Load Balancing in Beowulf Clusters

Chandramohan Rangaswamy
Department of Electrical and Computer Engineering
University of Illinois at Chicago
July 07, 2001

1 Abstract

Beowulf[1] Clusters are growing in popularity as a means for cheap and powerful distributed systems. Distributed systems demand a new set of tools for running the systems at optimal efficiency. One such tool is load-balancing. It is well known that distributing the loads in a distributed system, so that all the computers in the cluster run at more or less the same load, increases the efficiency of cluster. A heavily skewed cluster, has one or two workstation running all the tasks in the system, while other workstations remain idle. Jobs submitted to the heavily-loaded workstations take longer time to complete. Distributing the jobs submitted, to all the systems in the cluster reduces the time required to execute jobs and also increases the throughput of the cluster. A load balancing tool thus forms an invaluable part of any distributed system. Developing a load balancing tool for the Beowulf clusters is the goal of the project.

2 Beowulf Cluster computing

Cluster computing consists of running a network of computers as one distributed computer. Clusters can be formed using a Network of Workstations (NOW), or Cluster of Workstations (COW). In a Network of Workstations, workstations are loosely coupled, a computer lab can simply be used as a cluster. A Cluster of Workstations is more tightly coupled in that systems are more or less dedicated to be used as one distributed system. Beowulf clusters is in a way a misnomer since a
NOW can be run as a “cluster”. Many of the Beowulf clusters presently available are pure clusters in that dedicated machines are used to form the cluster. The main criterion for a cluster to be a Beowulf is that it should run Linux operating system. Although, now there are Windows NT clusters and Solaris clusters, only Linux clusters count as Beowulf.

A typical Beowulf cluster consists of a server which is connected to a network of workstations. Only the server has access to external networks. No special type of network topology is used for interconnection. All the systems are connected together through Ethernet. All the components in the Beowulf cluster are Commercial Off The Shelf (COTS) systems. All the systems except the server are bare-bone systems. They don’t have a monitor or keyboard or mouse connected to them. Normal interaction is through the network by using rsh. For maintenance purposes, a KVM kit is used. A KVM kit is a Keyboard-Video-Mouse kit, and is connected to the host workstation whenever maintenance work is to be done on the workstation. In some of severely cash-strapped clusters, even no video card is installed. In this case all maintenance work is done through serial port. A Linux terminal can run through serial port. Hence, even video cards are done away with. Only a network card is connected to the motherboard.

The operating system used is of course, Linux. Only the bare essentials are installed. This usually consists of development packages and applications. Normally X Window system is not installed unless the cluster is a Network of Workstations like a computer lab. Usually, software on all the nodes of the cluster are exactly the same (unless the system administrator really wants a nightmare). The home directory (/home) is usually located on the server only. The server is also a NFS server. This allows the nodes’ software to be easily replaceable. If any one workstation goes down or crashes, it can easily be brought up. Backup of entire cluster need not be made. Backup of only the server is required. Other nodes can be restored easily just by installing the OS again. Though, a backup of nodes' network settings will make the job even more easier.

Beowulf systems are fairly new in that have been around for only a few years. Hence special tools for running these clusters are not available or limited in their abilities. For example, load balancing tools available for Linux clusters are Condor[2], GNU Queue[3], MOSIX[4] and PANTS[5]. Each of the tools mentioned above have their advantages and disadvantages. These tools, however, have drawbacks in that they are not transparent (working unknown to user) or being excessively large. Before taking a look at the workings of each systems, an explanation of what is expected of a load balancing system is necessary.

The first and foremost task for a load balancing system is distributing tasks
in a cluster to all the systems in the cluster, so that all the systems are loaded equally. Load of the machine is defined as the cumulative of number of tasks running in the system and the intensity of use of system resources by the tasks. Lightweight tasks like shell are not at all resource intensive. For example, the bash shell has a relatively very low memory foot print (about 1.3 MB) and very low CPU usage (0.0). On the other hand, the X window system server running the new KDE 2.0 window manager has a large memory foot print (about 76MB), the GNU C compiler has a foot print varying from 9MB to 30MB and is very CPU intensive. A load balancing system should take all these variations in resource usage when allocating tasks to systems on the cluster.

Load balancing system should not be resource intensive, and should be fast enough so that there is no visible performance penalty on account of running the load balancing program.

3 General Techniques for Load Balancing

The problem of load balancing can be approached from different viewpoints and hence different solutions exist. These solutions fall into two techniques.

- Scheduling
  - Kernel space scheduling
  - User space scheduling
- Process migration

(Process migration in user space is difficult to the point of being impossible. Hence process migration is performed only in kernel space)

3.1 Scheduling

In the scheduling solution, whenever a new job is to be submitted to the cluster, the job is submitted to the node on the cluster which is optimally suited to take that load. Optimal load may mean the node with the lowest load on a cluster of same type and hence equally powerful machines. In case of clusters with dissimilar nodes, one or two machines might be much more powerful than other machines in the cluster. Hence, machines with optimal load may not be the machines with the lowest load.
3.1.1 User space scheduling

User space scheduling is the easiest of the load balancing both in theory and implementation. In user space scheduling, jobs are submitted to the cluster through the load balancing program. The load balancing spawns the job on the node with optimal load. Signals that are delivered to the program might be delivered through a stub (proxy) or may not be delivered at all. Most of the User space scheduling programs do not support programs which fork new processes and which use Interprocess Communication (IPC). To support IPC and fork, the programs may have to be recompiled or relinked with new libraries which implement system calls so that IPC and forking is possible over a network.

3.1.2 Kernel space scheduling

Kernel space scheduling makes changes in the kernel scheduling code so that jobs are scheduled on the node with optimal load. Scheduling is completely transparent to the user. Programs which make use of IPC, forks, execs and spawns are also supported. Whenever a job is submitted to the node, the kernel of that node checks if the current node is the optimal load for the execution of the job. Otherwise it finds the appropriate node and submits the task to that kernel for execution. Note has to be made here that programs once started in a particular node run to completion on that node only. Changes to the kernel code are required both in the task allocator and also in the signal delivery mechanisms. The task allocator is changed so that it checks the load condition whenever it creates a new task. The signal delivery mechanisms are changed so that signals are delivered over the network.

Kernel scheduling, although much better than user space scheduling, is much more complicated to implement as changes has to be made in many parts of the kernel. Also, the code for kernel scheduling has to change whenever changes are made to the kernel source code. Linux kernel is changed once every six month. Although no major changes take place in the kernel scheduler, kernel scheduling is tightly bound to the kernel source code and hence patches have to be released whenever a new kernel is released.

Another problem in case of kernel scheduling is that whenever program is created, the kernel has to find the cluster’s load. This results in delays even in creating programs which may not change the load of the node very much.

The scheduling technique has another inherent flaw, in that load balancing is not continuous. Load balancing is done only when a new program is created.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Memory</th>
<th>CPU Time</th>
<th>Load Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle (no image)</td>
<td>15MB</td>
<td>0.1</td>
<td>0.04</td>
</tr>
<tr>
<td>New Image (256x256x400ppi)</td>
<td>29MB</td>
<td>0.1-5.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Script-Fu: Line Nova</td>
<td>39MB-109MB</td>
<td>0.1-82.9</td>
<td>1.09-1.72</td>
</tr>
</tbody>
</table>

Table 1: Gimp Run

Once a program is created, changes in load condition cannot be accommodated. Due to this nature, the load on the nodes can become unbalanced even when load balancing program is running. Allocation of a task is dependent only on the load at the point of creation of task. The load on the node might increase after the task has been allocated. A program that was waiting for user input might use very little memory and CPU time, once user input is obtained, its memory and CPU time use might change drastically. For example, the data in Table.1 was recorded while running the gimp program.

As can be seen, the resources utilized by the program varies from time to time. If another resource intensive process is created before running the Script-Fu script, both the processes will be contending for resources, raising the load on the system and reducing throughput. But during this time, there is no way of transferring one of the processes to another system. The Script-Fu script took about 4 minutes to complete when it was run without any other CPU intensive programs. It took about 7.13 minutes when it was run along with kfract, the KDE fractal generator.

These deficiencies are overcome by the Process Migration technique.

### 3.2 Process Migration

Processes that are already created and being run in a particular node, can be moved to another node. Moved processes can then continue to run in that node. This is known as Process migration. Migrating a process, while tedious and cumbersome provides for finer control over the load in a cluster. This is particularly important in clusters with single point of entry like the Beowulf clusters. With single point of entry, all the processes tend to be created on a single machine, unless one pays regard to the load on the machine and starts his/her programs on another node using rsh. Moving the processes to a different machine reduces the load on the machine in a transparent way.

Load balancing programs using process migration can continuously monitor
the load on the cluster and react to changes in load. Although process migration is better for load balancing than other techniques not many programs use process migration (except for MOSIX[4]). The reasons for this include tedious nature of coding for process migration and non-availability of kernel sources (for Unix systems other than FreeBSD and Linux).

More detailed explanation of the Process Migration is provided in the Proposed Solution section.

4 Currently Available Systems

4.1 Condor

The Condor project[2], started development at University of Wisconsin, Madison in 1988. It was based on Remote-Unix project developed through the efforts and direction of Professors D. Dewitt, R. Finkel and M. Solomon. The Condor project was directed by Professor M. Livny. (Source: http://www.cs.wisc.edu/condor/background.html).

Condor system is currently available for both Unix and NT systems. Condor is essentially a batch processing system. Jobs that are submitted are put in a queue, the jobs are then run on different machines depending on the queuing mechanism, scheduling policy, priority scheme and resource classifications. Users are then informed about the results of execution. Machines in the Condor’s pool can be dedicated compute servers or can be non-dedicated machines. That is machines running idle in a computer lab can easily be included in the Condor pool.

Condor systems present many features. One of those important feature is Checkpointing and migration. Checkpointing is collecting important execution information about a process at regular intervals or at selected times. This way processes can be executed from the checkpoint on another machines. This is helpful in case of system failure like crashes or removal of the machine from pool or machine being shutdown. Since run-time information of machines is available, checkpointed processes can be moved from one machine to another. In Condor, Checkpointing and migration requires relinking of programs.(Some other features of Condor also require relinking with Condor libraries). Checkpointing and migration is useful for providing priority for users. Condor provides priority for machine owners rather than to the programs run by itself.

Condor also supports remote system calls over network. Another feature of
Condor is the **ClassAds** mechanism. In ClassAds mechanism, systems advertise their resources and users advertise their requirements. Condor system matches user with the system such that user’s requirements are fulfilled by the system assigned. For example, if an user requires 96 MB RAM to run a task, then the Condor system might assign that task to a system which announces that it has more than 96 MB of free memory.

Programs that are to be checkpointed and migrated have to be relinked with Condor libraries and have severe limitations imposed on them. The limitations are

1. Processes cannot fork, exec or spawn.
2. Interprocess communication is not possible.
3. Processes can open sockets, but the communication has to be brief.
4. Some of the signals, alarms, timers and sleeping are not allowed.
5. Multiple kernel-threads are not supported. User level threads can be used.
6. Processes cannot use mmap or munmap. (Memory mapped files)
7. File locks are not retained between checkpoints.
8. Files can only be opened read-only.
9. Jobs must be statically linked. Dynamically linked cannot be checkpointed.

The typical mini-HOWTO for running a job in Condor is,

1. Choose a run-time environment (**Universe**, Vanilla, Standard)
2. Create and submit a description file for the program.
3. Submit the job using **condor_submit**

The run-time environment for a Condor system is called an **Universe**. There are five types of universe to choose from. They are Standard, Vanilla, PVM, MPI and Globus. The typical universes are Vanilla and Standard. Vanilla universe is used to run programs that cannot be relinked. These programs cannot be checkpointed and hence cannot be migrated. In this case, programs have to run to completion in the system they are started from. Otherwise, the job has
to be restarted from the beginning on another machine. If the machine that is used for running the program is required by its owner, the Condor system either suspends the task or starts the job from the beginning in another machine. This also requires that all data are accessible from different machines through NFS or AFS.

Standard universe is used for programs which can be relinked. These programs are checkpointed and hence carry the limitation mentioned above. Programs are relinked using the `condor_compile` program.

The description file contains commands and keywords for directing the queuing of jobs. A ClassAd job is created depending on the description file. The job itself is run by through the `condor_submit` program.

The program can only be installed by root. It starts a variety of daemon processes, all running either under root or in user `condor`. The main daemon, `condor_master`'s task is to make sure that all of Condor's daemon's are running in all the machines in the pool. It restarts any daemons that crash and removes any machine from its available pool of machines if that machine is down. One of the machines has to be set as central manager machine and Condor has to be started on this machine first.

The Condor system provides quite so many options. This makes it a very versatile system. But it is a user space program and hence is limited in its reach. Most of the computation-hungry programs are not available in source, object form. Even if they are available in source form or in object form, a system administrator has to spend a great deal of time maintaining the system. He/she has to relink all the programs with Condor libraries. If these programs are relinked, these programs has to meet the requirements for checkpointing. The list of conditions for checkpointing effectively precludes many of the programs, for any non-trivial program will have some file read/write at the minimum.

Programs that are not relinked (running in Vanilla universe) on the other hand cannot be migrated. This means restarting an application from the start on another machine. While this can be done for programs with small run-time, doing this for programs which run for days (physics programs, VHDL compilers, simulators) is not feasible and is a wastage of resources.

4.2 GNU Queue

GNU Queue is a user space network load-balancing scheduler. It acts as a proxy to the programs that are run by it on another machine. As quoted in the developer's website it is similar to telnetting into the machine and running the application
there, except that this process is done in a machine selected by the Queue system. The queue program runs a daemon on all the machines in the cluster. Whenever an application is submitted to the queue program it queries all the cluster queued daemons for the load status. Depending on this data it runs the process on the machine with least load. The queue itself starts running as a proxy for the program that is running in the remote host. Any signals that are to be delivered to the program are delivered to the proxy which delivers the signals to the program.

The queue system is much simpler than Condor and has less footprint in regard to both CPU usage and memory. Installation and running queue is much simpler. Installation can be both done through users and through system administrator. Installation by user is less secure than installation by super user. Installation follows the standard Gnu program installation. (configure;make; make install). Source code of the project is available under GNU license and hence can be modified by users.

To execute a program in queue, the queued daemon must be installed in all the hosts specified in host access control list. After this installation, 
```
queue -h hostname --program
```
runs the program in hostname. If the hostname is omitted, queue runs the program in any machine with the least load or best machine depending on the Queue’s profile file. Profile file allows one to set various options for controlling queue operation. Options like minimum free space required, load average of system can be specified in the profile file. Queue supports MPI and PVM support is forthcoming.

### 4.3 MOSIX

MOSIX is being developed by Prof. Amnon Barak of Institute of Computer Science at The Hebrew University in Jerusalem, Israel. He and his team have developed MOSIX for many of Cluster OSES like Linux and NT. Infact the Linux implementation is seventh implementation of the MOSIX system. MOSIX system is the most complete implementation of a load balancing system for a Beowulf Cluster. It turns the cluster into almost a Single system with multiple processors (like an SMP system).

MOSIX is a kernel level program with preemptive process migration. Processes are migrated to nodes with less load as soon as load on the current node is greater than the threshold value. It also implements a separate file system of its own for its operation. The file system implemented is a cluster-wide shared
filesystem called MFS.

Installation of MOSIX is a task for the system administrator alone as it involves a kernel level change. MOSIX distribution site provides kernel patches for the kernel. This kernel changes are to be applied to the kernel source and the new kernel has to be compiled. This kernel has to be implemented in all the hosts in the cluster. An installation tool has been developed to automate some of these tasks.

MOSIX can be configured for operating in a Multi-user, time sharing environment in four different configurations.

Single-pool All the servers and workstations belong to a single cluster and are shared like-wise.

Server-pool Only the servers are shared, workstations work as a single units.

Adaptive-pool All servers are shared, workstations join or leave the cluster independently.

Half-duplex pool Servers are shared. Processes from workstations can be run on the server pool cluster.

These configurations may not be static. These configurations can be dynamically changed. A queuing system is also implemented to run processes in a batch.

The MOSIX system implements kernel level changes and hence is completely transparent to the user. However MOSIX system does have some disadvantages. It does not implement checkpointing. It uses `copy_from_user` and `copy_to_user` primitives over a network to communicate between user space and kernel space. That is, if a kernel in one machine needs to communicate with a program running on another host (for example, to deliver a signal) then the kernel has to move data from user space to kernel space and vice versa. This is done often in system calls. When this is done over a network it leads to reduction in system performance. Even though, MOSIX does pre-fetch data needed, it still leads to a lot of network traffic and more than network traffic, it leads to reduction in performance.

Another main problem with MOSIX is that there are frequent changes in the kernel and hence the system has to be modified for each kernel version. Hence, installing a new kernel means, download the kernel source, download the MOSIX package, apply the patch, compile the kernel and install the kernel. If the system
is modified so that it can be installed as a kernel module, then this problem might be solved.

4.4 PANTS

PANTS is a process migration tool. PANTS is PANTS Application Node Transparency System. PANTS uses checkpointing to migrate processes from one host to another. The initial version of PANTS was developed by Jeffrey Moyer. It was developed in Worcester Polytechnic Institute (WPI). The first version of PANTS used Preemptive Process Migration, but since it was architecture dependent, it was removed in PANTS v2.0[6]. The first version uses the EPCKPT[7], a patch to the Linux Kernel which adds process checkpointing and restart capability. PANTS used this to stop a running program, package it and transport it to an available node in the cluster and restart it remotely.

A multicast-based system is used to communicate with all the nodes. All the nodes which can accept a load can initiate a transfer of programs. This is known as receiver-initiated transfer. Similarly nodes which require a transfer of load, can initiate a transfer. This is known as sender-initiated transfer. PANTS uses a leader-based multicasting with sender initiated transfer.

PANTS v2.0 is in a way, a complete rewrite of PANTS. It is being developed by Kevin Dickson, Chuck Homic and S. Bryan Villamin. Version 2 does not implement a process migration. Load sharing is done only when there is a process being created. For this the execve system call is patched and is replaced. The replaced version talks with the PANTSD daemon when a process is created. This allows load sharing, but no process migration. However since this patch is simple enough to be applied for all architectures, the PANTS program is not independent of architecture. PANTS does not support IPC as of now.

5 Proposed Solution

A system which uses process migration is proposed for load balancing system. Load balancing based on process migration allows a more rigorous load balancing. The system is to be developed as a kernel module. This allows the system to be much more independent of kernel source and is more easily accepted for inclusion in kernel sources for distribution.

Load balancing systems use many different type of communication networks for communicating with other nodes. Some of the types are
• Broadcast
• Multicast
• Neighborcast

5.0.1 Broadcast

Broadcast type systems broadcast all their requirements to the whole network. This guarantees that there is no central point of failure. But this increases the network traffic heavily. Once a sender (the node that needs a process to be transferred) sends out a request for transferring a process, then it is sent to all the systems in the network. Then all the receivers (the nodes that receive the transferred process) respond to that request by sending out a positive or negative acknowledgment. So, this system without any optimization generates \((n - 1)^2\) requests and acknowledgments for one transfer. Besides that, a good arbitration policy, that does not generate any more traffic is needed to select which node finally receives the process to be transferred.

5.0.2 Multicast

Multicast systems have two different multicast subnets. One is the sender multicast group and the other is the receiver multicast group. Both the groups are dynamic in that, node that is heavily loaded joins the sender group while lightly loaded nodes join the receiver multicast. This reduces the traffic that takes place whenever there is a request for transfer. In this case, a request is sent only to the receiver group and only the receiver group nodes acknowledge the requests. This still needs a good arbitration policy. But any arbitration policy with this kind of setup is guaranteed to generate additional traffic. Hence a setup with a multicast leader is normally used. In this type of system, all requests and acknowledgments are routed through a separate node. This introduces a central point of failure, but reduces the bandwidth requirement.

5.0.3 Neighborcast

“Neighborcast” is transmitting only to the neighbors. Here a sender has only two receivers (Except for edge nodes, which have only one receiver). Its left and right neighbors. Whenever a process is to be transferred, the node selects either its left or right neighbor depending on the load on those machines. The
neighborcast system produces a less balanced cluster, since it takes a longer time for all the nodes to reach equilibrium. But this system reduces network traffic, does not need an arbitration policy and most of all, does not have a central point of failure. This solution is also inherently scalable. The nodes have to care about only its neighbors. But since all neighbors are covered, this system leads to a balanced cluster. Traffic generated by broadcast and multicast systems increases proportionally as number of nodes increase. But in this system, the traffic increases only linearly.

The neighborcast system is selected for implementing this project.

All load balancing systems should implement a transfer, selection, information and location policy.

5.1 Location Policy

Location is the node to which the selected process goes. This is either the left or right neighbor. The transfers can be propagated by the nodes, that is, if a node is unable to take the request, the request is passed on to its neighbor and so on. While this propagation can be used to reach the least loaded node to run the process, it is not used as it can lead to more hops and also to race conditions. That is, if two nodes send out a request for transfer, and both request land at the same node, then which request to choose? Also what happens to the request that was denied? Where does the request go to? To avoid these kind of problems, a neighboring node may not reject a request in most cases.

5.2 Transfer Policy

Transfer policy determines how the transfer begins. The most logical transfer policy would be the one where the sender initiates the transfer. The other policies possible are receiver-initiated and joint-initiated. When the load on the whole cluster is low or is heavily skewed, then receivers can initiate a transfer. This policy does that have its advantage that the already heavily-loaded sender has to spend less CPU cycles negotiating for a transfer. In the joint initiated, both the sender and receiver can initiate a transfer. In this system, a sender-initiated is used.
5.3 Information Policy

The information policy becomes the most important policy when the neighborcast system is used. A transfer is initiated only when the load of the node is greater than a threshold value. This threshold value is dependent on the overall cluster load and is a function of the overall load. Due to this, the way, cluster load is collected becomes important. The cluster load is collected as a moving average of the loads of individual nodes. This process starts with the first machine that enters the cluster. What is the information that is to be collected? These can be the disk utilization, CPU time, etc.

5.4 Selection Policy

Selection policy is selecting which process is to be transferred to the next node. The selection policy should not select a process that may just run for a few seconds for the transfer. Also, transfer of processes with large footprints should be avoided as much as possible.

Processes not only use CPU resources, but also use other system resources like files, IPC etc. In this case, selection of process to transfer becomes important. As much as possible, a process which does not use many of the system resources should be selected. Of course, processes may use files or other resources after they are moved. In this case implementation should provide for a way to access these resources over a network.

To aid in moving the processes, the processes must be checkpointed. Fortunately, a checkpointing solution for Linux is already available from Eduardo Pinheiro from Rutgers University. Although its availability as a module remains to be investigated, the checkpointing library can be used for moving the processes.

6 Summary

There are many approaches available for solving the problem of load balancing. The proposed implementation is just one of them. Although the proposed solution may not lead to a very quick load balancing to start with, it will definitely lead to a stable and fault-tolerant load balancing system.
References


