

# Infrastructure for Load Balancing on Mosix Cluster

MadhuSudhan Reddy Tera and Sadanand Kota  
Computing and Information Science, Kansas State University  
Under the Guidance of Dr. Daniel Andresen.

## Abstract

*The complexity and size of software are increasing at a rapid rate. This results in the increase in build time and execution times. Cluster computing is proving to be an effective way to reduce this in an economical way. Currently, most available cluster computing software tools which achieve load balancing by process migration schemes, do not consider all characteristics such as CPU load, memory usage, network bandwidth usage during migration. For instance, Mosix, a cluster computing software tool for Linux, does not consider network bandwidth usage and neither does it consider CPU usage and memory characteristics together. In this paper we present the infrastructure for efficient load balancing on Mosix cluster through intelligent scheduling techniques.*

## Introduction

As computers increase their processing power, software complexity grows at an even larger rate in order to consume all of those new CPU cycles. Not only does the running of the new software require more CPU cycles, but the time required to compile and link the software also increases. The basic idea behind clustering approach is to make a large number of individual machines act like a single very powerful machine. With the power and low prices of today's PCs and the availability of high performance Ethernet connections, it makes sense to combine them to build High Performance Computing and Parallel Computing environment. This is the concept

behind any typical clustering environment such as Beowulf parallel computing system, which comes with free versions of UNIX and public domain software packages.

Mosix is a software that was specifically designed to enhance the Linux kernel with cluster computing capabilities. It is a tool consisting of kernel level resource sharing algorithms that are geared for performance scalability in a cluster computer. Mosix supports resource sharing by dynamic process migration. It relieves the user from the responsibility of allocation of processes to nodes by distributing the workloads dynamically. In this project, we are concentrating on homogeneous systems, wherein we have machines with same family of processors running the same kernel.

The resource sharing algorithm of Mosix attempts to reduce the load differences between pairs of nodes (systems in the cluster) by migrating processes from higher loaded nodes to lesser loaded nodes. This is done in decentralized manner i.e. all nodes execute the same algorithms and each node performs the reduction of loads independently. Also, Mosix considers only balancing of loads on processors and responds to changes in loads on processors as long as there is no extreme shortage of other resources such as free memory and empty process slots. Mosix does not consider certain parameters such as network bandwidth usage by a process running on a node. In addition, Mosix distributes the load evenly and does not give the user, the control of load distribution i.e. if the user wants only few of the machines to be evenly

loaded and few others to be heavily /lightly loaded, he will not be able to do this. Our project aims to overcome these shortcomings. Our initial scheduling technique is decentralized and tries to give user, the control of balancing the load on various machines. The scheduling algorithms, we use, try to achieve balance in load, memory and network bandwidth by collecting performance metrics of a process through Performance Co-pilot (PCP), a framework and services to support system-level performance monitoring and performance management from SGI. We also propose an implementation, which is based on a centralized scheduler, which tries to eliminate the problems of decentralized scheduling, such as every node trying to move their CPU intensive processes to a lightly loaded node. The centralized scheduler also takes care of network bandwidth usage of a process and tries to reduce the overall network bandwidth consumption by migrating the communicating processes to single node in addition to balancing the load on individual processes as required by the user.

### **Load Balancing**

The notion in Mosix cluster is that whenever a system in the cluster becomes heavily loaded, then the load is distributed in the cluster. The dispatching of tasks from heavily loaded system and scheduling it to a lightly loaded system in the cluster is called *load balancing*. Load balancing can be divided into following phases:

a. *Load Evaluation* phase: “ The usefulness of any load balancing scheme is directly dependent on the quality of load measurement and prediction.” [Watts98] Any good load balancing technique not only has good measurement of load, but also, sees that it does not affect the actual load on the system.

b. *Profitability Determination* phase: We should perform load balancing only when the cost of imbalance is greater than cost of load balancing. This comparison of cost of imbalance vs. cost of load balancing is the profitability determination. Generally if cost is not considered during actual migration, an excessive number of tasks can be migrated, and this will have negative influence on the system performance.

c. *Task Selection* phase: Now we must select a set of tasks that must be dispatched from the system so that the imbalance is removed. This is done in the task selection phase. The task should be selected in such a way that moving the task from the system would remove the imbalance to a large extent. For instance, we can see the proportion of CPU usage of the task on the system. We should also consider the cost of moving the task over the link in the cluster, size of the transfer, since larger tasks will take longer time to move than the smaller ones.

d. *Task Migration* phase: This is the final phase of load balancing in the cluster. This step must be done carefully and correctly to ensure continued communication integrity.

In the following sections, we will explore two most popular cluster computing technologies namely, Mosix and Condor and also conclude why we chose Mosix. We continue the paper with our implementation and provide sample test results.

### **Mosix**

Mosix is a cluster-computing enhancement of Linux, which allows multiple uni-processors, and SMP's running the same version of kernel to share resources by preemptive process migration and dynamic load balancing. Mosix implements resource-sharing algorithms, which respond to load variations on individual computer systems by migrating processes from one workstation to another, preemptively. The

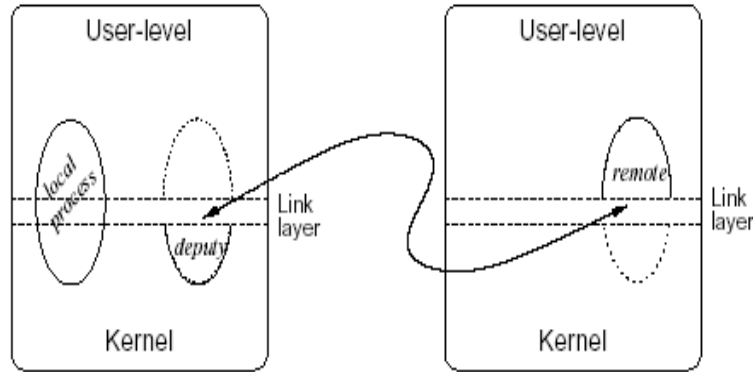


Figure 1: A local process and a migrated process

goal is to improve the overall performance and to create a convenient multi-user, time-sharing environment for execution of applications. Unique features of Mosix are

a. *Network transparency*: For all network related operations, the application level programs are provided with a virtual machine that looks like a single machine i.e. the application programs do not need to know the current state of the system configuration.

b. *Preemptive process migration*: It means that it can migrate any user's process, transparently to any available node at any time. Transparency in migration means that the functional aspects of the system behavior should not be altered as a result of migration.

c. *Dynamic load balancing*: As explained earlier, Mosix has resource sharing algorithms, which work in decentralized manner.

The granularity of work distribution in Mosix is process. Each process has Unique Home Node (UHN) where it was created. Process that migrates to other nodes (called 'remote') use local (in the remote node) resources whenever possible, but interact with the user's environment through the UHN, for e.g. `gettimeofday()` would get the time from the UHN. Preemptive process

migration in Mosix is implemented by dividing the migrating process into two contexts: user context ('remote')– that can be migrated, and system context ('deputy') – that is UHN dependent and cannot be migrated (see figure 1). The 'remote' consists of stack, data, program code, memory maps and registers. The 'deputy' consists of description of the resources, which the process is attached to, and a kernel-stack for the execution of the system on behalf of the process. The interaction of 'deputy' and the 'remote' is implemented at the link layer as shown in figure 1, which also shows two processes sharing a UHN, one local and a deputy.

Remote processes are not accessible to other processes that run at the same node and *vice versa*. They do not belong to any particular user nor can they be sent signals or otherwise manipulated by any local process. They can only be forced by system administrator to migrate out. The deputy does not have a memory map of its own. Instead, it shares the main kernel map similar to kernel thread. The system calls that are executed by the process (remote) are intercepted by remote site's link layer. If the system calls are site independent, it is executed by the 'remote' locally, else, the system call is forwarded to the 'deputy'. The

'deputy' then executes the call and returns the result back to remote site.

## **Condor**

Condor is a High Throughput Computing environment that can manage very large collections of distributively owned workstations. The environment is based on a novel layered architecture that enables it to provide a powerful and flexible suite of Resource Management services to sequential and parallel applications. The following are the features of Condor:

*a. Checkpoint and Migration:* Where programs can be linked with Condor libraries, users of Condor may be assured that their jobs will eventually complete, even in the ever changing environment that Condor utilizes. As a machine running a job submitted to Condor becomes unavailable, the job can be checkpointed. The job may continue after migrating to another machine. Condor's periodic checkpoint feature periodically checkpoints a job even in lieu of migration in order to safeguard the accumulated computation time on a job from being lost in the event of a system failure such as the machine being shutdown or a crash.

*b. Remote System Calls:* Despite running jobs on remote machines, the Condor standard universe execution mode preserves the local execution environment via remote system calls. Users do not have to worry about making data files available to remote workstations or even obtaining a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, no matter on which machine it really ends up executing on.

*c. Jobs can be ordered:* The ordering of job execution required by dependencies among jobs in a set is easily handled. The set of jobs is specified using a directed acyclic graph, where each job is a node in the graph. Jobs are submitted to Condor following the dependencies given by the graph.

*d. Condor Enables Grid Computing:* As grid computing becomes a reality, Condor is already there. The technique of glide in allows jobs submitted to Condor to be executed on grid machines in various locations worldwide. As the details of grid computing evolve, so does Condor's ability, starting with Globus-controlled resources.

*e. Sensitive to the Desires of Machine Owners:* The owner of a machine has complete priority over the use of the machine. An owner is generally happy to let others compute on the machine while it is idle, but wants it back promptly upon returning. The owner does not want to take special action to regain control. Condor handles this automatically.

*f. ClassAds:* The ClassAd mechanism in Condor provides an extremely flexible, expressive framework for matchmaking resource requests with resource offers. Users can easily request both job requirements and job desires. For example, a user can require that a job run on a machine with 64 Mbytes of RAM, but state a preference for 128 Mbytes, if available. A workstation owner can state a preference that the workstation runs jobs from a specified set of users. The owner can also require that there be no interactive workstation activity detectable at certain hours before Condor could start a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful expressions, resulting in Condor's adaptation to nearly any desired policy.

Condor has some limitations on jobs that it can transparently checkpoint and migrate which are the following

- a. Multi-process jobs are not allowed. This includes system calls such as *fork()*, *exec()*, and *system()*.
- b. Inter-process communication is not allowed. This includes pipes, semaphores, and shared memory.
- c. Network communication must be brief. A job *may* make network connections using system calls such as *socket()*, but a network connection left open for long periods will delay checkpointing and migration.
- d. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
- e. Alarms, timers, and sleeping are not allowed. This includes system calls such as *alarm()*, *getitimer()*, and *sleep()*.
- f. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
- g. Memory mapped files are not allowed. This includes system calls such as *mmap()* and *munmap()*.
- h. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.

- i. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.

The following are the reasons for selecting Mosix over Condor for our implementation

- a. Condor has too many limitations on the type of process it can migrate as we have seen above.
- b. Condor is not an open source project. Also it does not provide any API for migrating the processes. Mosix API and its source code are available free of cost through GPL for programmers to develop the existing product or to utilize them in upper layers. Also it has an option to restrict its resource sharing algorithms so that users can develop their own resource sharing/scheduling algorithms in order to gain a better control over load distribution.

### **Performance Co-Pilot (PCP)**

PCP is a framework and services to support system-level performance monitoring and performance management. Performance data may be collected and exported from multiple sources, most notably the hardware platform, the IRIX kernel, layered services and end-user applications. The diagram below shows architecture of PCP.

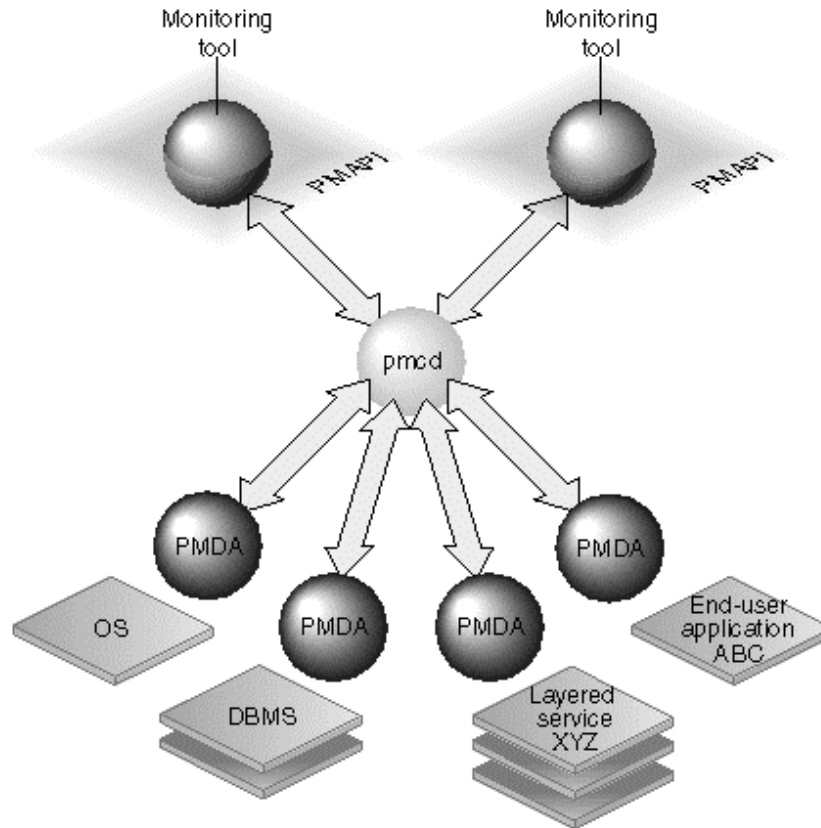


Figure 2: PCP architecture

PCP consists of several monitoring and collecting tools. The monitoring tools consume and process performance data using a public interface, the Performance Metrics Application Programming Interface (PMAPI). Below the PMAPI level is the Performance Metric Collector Daemon (PMCD) process, which acts in a coordinating role, accepting requests from clients, routing requests to one or more Performance Metrics Domain Agents (PMDA), aggregating responses from the PMDAs, and responding to the requesting client. Each performance metric domain (such as IRIX or some Database Management System or NETSTAT in our case)) has a well-defined name space for referring to the specific performance metrics it knows how to collect. Each PMDA

encapsulates domain-specific knowledge and methods about performance metrics that implement the uniform access protocols and functional semantics of the PCP. There is one PMDA for the operating system, another for process specific statistics, one each for common DBMS products, and so on. Connections between PMDAs and PMCD are managed by the PMDA functions. There can be multiple monitor clients and multiple PMDAs on the one host, but there may be only one PMCD process. PCP also allows extending the functionalities by writing agents to collect performance metrics from uncharted domains, or to program new analysis or visualization tools using the Performance Metrics Application Programming Interface (PMAPI).

## Design and Implementation

Figure below shows the various modules interact in final application (on every machine).

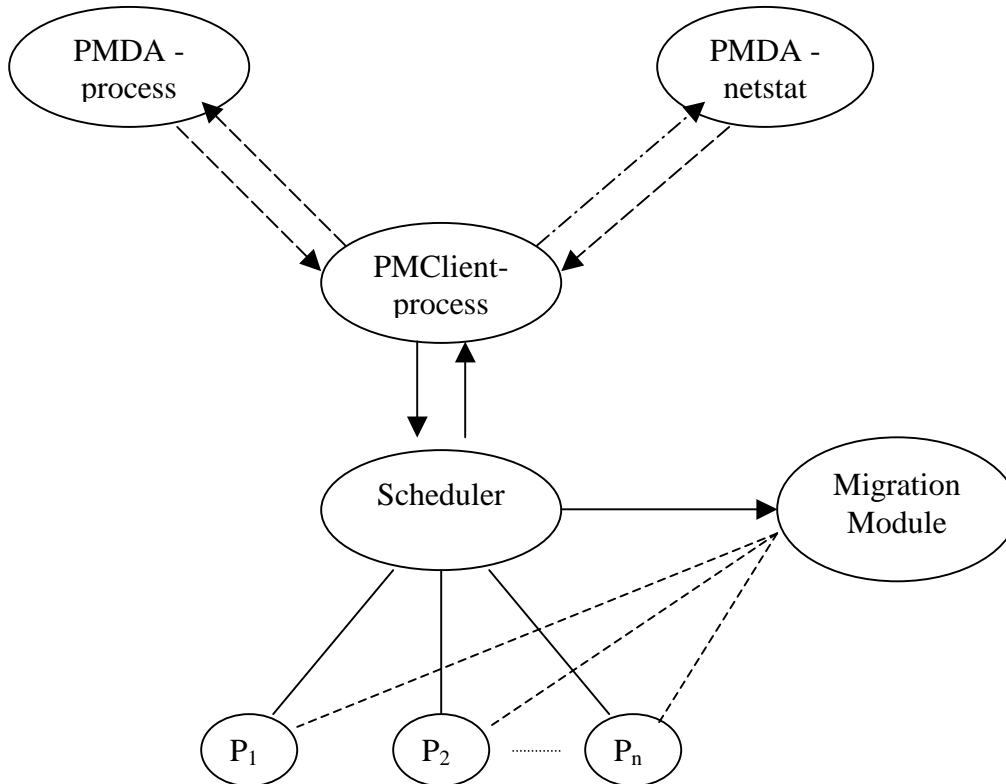


Figure 3: Interaction of Modules

The following are the modules in our implementation

a. *PMDA*: PCP provides PMDAs for collecting process metrics such as number of processes running, user time, system time, memory used and so on. We have used these PMDAs through the PMClient in order to get the process metrics. As the PCP does not have any PMDAs to give the network characteristics of a process, we have written a PMDA, using the common netstat utility to give metrics concerning network characteristics of each process such as intensity of communication (in bytes/sec), address of the machine on which the process is running on the other machine, source port number and destination port number (port number of the process with which its

communicating). The name of this PMDA is 'netstat' and user's can access its functionalities by using the name 'netstat'. E.g.: `pminfo -f netsat` gives the metrics of the communicating processes.

Precisely, the metrics of a process being considered presently are system time and netstat metrics. A CPU intensive process will have a high value for the ratio of system time to the total time taken. Whenever there is a load imbalance, we consider the process with maximum CPU intensity for migration. This step corresponds to the 'Load Evaluation Phase', mentioned earlier.

b. *PMClient*: It is responsible for talking with PMDA through PMAP Interface. It

fetches the metric values, which can be used for making the decision upon process migration.

c. *Scheduler*: It acquires all the performance metric values of all processes and then determines if any scheduling is required based upon the scheduling algorithm. This corresponds to the 'Task Selection Phase' (We do not consider 'Profitability Determination phase' as the cost of load balancing, such as the time/ load taken for running the scheduler and the time taken for actual migration of a process are very less as compared to the load of a process).

We have implemented two different scheduling techniques for balancing CPU loads. (All the algorithm implementations were done on a cluster of two machines).

Algorithm 1:

1: *fetch metrics for all process.*  
2: *fetch load on two machines*  
3: *check if loads are different (with difference greater than a threshold value).*  
    3a: *if load on current machine is less*  
        → *sleep for few seconds and jump to step 1.*  
    3b: *If load on current machine is much higher than load on other machine* →  
        3b1: *select the process, which is causing maximum load and also running on current machine.*  
            *If the process is migratable, move the process to other machine. Then sleep for few seconds and go step 1*  
        3b2: *If process already migrated, repeat step 3b1 for remaining processes.*

Algorithm 2 differs from Algorithm1 in the case of handling processes which have been already migrated. i.e. Algorithm1 waits for the process to return back by itself (

when the execution ends) whereas Algorithm2 gets the migrated process back if it sees that the load on the machine to which the process migrated has increased more than the current machine by a certain threshold.

Algorithm2:

1: *fetch metrics for all process.*  
2: *fetch load on two machines*  
3: *check if loads are different (with difference greater than a threshold value).*  
    3a: *If load on current machine is very less* →  
        3a1: *Check if any process has migrated to other machine. If so, get the process with highest load, back to current machine. Wait for few seconds and jump to start.*  
    3b: *If load on current machine is much higher than load on other machine* →  
        3b1: *select the process, which is causing maximum load and also running on current machine (by system time value).*  
            *If the process is migratable, move the process to other machine. Then sleep for few seconds and go step1.*  
        3b2: *If process already migrated, repeat step 3b1 for remaining processes.*

These algorithms correspond to the 'Task Selection and Migration Phase'. The 'Migration Module' handles the actual migration by using Mosix API. Both the algorithms give a better performance over process assignment (static) without any scheduling as they try to balance the load on both the machines. Algorithm2 is more efficient than algorithm1 as the later does not does not migrate the processes back home even when the load on other machine become high. The graphs, later in the paper show the improvement over the completion

time of processes scheduled with algorithm over the ones run without algorithm.

The advantages of the above scheduling algorithms over Mosix's resource sharing algorithm are that they give the control of load balancing to the user. User can specify the threshold of loads on each machine for the scheduler to make better decisions on load balancing. Above methodologies have certain shortcomings as for example 1) They are decentralized, in which every machine tries to run the scheduling algorithms and it might happen that all highly loaded machines try to migrate their processes onto lightly loaded machine. This causes the lightly loaded machine (say 'X') to become heavily loaded within a short span of time. Now, 'X' tries to migrate its own processes or all other machines, which migrated their processes, might take their processes back. This series of actions will lead to frequent increase and decrease of loads and would not help in any way for load balancing. Also, every machine might try to migrate the communicating process to the machine with

which it is communicating. This might lead to just swapping of processes with respect to machines and does not reduce the communication intensity. Later we propose a solution for all the above issues along with providing the flexibility of scheduling control to the user by means of a central scheduler.

### Results

The diagrams below shows the performance of the machines with sample test programs, which are either CPU, bound or I/O bound. The CPU bound processes were benefited by our scheduling algorithms which is evident from their faster execution as also seen in the graphs. Figure 4 & 5 show the performances of the algorithm2 and algorithm2 respectively. The results show that algorithm2 is more efficient than algorithm1 as expected .The I/O intensive processes are not benefited much by the schedulers as the system calls for files are always redirected towards the 'deputy' in the home node (see figure 6).

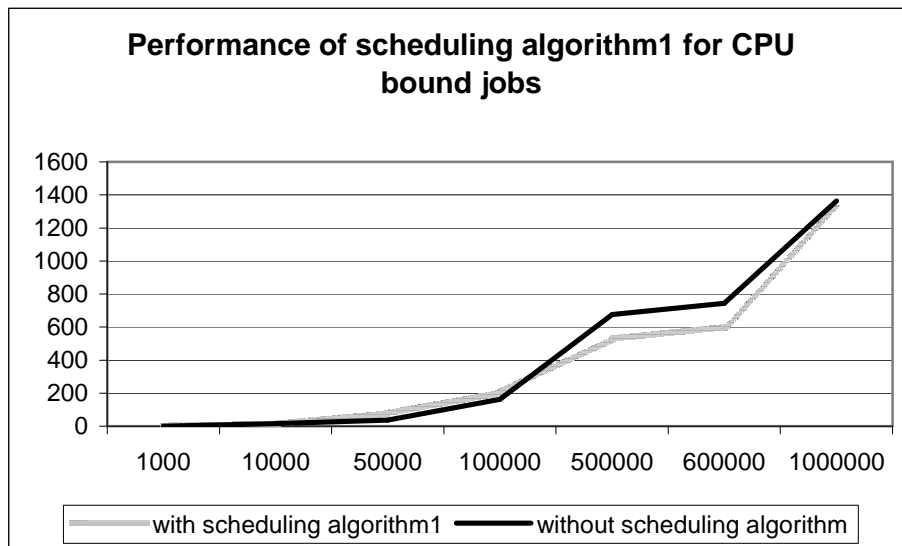


Figure 4: Graph of program size (X Axis) vs. execution time (Y Axis)

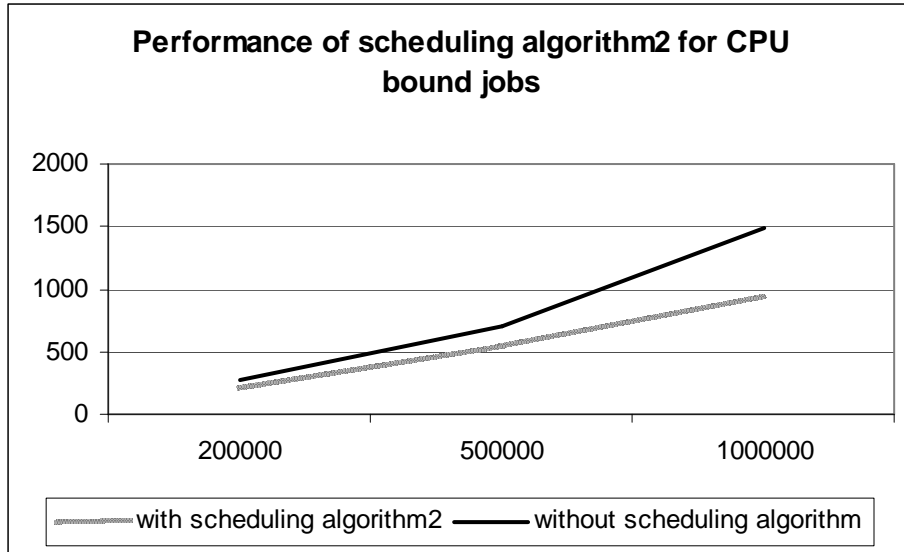


Figure 5: Graph of program size (X Axis) vs. execution time (Y Axis)

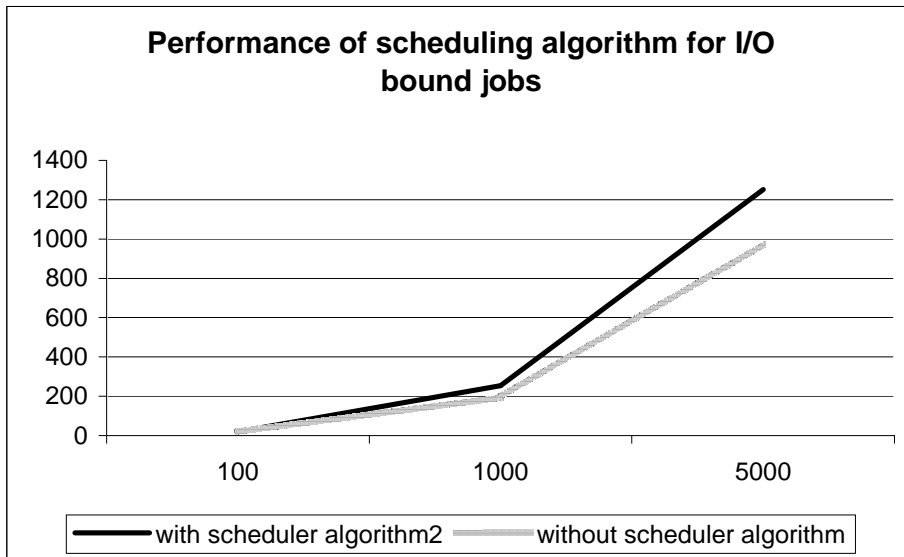


Figure 6: Graph of program size (X Axis) vs. execution time (Y Axis) without



b. For all jobs with considerable network intensity, the centralized scheduler tries to create a graph/table which determines the communicating pairs (or a set) of processes. The scheduler then tries to reduce the network intensity by having the communicating processes on the same machine. Also, the central scheduler tries to balance CPU load, which is caused by the processes, which are network intensive. The phases of getting the metrics, constructing the graph of communicating processes, getting the loads of various machines are separated from the actual scheduling mechanism. This will help in changing the scheduling mechanism based on user's requirements as and when needed. The centralized scheduler will then send a message to the individual machines (servers) to actually migrate the processes. However, there are certain issues to be considered in centralized scheduler implementation. The amount of information passed from the individual machines to the central scheduler might be enormous. As for example if a machine passes the following structure on every request for metrics, (values in braces indicates the size of each field)

```
struct info {
char pid[2];(2) // pid
float CPU_Load; (4) //the cpu load of this
process
int sport ;(2) //source port
int dport; (2) //destination port
char daddr[8]; (8) // destination IP address
float net_intensity; (4) // network intensity
}
```

The total size of the structure is 22 bytes. The values for sport, dport and net\_intensity will be zero non-communicating processes. If machine is loaded with many processes, then sending the above information for all processes frequently (say every 5 – 10 seconds), then it will be an overhead on the

network in addition to the actual communication of the processes themselves. We can reduce this overhead to a large extent by making the local schedulers more intelligent. All processes with only CPU load can be handled for migration, by the local schedulers themselves. The local scheduler will then only passes the information about the processes, which have some communication. The total amount of information passed would be very less as compared to earlier amount of information passed if the network intensive processes are a small subset of the total processes running in the system. This scheme will make the whole process efficient by dividing the scheduling overhead to different machines and also reduces the network load when the scheduling is done frequently. If the scheduling is done occasionally (say, once every 15 minutes) then making the central scheduler handle all the work (scheduling both CPU intensive and network intensive processes) will be the efficient way.

### Conclusions and Future Work

In all the work done so far, we have prepared the infrastructure for load balancing on Mosix Cluster. We have tested this with our (local) schedulers based on simple load balancing algorithms that we discussed in the earlier sections. As we have mentioned earlier, the local schedulers have their own disadvantages such as frequent variation of loads, frequent swapping of communicating processes. We plan to overcome these disadvantages by implementing the central scheduler. Also, extensive scripts will facilitate large scale testing with varying parameters for complex scheduling algorithms.

## References

- [1] <http://www.mosix.org/>
- [2] <http://oss.sgi.com/projects/pcp>
- [3] <http://www.cs.wisc.edu/condor/manual>
- [4] Watts, Jerrell and Taylor, Stephen, "A Practical Approach to Dynamic Load Balancing, " IEEE Transactions on Parallel and Distributed Systems, Vol 9, No 3, March 1998.
- [5] Amnon Barak, Oren La'adan Amnon Shiloh, "Scalable Cluster Computing with MOSIX for LINUX."
- [6] Amnon Barak, Avner Braveman, Ilia Gilderman and Oren Laden, Performance of PVM with the MOSIX Preemptive Process Migration Scheme."
- [7] Steve McClure, Richard Wheeler, "Mosix: How Linux Clusters Solve Real World Problems."