Simulation of batch scheduling using real production-ready software tools

Alejandro Lucero[**1]

Barcelona SuperComputing Center

Abstract. Batch scheduling for high performance cluster installations has two main goals: 1) to keep the whole machine working at full capacity at all times, and 2) to respect priorities avoiding lower priority jobs jeopardizing higher priority ones. Usually, batch schedulers allow different policies with several variables to be tuned by policy. Other features like special job requests, reservations or job preemption increase the complexity for achieving a fine-tuned algorithm. A local decision for a specific job can change the full scheduling for a high number of jobs and what can be thought as logical within a short term could make no sense for a long trace measured in weeks or months. Although it is possible to extract algorithms from batch scheduling software to make simulations of large job traces, this is not the ideal approach since scheduling is not an isolated part of this type of tools and replicating same environment requires an important effort plus a high maintenance cost. We present a method for obtaining a special mode of operation for a real production-ready scheduling software, SLURM, where we can simulate execution of real job traces to evaluate impact of scheduling policies and policy tuning.

1 Introduction

Simple Linux Utility for Resource Manager (SLURM) [1] has been used at Barcelona Supercomputing Center (BSC) [2] since 2005. Initially designed as a resource manager for jobs execution through large linux cluster installations, it has today extended features for job scheduling. This software is a key tool for a successful management of such a large machine as Marenostrum, a 2500 node and more than 10000 cores installed at BSC. SLURM first versions supported a very simple scheduler: a First Come First Served algorithm. Last versions improve scheduling with a main algorithm based on job priority allowing to define different quality of service queues with users getting access to specific ones. A second scheduling algorithm is present with SLURM as configurable: the backfilling algorithm. It can work together with the main scheduler, its goal being to use free resources by lower priority jobs as long as none higher priority job is delayed. SLURM offers other features as reservations, special job requests and job preemption.

It is worth to mention SLURM scheduling is fully dependent on how resources are managed. For a better understanding of this relationship, the scheduling should be thought of as a two step process:

[**] alejandro.lucero@bsc.es

1. A first step selects a job which requisities can be granted based on pure scheduling algorithm. At this point a job was selected and there are enough resources to execute it.
2. A second step selects best nodes for job execution trying to use resources in such a way that job requisities are accomplished and no more resources than really needed are wasted.

For example, jobs can ask for an amount of memory per node, so resource management needs to know which nodes own such amount of memory, and in case of other jobs running at the nodes already, it needs to know current memory available. Resource management is currently done with cpus/cores/threads and memory but it would be possible to use others like disk, network bandwith or memory bandwith.

Taken SLURM as the batch scheduler the simulation software will be based on, it would be possible to extract the SLURM code related to scheduling and work with it. However, as we have seen above, scheduling is linked to resource management so it would be needed to extract resource management code as well if we really want to work with a real scheduling software. Assuming this task is feasible without a high effort, there are other components we would need for using this code:

- job submission
- supporting special job requests
- reservations
- controlling nodes
- controlling job execution
- job preemption
- ...

For a valid simulation all of these possibilities should be contemplated therefore extracting just scheduler code from SLURM is not enough. It would be possible to build a simulation bench based on SLURM code but there are two main problems with this approach:

1. It would require an important effort implementing all the components needed
2. It would have a high maintenance cost for every new SLURM version.

We present a solution for job trace execution using SLURM as the simulation tool with minor SLURM code changes. Our design goal is to hide simulation mode from SLURM developers and to have everything SLURM supports available in simulation mode. We think the effort should be done once, then new SLURM features in future new versions will be ready for simulation purposes with minor effort making easier simulation maintenance.

The rest of the paper covers related work, design, implementation and preliminary results.

2 Related Work

Process scheduling is an interesting part of operating systems and a focus of research papers for decades. It is well known there is no perfect scheduler covering whatever the workload and in specific systems like those with real time requirements scheduling is tuned and configured with detail. Batch scheduling for supercomputers clusters shares same complexity and problems as process or I/O scheduling presented at operating systems. Previous work trying to test scheduling algorithms and scheduling configurations has been done as in [3] where process scheduling simulation is done at user space copying scheduler code from Linux kernel. That work suffers from one of the problems listed above: maintenance is needed to keep same scheduler code at user space.

Another work more related to batch scheduling for supercomputers is presented at [4] where it is well-described the importance of having a way of testing scheduling policies. But this work also faces problems since simulator code replaces real scheduler and wrappers for resource management software are specifically built, so simulation is done assuming such an enviroment being realistic enough. Other works [5] use CSIM [6] modelling batch scheduling algorithms with discrete events. Although this is broadly used for pure algorithm research it requires an important effort and it is always difficult to reproduce the same environment what can lead to understimate other parts of real scheduler systems like resource management. Research using a real batch scheduler system has been done [7] but without a time domain for simulation: assumption is made for dividing per 1000 time events obtained from a job trace. In this case batch scheduling software runs as usual but every event take 1/1000 of its real time. This approach has some drawbacks as for example what happens with jobs taking less than 1000 seconds to execute. Clearly it is not possible to simulate a real trace and it is important to notice small changes of job scheduling can have a huge impact on global scheduling results.

It is worth to mention Moab [8] cluster software which is currently used at BSC along with SLURM. Moab takes care of scheduling with help from SLURM to know which are the nodes and running jobs state. Moab has had support for a simulator mode since initial versions although it has not always worked as marketing said. In fact, this work was started to overcome the problem trying to use this Moab feature in the past. Being Moab a commercial software there is not documentation about how this simulation mode was implemented. However, we got some information from Moab engineers commenting how high was the cost of maintaining this feature with new versions.

We have not found any work using the preloading technique with dynamic libraries for creating the simulation time domain and neither using this approach with such a multithread and distributed program as SLURM.

3 Simulator Design

SLURM is a distributed multithreading software with two main components: a slurm controller (slurmctl) and a slurm daemon (slurmd). There is just one slurmctl by cluster installation and as many slurmd as computing nodes. The slurmctl tasks are resource management, job scheduling and job execution, although in some installations SLURM is used along with an external scheduler as MOAB. Once a job is scheduled for execution the slurmctl communicates with those slurmd belonging to the nodes selected for the job. Behind the scene there is a lot of processing for allowing job execution on nodes, like preserving user environment when job was submitted. SLURM design had scalability as a main goal so it supports thousands of jobs and thousands of nodes/cores per cluster. When slurmctl needs to send a message to the slurmds it uses agents (communication threads) for supporting a heavy peak of use as, for example, signaling a multinode job for terminating, getting nodes status or launching several jobs at the same time. Besides those threads, created for a specific purpose and with a short life, there are other main threads as the main controller thread, power management thread, backup thread, remote procedure call (RPC) thread, and backfilling thread. In the case of slurmd there is just one main thread waiting for messages from the slurmctl, an equivalent to the slurmctl RPC thread. On job termination slurmd sends a message to the slurmctl with the jobid and job execution result code.

Figure 1 shows SLURM system components. Along with slurmctl and slurmd there are several programs for interacting with slurmctl: sbatch for submitting jobs, squeue and sinfo for getting infotmation about jobs and cluster state, scancel for cancelling jobs, and some other programs not shown in the picture.

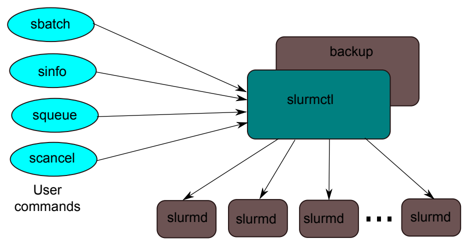

Fig. 1. Slurm Architecture

From a simulation point of view the task done by slurmd is simpler since it is not necessary job execution, just to know how long will it take, information we get from the job trace. A simple modification for simulation mode in the slurmd will support this easily just sending a message when the job finishes knowing how long ago the job was launched. The key is how to know the current time since in simulation mode we want to execute a trace of days or months in just hours or minutes. Another issue is to support simulation for thousand of nodes: it is not necessary to have one slurmd by node but just one super-slurmd taking control of jobs launched and job duration. No change is needed for this super-slurmd since SLURM has an option called FRONTEND mode which allows this behaviour although for other purposes. Although slurmd by design can be in a different machine than slurmctl, for simulation this is not needed.

For slurmctl no changes are needed. However it is a good idea to avoid some threads execution related to power management, backup, slurm global state save and signal handling, which make no sense under simulation mode. Also, during a normal execution, SLURM takes control of all computational nodes running a slurmd. Periodically the slurm controller pings the nodes and ask for current jobs being executed. This way SLURM can react to nodes or jobs problems. Under simulation we work within a controlled environment so no problem is expected, therefore that funcionality is not needed.

As in the case of slurmd, slurmctl execution is based on time elapsed: both main scheduler algorithm and backfilling are infintite loops executed within a speficic period; a job priority is calculated based on fixed parameters dependent on user/group/qos, but usually priority increases proportionally to job's time queued; jobs have a wall clock limit fixing maximum execution time allowed for the job; reservations are made for future job execution.

In simulation mode we want to speed up time therefore execution of long traces in a short time can be done for studying best scheduling policy to apply. Our design is based on the LD_PRELOAD [9] functionality available with shared libraries in UNIX systems. Using this feature it is possible to capture specific function calls exported by shared libraries and to execute code whatever the necessity. As SLURM is based on shared libraries we can make use of LD_PRELOAD to control simulation time, so calls to time() function are captured and current simulated time is returned instead of real time. Along with time() there are other functions time related used by SLURM which are wrappered: gettimeofday() and sleep(). Although there are other functions time related, like usleep() and select(), SLURM uses them for communications what is not going to affect simulation. Finally, as SLURM design is based on threads, and calls to time functions are made indistincly by all of them, some POSIX threads related functions are wrappered as well.

Figure 2, shows how simulator design works: a specific program external to SLURM is created, sim_mgr, along with a shared library, sim_lib, containing wrappers for time and thread functions. During initialization phase, sim_mgr waits for programs to control: slurmctl and slurmd. Once those programs are registered (executing init function related to sim_lib, 0a and 0b in the figure), every time pthread_create is called, wrapper code at sim_lib registers the new thread(1a and 4b1).

In the figure, lines with arrows represent simulation related actions, with first number at the identifier representing the simulation cycle when that action happens. The letter at the identifier is just to differenciate the two threads shown,

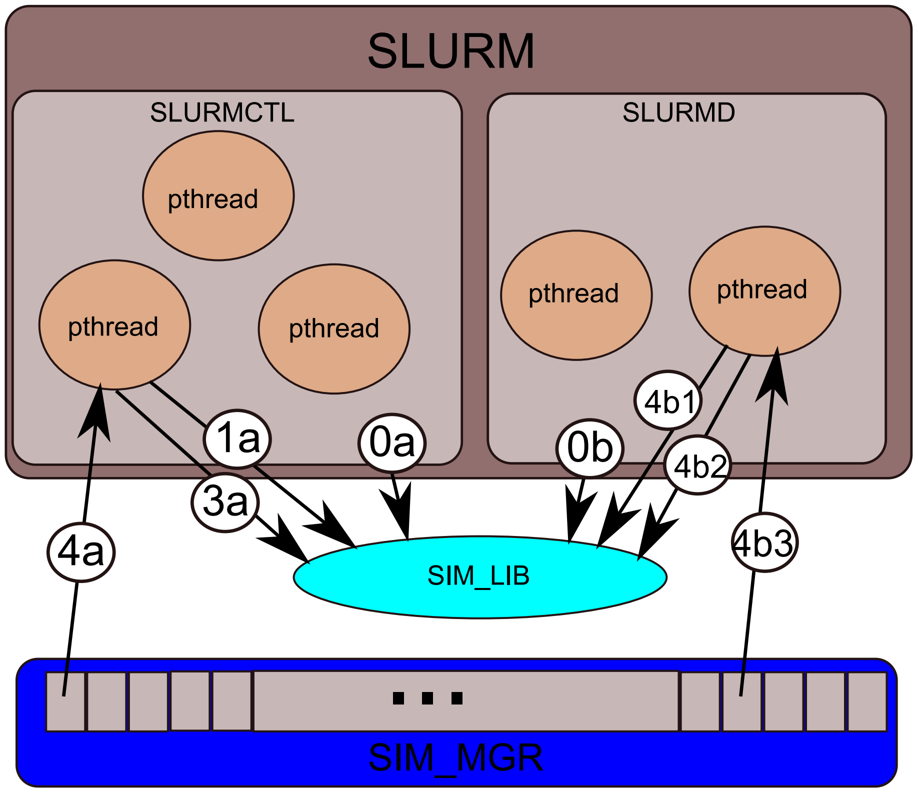

Fig. 2. Simulator Design

and the second number indicates the sequence of actions in the same cycle.

So the slurmctl thread was created at simulation cycle 1 and the slurmd thread was created at simulation cycle 4. Inside pthread wrappers there is a call to the real pthread functions, what is not true for the time wrappers which fully replace the real time functions. Registration links a thread to a POSIX semaphore used for simulation synchronization and it also links the thread to a structure containing sleep value for that thread and other fields used by the sim_mgr.

A simulation clock cycle is equivalent to one real time second and in each cycle sim_mgr goes through an internal thread array working with registered threads. There are two classes of threads from sim_mgr point of view:

- those with do sleep calls running through several simulation cycles (and someones through all the simulation). Sim_mgr detects a sleepy thread and keeps a counter for waking up the thread when necessary.
- those which are created, do something and exit in the same simulation cycle.

Back to figure 2, thread A does a sleep call at simulation cycle 3 (3a) and sim_mgr wakes up the thread at simulation cycle 4 (4a). So we know it was a sleep call for just one second. At that point sim_mgr waits till that thread does another sleep call or maybe a call to pthread_exit. The main threads in slurmctl have this behavior: infinite loops with a frequency defined by thread. The other type of threads is shown with the B thread in the figure. This thread is created at simulation cycle 4 (4b1) and at the same cycle it calls to pthread_exit (4b2). When sim_mgr needs to act with that thread there are two situations: the thread did not finish yet or it did finish. In the first case sim_mgr waits till the thread calls to pthread_exit. In the second one sim_mgr just sets the array thread slot free for new threads.

4 Simulator Implementation

A main goal for the simulator is to avoid SLURM modifications or, if necessary, to make those changes trasparent for SLURM developers. The ideal situation would be simulator feature going along future SLURM versions smothly just with minor maintenance. If this turns out to be an impossible task a discussion should be opened at SLURM mailing list exposing the problem and studying the cost. Just if the simulator becomes a key piece for SLURM, could it make sense major changes of core SLURM functionality.

Simulator design implies to implement two pieces of code outside SLURM code tree: the simulation manager and the simulation shared library. The simulation manager, sim_mgr, keeps control of threads and simulation time, waiting for threads to terminate or waking up threads when sleep timeouts requested by those threads expire. The simulation lib captures time related calls thanks to LD_PRELOAD functionality and synchronize with sim_mgr for sleep calls or getting simulation time. This design is quite simple for a single process or thread but it gets more complex for the multithread and distributed SLURM software. Sim_mgr needs a way to identify threads and to control each one independently, therefore sim_lib implements wrappers for some pthreads functions like pthread_create, pthread_exit and pthread_join, which give information to sim_mgr when they are called. A first implementation with sockets inside those pthread wrappers was made but it made sim_mgr less intuitive to understand and debug. A second implementation was done using POSIX shared memory containing global variables and per thread variables. One of the structures inside the shared memory is an array of POSIX semaphores used for synchronizarion between sim_mgr and slurm threads. A limitation of this design using POSIX mechanisms is slurmd needs to be at the same machine than slurmctl and sim_mgr, although we have commented above that this is not a problem for simulation purposes.

When sim_mgr is launched shared memory and semaphores initialization is done. After that, a new thread is registered when pthread_create wrapper is called. Both, slurmctl and slurmd can create threads so it is necessary registration to be atomic for getting a simulation identifier by thread. Linked to this identifier are two main fields: a sleep counter and a semaphore index. For tracking which thread has which identifier a table is created using pthread_t value returned by real pthread_create function. A thread calling pthread_exit, pthread_join or sleep functions will use that table and value of pthread_self() to finding out which semaphore to use for synchronization with the sim_mgr.

When a thread calls time function, sim_lib wrapper for time is executed. Inside this wrapper there is just a read of a global simulation address using the shared memory space containing the current simulation time. This is the same for gettimeofday with a difference related to microseconds value returned by this function. Initially we do not need to have a simulation with such a fine grain precission so microseconds is a fake value wich increments every time gettimeofday is called inside same simulation cycle.

When a thread calls sleep function the sim_lib wrapper is executed as follows:

1. wrapper code gets which thread identifier that thread has inside simulation. For this a search through a table linking pthread_t values coming from pthread_create and simulation thread identifiers is made. A thread can know which is its pthread_t value with a simple call to pthread_self which is part of standard pthread library. A potential problem with this approach is the possibility of getting two threads with same pthread_t value. This is not possible for a single Linux process but SLURM is a distributed program with at least two processes, a slurmctl and a slurmd, so it could be possible to have two threads with same pthread_t value. Linux uses the stack address of the created pthread for that pthread_t value so it is quite seldom for two different programs to get same pthread_t values. Nonetheless a table using pthread_t along with process identifier (pid) would be safer.
2. Using the thread identifier as an array index to thread info structure, the thread sleep field is updated to the value passed as a parameter to sleep function.
3. Next step is synchonization with sim_mgr. There is a POSIX semaphore by pthread by design but this is not enough for getting synchronization right so there are two semaphores by thread. First semaphore is for thread waiting to be woken up by the sim_mgr meanwhile the other thread is for sim_mgr to wait thread coming back to sleep or exiting.
4. If the thread calls sleep again the loop goes back to 1). If pthread_exit is called by the thread a sem_post(sem_back[thread_id]) is done inside pthread_exit wrapper.

We have seen what wrappers do except for pthread_join. This wrapper is quite simple, adding a new valid thread state, joining, for simulation. This is necessary for avoiding interlocks when SLURM agents are used.

`Simulation manager`

Wrappers implemented inside sim_lib library are useful for connecting slurm threads with the simulation manager, sim_mgr. The global picture of how simulation works can be clarified with a detailed view of what sim_mgr does in a simulation cycle:

1. for each registered thread being neither a never ending thread nor a new thread:
   (a) it waits till that thread sleeps, exits or joins
   (b) if thread is sleeping and sleep counter is equal to 0
      i. it does a sem_post(sem[thread_id])
      ii. it does a sem_wait(sem_back[thread_id])

   The term never ending thread needs some explanation. What sim_mgr does is to allow just one slurm thread to execute at thread same time but allowing each thread to execute within a simulation cycle. However, some specific threads need less control for avoiding an interlock between threads. This is the case of threads getting remote requests at both slurmctl and slurmd. Usually those threads call to pthread_create when a new message arrive and the new thread will be under sim_mgr control as usual.
2. Using the trace information, if there are new jobs to be submitted at this simulation cycle, sbatch, usual slurm program for job submission, is executed with the right parameters for the job.
3. For each new thread sim_mgr waits till that thread sleeps, exits or joins. As slurm can have some specific situations with a burst of new threads, sim_mgr needs a way of controlling how threads are created avoiding to have more than 64 threads at the same time, since this is the current limit sim_mgr supports. So this last part of the simulation cycle manages thread creation delaying some pthread_create calls and identifying what we have named proto-threads, this is almost created threads. This is loop managing this possible situation:
   (a) For each new thread, if the thread is still alive, wait till it does a call to sleep, exit or join. In other case, it does mean this thread did its task quickly and called pthread_exit.
   (b) When all new threads have been managed, all threads which did a call to pthread_exit during this simulation cycle can release their slot so new threads can be created. If there were not any free slot it could be possible a pthread_create call is waiting for getting a simulation thread identifier. A global counter shows how many proto-threads are waiting for free slots. If this counter is greater than zero then sim_mgr leaves some time for those pthread_create calls to finish then goes back to previous step looking again for new threads. Otherwise simulation cycle goes on.
4. At this point any thread alive or any thread created through this simulation cycle has had a chance to execute. So if the thread is still alive it is waiting on a semaphore. Sim_mrg can now decrement sleep value for all of those threads.
5. All threads slots belonging to threads which called to pthread_exit during this cycle are released.
6. Last step is to increment the simulation time is one second.
7. Go back to 1

5 Slurm code changes

Our main design's goal was to use SLURM program for simulation without major changes. In the design section were described which changes are needed for the slurmd and slurmctl in order to support the simulation mode. Once implementation has been done, changes can be shown in detail:

- Our design implies threads to explicitly call pthread_exit. Usually this is not needed and the operating system/pthread library know what to do when a thread function is over. Therefore we have added pthread_exit calls to each thread code routine what is not an important change for SLURM code.

- Under simulation we build up a controlled environment: we know when threads will be submitted, when a node will have problems (if simulation supports this feature) and how long a job will run. There will not be unexpected job or node failures, therefore we can avoid code execution related to monitor jobs and nodes state when simulation mode is on.

- As jobs are not executed it is not necessary to send packets to each node for job cancelling, job signaling, job modification, etc. Removing this part of SLURM simplifies simulator work significantly since slurmctl uses agents for these communications. In clusters with thousands of nodes could it be possible jobs with thousand of nodes so sending messages to each node is not negligible at all. SLURM design was made for supporting those communication bursts and agents take care of sending messages from slurmctl to slurmds. The problem with agents is they are based on threads strongly: agents are threads themselves, there are pools of available agents with a static maximum limit and watchdog threads are specifically created to monitor agent threads. Although this is working fine for SLURM getting a high scalable software, such a mesh of threads is too much complex for our simulation design since interlocks between threads appear easily.

- As we said before, slurmd is simpler under simulation since no job needs to be really executed. By other hand, an extra task needs to be done: to know how long a job will run and to send a message of job completion when simulation time reaches that point. A new thread is created, a simulator helper, which each second checks for running jobs to be completed taken into account current time and job duration, and sends a message to slurmctl if necessary.

- Finally, slurmctl works as normal SLURM execution. Just some changes for sending messages without using agents has been done which makes sim_mgr simpler and facilitates syncronization for obtaining determinsm through simulations using the same job trace.

6 Preliminary Results

A first simulator implementation is working and executing traces with jobs from different users and queues. A regression testsbench has been build up using synthetic traces, useful for testing exceptional situations like a burst of jobs starting or completing at the same cycle. Several simulations using same synthetic trace are executed to validate design expecting same scheduling results which implies we achieve determinsm under simulation.

After some simulator executions we discovered a problem hard to overcome since it is linked to normal SLURM behavior. As we have seen, sim_mgr controls simulation execution leaving threads to execute but in sequential order, one by one. This is just the opposite threads were designed for, but it allows to execute traces with determinism. Leaving threads without control would mean small differences like a job starting one second later. Althouhg this looks harmless it could mean other jobs being scheduled in a different way with such a change being spread through the whole simulation, ending with a totally different trace execution. So sim_mgr leaves a thread to execute and waits till that thread calls sleep function again. There is a slurmctl thread for backfilling which is executed two time per minute by default, and the problem is the task done by this backfilling thread can easily take more than a second: the more jobs queued the more time will it take. It is also related to amount of resources, then doing simulations with a 1000 4-cpus nodes cluster and with 1000 jobs, with jobs duration from 30 seconds to an hour, it is normal for the backfilling thread to take up to 5 seconds to complete. Therefore, what is the goal behind simulation, to speed up job execution, is compromised due to this natural SLURM behaviour.We hope SLURM simulator can help to overcome this problem for real SLURM, then simulator itself can take benefit for executing long traces faster.

Next table shows results using a Intel Core Duo at 2.53Mhz and synthetic traces of 3000 jobs executed on a 1000 4-cpus node cluster:

Trace number	Executions	Sim Time	Exec Time	Speedup
1	5	172800	5564	31.05
2	5	171900	5487	31.32
3	5	176360	5324	32.23
4	5	173600	5422	32.01
5	5	182360	5821	31.32

Time results per trace number are mean of 5 executions. Simulation of four days of jobs execution takes around one hour and a half. Extrapolating these results, for a one month trace simulation will last for a bit more than 10 hours.

7 Conslusions and Future Work

A solution is presented for using a real batch scheduling software as SLURM for job trace simulation. Although SLURM design is strongly based on threads it is possible to keep the original SLURM functionality with minor changes. For this some calls done to standard shared libraries are captured and specific code for simulation purposes executed.

Implementation complexity has been left outside SLURM code so main SLURM developers will not to be aware of this simulation mode for future SLURM enhancements and simulator maintenance will be easier.

Next step will be to present this work to SLURM core developers and users and if it is welcomed to integrate the code with SLURM code project.

References

1. M. Jette, M. Grondona, *SLURM: Simple Linux Utility for Resource Management*, Proceedings of ClusterWorld Conference and Expo, San Jose, California, June 2003
2. Barcelona SuperComputing Center, www.bsc.es
3. LinSched: The Linux Scheduler Simulator, John M. Calandrino, Dan P. Baumberger, Tong Li, Jessica C. Young, and Scott Hahn. http://www.cs.unc.edu/ jmc/linsched/
4. Impact of Reservations on Production job Scheduling Martin W. Margo , Kenneth Yoshimoto , Patricia Kovatch , Phil Andrews. Proceedings of the 13th international conference on Job scheduling strategies for parallel processing 2007
5. A measurement-based simulation study of processor co-allocation in multicluster systems. S. Banen, A. I. D. Bucur, and D. H. J. Epema In Job Scheduling Strategies for Parallel Processing,2003
6. http://www.atl.lmco.com/projects/csim/
7. Parallel Job Scheduling Under Dynamic Workloads (2003). Eitan Frachtenberg , Dror G. Feitelson , Juan Fernandez , Fabrizio Petrini. In Job Scheduling Strategies for Parallel Processing, 2003
8. www.clusterresources.com
9. ld-linux.so(8), Linux Programmer's Manual Administration and Privileged Commands