

Development of usage scripts for Virgo users

A Project Report

submitted by

M B PRASANNA KUMAR, EE14B034

*in partial fulfilment of requirements
for the award of the degree of*

BACHELOR OF TECHNOLOGY



**DEPARTMENT OF Electrical Engineering
INDIAN INSTITUTE OF TECHNOLOGY MADRAS**

MAY 2018

THESIS CERTIFICATE

This is to certify that the thesis titled **Development of usage scripts for Virgo users**, submitted by **M B Prasanna Kumar, EE14B034**, to the Indian Institute of Technology Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Harishankar Ramachandran

Project Guide

Professor

Dept. of Electrical Engineering

IIT Madras, 600036

Place: Chennai

Date: 25 May 2018

ACKNOWLEDGEMENTS

This work would not have been possible without the guidance and the help of several people. I take this opportunity to extend my sincere gratitude to all those who made this thesis possible. First, I would like to thank all my teachers who bestowed me with good academic knowledge. I am indebted to my advisor Prof. Harishankar Ramchadran whose expertise, generous guidance and support made it possible for me to work on a topic that was of great interest to me. I would like to thank my family for giving support and guidance all through my life. I would also like to thank all my friends and well-wishers for helping me in difficult times and being a good source of support and guidance.

ABSTRACT

KEYWORDS: IBM loadleveler, Slurm, Commands, Workload manager, Daemons, Architecture

Workload management(WLM) is a process for determining the proper jobs distribution in order to provide optimal performance for applications and users. WLM also manages the use of system resources, such as processors and storage. The workload management softwares under discussion are IBM's Loadleveler(1) and Slurm(2). IBM's LoadLeveler schedules jobs, and provides functions for building, submitting, and processing jobs quickly and efficiently in a dynamic environment. SLURM is Linux Cluster's locally developed C-language Simple Linux Utility for Resource Management. Slurm is designed to be a replacement for IBM's LoadLeveler. This report gives a brief introduction to the fundamentals of both architectures and analyses their similarities and differences. It explains the work done in the attempt to **Develop usage scripts for Virgo** (IITM Super computer,uses Loadleveler for workload managing) users which will be available to users who logs in with their userid and password.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
1 Loadleveler	1
1.1 Introduction	1
1.2 Job Scheduling	1
1.3 Loadleveler Daemons	2
1.4 Job Cycle	3
2 SLURM	4
2.1 Introduction	4
2.2 Slurm Features	5
3 Comparision of architectures	7
3.1 Analysis of Commands	7
3.2 Usage of environment variables	8
3.3 Performance Comparison	9
4 Development of usage scripts	10
4.1 Intial work done	10
4.2 Finding average job time	11
4.3 Implementation in SLURM	16
5 Conclusion	17

CHAPTER 1

Loadleveler

1.1 Introduction

LoadLeveler is a job management system that allows users to run more jobs in less time by matching the jobs' processing needs with the available resources. LoadLeveler schedules jobs, and provides functions for building, submitting, and processing jobs quickly and efficiently in a dynamic environment.

LoadLeveler has three types of interfaces that enable users to create and submit jobs and allow system administrators to configure the system and control running jobs. They are control files (configuration files, administration files, job command files), command line interface and application program interface (API). The commands and APIs permit different levels of access to administrators and users.

Each machine in the LoadLeveler cluster performs one or more roles in scheduling jobs. Depending on the roles performed in scheduling jobs by each machine in the LoadLeveler cluster, a machine can be one of the following type: Job Manager Machine, Central Manager Machine, Executing Machine, Resource Manager Machine, Region Manager Machine and Submitting Machine. The Resource Manager is responsible for managing all machine and job resources, and the scheduler is responsible for scheduling jobs on the resources provided by the resource manager.

1.2 Job Scheduling

After a job is submitted, LoadLeveler examines the job command file to determine which machine, or group of machines, is best suited to provide the resources required, then dispatches the job to the appropriate machines. This process is aided by Job Queues. A job queue is a list of jobs that are waiting to be processed. When a user submits a job to LoadLeveler, the job is entered into an internal database, which resides on one of the machines in the LoadLeveler cluster, until it is ready to be dispatched to run on another machine.

A job can be dispatched to either one machine, or in the case of parallel jobs, to multiple machines. LoadLeveler examines the requirements and characteristics of the job and the availability of machines, and then determines the best time for the job to be dispatched which need not necessarily be first come first serve basis.

LoadLeveler also uses Job Classes to schedule jobs to run on machines. A Job Class is a classification to which a job can belong. The system administrator can define these job classes and select the users that are authorized to submit jobs of these classes. LoadLeveler also examines a job's priority to determine when to schedule the job on a machine. A priority of a job is used to determine its position among a list of all jobs waiting to be dispatched.

1.3 Loadleveler Daemons

The LoadLeveler daemons are programs that run continuously and control the processes that move jobs through the LoadLeveler cluster. A **Master daemon** (LoadL-master) runs on all machines in the LoadLeveler cluster and manages other daemons. The **Schedd daemon** (LoadL-schedd) receives jobs from the llsubmit command and manages them on machines selected by the negotiator daemon. The **Startd daemon** (LoadL-startd) monitors job and machine resources on local machines and forwards information to the negotiator daemon.

The **Negotiator daemon** (LoadL-negotiator) runs on the central manager machine. It monitors the status of each job and machine in the cluster and responds to queries from llstatus and llq commands. The **Keyboard daemon** (LoadL-kbdd) monitors keyboard and mouse activity. The **Gsmonitor daemon** (LoadL-GSmonitor) monitors for down machines based on the heartbeat responses of the MACHINE-UPDATE-INTERVAL time period.

1.4 Job Cycle

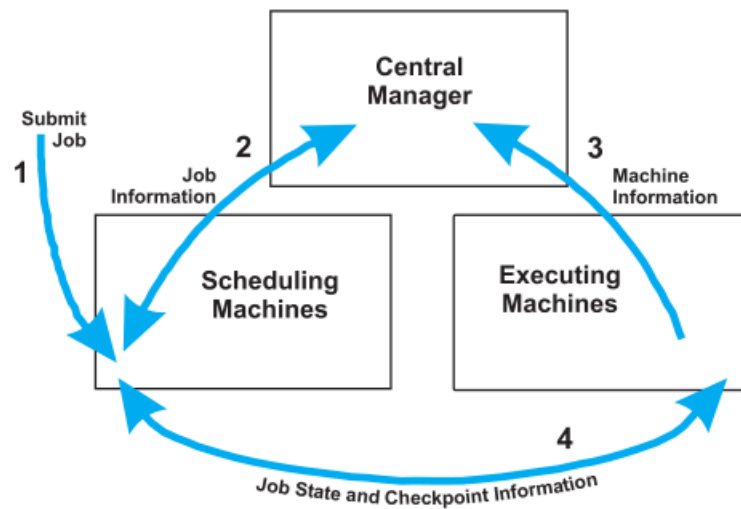


Figure 1.1: Job Cycle (1)

The job is submitted to the scheduling machine on the loadleveler. The Schedd daemon, on the scheduling machine, stores all of the relevant job information on local disk. Later the Schedd daemon sends job description information to the Negotiator daemon. At this point, the submitted job is in the Idle state.

The Negotiator daemon checks to determine if a machine exists that is capable of running the job. Once a machine is found, the Negotiator daemon authorizes the Schedd daemon to begin taking steps to run the job. This authorization is called a permit to run. At this point, the job is considered Pending or Starting.

Later, the Schedd daemon contacts the Startd daemon on the executing machine and requests it to start the job. The executing machine can either be a local machine (the machine from which the job was submitted) or a remote machine (another machine in the cluster). Then the Startd daemon on the executing machine spawns a starter process for the job. The Schedd daemon sends the starter process the job information and the executable. The Schedd daemon notifies the negotiator daemon that the job has been started and the negotiator daemon marks the job as running.

When the job completes, the starter process notifies the Startd daemon, which in turn notifies the Schedd daemon. The Schedd daemon examines the information it has received, and forwards it to the Negotiator daemon. At this point, the job is in Completed or Complete Pending state.

CHAPTER 2

SLURM

2.1 Introduction

Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Its features suit it to large-scale, high-performance computing environments, and its design avoids known weaknesses (such as inflexibility or fault intolerance) in available commercial resource management products for supercomputers. It requires no kernel modifications for its operation and is relatively self-contained. Slurm is the workload manager on about 60 percent of the TOP500 supercomputers with six of the top ten systems including the number 1 system, Sunway TaihuLight with 10,649,600 computing cores.

As a cluster workload manager, Slurm has three key functions. **Allocate nodes** - Give users access to computer nodes for some specified time range so their job(s) can run. **Control job execution** - Provide the underlying mechanisms to start, run, cancel, and monitor the state of parallel (or serial) jobs on the nodes allocated. **Manage contention** - Reconcile competing requests for limited resources, usually by managing a queue of pending jobs.

At LC, an adequate cluster resource manager needs to meet some other general requirements like scalability, portability, fault tolerant, open source and modular. No commercial (or existing open source) resource manager meets all of these needs. So since 2001 Livermore Computing, in collaboration with Linux NetworX and Brigham Young University, has developed and refined the "Simple Linux Utility for Resource Management" (SLURM).

SLURM was originally used as a resource manager for Linux (specifically for CHAOS) systems. But starting in 2006, LC began gradually replacing IBM's native LoadLeveler with SLURM on its AIX(Advanced Interactive eXecutive) systems as well. The AIX-SLURM combination behaves slightly differently than the CHAOS-SLURM combination.

2.2 Slurm Features

SLURM consists of two kinds of daemon and five command-line user utilities, whose relationships appear in this simplified architecture diagram:

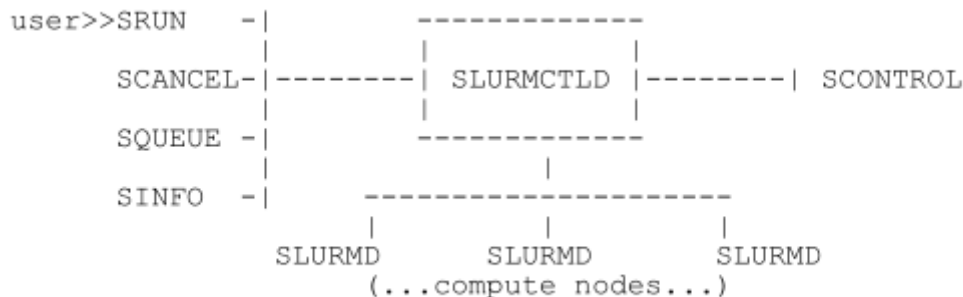


Figure 2.1: SLURM Architecture (4)

SLURM's central control daemon is called SLURMCTLD. SLURMCTLD runs on a single management node, reads the SLURM configuration file, and maintains state information on nodes, partitions, jobs and job steps. The SLURMCTLD daemon in turn consists of three software subsystems namely Node Manager, Partition Manager and Job Manager, each with a specific role.

Node Manager monitors the state and configuration of each node in the cluster. It receives state-change messages from each compute node's SLURMD daemon. Partition Manager groups nodes into disjoint sets (partitions) and assigns job limits and access controls to each partition. It also allocates nodes to jobs based on job and partition properties. Job Manager accepts job requests, places them in a priority-ordered queue, and reviews that queue periodically or when any state change might allow a new job to start.

The SLURMD daemon runs on every compute node of every cluster that SLURM manages and it performs the lowest level work of resource management. SLURMD carries out five key tasks and has five corresponding subsystems namely Machine Status, Job Status, Remote Execution, Stream Copy Service and Job Control.

The command line utilities offer users access to remote execution and job control and permit administrators to dynamically change the system configuration. These commands use SLURM APIs that are directly available for more sophisticated applications.

scancel cancels a running or a pending job or job step, subject to authentication and authorization. This command can also be used to send an arbitrary signal to all processes on all nodes

associated with a job or job step. **scontrol** performs privileged administrative commands such as bringing down a node or partition in preparation for maintenance. **sinfo** displays a summary of status information on SLURM-managed partitions and nodes.

squeue displays the queue of running and waiting jobs (or "job steps"), including the JobId and the nodes assigned to each running job. A wide assortment of filtering, sorting, and output format options are available for both **squeue** and **sinfo** commands. **srun** is used for allocating resources, submitting jobs to the SLURM queue, and initiating parallel tasks (job steps). SLURM associates every set of parallel tasks ("job steps") with the SRUN instance that initiated that set, and SRUN gives you elaborate control over node choice and I/O redirection for your parallel job.

SLURM achieves portability (hardware independence) by using a general plugin mechanism with about 100 optional plugins. SLURM's configuration file tells it which plugin modules to accept. A SLURM plugin is a dynamically linked code object that the SLURM libraries load explicitly at run time. Each plugin provides a customized implementation of a well-defined API connected to some specific tasks.

By means of this plugin approach, SLURM can easily change its interconnect support, security techniques, metabatch scheduler, low-level job scheduler for locally prioritizing and initiating SRUN-managed jobs and between-node communication "layers".

SLURM is not a comprehensive cluster administration or monitoring package. While SLURM knows the state of its compute nodes, it makes no attempt to put this information to use in other ways, such as with a general purpose event logging mechanism or a back-end database for recording historical state. It is expected that SLURM will be deployed in a cluster with other tools performing those functions. SLURM was expressly designed to provide high-performance parallel job management while leaving scheduling decisions to an external entity.

CHAPTER 3

Comparison of architectures

3.1 Analysis of Commands

The primary SLURM job-control tool is SRUN, which fills the general role of PRUN (on former Compaq machines) or POE (on IBM computers). The choice of run mode ("batch" or interactive) and the allocation of resources with SRUN strongly affect the job's behavior on machines where SLURM manages parallel jobs. SLURM works collaboratively with POE on AIX machines where SLURM has replaced IBM's LoadLeveler.

To monitor the status of SRUN-submitted jobs, the SLURM utility called **SQUEUE** is used. Similar work is done by the command **llq** in case of IBM's Loadleveler. To monitor the status of SLURM-managed compute nodes the complementary tool called **SINFO** is used. In Loadleveler cluster **llstatus** is used to return status information about machines. All these commands can be formatted and sorted to get data in required format.

On BlueGene/L only, SLURM provides an additional user tool called SMAP to reveal topographically how nodes are allocated among current jobs or partitions. SMAP takes over the terminal window in which it runs. So executing it as a controllee of XTERM, in a separate window dedicated to its output, is a good strategy. The xterm program is a terminal emulator for the X Window System. SMAP needs a window wider than 80 characters to display its character-based "map" of job/node allocations effectively. Generally a 100-character-wide window is requested with XTERM's geometry option.

Because of its prerequisites (above), a typical appropriate SMAP run could begin with an execute line such as this **xterm -geometry 100x30 -e /usr/bin/smap -Dj > /dev/null**

SMAP's character-based map of job/node allocations typically looks like this

```

  a a a a b b d d ID JOBID PARTITION USER NAME ST TIME NODES NODELIST
  a a a a b b d d a 12345 batch joseph tst1 R 43:12 64 bgl[000x333]
  a a a a b b c c b 12346 debug chris sim3 R 12:34 16 bgl[420x533]
  a a a a b b c c c 12350 debug danny job3 R 0:12 8 bgl[622x733]
  d 12356 debug dan colu R 18:05 16 bgl[600x731]
  e 12378 debug joseph asx4 R 0:34 4 bgl[612x713]
  a a a a b b d d
  a a a a b b c c
  a a a a b b c c

  a a a a . . d d
  a a a a . . d d
  a a a a . . e e
  a a a a . . e e

  a a a a . . d d
  a a a a . . d d
  a a a a . . . .
  a a a a . . . #

```

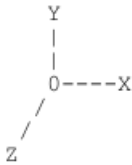


Figure 3.1: Sample smap output (4)

3.2 Usage of environment variables

SRUN and IBM-AIX(Advanced Interactive eXecutive)’s POE (Parallel Operating Environment) both use UNIX environment variables to manage the resources for each parallel job that they run. Environmental variables for a job helps to know certain things about how it was scheduled, what is the job’s working directory, or what nodes were allocated for it. Examples for environmental variables in SLURM are SLURM_CPUS_PER_TASK, SLURM_JOB_NODELIST, SLURM_NTASKS_PER_NODE etc. Examples for environmental variables in Loadleveler are LOADL_STEP_CLASS, LOADL_STEP_NAME, LOADL_PROCESSOR_LIST etc. The variables with comparable roles have different names under each system and both systems have many other environment variables for other purposes too.

Three major differences in usage of environment-variable by SRUN and POE are as follows:

SRUN assigns values to its resource-management variables by means of its own interactive options, one option for each environment variable (plus extra control options, such as -j). Instead, POE uses the usual SETENV or EXPORT utilities to assign values to its environment variables.

POE’s LoadLeveler ignores many environment variables when it run batch jobs under AIX on LC machines. SLURM does not ignore the corresponding environment variables when set by SRUN, even for batch runs.

On LC Linux clusters, the completion of your SLURM-managed batch job on any compute node(s) also automatically terminates any accompanying interactive processes run by you on those same compute nodes (a policy started in August, 2007). Such processes may continue to

run on AIX machines.

This chart lists the SLURM (SRUN-set or inferred) resource-management environment variables for which direct POE counterparts exist.

Environment Variable Role	SRUN Option To Set	SLURM Variable Name	POE Variable Name
Total processes to run	-n	SLURM_NPROCS	MP_PROCS(*)
Total nodes allocated	-N	SLURM_NNODES	MP_NODES(*)
Node list for this job	(inferred)	SLURM_NODELIST	MP_SAVEHOSTFILE(*)
MP ID of current process	(inferred)	SLURM_PROCID	MP_CHILD
Output mode choice	-o (lc)	SLURM_STDOUTMODE	MP_STDOUTMODE
Partition for this job	-p	SLURM_PARTITION	MP_RMPOOL(*)
Debug message level	-d	SLURMD_DEBUG	MP_INFOLEVEL
Output message level	-v (lc)	SLURM_DEBUG	MP_INFOLEVEL
Output label choice	-l	SLURM_LABELIO	MP_LABELIO

(*)Ignored by LoadLeveler for batch jobs on AIX machines at LC.

Figure 3.2: (4)

3.3 Performance Comparison

For research purposes some SLURM tests were performed on a 1000-node cluster in November 2002. Some development was still underway at that time and tuning had not been performed. The results for executing the program/bin/hostname on two tasks per node and various node counts are shown in figure below. It was observed that SLURM performance is comparable to the Quadrics Resource Management System (RMS) for all job sizes and about 80 times faster than IBM LoadLeveler at tested job sizes.

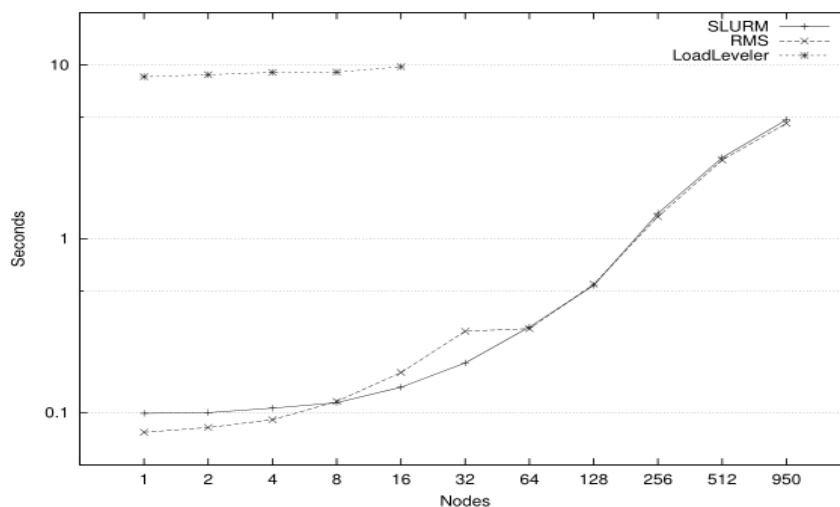


Fig. 8. Time to execute /bin/hostname with various node counts

Figure 3.3: Performance comparison(4)

CHAPTER 4

Development of usage scripts

4.1 Initial work done

The initial idea of the project was to **Develop usage scripts for Virgo** (IITM Super computer, uses Loadleveler for workload managing) users which will be available to users who logs in with their userid and password. They are meant to display user's own data along with general usage information so he can understand how to use the system. The scripts would help the users to get a rough estimate of the time taken to finish a particular job given its requirements. This was expected to be done using the details of jobs already available in the queue and the data about previously finished jobs under various conditions which can be known from history and log files.

After learning the Loadleveler basics and useful commands, I received log files of negotiator, resource manager and scheduler daemons besides LoadL-admin, LoadL-config and History files. These log files include messages indicating what the daemon or process is doing and when the processing is occurring, using timestamps. This includes what transactions being received from and sent to other daemons or processes, and indication of error conditions encountered.

The **NegotiatorLog** contains details about the machines that are contacted for various job-steps given by various users. The **SchedLog** contains information about the job state for various Step ids' at various time instances. The **LoadL-admin** file consists of machine, machine-group, class, group, region stanzas. For each type defaults are defined along with required varieties in them. The **LoadL-config** file consists of definitions of various kinds of schedulers, various daemons macros and other system related information.

All these log files were either giving status of the machine or whether a job-step has started but didn't have information about the exact end time of those jobs. That information is expected to be available in the **history file** which is a large file(20 GB) containing data of finished jobs, collected from all the machines identified in the administration file, but when tried to read, it is showing strange characters in maximum part of the file. The result remained the same even after using various text editors for the reading purpose.

After this, I started working with **llsummary files** which contain job resource information on completed jobs. In standard listing format it provides number of jobs, number of steps performed and total CPU time utilized by each user in required range of time interval. The main issue with llsummary command is that we don't have formatting and sorting options for the output generated by the llsummary command except for three different listing options. The Long Listing would give much more than the required details about each job which is unproductive for the considered task.

4.2 Finding average job time

With the available shortlisting format of llsummary, I wrote an algorithm to find the average cpu time consumed per job for all users that submitted job during the required time interval. It accepts one or more llsummary files as input and generates the combined output.

Listing 4.1: Finding Active Rays

```
1 import sys
2 import re
3
4 def main():
5     a= [x for x in input().split(",")]
6     name = []
7     jobs = []
8     job_cpu = []
9     avg_cpu_time_value = []
10    avg_cpu_time=[]
```

This algorithm accepts one or more llsummary files as input, which are separated by ",". Five different lists are created to store the user names, number of jobs they performed, total cpu time utilised, the average cpu time taken in seconds and hours formats respectively.

```
1     for x in a:
2         f = open(x, "r")
3         fr = f.readlines()
4         for l in fr:
```



```

5     m = re.search(r'(\w+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+:\d+↵
        +:\d+)\s+',1)
6     if m:
7         user_name = m.group(1)
8         no_of_jobs = int(m.group(2))
9         job_cpu_time = (int(m.group(4))*24+int(m.group(5).↵
            split(":")[0]))*3600 + int(m.group(5).split(":")↵
            [1])*60 + int(m.group(5).split(":")[2])
10        if(name.count(user_name)>0):
11            index = name.index(user_name)
12            no_of_jobs = no_of_jobs + jobs[index]
13            job_cpu_time = job_cpu_time + job_cpu[index]
14            avg = job_cpu_time/no_of_jobs
15            k = str(int(avg/3600))+":"+str(int(int(avg↵
                %3600)/60))+":"+str(int(avg%60))
16            jobs[index]= no_of_jobs
17            job_cpu[index] = job_cpu_time
18            avg_cpu_time_value[index] = avg
19            avg_cpu_time[index] = k
20        else:
21            avg = job_cpu_time/no_of_jobs
22            k = str(int(avg/3600))+":"+str(int(int(avg↵
                %3600)/60))+":"+str(int(avg%60))
23            name.append(user_name)
24            jobs.append(no_of_jobs)
25            avg_cpu_time_value.append(avg)
26            job_cpu.append(job_cpu_time)
27            avg_cpu_time.append(k)
28
29        else:
30            n = re.search(r'(\w+)\s+(\d+)\s+(\d+)\s+(\d+:\d+:\d+↵
                +)',1)
31            if n:
32                user_name = n.group(1)
33                no_of_jobs = int(n.group(2))
34                job_cpu_time = int((n.group(4).split(":")[0]))↵

```

```

*3600 + int(n.group(4).split(":")[1])*60 + ←
int(n.group(4).split(":")[2])
35 if(name.count(user_name)>0):
36     index = name.index(user_name)
37     no_of_jobs = no_of_jobs + jobs[index]
38     job_cpu_time = job_cpu_time + job_cpu[index←
        ]
39     avg = job_cpu_time/no_of_jobs
40     k = str(int(avg/3600))+":"+str(int(int(avg←
        %3600)/60))+":"+str(int(avg%60))
41     jobs[index]= no_of_jobs
42     job_cpu[index] = job_cpu_time
43     avg_cpu_time[index] = k
44     avg_cpu_time_value[index] = avg
45 else:
46     avg = job_cpu_time/no_of_jobs
47     k = str(int(avg/3600))+":"+str(int(int(avg←
        %3600)/60))+":"+str(int(avg%60))
48     name.append(user_name)
49     jobs.append(no_of_jobs)
50     job_cpu.append(job_cpu_time)
51     avg_cpu_time.append(k)
52     avg_cpu_time_value.append(avg)

```

The first for loop runs until all the input files are parsed. In the input file the total cpu time utilized can be in two formats. One is **days + hours:minutes:seconds** and the other is **hours :minutes :seconds**. The two cases are dealt separately. The pattern is recognized using regex in python. A for loop parses through all the lines one at a time till it reaches the end of the file. Once if the pattern matches with one of the two possible patterns, the information regarding the username, number of jobs and job cpu time are retrieved to new variables. Then it is checked whether the username is already available or not. If it is previously available the details associated with the username are updated with the new data. Else it gets appended as new username and it's information is stored.

```

1     z=0
2     print("Davg CPU Time = Avg CPU Time for the user - Total Avg CPU
          CPU Time for all the users")
3     print ("User Name      Jobs          Avg CPU Time(h:m:s)      Davg CPU
          Time(h:m:s) ")
4     for i in range(0, len(name)):
5         if(str(name[i])=="TOTAL"):
6             z=i
7             break
8     for i in range(0, len(name)):
9         if(i!=z):
10            if(avg_cpu_time_value[i]> avg_cpu_time_value[z]):
11                diff = avg_cpu_time_value[i] - avg_cpu_time_value[z]
12                val = "+" + str(int(diff/3600))+":"+str(int(int(
                    diff%3600)/60))+":"+str(int(diff%60))
13            else:
14                diff = avg_cpu_time_value[z] - avg_cpu_time_value[i]
15                val = "-" + str(int(diff/3600))+":"+str(int(int(
                    diff%3600)/60))+":"+str(int(diff%60))
16            print (str(name[i]).ljust(10)+"      "+str(jobs[i]).ljust(
                    10)+"      "+str(avg_cpu_time[i]).ljust(15)+"      "+
                    val.ljust(10))
17
18
19     print (str(name[z]).ljust(10)+"      "+str(jobs[z]).ljust(10)+"
          "+str(avg_cpu_time[z]).ljust(10))
20
21 if __name__ == "__main__":
22     main()

```

The index of the username as "TOTAL" is found out using a for loop. Using this the difference between the total average cpu time and average cpu time is calculated and displayed in the output. Due to the inavailability of admin access to me, I wrote the algorithm which takes

Name	Jobs	Steps	Job Cpu	Starter Cpu	Leverage
me16d403	1	1	690+23:47:05	00:00:06	9950270.8
me13d210	2	2	149+22:48:00	00:00:02	6477840.0
ph14d302	3	3	449+16:16:39	00:00:07	5550314.1
me15d403	3	3	313+03:51:53	00:00:01	27057113.0
ph13d014	11	11	102+04:16:04	00:00:06	1471360.7
aml7d015	9	9	1427+10:08:55	00:00:03	41109778.3
aml5d018	16	16	597+07:37:57	00:00:02	25804138.5
ph14d038	3	3	41+22:29:35	00:00:00	(undefined)
me12d001	2	2	184+00:22:12	00:00:01	15898932.0
ph14d034	3	3	267+14:26:05	00:00:01	23120765.0
ae15d200	3	3	6+23:31:06	00:00:00	(undefined)
vc	1	1	118+17:36:46	00:00:02	5129303.0
me15d424	1	1	80+00:39:59	00:00:00	(undefined)
nandiga	2	2	230+16:17:01	00:00:01	19930621.0
cy15d099	15	15	830+06:37:28	00:00:10	7173584.8
ph13d079	8	8	38+16:44:30	00:00:02	1671735.0
ph13d032	3	3	24+11:01:27	00:00:01	2113287.0
mml6d401	1	1	9+23:48:18	00:00:01	863298.0
murali	2	2	131+02:12:32	00:00:02	5663176.0

Figure 4.1: Sample Input

Davg CPU Time = Avg CPU Time for the user - Total Avg CPU Time for all the users

User Name	Jobs	Avg CPU Time(h:m:s)	Davg CPU Time(h:m:s)
me16d403	1	16583:47:5	+16397:29:10
me13d210	2	1799:24:0	+1613:6:5
ph14d302	3	3597:25:33	+3411:7:38
me15d403	3	2505:17:17	+2318:59:23
ph13d014	11	222:56:0	+36:38:5
aml7d015	9	3806:27:39	+3620:9:45
aml5d018	16	895:58:37	+709:40:42
ph14d038	3	335:29:51	+149:11:57
me12d001	2	2208:11:6	+2021:53:11
ph14d034	3	2140:48:41	+1954:30:47
ae15d200	3	55:50:22	-130:27:32
vc	1	2849:36:46	+2663:18:51
me15d424	1	1920:39:59	+1734:22:4
nandiga	2	2768:8:30	+2581:50:36
cy15d099	15	1328:26:29	+1142:8:35
ph13d079	8	116:5:33	-70:12:20
ph13d032	3	195:40:29	+9:22:34
mml6d401	1	239:48:18	+53:30:23
murali	2	1573:6:16	+1386:48:21

Figure 4.2: Sample Output

the input manually. This can be implemented either as a command or as a Cron job to gather information on average cpu times consumed by various users in required time intervals.

4.3 Implementation in SLURM

Accounting information for jobs invoked with Slurm are either logged in the job accounting log file or saved to the Slurm database. The **sacct** command displays job accounting data stored in a variety of forms for the analysis. This can be useful for monitoring job progress or diagnosing problems that occurred during job execution. By default, **sacct** will report Job ID, Job Name, Partition, Account, Allocated CPU Cores, Job State, and Exit Code for all of the current user's jobs that completed since midnight of the current day.

The *-o*, *-format* option allows the users to customize output of job usage statistics. For the required data the following format of **sacct** command can be used:

```
sacct -format=User,JobID,time,start,end,elapsed,NCPUS,NNodes,NTasks
```

Using the above command we will get information about the time at which the job was submitted, started, ended, number of cpus, nodes and tasks associated with it. From this we can get the average waiting time, cpu time for all users, formulate a dependence between them and number of cpus and nodes used, number of tasks per cpu and number of tasks per job using machine learning concepts. For more useful results, we have to include the queue length at the time of submission of the job, finding the expected time to finish the jobs in the queue and updating the dependency relation by comparing with the actual time taken to finish those jobs.

CHAPTER 5

Conclusion

In this technical report a comparison of two workload managers: IBM's Loadleveler and SLURM is provided with the aim of helping the readers to get a brief understanding about their architectures and advantages of using SLURM over Loadleveler.

LoadLeveler is a parallel job scheduling system that allows users to run more jobs in less time by matching each job's processing needs and priority with the available resources, thereby maximizing resource utilization. It was primarily available only for AIX operating system, however it is now also available for POWER and x86 architecture Linux systems.

SLURM is an open-source, fault tolerant job scheduling system started at LLNL and that is now very popular in large supercomputing centers because it offers high scalability and pretty complete functionality by using additional SLURM plugins. Being a Free Open Source Software means that one has access to the code so that they are free to use it, study it, and/or enhance it. These reasons contribute to Slurm being subject to active research and development worldwide, displacing proprietary software in many environments.

As of now, IBM seems to have ended the development of Loadleveler and continuing efforts concentrated around OpenPower, using accelerators such as FPGAs and GPUs. The inadequate availability of documentation and support from IBM for the Loadleveler users and administrators is a huge setback for Loadleveler. In terms of future development, SLURM seems to be the most promising compared to batch systems like **SGE**(Sun Grid Engine), **TORQUE** etc, evaluated including a strong development community.

REFERENCES

- [1] *"Using and Administering"*for IBM LoadLeveler for AIX 5L,
<http://www.hpcx.ac.uk/support/documentation/IBMdocuments/a2278810.pdf>
- [2] *SLURM Workload Manager Documentation*, <https://slurm.schedmd.com/>
- [3] *"Analysis of Batch Systems"* Technical Report - Cesga,
<https://cesga.es/en/biblioteca/downloadAsset/id/753>
- [4] *"SLURM Reference Manual"*, https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c01858965
- [5] *"SLURM: Simple Linux Utility for Resource Management"* by Moris Jette and Mark Gron-
dona, <https://e-reports-ext.llnl.gov/pdf/241984.pdf>
- [6] *"User Commands PBS/Torque Slurm LSF SGE LoadLevel"*,
<https://slurm.schedmd.com/rosetta.pdf>