the cluster.

A possible interface to Maui may be created using partitions, the `LOCAL` allocation algorithm function, and developing the four primary functions. Other manipulations might be needed, but the requirements for development are certainly available.

### 2.1.2 Bbq

The Scyld Beowulf Batch Queuing System, or Bbq [7], is an allocator, scheduler, and queuing system all in one. The allocator is very basic and only takes the next available node in exclusive use by setting the user and group permissions on each node. While the allocator does not fulfill our requirements, it is worth noting that enforcement of Bbq can be accomplished by setting Bproc permissions. This will be helpful later on.

The queuing system is designed to be easy for user interaction and is also fairly basic. Several different queues are available and labeled with a single character from `a` to `z` and `A` to `Z`. The higher letters have lower priorities and `a` is the default for Bbq. There is also a special queue, labeled `=`, and is specifically for currently running jobs. These queues are sorted by the job start times.

Bbq also contains a scheduler that is a job batching system based on the Linux command, at [14]. At was developed by Thomas Koenig and David Parsons. It is fairly simple to use which has made it very attractive for researchers. However, its simplicity has limited its functionality, and many users soon move on to more complex schedulers.

The scheduling in at is straight forward and is created specifically for future reservations. Using the `at` command a process can be scheduled to be run at a later time, but no pre-processing is done to make sure that jobs will not overlap or have to wait for their specified start time.

The at system comes in two parts. The first is a daemon. called `atd`, which runs jobs that are already queued. This daemon will run batched jobs based on a limiting load factor for the system. Administrators can override this factor however, by setting a different threshold.

The second part to at, is a text based interface, where users can interact with the `atd` daemon. The command `at` will schedule a job by a specified time, or by typing `now`, `midnight`, or `noon`. This interface also contains the `atq` command, which will list all scheduled jobs in a queue, and the `atrm` command, which will remove a job from the queue.

While this scheduler gives us a better understanding of the interactions necessary for integration with the Scyld operating system, it does not to fit our needs for a scheduler or allocator in the long run. At present, at is not suitable for a system where users are competing for resources, and would not work well in a Mini-Grid architecture with several different research groups vying for nodes. However, it may be worth finding a temporary interface for testing purposes.

## 2.2 Allocators

Many allocators exist for different architectures. Most often new tools must be designed, because of new hardware requirements or the availability of a new allocation mechanism or algorithm. Several examples of allocation tools will be discussed in this Section and the possibility of incorporation into the Mini–Grid architecture will be examined.

### 2.2.1 Connection Machine

The Connection Machine, or CM5 [4] and [13] was first released by Thinking Machines in October, 1991. It tried to combine the positive aspects of both the MIMD and SIMD machines. The "CM5 supports the full data parallel model by providing high performance for branching and synchronization alike" [4].

The CM–5 operating system, CMOST, is an enhanced version of the UNIX operating system. It supports most of the standards in UNIX and uses the network standards to communicate to all of its processors through three separate network connections.

The basic architecture of the CM–5 can be seen in Figure 2.2. The three networks that connect the processing elements are the control, diagnostic, and data networks. The control network is used for communications that involve all processors including

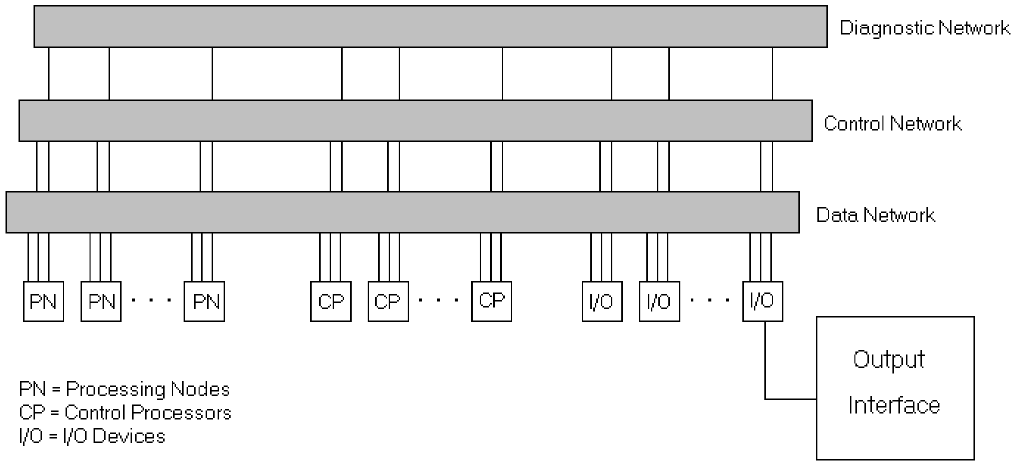


Figure 2.2: Connection Machine, CM–5, Architecture.

broadcasting and synchronization. The diagnostic network is used for a "back–door" entrance for administrators to gain access to all parts of the machine. The data network is used for interprocessor communication.

The processing elements that appear in Figure 2.2 contain two types. The first is the actual Processing Nodes (PN) that do the computations for programs. The number of PNs can be anywhere from several tens to thousands of processors. These nodes contain general purpose processors based on the RISC architecture and usually are upgraded to contain high–performance arithmetic accelerators. These are the nodes that are allocated to specific jobs.

The second type of elements are the Control Processors (CPs). These nodes contain a SPARC microprocessor that is based on the RISC architecture. They are more streamlined than the PNs and are specifically made for making decisions about communications, allocations, and running system calls. These processors can be designated one of two types, either an I/O Control Processor (IOCP) or a Partition Manager (PM).

The Partition Manager controls and allocates the nodes in partitions. A partition is a grouping of the Processing Nodes created by an administrator that would all perform the same approximate function. A Control Processor would be designated a PM for each partition created, and all allocation and scheduling decisions would be made by hardware. The available modes for these partitions are running a single high–priority job, a batch mode, and a time–sharing mode.

These partitions can be rearranged to include any number of processors; the only restriction on the number of partitions is the number of PM's. However, by combining all the partitions into one, it is possible to use the entire CM–5 as a single machine. This rearrangement could be very useful, but it must be done manually by an administrator and can not be done by the hardware or software without strict instructions.

The idea behind partitions is valuable and their control hierarchy is very similar to a grouping of clusters, but without software dynamically changing ownership of these partitions, this allocation scheme cannot be developed to work on a Mini-Grid. The allocation of the actual nodes is also done with hardware in the Partition Managers and would be very difficult to probe for more in depth allocation algorithm information. We should be able to think of the partitions as separate clusters in the Mini-Grid, and hopefully use this idea to develop a similar hierarchy to a new allocation tool.

### 2.2.2 Globus Resource Allocation Manager

With the complexity involved in a Grid, a way is needed to manage jobs. The Globus Project created and released the first version of the Globus Grid Programming Toolkit [16] in November of 1998. This toolkit "provides a set of standard services for authentication, resource location, resource allocation, configuration, communication, file access, fault detection, and executable management" [16]. Not all tools need to be installed, and can be combined for the user's specific needs.

The allocation tools of Globus come in five separate pieces. These pieces include:

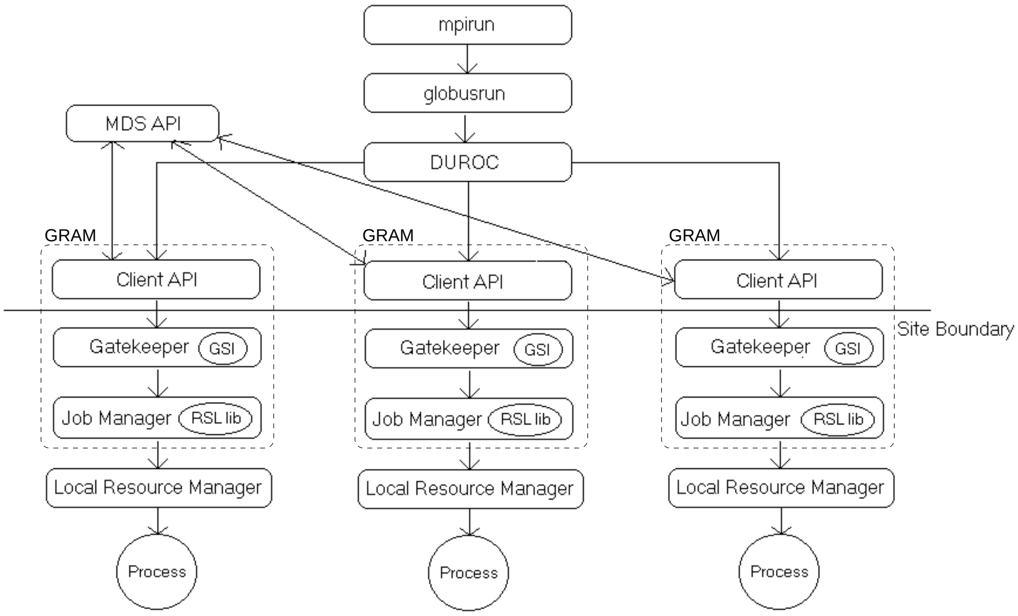


Figure 2.3: Globus Resource Allocation Scheme.

- DUROC: Dynamically Updated Request Online Co–allocator,
- GRAM: Globus Resource Allocation Manager,
- MDS: Metacomputing Directory Service,
- RSL: Resource Specification Language, and
- the Local Resource Manager.

The steps used to allocate resources and start a process can be seen in Figure 2.3. The main Globus programs start with DUROC, include several pieces to GRAM, and the MDS daemon. The Local Resource Manager is dependent on the site architecture and operating system. The RSL Library is a communication tool used by Globus to allow a heterogeneous Grid to specify necessary resources in general terms.

The Globus Toolkit supports an algorithm known as co–allocation, or the simultaneous allocation of a resource to two or more sets in a shared state. The tool that keeps track of this information is the Dynamically Updated Request Online Co–allocator, or DUROC. DUROC keeps track of the requests for resources and initializes the processes. It monitors the system and keeps track of new or failed nodes. Once it receives a new request, it discovers any necessary information about the state of the system and then contacts the GRAM tool to actually start the process on the remote machines.

Before GRAM can start the processes, it must contact the MDS to find out where the resource is located and what kind of hardware and applications exist on the machines. This directory can be updated by the Globus system, an application, or another information provider. The MDS helps the GRAM Client know which Gatekeepers, or remote security daemons, to contact for allocation.

The main program used for allocation is the Globus Resource Allocation Manager, or GRAM. It contains four components. The first is the GRAM Client which sends requests to the remote machines to start a particular process. The component it contacts is called the Gatekeeper, which accesses its Globus Security Infrastructure to authenticate the user trying to start a new process. Once the Gatekeeper allows the request through, it is passed to the Job Manager. The third part, the Job Manager translates the request sent by the GRAM Client to the necessary calls for the Local Resource Manager on that machine. The RSL library is a common language for specifying the job requirements for a particular machine and is considered the power behind GRAM. The library makes the communications in a heterogeneous network, like a Grid, possible.

The last piece is resource dependent. The Local Resource Manager keeps track of the available resources and the allocation of each to specified users. It often has information stored in a database or in configuration files. Globus currently supports the following managers: POE [3], Condor [17], Easy–LL [10], NQE [11], Prun [15], Loadleveler [1], LSF [2], PBS [5], GLUnix [6], and Pexec [12].

The allocation tools in the Globus Toolkit were designed for separate resources and clusters that would never combine, unlike the Mini-Grid architecture. The layering and functionality of this software would be of great use on the Grid, but its focus on the heterogeneous nature of a Grid would cause it to be too inefficient for our purposes.

Many allocators are architecture dependent, such as the CM-5. Others that have tried to be very generalized for a Grid are very complex, such as the Globus Toolkit, and are usually not very efficient. Something in between is necessary for our allocation purposes, that would combine the flexibility of the Grid and clusters, but with the efficiency that would allow a program to run on one cluster with borrowed nodes.

# Chapter 3

# Balloc

## 3.1 Introduction

The research discussed in this chapter was started because of the need for a new allocation tool that would be usable on the new Mini–Grid architecture. This Grid is comprised of separate, homogeneous Beowulf clusters that are completely connected, including the remote nodes. Since these nodes can belong to any of the clusters, a new allocation program was needed that could dynamically and transparently “borrow” these nodes from one cluster to another. By allowing this functionality, the idea of the Scyld Beowulf is continued with the user seeing only a “single computer” as opposed to the large Beowulf Grid. The use of the tool should be transparent and needs to be enforced. This allocation tool needs to be extensible, simple for the user, and robust. It must be able to process multiple users requesting large numbers of nodes and run in an appropriate amount of time when allocating and freeing resources.

The Beowulf Allocator, or Balloc, was created as an allocation tool for the new Mini–Grid. It contains a system daemon that keeps track of allocations and resources, and a user API that allows sets to be obtained through an `mpirun` call or directly through the daemon itself. The functionality of Balloc will be discussed through a description of the user interaction. The actual implementation will be discussed through a description of the system daemon and the algorithms used for allocation.

## 3.2 End User Functionality

Balloc is purely an allocation tool. When a request arrives, it responds by returning which nodes are now reserved for use by the job and user. This list of nodes is considered a set, available to the user until the set is freed. It keeps a log of which users are allowed on which nodes and tracks all sets, available nodes, and allocated resources.

Using Balloc can either be direct or indirect, but will always occur when a user executes a parallel program. A process must call Balloc to get appropriate usage permissions set. The only exception to this is the root user, which can execute processes on any node. Normally the user will not work directly with the Balloc interface, but will use MPI calls, although the direct interaction is available via simple function calls in an API.

There are three main ways for a user to access the information and resources controlled by Balloc. These include:

- Using the MPI interface and calling `mpirun`,
- Using the Balloc user interface and the set and node numbers returned, and
- Using the Balloc control manager to allocate and free nodes.

### 3.2.1 Incorporation into MPI

The Message Passing Interface, or MPI, is a commonly used parallel programming library that stresses the use of standard message passing functions that allows easier communication programming between remote processes. It is widely available and

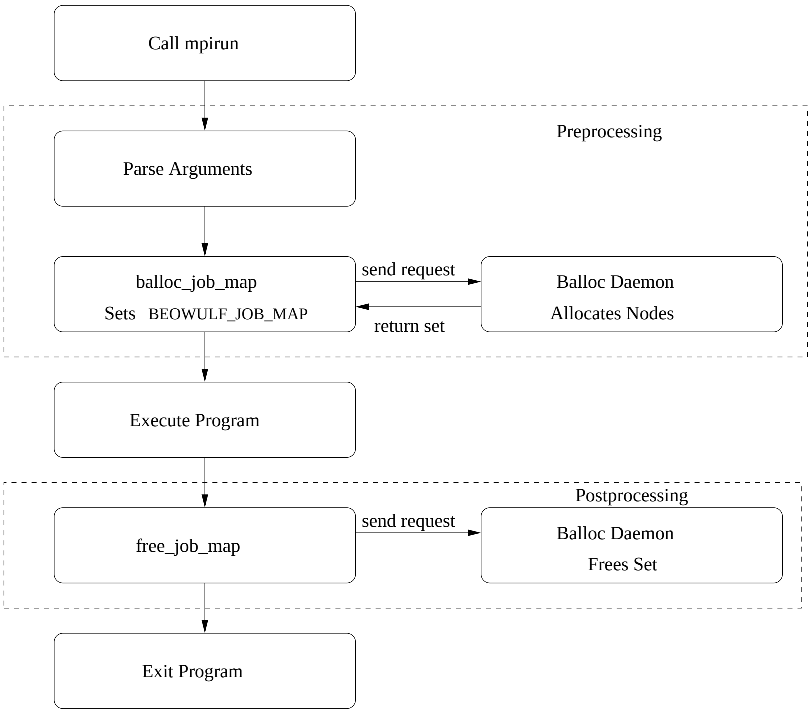


Figure 3.1: `Mpirun` execution sequence.

both free and vendor versions exist. The user is able to start an MPI program with the `mpirun` script. This script attempts to hide some of the background work that starts, executes, and cleans up an MPI parallel program.

The Scyld operating system has adapted `mpirun` to execute Bproc code that migrates all of the remote processes. Bproc decides where to send these processes in two separate ways. The first is by defaults. The number of processors requested is placed in an environment variable `NP`, or the number of processors. By default, Bproc places the first process on the head node, and the rest on consecutive nodes. The second way takes a different environment variable, `BEOWULF_JOB_MAP`. This variable contains a list of nodes that the program wishes its processes to run on. The argument to `BEOWULF_JOB_MAP` looks like "3:15:8:2:25", where each value is a node number. This example argument would migrate five processes, the first to node 3, the second to node 15, and so on.

Using existing programs, we took advantage of the fact that Scyld's designation of resources is as easy as setting an environment variable. By modifying the `mpirun` script we were able to create a way that the user can interact with Balloc indirectly. Often the user may not even know that Balloc was accessed. This occurs when the user calls `mpirun` and the `mpirun` script accesses two C programs called `balloc_job_map` and `free_job_map`. The function order that occurs when a user calls `mpirun` can be seen in Figure 3.1

When a user calls the `mpirun` script, it first parses all arguments used in the command line. The available arguments that are Balloc specific will be discussed in more detail in the Allocation Options Subsection in Sections 3.3.1 and 3.3.2. They include:

- `--ba--exc` = Allocate nodes in Exclusive mode (Default is Shared mode),
- `--ba--less` = Accept a set with less nodes than requested (Default is strict setting),
- `--ba--borr` = If necessary borrow nodes from another cluster (Default is no borrowing),
- `--ba--one` = Only allocate resources on one cluster (Default is mixed allocations allowed),
- `--ba--group` "char" = Borrow nodes from clusters with specified "m", by most first, or "p", by priority grouping (Default is by cluster priority).

After the arguments have been parsed, the script then calls the `balloc_job_map` program. This program sends an allocation request to Balloc. When Balloc returns the node information, `balloc_job_map` configures the node numbers into the correct argument sequence for `BEOWULF_JOB_MAP`, and sets the environment variable. This is not the only environment variable set however. `BALLOC_SETNUM` is also exported. This variable is the set number used to designate the grouping of nodes allocated by `balloc_job_map`'s call to Balloc.

After the preprocessing, the actual program executes and Bproc migrates the processes to the correct nodes. Once the program is completed, some clean up is needed. This is where `BALLOC_SETNUM` is used. `Mpirun` calls `free_job_map`, which sends a free request to Balloc. This request specifies which grouping of nodes to free by `BALLOC_SETNUM`. If everything executes appropriately, the script then exits. This interaction with Balloc should be the safest, but user requests are not always going to be in the form of an `mpirun` call. This is where the other two available interfaces become useful.

### 3.2.2 Balloc API Functions

It may be necessary for a user to allocate and manipulate a set within a program. The user API was created for this purpose to allow direct interaction with the Beowulf Allocator. The function calls available set up the necessary request packets and communicate with Balloc without the user having to worry about any socket programming. The API does the entire communication including formatting, sending, and receiving the packets. It even manipulates the byte order of the responses, in case different operating systems or architectures are being used. Currently the API is only in C, but a Java API would be useful in creating a web based monitoring system. More information about each function and its declaration can be found in the user manual in Appendix A.

The allocation function calls available to the user are:

- `int balloc(int nodes, int mode, node_info *data, int gro, int all, int bor, int ltn, char *bheadname)`
- `int bfree(int set, char *bheadname)`
- `int free_node_info(node_info *data)`
- `int bnodestat(int *data, int *datanode, int node, int *mode, int *sets, int *count, char * bheadname)`

- `int btypestat(node_info *data, int mode, int *nodes, char *bheadname)`
- `int bsetstat(node_info *data, int set, int *uid, int *mode, int *nodes, char *bheadname)`
- `int bactset(int *data, int *sets, char *bheadname)`

The first three calls should be all that the user needs to interface directly with Balloc. The first call `balloc`, is the function that actually allocates the resources requested. The `nodes` argument contains the number of nodes requested in the specified `mode`. When Balloc receives this request, it does all of the processing necessary to create a set of nodes on the system. The remote nodes in this set might be spread out across several different clusters, but the user will only see them as nodes belonging to the cluster that the request occurred on. The node numbers, node addresses, and number of users on each node are returned in the `node_info` linked–list structure. The function `free_node_info` simply frees the memory malloced for this structure. The arguments `gro`, or groupings of nodes, `bor`, or whether or not to borrow from another cluster, `all`, or nodes all on one cluster, and `ltn`, or whether or not less nodes would be accepted, are discussed in further detail in the Allocation Options Subsection in Sections 3.3.1 and Section 3.3.2.

Once the user has executed the program, the user frees the set that was created. When `bfree` is called, the set number and all memory allocated to keep track of the group is freed. The nodes are set to mode Free, if no other users are on those machines. If `bfree` is not called, the set continues to exist until the user is finished with his project and calls `bfree`. If the user forgets, an administrator must call `bfree` or a reboot of the head node will free the nodes.

The API also includes function calls that will monitor and return the status of the condition of the system. These calls are especially helpful with the control manager developed in Section 3.2.3. These functions are fairly self–explanatory and use mainly pointers to return the requested information. The `bnodestat` function returns the status information of the processor with the number `node`. The `btypestat` function returns a list of all nodes that are in the specified state, `mode`, `bsetstat` returns all information about the specified grouping of nodes with the set number, `set`, and `bactset` returns a list of all active sets. Together these four status functions can give a general idea about the condition of the cluster.

There are also several functions that are listed in the API, but would normally only be used by the Balloc daemon, an administrator, or for testing purposes. These include:

- `int bresv(int nodes, int *return_nodes, char *bheadname)`
- `int ballocresv(int nodes, int setnum, borr_node_info *data, char *bheadname)`
- `int free_borr_node_info(borr_node_info *data)`
- `int breturn(int set, int importance, char *borrclustername, char *bheadname)`
- `int breset(char *bheadname)`

Both `bresv` and `ballocresv` are used when one cluster must try and borrow nodes from another. The `borr_node_info` structure returned in `ballocresv` is the same as `node_info`, but includes the Ethernet address of each node. The implementation of these two calls is discussed further in Section 3.3.2. `free_borr_node_info` is a cleanup function and frees the information returned in the `borr_node_info` structure. The `breturn` function forces a return on a borrowed set from a cluster. This can either be done when the nodes are finished running the current jobs, or causes the return to be immediate with no regard to the executing jobs. More information about the implementation can be found in Section 3.3.2. The final function in Balloc is a bail out function called, `breset`. This completely reinitializes the cluster immediately. This call should be removed, and only implemented when expansion and testing of Balloc are in process.

### 3.2.3 Balloc Control Manager and Status Reports

The Balloc Control Manager, or bactl, is a text application that allows an administrator to check and control the status of the system. It can be run on any Linux machine on the network, and connects to the Beowulf cluster by use of sockets. The instruction `newcluster` can be used to instruct bactl to connect to a specific head node, There are several different commands available that can be listed with the `?` help instruction. All of these commands work by executing the user API functions described in the previous Section.

The status checks available include a request on each state, such as `freestat`, or Free nodes, and `excstat`, or Exclusive nodes. Actual allocations and releases of resources can be accomplished from this program as well. If an administrator needed to free all sets held by a specific user, an `actsets` could be executed which would return a list of all active sets. This list of set numbers could then be passed to `setstat`, which would return all the information about that set, including the user id. If the specific user owned the set a `freeset` command could be called, and would release those resources. All information returned is printed to the screen in a readable format.

## 3.3 System Daemon Balloc

The daemon Balloc runs with the Bproc daemon on the head node of each separate cluster. It contains node information necessary for allocation purposes. This information is gathered using system calls to Bproc, including `bproc_nodestatus`, `bproc_numnodes`, and `bproc_nodeinfo`. `Bproc_nodestatus` returns the state of a specified node, which in Bproc can be boot, up, down, error, unavailable, reboot, halt, and pwroff. `Bproc_numnodes` returns the total number of nodes currently installed on the system. `Bproc_nodeinfo` is only used for initialization and contains state, IP address, and user and group permissions. The Ethernet address is obtained by reading the Beowulf configuration file located in /etc/beowulf/config. Hardware information could also be available by adding a separate configuration file or a exploratory program. The hardware structure is incorporated into the node database, but not used for any of the current allocation algorithms because the Mini–Grid is a homogeneous system.

The implementation of Balloc can be broken down into two groups. The first group is the basic allocation mechanism necessary on any cluster or parallel computer to keep track of local resources. This includes databases, allocation types, options for allocation, and enforcement, or security. The second group of functions are new to allocators. These have to do with the necessary interactions for "borrowing" nodes between clusters. Separate databases for local and borrowed nodes must be used. New functions, including borrowing, returning, reserving, and freeing must be implemented to work with the Scyld operating system and the Bproc daemon.

### 3.3.1 One Cluster Implementation

Using other available allocation programs as a starting point, the first goal of this project was to create a reliable allocation tool that could be run on a single cluster. This tool had to interface correctly with the Scyld operating system and MPI. The MPI interface is discussed more thoroughly in Section 3.2.1. This daemon had to allow quick access to nodes, since the majority of use of the Grid is assumed to be located within the bounds of a single cluster.

The first step in developing this new tool was to examine what types of databases would be required for quick, direct access to complete node and allocation information. Two separate databases were used for this purpose. The first database was designed to keep the state information for the local nodes. This database is an array of a node structures, which includes the state, the IP address, and the Ethernet address of the node. The node numbers are assigned by Scyld, and the information is placed in the database such that the index corresponds to the node number. The second database required was used to keep track of the allocations. Each grouping of nodes is considered a "set" of nodes. A set is considered "active" if the grouping is designated to a user. Different active sets can contain the same nodes, if those nodes are in a multi–user state. The active sets database is an array of nodeset structures, which includes the user id that owns the set, the request id of the packet that allocated the set, the number of nodes in the set, and a list of node numbers that have been allocated to that set. Set numbers are assigned to each active set and are the index into the database.

Once the databases were designed the manipulation of nodes needed to be implemented. Several topics will be discussed in the next few Subsections, including the necessary states or modes for the nodes, options available for allocations, and enforcing the use of the Balloc program.

### Allocation Types

Several states for the nodes need to be defined for the correct allocation and access in each set. The modes used in Balloc for the one–cluster case are Free, Shared, Exclusive, Unknown, and Down. The Unknown and Down states are self–explanatory and correspond to the state information returned by Bproc. A flow chart of the rest of the available states is found in Figure 3.2.

When Balloc first initializes, all nodes that are not Unknown or Down will be placed in the Free state. This state can transition into any other state. Whenever a new allocation is requested, these nodes are allocated first, since they should not currently be used by any processes.

The Shared mode, corresponds to the required multi–user access to particular nodes. This state allows for multiple sets, owned by different users, to contain the

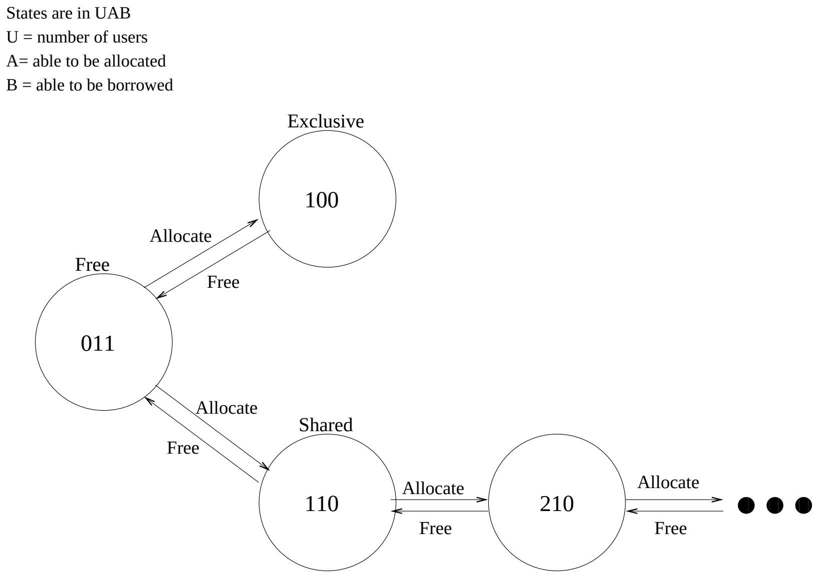


Figure 3.2: State transitions available for nodes in Balloc.

same node. This state would normally be used by researchers that are not testing timing issues or researchers with tasks that do not have timing constraints. If a user requests a Shared set, the Free nodes are first searched, and then the nodes that already contain users are allocated. By allocating the Free nodes first, the most system resources are being utilized and the best performance possible will be delivered to each job.

Some users with higher priorities might choose to run a large job and do not wish anyone else to be able to access the nodes executing the processes. The Exclusive state was created for a single–user access to a set. Unfortunately, if all the nodes in a cluster are placed in an Exclusive state, a Shared job cannot be run until nodes become available. This state gives excellent user performance, but limits the ability of the cluster to fulfill other jobs.

### Available Options For Allocation

Users will want to allocate nodes on a cluster in different ways. Options must be implemented to allow flexibility with Balloc. Since the current hardware configuration is a homogeneous system, not many options were implemented, but the base format was created. This format is extensible for future use, because it uses an integer flag with bit–representations of each option.

The main option developed was to allow Balloc to allocate and return a set of nodes that contains less than the number of nodes that were initially requested. This flag `LESSTHAN` is one bit in an integer flag that is set in the API function called `balloc`. When calling `balloc`, `LESSON` or `LESSOFF` can be placed in the `ltn` argument. If using `mpirun`, `--ba--less` can be used to set this option. If `LESSTHAN` is not set, Balloc will try to fill the request, but if it is unable to do so, it returns an error.

More options were developed for the two or more cluster case and are discussed in Section 3.3.2.

### Enforcement

Enforcing the use of Balloc was important in creating a stable environment for a scheduler to be able to fill the appropriate tasks. If every user did not contact Balloc for nodes, then jobs that were supposed to run on Exclusive nodes might have multiple users. This also becomes a problem with borrowing nodes as can be seen in Section 3.3.2. If users started jobs on nodes that Balloc thought were Free, and then tried to loan those nodes out, the nodes would be rebooted and all information about the process would be lost.

The Scyld operating system has its own enforcement policies. Bproc only allows users that have the appropriate permissions on nodes access to those resources. This information is stored in the /etc/beowulf/config file for initialization of Bproc. Root can change these permissions by calling `bproc_chown(node number, user id, group id)`, which sets the user and group id's to have access to the specific node number.

Balloc changes the configuration file and calls `bproc_chown` every time an allocation or free is requested. The configuration file is changed in case the Bproc daemons are restarted. This way on initialization, the correct users have the appropriate permissions and can continue working.

On startup all nodes are initialized to allow only root access. This forces all allocations to be done by Balloc, but still allows root to have access to the nodes. If the nodes are allocated in Exclusive mode, the user id is set to the user who sent the request and the group id is kept as root. If the nodes are allocated in Shared mode, the user id is set to `any` to allow multiple users and the group id is kept as root. The reason `any` must be set is there is currently no way to specify two or more users. Groups could be set up, but since nodes can be in multiple sets, and no two nodes need be in the same combination of sets, a group would have to be created for each node. Since a node in Shared mode can not be borrowed and access to these clusters is fairly restrictive, the `any` option was chosen for use in this version of Balloc.

### 3.3.2 Two or More Cluster Implementation

The main purpose of Balloc was to create a new allocation program that can transparently "borrow" nodes between clusters. It was designed for the new architecture that can be seen in Figure 1.1. This Figure illustrates the new Mini–Grid, which can be depicted as a large cluster broken down into separate sub–clusters. These sub–clusters are completely connected in the background and any remote node can belong to any of the head nodes.

To allow these sub–clusters to combine, Balloc had to be able to move nodes to different clusters, or "borrow" nodes. Borrowing nodes consists of transferring ownership of a node from one cluster to another so, while a node is still physically closer to the loaning head node, it nows receives instructions and processes from

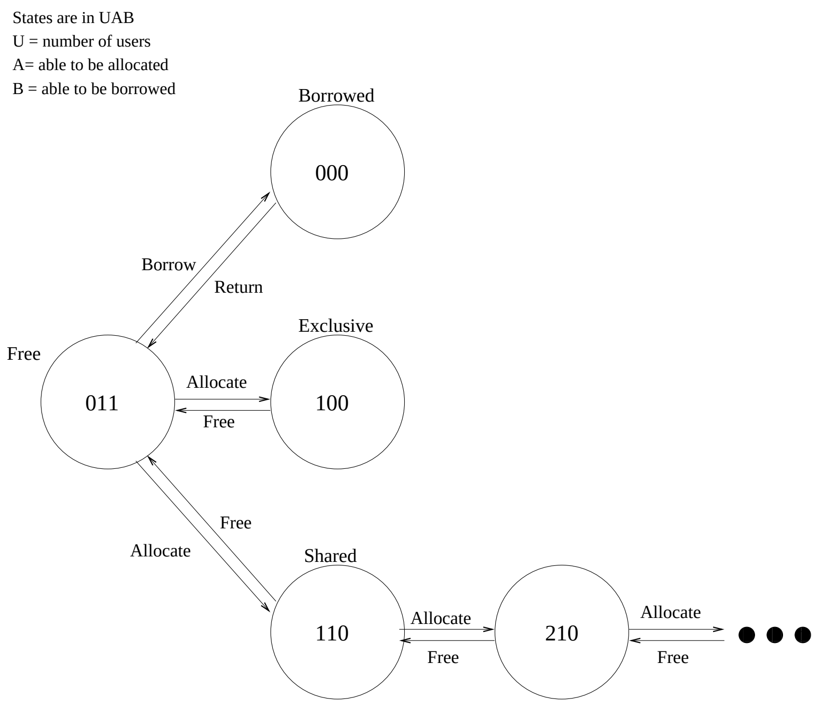


Figure 3.3: State transitions by a loaner, available on Beowulf Grid.

the borrowing head node. With the connections between the local nodes and the borrowed nodes, latency should be fairly negligible.

Borrowing nodes can be seen from two different perspectives. The first is from the loaner. The available state transitions for the owning node can be seen in Figure 3.3. This is very similar to the single cluster case, but a new mode Borrowed is introduced. The loaning cluster does not keep track of who the borrowed nodes were allocated to, just which cluster borrowed the nodes. For this reason, the Borrowed state only exists on the loaning cluster.

The borrowing cluster sees the node as one of its own and places it in a different state in a separate database. The state transitions available to the borrowing cluster can be seen in Figure 3.3. This state diagram contains some unusual states. The main ones are the Exclusive modes with multiple users. This occurs when a cluster requests that its nodes be returned, and is discussed further in following Subsections.

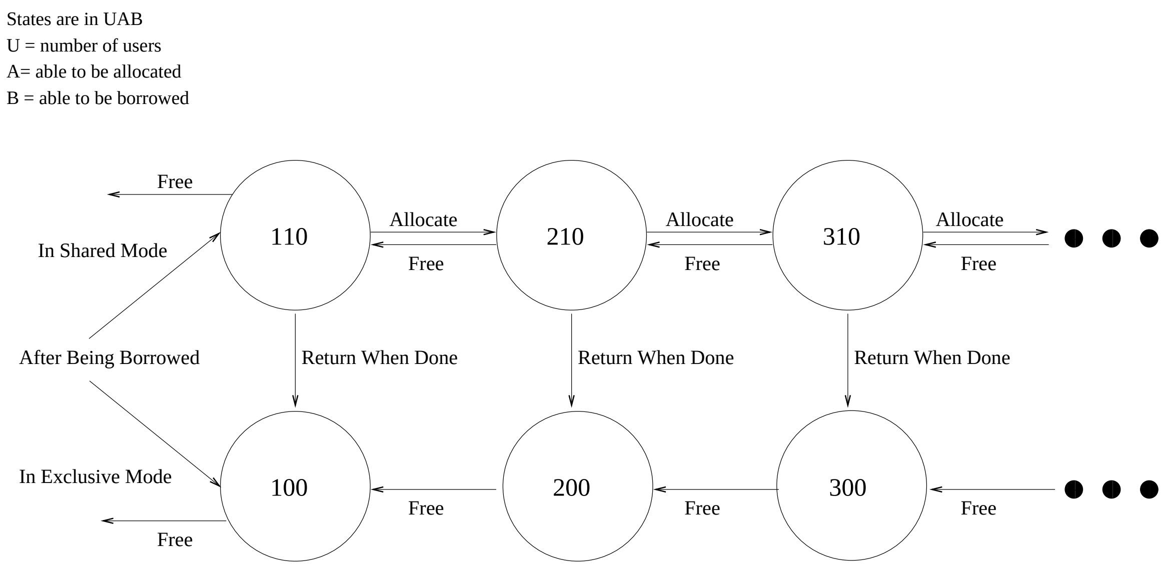


Figure 3.4: State transitions by a borrower, available on Beowulf Grid.

The borrowing cluster treats the node as a local one, allocating and freeing it as it would any remote node. The only exception is where the information is stored. A separate database for borrowed nodes was created. This database contains nodes that were borrowed from other clusters, and is comprised mostly of the same information as in the local node database. This database however, must contain additional information, including the original set number, the original node number, and the original owning cluster.

The necessity for keeping track of the original information, especially the set number, can be seen in Figure 3.5. When the cluster borrows the set, it must keep track of the original set number and owner to be able to call `bfree` on that set when the entire group is Free. The user never sees this set number and it would only appear on the loaning cluster's `bactset` function call. This helps keep the borrowing transparent from the user.

Borrowed nodes are added on to the end of the list of available nodes. They are the last to be allocated in any request. The local node database has a specific size, which is the number of nodes available on the cluster. When a node is borrowed its

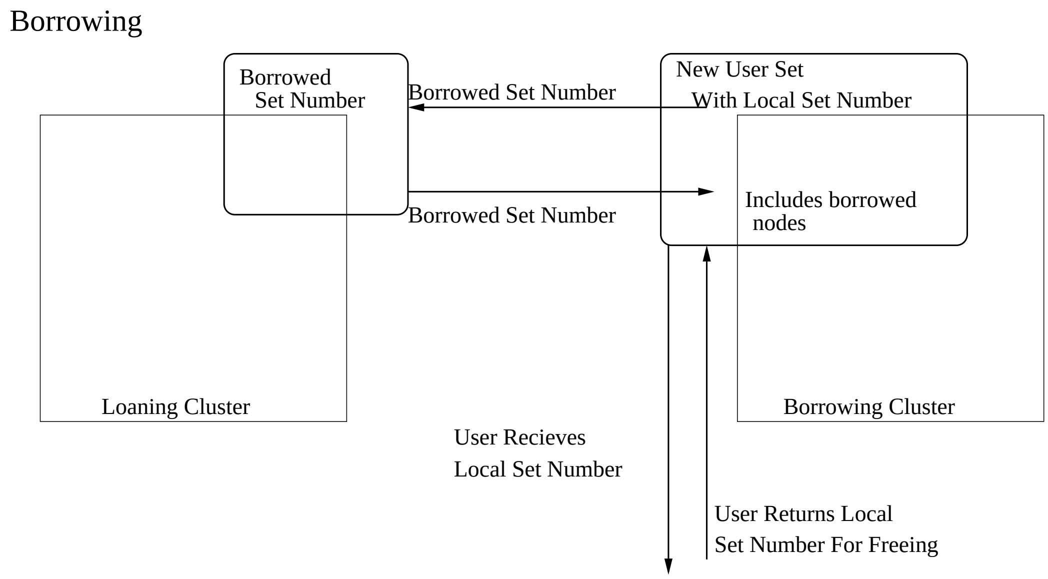


Figure 3.5: Tracking set numbers while borrowing nodes.

node number becomes its index into the borrowed database plus the local database size. The index might not be the same on consecutive borrows of the same node, because the node numbers are filled as nodes are received from another cluster.

The next steps were deciding on available options for two or more clusters an actually implementing the functions that borrowed and returned nodes. The options implemented for the Grid are discussed in the next Subsection. To borrow a node, a remove from the original cluster and a add to the new cluster were required. To return a node, a return from the new cluster and a reset on the original cluster were required. These implementations are discussed in later Subsections.

### Available Options For Allocation

The available options for allocating nodes on more than one cluster include three different alternatives. All are implemented using the integer flag created in the `balloc` API call. The three options and their corresponding arguments in the API `balloc` call in Section 3.2.2 and `mpirun` call in Section 3.2.1 are:

- option; API arg.; MPI arg.
- borrowing; `bor`; `--ba--borr`
- grouping; `gro`; `--ba--group`
- allonone; `all`; `--ba--one`

The first bit added to the flag, is whether or not to borrow nodes if necessary. Balloc always tries to allocate nodes on the local machine, if possible, but if this flag is set it will try to fill any remaining nodes on other clusters on the Grid. The API can be filled in with `BORROWON` or `BORROWOFF`. The way Balloc chooses to allocate nodes is set by the other two flags.

The grouping flag takes four bits in the integer flag and can currently either be `PRIORITY` or `MOSTFIRST`. This preference specifies which clusters to borrow nodes from. If `PRIORITY` is set, the nodes are taken from the cluster with the highest priority listed in its cluster configuration file. If `MOSTFIRST` is set, the nodes are taken from the cluster with the most available free nodes first. If `MOSTFIRST` is set and two or more clusters contain the same number of free nodes, the cluster with the highest priority is allocated first. The default for this flag is to allocate by `PRIORITY`.

The final preference implemented is the allonone bit flag, which can be set in the API with `ALLON` or `ALLOFF`. This option specifies whether or not the entire set of nodes must physically exist on one machine. Borrowed nodes are allowed, if they all come from the same cluster. This flag takes precedence over the `LESSTHAN` flag described in Section 3.3.1; if this flag is set Balloc will not return fewer nodes than the amount requested. The default for this flag is `ALLOFF`.

#### Borrowing Implementation

When borrowing nodes, Balloc has to do some processing before the allocation actually occurs. Depending on the policy of the system, Balloc might chose one available set over another on a different cluster. One example of this occurrence might be if a cluster needed to borrow five nodes. If one cluster had three nodes free and another had five, it would make more sense to take the five nodes, than three and two.

To allow an exploration of the available nodes on the Grid, a reservation system was used. This system uses the `bresv` API call to reserve a specified number of nodes. If that number of nodes is not available, the maximum number of reservable nodes is returned. The reserved nodes are placed in a set on the loaning cluster. The borrowing cluster then sorts through the available nodes, and sends a `ballocresv` call to the loaning cluster. This function takes a set number, which is the reserved set, and the number of nodes to allocate from that set. The loaning cluster sets an alarm to go off, allowing ten seconds for the borrowing cluster to decide. If it does not receive a `ballocresv` call on the set number by the alarm, it frees the nodes that were reserved.

If a `ballocresv` is received from the borrowing cluster, the nodes requested must be removed from the Bproc on the owning machine. When a remote node is booted in the Scyld operating system, it is a two step process. The node first sends out a RARP, or a request to find an owning cluster. A Beoserv daemon, part of Bproc that handles the remote node bootings, will respond and reboot the node with the correct kernel version. To remove a node from a cluster the Bproc and Beoserv daemons must be told that this node no longer belongs to this cluster.

To get the daemons to realize this, the configuration file, /etc/beowulf/config, must be changed. The file is first read into a buffer and several node structures. The file is then rewritten, and as the nodes are being printed, each one is checked whether or not it is in the set being allocated. If it is, it is turned off by writing “node off root root” in the corresponding node line. Once the configuration file has been altered, the set must be rebooted by calling `bproc_setnodestatus` for each node. The daemons must then be HUPed to force them to reread the new configuration file, and know not to respond to the RARPs of the rebooted nodes. The node is now effectively removed from the cluster.

Once removed, the borrowing cluster must add the rebooted nodes to its list. This is basically done the same way as removing the nodes from the original cluster. A couple of differences are the configuration file is changed to include the node, e.g. "node ETHERNETADDRESS root root", the nodes are not rebooted, and the /var/beowulf/unknown_addresses file must be altered. The unknown address file is used by Bproc to know which addresses not to respond to. It contains a list of Ethernet addresses that a RARP was received from, but no node number was assigned to. This file is read in by Balloc and checked for any of the Ethernet addresses of the borrowed set. If any are present they are removed before the Bproc and Beoserv daemons on the borrowing cluster are HUPed. Once these daemons are restarted, the nodes are then booted into the cluster.

Once the borrowing is complete, the nodes must be booted with the correct kernel from the new cluster. This may take up several minutes depending on the machine speed and the whether or not an error occurs with the remote node. The new cluster will try rebooting the machine up to three times. If the node still will not come up, it is marked as down, and may be returned to the original cluster when a free occurs. If this is the case, an administrator may have to go in and reset the node by hand.

If Balloc returns while the nodes are in the boot state, the user will not be able to use the borrowed nodes and an error occurs. Therefore, right before Balloc sets the permissions on those nodes, it waits for each one to come up. As stated this can take several minutes, but this design is assuming large jobs, that take a long time to run. With only a few minutes before and after, for preprocessing, this is not expected to be a major factor.

### Returning Implementation

When returning nodes, the entire set must be returned. A single node cannot be restored, unless it is a set of one. The reason for this is to keep consistency and simplicity for borrowing. As can be seen in Figure 3.5, the original set number is required for returning a set to its owner. This allows the borrowed nodes to be treated much the same as local nodes.

Returning nodes is done much the same way as borrowing, but there are two ways a return can be started. The first is with a `breturn` API call. The `breturn` contains a flag which commands a return to be done `NOW` or `WHENDONE`. A `NOW` argument causes the entire set to be returned immediately with no regard for any user job currently running on that machine. The nodes are returned and any set containing those nodes has a -1 placed in the node list. A `WHENDONE` argument only has effect if the nodes were allocated in a Shared state as can be seen in Figure 3.4. It causes the nodes to move into an Exclusive state such that no other jobs can be placed on them. The nodes will return when the current users are finished and no other jobs can be started.

The `breturn` is not called by Balloc, but is actually called by the administrator, normally using the Balloc Control Manager. The call could be placed in Balloc, but the timing of when it would be called needs to be strongly considered.

The other way to return nodes, is simply when every node in the set is Free, to immediately give back the borrowed nodes. In Figure 3.4, there is a final free that allows the nodes to no longer be part of the new cluster. This free checks every node in the set, and returns only if the entire set is free. If it is entirely free not, these nodes remain part of the cluster and can be allocated again if in Shared mode.

The implementation of returning and reseting the nodes is the same as borrowing them. The /etc/beowulf/config and /var/beowulf/unknown_addresses files are altered accordingly and the appropriate nodes are rebooted. The daemons are then HUPed and forced to reread the files.

However the loaning cluster does not wait for the nodes to be rebooted correctly before responding to the borrowing Balloc. This saves some time in the post-processing for the user. The loaning cluster sends the response, and then waits for the nodes before setting the appropriate permissions.

Once the nodes have been set as up on the owning cluster, the entire borrowing process has been completed. This process can be repeated whenever necessary, however the user wishes to allocate the nodes.

# Chapter 4

# Results

Balloc was designed specifically for a Mini-Grid architecture and the Scyld Operating System for a Beowulf cluster. The main goals of this research have been met by the implementation of the allocator. The efficacy of borrowing nodes can be demonstrated with a couple of examples. These objectives include maintaining the state of all nodes distributed across the Grid, the implementation of a borrowing mechanism, multi–user and single–user support, available options for policy exploration, and transparency to the user. Example usage is explored in Section 1.2.

One goal, simplicity of use corresponds to the transparency to the user and is fulfilled with the incorporation of the three ways for a user to access the information, including `mpirun`, the API, and the Balloc Control Manager. In the first case the user does not even need to access the Balloc daemon, therefore being transparent, and in the other two a user interface allows any interaction to be limited to simple function calls. Borrowing is also transparent, because the user sees all allocated nodes as belonging to the queried cluster. The only hint that this occurs, is through the option that must be set.

Extensibility is available through development of the user API, options for allocation, and new functions that can be developed into Balloc. In each one of these

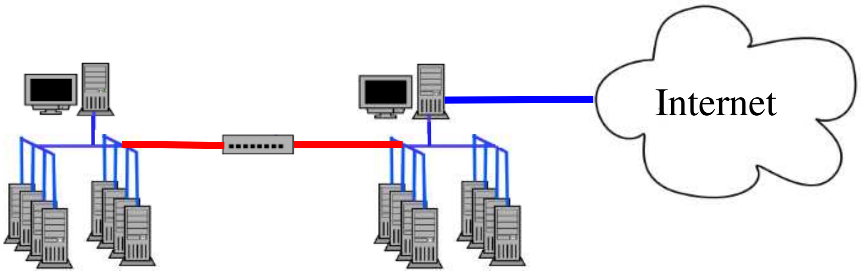


Figure 4.1: Grendel Testbed Architecture

parts, a format is followed that would allow a programmer to develop new policies and implementations that would expand the capabilities of Balloc.

The last two goals that were required for this program were an efficient implementation and an enforcement policy. Efficiency is discussed further in Section 4.2. Enforcement of Balloc's allocations occurs with the application setting the appropriate permissions for the request's user id. For our purpose, this policy is enough security to necessitate access to Balloc before any computations can be performed.

## 4.1 Testbeds

To test the new allocation tool, Balloc, a testbed had to be created. The initial testbed consists of two clusters of six remote nodes and one head node each, and whose architecture can be seen in Figure 4.1. This testbed was slow in comparison to the final target architecture discussed below, but was sufficient for implementation testing. The nodes consisted of 150 MHz Pentium processors, 64 MB EDO DRAM, two GB IDE hard drives, and SMC Tulip-Based Fast Ethernet cards. This test Mini-Grid was developed from an older Beowulf Cluster named Grendel. Only the initial head node was allowed to have access to the outside network.

The target architecture for Balloc was the Clemson University Mini-Grid, as seen in Figure 4.2. The Clemson University Mini–Grid can be illustrated as one large

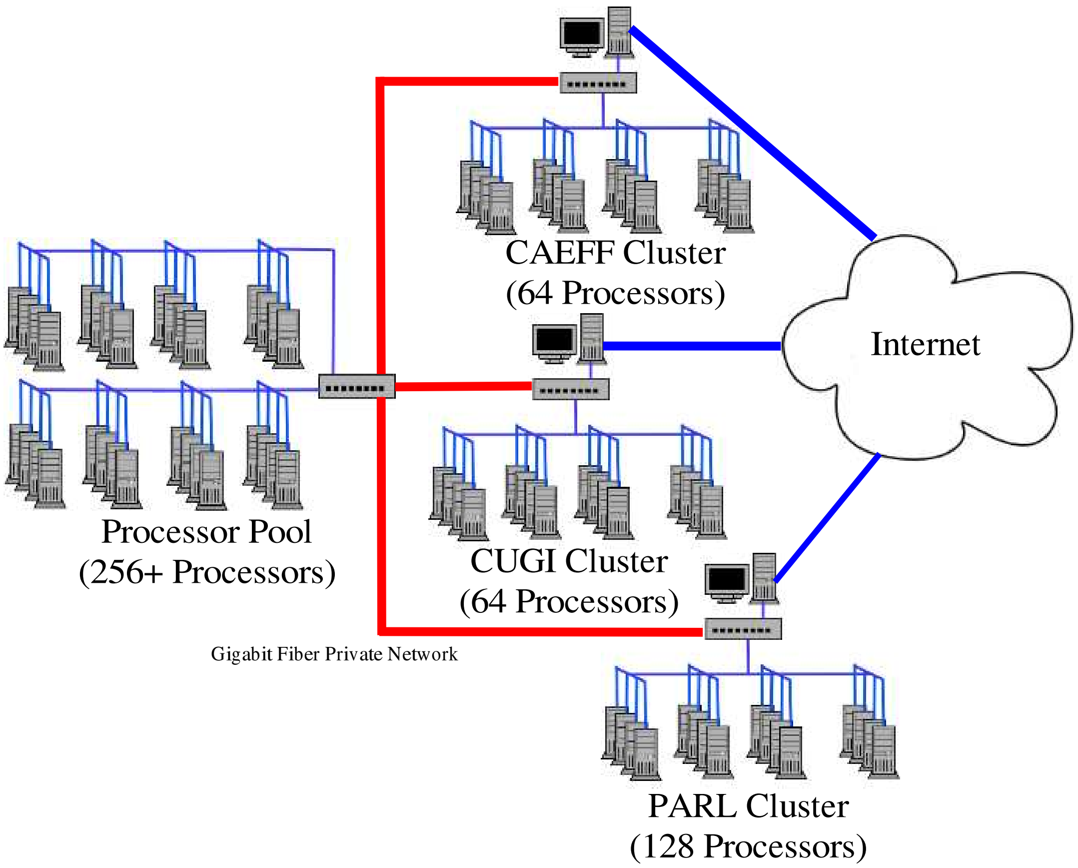


Figure 4.2: Clemson University's Mini-Grid Structure

cluster that is broken into four separate smaller sub–clusters. These sub–clusters belong to different research groups on campus, allowing them to combine resources to build an even larger parallel computer than one group alone could afford. The three groups working on this project are the Center for Advanced Engineering Fibers and Films (CAEFF), the Clemson University Genomics Institute (CUGI), and the Parallel Architecture Research Lab (PARL).

These sub–clusters exist around the campus and each cluster is connected with gigabit fiber on a private network. The nodes use Ethernet switches to allow any node to belong to any head. Each cluster is configured to contain 256 nodes, the total number of nodes in the Grid, and each node contains dual Intel Pentium 3 1GHz processors. Due to hardware problems, I was unable to do any testing on the Clemson Mini-Grid, but Balloc was run successfully on the Grendel Testbed.

## 4.2 Experiments Completed with Timing Information

Efficiency is a small issue that will be addressed briefly. The main aspect is that the allocation tool should not slow down development. Given the expected size of a Mini-Grid and its subclusters, it can be assumed that when a user forces Balloc to borrow nodes, the number of nodes requested is very large. Since the number of nodes in the set is very large, the size of the computations must also be very large and involve a great deal of time. Therefore, if allowing the user to spread the computation over nodes that may have fewer users on them decreases the amount of time to perform the functions, then some leniency can be given to the allocation tool.

A test program was used to time the computations with and without Balloc. This program multiplied two matrices one of $1280 \times 320$ and the other $320 \times 160$. Using the Grendel Grid, the program was run on different number of processors ten times. The run time of the program was taken using the wall clock time, and can vary depending on the state of the system.

Taking this into consideration, the common case, or one cluster case, must take place quickly, while borrowing nodes may take a greater amount of time. However, it was discovered that using Balloc for the common case, might actually speed up the test program. More studies would have to be performed for this statement to be justified, but with the test program running on five processors, the average time without Balloc was 34.22 seconds and the average time with Balloc was 31.91 seconds. While 6.75% is not a great difference, it is enough to take notice. This difference could be in the way Balloc and MPI on Scyld allocate nodes. With this example, it is shown that the common one-cluster case is not slowed down at all by using Balloc.

The rarer case, where nodes must be borrowed, comes in to play with very large jobs, or multiple jobs that require Exclusive access. The test program was used with twelve nodes, using the entire Grendel Grid. Unfortunately, the test program did not seem to be as efficient on multiple nodes, and processing took around 40 seconds with Balloc. Borrowing on the other hand is extremely slow on Grendel. With an average time of 5.5 minutes for borrowing all six nodes, when none fail or have to be rebooted, processing time would need to be much greater than that to justify borrowing nodes.

A problem occurs when a node fails or does not boot correctly and adds an average of 2 minutes per node to the borrowing time. On Grendel this occurred approximately 50% of the time. It is assumed that some of this occurs because of slow hardware and implementation on the Clemson Mini-Grid would help. Even though the nodes have a tendency to not boot, Balloc handles this and only one out of ten times did the node not eventually come up. In this case the node is deleted from the set and the set is sent with one fewer that requested.

# Chapter 5

# Conclusions and Future Work

This paper discussed the design and implementation of a new allocation tool for the Mini-Grid architecture. The requirements for such an allocator were examined and were determined to be to:

- Maintain the state of nodes distributed across the grid,
- Allow for multi–user and single–user access to nodes,
- Provide a mechanism for transferring nodes between clusters,
- Provide borrowing and allocation options for policy implementation in a scheduler,
- Be transparent to the user,
- Enforce usage policies,
- Have an efficient implementation, and
- Be able to integrate with existing software.

Several schedulers were examined for necessary interface components, and then several allocators were examined for possible integration. It was determined that the existing allocators did not meet the necessary requirements for this architecture.

The design and implementation of a new Beowulf allocation tool, Balloc, was discussed, including the development of a user API and a system daemon. The user API was shown to hide the user from the complexities of socket programming, and allowed simple access to Balloc. The system daemon was shown to complete all functionality requirements necessary including integrating with the Scyld software and providing a new borrowing mechanism. Balloc is robust enough to handle multiple users requesting large numbers of nodes. It is time efficient when allocating nodes and does not effect the timing statistics of certain types of MPI programs. When borrowing nodes, the allocation time is increased, but since borrowing nodes implies a large program, the time required is small in comparison.

There are many avenues available for future work on Balloc. The first is to test the allocator on the actual Clemson University Mini-Grid. This Mini-Grid is much larger in size and would give a greater estimate of the amount of time required for borrowing and allocating nodes. Database searches would be greater and rebooting large numbers of nodes might cause problems. Attempts should be made to see if a speedup is necessary for reading and writing configuration files. This larger testbed would give better insight into the operations of Balloc.

Another venue, is the integration of Balloc with an existing scheduler. This would make the use of Balloc on a large cluster more effective. This integration should be able to access Balloc for nodes and tell it when to allocate, but Balloc should decide which nodes to use. By scheduling jobs into the Mini-Grid, more throughput could be achieved.

Also, a reservation scheme with the scheduler will need to be implemented for future use. This requires developing a functionality that allows groups of users onto the same group of reserved nodes. One way to accomplish this might be to create a set management program. It could keep track of a larger set and be able to allocate smaller sets within that group. This would allow a research group to designate a set of nodes for their exclusive use, and only allow those people on the nodes in a shared functionality.

A program that might increase the recognition of the Clemson Mini-Grid is a GUI program that would be available on the web for use by anyone. This program would allow users to see the status of the Mini-Grid at any time. This application would require a Java API to be developed and should not be too difficult, since the communication packets are already placed in network-byte order. This API would need to contact Balloc and use the status requests to graphically display the current status.

Another GUI that might be developed is for the Balloc Control Manager. The current interface to the manager is purely text based and can make it hard for the admin to keep track of the status of the Mini-Grid. This GUI would make controlling and maintaining the Mini-Grid easier for an administrator.

Finally, Balloc could be developed to have more control over the way Bproc sees the cluster. Currently, the cluster must be configured to contain IP addresses and node numbers for the entire size of the Mini-Grid. The configuration files could be written to expand and collapse the size of the cluster according to when nodes are borrowed from another cluster. This configuration file contains all information about the cluster, including the IP address range, a list of node Ethernet addresses, and and the net mask used for communications.

Balloc was developed for use on the new Mini-Grid architecture. This thesis examined the necessary requirements for such an allocator. How these requirements were developed and implemented in Balloc was discussed.

# Appendices

# Appendix A

# Balloc API Manual

## A.1 Overview of Balloc API

The balloc application protocol interface (API) consists of nine user functions and four system functions. The prototypes of both the user and system functions are defined in the “balloc.h” header file, and the implementation of these functions is defined in the “balib.c” source file. The user functions provide a means to allocate, free, and query sets and a means to create a job map from a set. The system functions are provided primarily for use by the balloc daemon source and for the balloc API source itself. The system functions give the daemon source the ability to temporarily reserve nodes on a cluster for possible allocation. Other system functions give the API source the ability to convert data from network-byte-order to host-byte-order. A brief description of the user functions and system functions can be found in Tables 1 and 2 respectively. Next some coding examples are given. Finally, a more detailed description of each function is listed.

There are four user functions provided to create and free sets. These functions are balloc(), bfree(), breturn(), and breset(). The function balloc() is used to allocate sets. Several attributes of the set can be specified which includes the number of nodes in the set, the mode the set should be allocated in, the grouping of the set, whether or not all nodes must be allocated on one cluster, whether or not borrowing is allowed, and whether or not less nodes than requested will be accepted. Once a set has been allocated, it can be freed with bfree(). If a set has been loaned to another cluster it can be returned with breturn() when the current processes are finished or immediately. Finally, breset() can be used to reset the allocation tables. However, this function should only be used for testing purposes.

There are four user functions provided to query nodes and sets. These functions are bactset(), bnodestat(), bsetstat(), and btypestat(). The function bactset() will return a list of all active sets. The function bnodestat() is used to determine the status of the specified node. The returned information about the node includes the IP address, the mode, the number of users currently using the node, the number of sets that contain the node, and a list of the sets that contain the node. The function bsetstat() is used to determine the information about a specific set. The returned information includes the mode the set is in, the user ID of the owner of the set, the number of nodes in the set, and a list of the nodes in the set. The function btypestat() is used to get a list of nodes in a specific mode.

Once a set has been created with balloc(), the function node_info_to_job_map() is used to create a job map from the specified list of nodes. A job map is a string containing a list of node numbers seperated by colons. Typically, a job map is created for the environment variable BEOWULF_JOB_MAP. Once the BEOWULF_JOB_MAP variable is set to the job map, a job can be executed on nodes listed in the job map.

The two system functions used by the balloc daemon source are bresv() and ballocresv(). The bresv() function is first called to reserve the specified number of nodes for ten seconds while balloc determines whether or not the nodes are needed. If balloc decides the nodes are needed, the ballocresv() function is called to allocate the nodes in the set that were just reserved.

The other two system functions are ntoh_borr_info() and ntoh_node_info(). These functions are mainly used by the balloc API functions. They both take returned network-byte-order data and convert it to host-byte-order. The ntoh_borr_info() function organizes the data into borr_node_info structures and the ntoh_node_info() function organizes the data into node_info structures.

A typical scenario will involve first using balloc() to allocate some sets. Next querying may be done to determine the status of nodes and sets. Then the function node_info_to_job_map() will be called to create a job map. The job map will be used to set the BEOWULF_JOB_MAP environment varaiable. At this point jobs can be run on the nodes specified by the BEOWULF_JOB_MAP variable. Finally after all jobs have ended on a particular set, the set is freed.

## Table 1 - User Functions

| Function prototype | Description |
|---|---|
| int **bactset**(int *data, int *sets, char *bheadname) | Returns a list of active sets. |
| int **balloc** (int nodes, int mode, node_info *data, int gro, int all, int bor, int ltn, char *bheadname) | Allocates the number of nodes requested if the given constraints are met. |
| int **bfree**(int set, char *bheadname) | Frees the specified set. |
| int **bnodestat**(int *data, int *datanode, int node, int *mode, int *sets, int *count, char *bheadname) | Returns information on a specified node. |
| int **breset**(char *bheadname) | Resets balloc. All nodes are freed and removed from all sets. Should only be used for testing purposes. |
| int **breturn**(int set, int importance, char *borrclustername, char *bheadname) | Request the set that 'borrclustername' loaned to 'bheadname' to be returned to 'borrclustername'. |
| int **bsetstat**(node_info *data, int set, int *uid, int *mode, int *nodes, char *bheadname) | Returns information on the specified set. |
| int **btypestat**(node_info *data, int mode, int *nodes, char *bheadname) | Returns a list of nodes that are in the specified mode. |
| char ***node_info_to_job_map**(node_info *data) | Returns a job map of the form "X1:X2:...:XN" where X1 through XN are the node numbers stored in the list 'data'. |

## Table 2 - System Functions

| Function prototype | Description |
|---|---|
| int **ballocresv**(int nodes, int setnum, borr_node_info *data, char *bheadname) | Allocates nodes that are in the reserved set 'setnum'. |
| int **bresv**(int nodes, int *return_nodes, char *bheadname) | Reserves a set of nodes for 10 seconds that contains less than or equal to the number requested. |
| int **ntoh_borr_info**(int *new_data, int numnodes, borr_node_info *data) | Converts the returned data 'new_data' to a borr_node_info list. |
| int **ntoh_node_info**(int *new_data, int numnodes, node_info *data) | Converts the returned data 'new_data' to a node_info list. |

## A.2 Example Coding

1. Below is an example that allocates a set of 16 nodes on the local cluster-head. Borrowing will not be allowed and less nodes will not be accepted. The nodes will be allocated in SHARED mode. After the nodes are allocated, a job map is created from the nodes. Finally, the set of nodes are freed.

```
node_info * theNodes;
node_info *finger1, *finger2;
char bheadname[MAXHOSTNAME + 1];
char *jobMap;
int set = 0;
int status = 0;

// get the local host name
gethostname(bheadname, MAXHOSTNAME);

// allocate the set
theNodes = (node_info *) malloc(sizeof(node_info));
set = balloc(16, SHARED, theNodes, MOSTFIRST, ALLOFF, BORROWOFF,
                                            LESSOFF, bheadname);
if(set < 0)
{
   fprintf(stderr, "Failed to allocate Set!\n");
   exit(-1);
}

// get the job map
jobMap = node_info_to_job_map(theNodes);
if(jobMap == NULL)
{
   fprintf(stderr, "Failed to create job map!\n");
   exit(-1);
}
...
// use the job map to execute jobs
...
// free the set
status = bfree(set, bheadname);
if(status < 0)
{
   fprintf(stderr, "Failed to free set!\n");
   exit(-1);
}
```

```
// free the node_info list from memory
finger1 = theNodes;
while(finger1 != NULL)
{
   finger2 = finger1->next_node;
   free(finger1);
   finger1 = finger2;
}
```

2. Below is an example of allocating 32 nodes in EXCLUSIVE mode. Priority gouping will be used. Allocating all the nodes on one cluster will not be required. Borrowing will be allowed. Finally, less nodes than requested will be accepted.

```
node_info * theNodes;
char bheadname[MAXHOSTNAME + 1];
int set = 0;
int status = 0;

// get the local host name
gethostname(bheadname, MAXHOSTNAME);

set = balloc(32, EXCLUSIVE, theNodes, PRIORITY, ALLOFF, BORROWON,
                                                   LESSON, bheadname);
if(set < 0)
{
   fprintf(stderr, "Failed to allocate Set!\n");
   exit(-1);
}
...
status = bfree(set, bheadname);
if(status < 0)
{
   fprintf(stderr, "Failed to free Set!\n");
   exit(-1);
}
// free vars, etc...
```

3. Below is an example of how to use bactset() and bsetstat(). The returned list of sets from bactset() are printed to the display. For each set, the user id of the owner of the set, the mode the set is in, and the number of nodes in the set are retrieved using bactset() and printed to the display.

```
char bheadname[MAXHOSTNAME + 1];
int status = 0;
int i = 0;
```

```
int sets = 0;
int *data;

// get the local host name
gethostname(bheadname, MAXHOSTNAME);

status = bactset(&data, &sets, bheadname);
if(status < 0)
{
   fprintf(stderr, "Failed to get active sets!\n");
   exit(-1);
}

// print out all the set numbers
for(i = 0; i < sets; i++)
{
   node_info *node_list;
   int uid = 0;
   int mode = 0;
   int nodes = 0;

   printf("set #: %d\n",data[i]);
   status = bsetstat(node_list, data[i], &uid, &mode, &nodes,
                                                bheadname);
   if(status < 0)
   {
      fprintf(stderr, "Error getting status of set!\n");
      exit(-1);
   }

   printf("\tuser id: %d\n", uid);
   printf("\tmode: %d\n", mode);
   printf("\tnum nodes: %d\n", nodes);

   // free vars, etc...
}
```

4. Below is an example of how to use bnodestat() and btypestat().

```
char bheadname[MAXHOSTNAME + 1];
int *data;
node_info *node_list;
int count = 0;
int status = 0;
int sets = 0;
```

```
int datanode = 0;
int uid = 0;
int mode = 0;
int nodes = 0;

// get the local host name
gethostname(bheadname, MAXHOSTNAME);

node_list = (node_info *) malloc(sizeof(node_info));
status = btypestat(node_list, SHARED, &nodes, bheadname);
if(status < 0)
{
   fprintf(stderr, "Error retrieving list of nodes in
                                specified mode!\n");
   exit(-1);
}

if(nodes > 0)
{
   status = bnodestat(data, &datanode, node_list[0], &mode,
                              &sets, &count, bheadname);
   if(status < 0)
   {
      fprintf(stderr, "Error retrieving information on node!\n");
      exit(-1);
   }

   printf("node: %d\n", node_list[0]);
   printf("\tip address: %d\n", datanode);
   printf("\tmode: %d\n", mode);
   printf("\tnum sets: %d\n", sets);
   printf("\tnum users: %d\n", count);
}
else
{
   printf("There were no nodes found in the specified mode\n");
}

// free vars, etc...
```

## A.3 Function Descriptions

**bactset()**

### Function Prototype

int bactset(int *data, int *sets, char *bheadname);

### Description

The function 'bactset' will return a list of all active sets in the 'data' argument. The number of active sets found will be returned in the 'sets' argument.

### Arguments

**int *data : output**

**int *sets : output**

**char *bheadname : input**

The argument 'data' is used by bactset() to return a list of all the active sets.

The argument 'sets' is used by bactset() to return the number of active sets.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

### Return Value

If bactset() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

### Errno

EBADREQ The request was not valid.

**balloc()**

## Function Prototype

int balloc(int nodes, int mode, node_info *data, int gro, int all, int bor, int ltn, char *bheadname);

## Description

The function balloc() attempts to allocate the number of nodes specified by 'nodes'. The nodes can be allocated in one of two modes as described by the 'mode' argument. The 'ltn' argument allows the user to specify that less nodes than requested are acceptable.

balloc() can borrow nodes from other clusters if there are not enough available on the local cluster and the 'bor' variable is set to allow borrowing. When borrowing nodes, there are two options that specify how the nodes should be borrowed. The argument 'gro' allows the user to specify the grouping. The argument 'all' allows the user to specify whether or not all nodes must be borrowed from one cluster.

## Arguments

**int nodes : input**

**int mode : input**

**node_info *data : output**

**int gro : input**

**int all : input**

**int bor : input**

**int ltn : input**

**char *bheadname : input**

The 'nodes' argument specifies the number of nodes to allocate. This number should be greater than 0.

The 'mode' argument allows the user to specify what mode the nodes should be allocated in. The possible modes are either SHARED or EXCLUSIVE. SHARED means that the nodes allocated can belong to more than one set. EXCLUSIVE means that the nodes allocated cannot be allocated to any other sets until they are freed.

The 'data' argument is used by balloc() to return the list of nodes allocated to the set. The 'data' object must point to a valid node_info structure. The structure must have memory allocated to it before calling balloc().

The 'gro' argument allows the user to specify the grouping of borrowed nodes. The possible values of 'gro' are PRIORITY or MOSTFIRST. PRIORITY means that nodes should be borrowed from clusters with the highest priority first. The priority of the clusters is defined in the balloc cluster config file. MOSTFIRST means that nodes should be borrowed from clusters with the most available nodes first. If two or more clusters have teh same number of nodes available, then PRIORITY is used.

The 'all' argument is used to specify whether or not all of the borrowed nodes must come from one cluster. The possible values for this argument are ALLON or ALLOFF. ALLON means that all of the borrowed nodes must come from the same cluster. ALLOFF means that all of the borrowed nodes do not need to come from the same cluster. If there are enough available nodes on the local cluster, this argument is ignored.

The 'bor' argument is used to specify whether or not nodes should be borrowed if there are not enough available nodes on the local cluster. The possible values for this argument are BORROWON or BORROWOFF. BORROWNON means that balloc() should attempt to borrow nodes if there are not enough available. BORROWOFF means that balloc() should not attempt to borrow nodes.

The 'ltn' argument is used to specify whether or not less nodes are acceptable. The possible values for this argument are LESSON or LESSOFF. LESSON means that less nodes are acceptable. LESSOFF means that either the number of nodes requested are allocated or none are allocated.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If balloc() is successful, the set number is returned and the argument 'data' contains the returned node addresses. If balloc() cannot meet the request, -1 is returned and errno is set.

## Errno

| | |
|---|---|
| EBADREQ | The request was not valid. |
| EUNAVAIL | The amount of requested nodes are not available. |
| ENOSETS | No more sets could be created. |
| EBADNUM | There were more nodes requested than the number of reserved nodes. |

**bfree()**

## Function Prototype

int bfree(int set, char *bheadname);

## Description

The bfree() function will remove all nodes from the set specified by the argument 'set'. If the set contained a set of borrowed nodes and those nodes do not belong to any other set, bfree() will return the group of borrowed nodes to it's original cluster.

## Arguments

**int set : input**

**char *bheadname : input**

The 'set' argument is the set number that will be freed.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If bfree() is successful, 0 will be returned. If it is not successful, -1 will be returned and errno will be set.

## Errno

| | |
|---|---|
| EBADREQ | The request was not valid. |
| EBADSET | The set specified was not found. |

**bnodestat()**

## Function Prototype

int bnodestat(int *data, int *datanode, int node, int *mode, int *sets, int *count, char *bheadname);

## Description

The function bnodestat() returns information on the specified node 'node'. The information returned includes a list of set numbers that contain the node, address of the node, the mode of the node, the number of sets containing the node, and the number of users on the node.

## Arguments

**int *data : output**

**int *datanode : output**

**int node : input**

**int *mode : output**

**int *sets : output**

**int *count : output**

**char *bheadname : input**

The 'data' argument is used by bnodestat() to return a list of set numbers that the node belongs to.

The 'datanode' is used by bnodestat() to return the IP address of the node.

The 'node' argument is the specified node to look up.

The 'mode' argument is used by bnodestat() to return the mode that the node is in. The possible modes are: UNKNOWN, FREE, SHARED, EXCLUSIVE, RESERVED, DOWN, BORROWED. UNKNOWN means that the status of the node cannot be determined. FREE means that the node is currently available. SHARED means that the set belongs to one or more SHARED sets. The node can be allocated to more SHARED sets. EXLUSIVE means that the node belongs to an EXCLUSIVE set and cannot be allocated to another set until it is freed. RESERVED means that the node has been RESERVED by the system for future use. The node cannot be allocated. DOWN means that the node is not currently operating. BORROWED means that the node has been loaned to another cluster.

The 'sets' argument is used by bnodestat() to return the number of sets that the node belong to. This is the size of the 'data' list.

The 'count' argument is number of users that are currently using the node.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If bnodestat() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

| | |
|---|---|
| EBADREQ | The request was not valid. |
| EBADNODE | The node specified was not available |

**breset()**

## Function Prototype

int breset(char *bheadname);

## Description

The function breset() reinitializes the entire set database. All of the nodes are freed and removed from all sets. This function should only be used for testing purposes and should be deprecated.

## Arguments

**char *bheadname : input**

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If breset() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

EBADREQ   The request was not valid.

**breturn()**

## Function Prototype

int breturn(int set, int importance, char *borrclustername, char *bheadname);

## Description

The breturn() function returns the set of borrowed nodes to it's original owner. This request should be sent to the cluster head that borrowed the nodes.

## Arguments

**int set : input**

**int importance : input**

**char *borrclustername : input**

**char *bheadname : input**

The 'set' argument is the set that contains the loaned nodes.

The 'importance' argument can be either NOW or WHENDONE. NOW means that the nodes must be returned immediately. There is no warning given, and any set containing those nodes will loose those nodes. WHENDONE means to wait until the jobs scheduled on the nodes are done, but no additional jobs can be scheduled.

The 'borrclustername' argument must point to a null terminating string that contains the network name of the cluster head that loaned the set.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If breturn() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

| | |
|---|---|
| EBADREQ | The request was not valid. |
| EBADSET | The set specified was not found. |

**bsetstat()**

## Function Prototype

int bsetstat(node_info *data, int set, int *uid, int *mode, int *nodes, char *bheadname);

## Description

The function bsetstat() returns information about the set specified by 'set'. The information returned is the list of the nodes in the set, user ID of the owner of the set, mode the set is in, and number of nodes in the set. The information returned about each node in the set includes the node number, IP address, and the number of users currently on the node.

## Arguments

**node_info *data : output**

**int set : input**

**int *uid : output**

**int *mode : output**

**int *nodes : output**

**char *bheadname : input**

The 'data' argument is used by bsetstat() to return a list of nodes in the set. It must point to a valid node_info structure. This memory must be allocated before calling bsetstat().

The 'set' argument should contain the set number for the set that is being queried.

The 'uid' argument is used by bsetstat() to return the user ID of the owner of the set.

The 'mode' argument is used by bsetstat() to return the mode of the set. The mode can be one of the following: SHARED, EXCLUSIVE, or BORROWED. SHARED means that any node in the set can be allocated to another set that is also in the SHARED mode. EXCLUSIVE means that all nodes in the set are exclusively allocated to the set. None of the nodes can be allocated to future sets until they are freed from this set. BORROWED means that the set is not part of the cluster's current resources. The nodes in the set have been loaned to another cluster.

The 'nodes' argument is used by bsetstat() to returned the number of nodes in the set.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If bsetstat() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

| | |
|---|---|
| EBADREQ | The request was not valid. |
| EBADSET | The set specified was not found. |

**btypestat()**

## Function Prototype

int btypestat(node_info *data, int mode, int *nodes, char *bheadname);

## Description

The function btypestat() returns a list of nodes that are in the specified mode.

## Arguments

**node_info *data : output**

**int mode : input**

**int *nodes : output**

**char *bheadname : input**

The 'data' argument is used by btypestat() to return the list of nodes that are in the mode specified by 'mode'. The 'data' argument must point to a valid node_info structure. The structure must have memory allocated to it before calling btypestat().

The 'mode' argument should specify the mode to look up.

The 'nodes' argument returns the number of nodes that are in the specified mode.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If btypestat() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

EBADREQ   The request was not valid.

**node_info_to_job_map()**

## Function Prototype

char * node_info_to_job_map(node_info *node_count);

## Description

The function node_info_to_job_map() takes a list of nodes 'node_count' and returns a list of node numbers in the job map format for the environment variable BEOWULF_JOB_MAP.

## Arguments

**node_info *node_count : input**

The 'node_count' argument should contain a list of nodes that will be returned in job map format.

## Return Value

If node_info_to_job_map() is successful, a job map is returned with the form nodenumber: nodenumber:...:nodenumber. If it is not successful, NULL is returned and errno is set.

## Errno

EBADREQ The request was not valid.

**ballocresv()**

## Function Prototype

int ballocresv(int nodes, int setnum, borr_node_info *data, char *bheadname);

## Description

The function ballocresv() allocates previously reserved nodes in the specified set.

## Arguments

**int nodes : input**

**int setnum : input**

**borr_node_info *data : output**

**char *bheadname : input**

The 'nodes' argument specifies the number of nodes to allocate. This can be less than or equal to the number originally reserved.

The 'setnum' argument specifies the reserved set that the nodes are in.

The 'data' is used by ballocresv() to return the list of nodes that have been allocated. The 'data' object must point to a valid borr_node_info structure. The structure must have memory allocated to it before calling ballocresv().

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. This argument will typically be localhost for users.

## Return Value

If ballocresv() is successful, the new set number is returned. If it is not successful, -1 is returned and errno is set.

## Errno

| | |
|---|---|
| EBADREQ | The request was not valid. |
| EBADNUM | There were more nodes requested than the number of reserved nodes. |

**bresv()**

## Function Prototype

int bresv(int nodes, int *return_nodes, char *bheadname);

## Description

The function bresv() reserves the number of nodes specified by 'nodes' for 10 seconds. During the 10 seconds balloc() will decide whether or not it needs to borrow those nodes. If balloc does not claim the nodes within 10 seconds, they are freed.

## Arguments

**int nodes : input**

**int *return_nodes : output**

**char *bheadname : input**

The 'nodes' argument specifies how many nodes to reserve.

The 'return_nodes' argument is used by bresv() to return the actual number of nodes reserved, because balloc assumes less nodes are acceptable for reservation. This allows the number of nodes on clusters to be compared.

The 'bheadname' argument must point to a null terminating string that contains the network name of the cluster head to send request to. The reserved nodes will be reserved on the cluster referenced by this argument.

## Return Value

If bresv() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

| | |
|---|---|
| EBADREQ | The request was not valid. |
| ENOSETS | No more sets could be created. |

**ntoh_borr_info()**

## Function Prototype

int ntoh_borr_info(int *new_data, int numnodes, borr_node_info *data);

## Description

The function ntoh_borr_info() takes the returned data 'new_data' and converts it to the borr_node_info list 'data'. The data is translated from network to host byte order and organized into the 'data' list. This function is used by balloc calls to change byte order of data returned from the balloc daemon.

## Arguments

**int *new_data : input**

**int numnodes : input**

**borr_node_info *data : output**

The 'new_data' argument contains a list of data that is in network byte order. The sequence of data, as found int the borr_node_info structure, is nodenum, nodeaddr, count, eaddr, repeat.

The 'numnodes' argument is the number of nodes in the list.

The 'data' argument is used by ntoh_borr_info() to return the list of nodes.

## Return Value

If ntoh_node_info() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

EBADREQ The request was not valid.

**ntoh_node_info()**

## Function Prototype

int ntoh_node_info(int *new_data, int numnodes, node_info *data);

## Description

The function ntoh_node_info() takes the returned data 'new_data' and converts it to a node_info list. The data is translated from network to host byte order and organized into the 'data' list. This function is used by balloc calls to change byte order of data returned from the balloc daemon.

## Arguments

**int *new_data : input**

**int numnodes : input**

**node_info *data : output**

The 'new_data' argument contains a list of data that is in network byte order. The sequence of data, as found in the node_info structure, is nodenum, nodeaddr, count, next_node, repeat.

The 'numnodes' argument is the number of nodes in the list.

The 'data' argument is used by ntoh_node_info() to return the list of nodes.

## Return Value

If ntoh_node_info() is successful, 0 is returned. If it is not successful, -1 is returned and errno is set.

## Errno

EBADREQ The request was not valid.

# Bibliography


[1] IBM Corporation. RS/6000 SP System Management: Easy, Lean, and Mean. Technical report, International Technical Support Organization, June 1995.

[2] Platform Computing Corporation. LSF Reference Guide: Version 4.2. Technical report, June 1994.

[3] Platform Computing Corporation. Using LSF with IBM SP-2. Technical report, 2000.

[4] Thinking Machines Corporation. The Connection Machine: CM-5 Technical Summary. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, October 1991.

[5] Veridian Corporation. PBS Unix Manual Pages. Technical report, June 2000.

[6] Amin Vahdat Douglas Ghormley, David Petrou and Keith Vetter. Global Layer Unix: GLUnix. http://now.cs.berkeley.edu/Glunix/glunix.html, 1997.

[7] et. al. Dr. Walter B. Ligon, Dr. Daniel C. Stanzione. Scyld Beowulf Training. Technical report, Scyld Computing Corporation, 2001.

[8] Supercluster Development Group. Maui Documentation: Sections: History, Overview, Quick Start Guide, User's Manual, Administrator's Guide. http://www.supercluster.org, 2000.

[9] Supercluster Development Group. Wiki RM Interface Specifications Version 1.1. http://www.supercluster.org, 2000.

[10] Cornell University IBM SP. Extensible Argonne Scheduling System Load Leveler. http://www.tc.cornell.edu/UserDoc/SP/Batch/Easy.

[11] Cray Incorporated. Network Queueing Environment(NQE): Software Guide. http://www.cray.com/products/software/nqe.html, 2001.

[12] Southampton Oceanography Center James Rennell Division. Pexec man pages. http://www.soc.soton.ac.uk/JRD, October 1997.

[13] Charles E. Leiserson. The Network Architecture of the Connection Machine CM-5. pages 1–18, October 1992.

[14] LINUX. LINUX man pages: at, atd, 2000.

[15] University of Utah. Usage of the Compaq Sierra. Technical report, Center for High Performance Computing, July 2001.

[16] The Globus Project Team. The Globus Grid Programming Toolkit Tutorial. Technical report, The Globus Project Team, ANL, and USC/ISI, November 1999.

[17] University of Wisconsin-Madison The Condor Team. Condor High Throughput Computing. http://www.cs.wisc.edu/condor.

[18] Donald J. Becker Thomas Sterling, John Salmon and Daniel F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. 1999.