Utilizing High Performance Computing to Aid in the Analysis of Chemical Abundances of Globular Clusters

Molly Bischoff

Summer Research Scholarship Project 2012/2013

Supervisor: Professor Peter Cottrell
Abstract

Globular clusters are some of the oldest objects in the galaxy. Understanding the chemical evolution of globular clusters would give way to further understanding of the formation and early evolution of the galaxy. The data used for the analysis of the chemical abundances was from the globular cluster ω Centauri. This cluster is quite different from a typical cluster and so it is particularly interesting.

The analysis examined an observed spectrum for each of the 1088 stars. Using photometric observations of each star a synthetic spectrum was also generated by a modified version of MOOG. The data structure and the analysis code were altered so that the process could be run on the BlueFern system over multiple processors. The input data was altered so that it could be easy divided into any number of groups necessary. The observed and synthetic spectrum were then matched and the values of the chemical abundances were extracted.

The analysis of a large data sample is a time consuming task. The use of a high performance computer greatly increases the efficiency of the analysis. Prior to running on the supercomputer the analysis was taking a time of 5.5 days to complete. Currently on the supercomputer this is reduced to only 4.1 hours. This can continue to be reduced with the use of additional processors. Understanding the nuclear synthesis of elements of stars in globular clusters will bring about a new understanding of the formation and early evolution of the Milky Way Galaxy and the entire Universe.
Chapter 1

Introduction

1.1 Scientific Background

Globular clusters are tightly bound star clusters that are typically considered to be the oldest objects in the Galaxy. Globular clusters are mainly found within the galactic halo. The stars in globular clusters tend to have different chemical compositions than those residing in the relatively nearby galactic disk. The stars have been found to be metal poor, that is, depleted in heavy elements. The stars in globular cluster contain information about the formation and evolution of the Galaxy due to the differences (as compared to most other stars) in age and chemical composition[1].

![Figure 1.1: ω Centauri.](image1)

ω Centauri is one of the largest globular clusters in the Galaxy. However, it has numerous features that set it apart from a typical globular cluster. The main sequence is at least bifurcated[3] and the giant branch veritably contains multiple branches[4]. These various individual branches all correspond to varied chemical abundances. Also, ω Centauri contrasts with most mono-metallic clusters as it has a positive correlation with s-process (slow neutron capture process) elemental abundances and metallicity[5]. Most mono-metallic clusters do not show any variation in neutron-capture process abundances; the r- and s- process abundances. These differences object the classification of ω Centauri as a globular cluster[5]. There is speculation of it being the stripped core of a dwarf galaxy[6]. Whatever the formation of ω Centauri may be, analyzing the chemical evolution process within it and other globular clusters will give way to a greater understanding of galactic and early stellar formation. The spectrum of the stars in ω Centauri are received using a multi object spectrograph. The spectrograph can simultaneously obtain the spectra of hundreds of objects.

![Figure 1.2: Colour magnitude diagram of ω Centauri showing multiple branches[2].](image2)
1.2 Supercomputing Background

Supercomputers were introduced in the 1960s. At this time the processing capacity was quite limited as the machines only consisted of a few processors. By the 1990s the machines were utilizing thousands of processors and growing. Today the machines can use upwards of 250,000 processors [7][8]. Supercomputers are used for calculation intensive problems throughout a range of fields including astronomy, quantum physics, weather forecasting and physical simulations (e.g. nuclear weapon detonation).

When using multi object spectrographs to analyze a cluster of stars, a very large number of observed spectra are produced. A computerized process is essential to run the analysis of each of the numerous observed spectra to obtain the required information regarding the physical properties of the stars. Although there are numerous codes and processes to preform the analysis, the job can take a very long time when dealing with such a high volume of data. Taking multiple days to a week to analyze a set of data on a single processor is not an efficient method. Utilizing a supercomputer’s numerous processors can greatly increase the efficiency of the analysis when the data set can be split into groups and run simultaneously.

BlueFern[9] is the supercomputer at the University of Canterbury used for this project. BlueFern was established in 2006 and has several computing facilities with varied focuses. To submit jobs to the BlueFern system, the jobs must be submitted through the LoadLeveler. The LoadLeveler identifies what resources a job needs to run, determines if and when the job can run, and then disperses the job to the proper processor. For multiple jobs, it divides the jobs over the various available and suitable processors. BlueFern has numerous nodes, one of which can run Python scripts. This node is called BlueFern p755 and runs Linux. It has a host name of beatrice.canterbury.ac.nz (Beatrice). The analysis code runs Python scripts that call the program MOOG and so Beatrice is the node used for this project. MOOG has also been installed on the Beatrice node for use. Utilizing the BlueFern system allows for the possibility of the analysis of thousands of stars in a much reduced timeframe.
Chapter 2

Analysis Procedures

2.1 Overview

To analyze the chemical abundances in stars a matching method of a synthetic spectrum to an observed spectrum is used [5]. Different chemicals affect specific spectral lines and thus the chemical composition can easily be studied. The chemical composition and the evolution of the composition can clarify the understanding of the formation and evolution of globular clusters and the Galaxy. To use a matching method a synthetic spectrum needs to be synthesized. The analysis process generates a synthetic spectrum through a code MOOG [10] and matches the synthetic spectrum to the observed spectrum.

For a synthetic spectrum to be generated numerous attributes about the star are necessary such as the temperature, gravity, and metallicity. The data set used contains the photometric observations for each star and the analysis calculates the temperature, gravity, and the metallicity (using the calcium line at 4224Å) for each star. To be able to generate a synthetic spectrum a model file for each star needs to be given to MOOG. The model file provides MOOG with the necessary information to analyze the star. Once these properties are found a useful Python script called Atmosphy [11] can be used to build on databases such as the Catelli-Kurucz grid [12]. Atmosphy is able to determine stellar atmosphere models within Python assuming that the stars have parameters that lie on the grid. The use of Atmosphy allows the model files for each of the stars to be generated quickly. So the analysis code automatically generates a new model file for each star that is analyzes. With the model file, MOOG is able to generate a spectrum and compare it to the observed spectrum of each star. If the synthetic spectrum matches the observed spectrum it can be inferred that the properties used to generate the synthetic spectrum closely match the properties of the observed star.

The input data used is in the form of a comma separated table. The analysis code, Analyzer.py as written by Jeffrey Simpson, uses the input table to generate all of the required information to create and match the synthetic spectrum. Analyzer.py outputs a line per star to an output file that contains the star number, the calculated temperature and gravity, and all of the derived chemical abundances.

2.2 MOOG

To create a synthetic spectrum the local thermal equilibrium (LTE) program called MOOG was used. The spectrum is created via a system of algorithms according to astrophysical models. MOOG was created by Chris Sneden at the University of Texas at Austin. MOOG is often used in the determination of the chemical composition of a star. For the goal of analyzing numerous stars a modified version of
MOOG was used, MOOGsilent. MOOGsilent was developed by Wolfgang Kerzendorf. MOOGsilent is a non-interactive version of MOOG. MOOGsilent looks for a parameter file where the name of the file must be batch.par. The file must be named this because the code cannot stop to ask the user for the name of the parameter file[13]. Using MOOGsilent works very well for analyzing one star at a time. To analyze the data in parallel and on Beatrice, MOOGsilent needed to be modified further. It was modified by Dr. François Bissey to accept a random file name for the parameter file. This allows the code to be run on the BlueFern system. This version, MOOGsilent2, is used for the analysis for this project.

2.3 LoadLeveler

The LoadLeveler has the purpose of distributing multiple jobs over all the available processors. It allows jobs to be run in parallel. Any LoadLeveler script is submitted to the LoadLeveler using the command: llsubmit file.ll. A typical LoadLeveler script contains a header that specifies different attributes for the job. A typical header is seen below:

```bash
#!/bin/sh
#@ group = UC
#@ output = $(Executable).$(Cluster).out
#@ error = $(Executable).$(Cluster).err
#@ job_type = serial
#@ notify_user = mmb80
#@ notification = always
#@ class = p7linux
#@ wall_clock_limit = 50:00:00
#@ rset = rset_mcm_affinity
#@ task_affinity = core(1)
#@ LL_RES_ID=p2n14-c.45.r
#@ queue
#@ suggested environment settings:
export MP_EAGER_LIMIT=65536
export MP_SHARED_MEMORY=yes
export MEMORY_AFFINITY=MCM
```

See Appendix A.1 for explanation of each line of header.

For the LoadLeveler to split the jobs up they need to all be submitted in one group. For instance, if the LoadLeveler script read:

```
header
llsubmit file1.ll
llsubmit file2.ll
llsubmit file3.ll
...
```

The LoadLeveler would run file1, then file 2, then file 3. Submitted in this way the files are run in series as every executable line at the end of a load leveler script is considered a job. To get around this issue a shell script is used. The shell script contains nothing more than the llsubmit command for each of the files being submitted as seen below:

```
llsubmit file1.ll
llsubmit file2.ll
llsubmit file3.ll
...
```

Note that before being able to run, a shell script must be given permission to run by the following
command:

```shell
chmod 755 name_of_shell_script
```

This shell script is then called to run within the .ll file submitted to the LoadLeveler using:

```shell
./name_of_shell_script
```

Using the shell script means that all of the files are being submitted as jobs together at one time as a group. The LoadLeveler can then run the jobs in parallel.

### 2.4 Modifications to Analysis Code

Prior to the analysis code, Analyzer.py, running on the BlueFern system it ran on a single processor directly through Python. To create the job shell script containing groups of the original data set, some modifications were made to Analyzer.py. The goal was that the analysis code would iterate over each of the groups. To do this the analysis code was modified by turning the main portion of it into a function. The function is set to take the inputs as start and stop values. The start and stop values refer to the lines of the comma separated table. The table was also altered into a list of lists so that it could be easily indexed. These modifications allow the grouping method to be easily used with any data set so long as the format of the table is identical.

To easily check if the code ran properly the header line to be written to the output file was removed from the analysis code. This header was originally being written to the output file once for each group. Now the number of lines in the output file is equal to the number of stars, which is the number of lines in the original data table.

Other slight modifications to Analyzer.py were made to fix some errors that occurred when transferring the code to Beatrice. When the code was moved to Beatrice the Atmosphy grid became smaller. This created errors on a few different stars. The error occurred when the code was trying to make a star either very metal poor or very metal rich. The value would be off of the grid and a synthetic spectrum could not be generated. A catch was put into the Analyzer.py code to prevent the error from killing the overall process. The highest metallicity, [Fe/H] \(^1\), allowed is -0.2 and the lowest is -2.5.

### 2.5 Grouping

A python script was written that acts as the grouping process. The script divides the total number of lines (stars) in the input file in a manner that depends on the number of groups, n, specified. The number of stars per group is determined by:

\[
\text{number per group} = \frac{\text{number of stars}}{n},
\]

where \(\text{number of stars}\) is the length of the input file and \(n\) is the number of groups specified.

The script creates \(n-1\) groups that have an equal number of stars. The \(n\)th group has more stars as it is given the extra stars when the total number of stars does not divide into an integer number. For each group, a load leveler file is written by the python script in the form of (start)-(stop).ll. This file contains a typical LoadLeveler header and then a single line that calls the analysis code and runs the analysis function over the group start and stop values as below:

```shell
python2.7 -c "execfile('Analyzer.py'); run_Analyzer()"
```

\(^1\)\([\text{Fe/H}] = \log_{10}(\text{Fe/H})_{\text{star}} - \log_{10}(\text{Fe/H})_{\text{Sun}}\)
The python script then creates the shell script that contains all the jobs to be submitted to the LoadLeveler. The python script also gives this shell script permission to run.

The python script that performs this task of grouping is file_maker_jeffrey.py. It is submitted to the LoadLeveler through the starting.llfile.ll script. The starting file script contains the LoadLeveler header, runs the file_maker_jeffrey.py script then runs the job script. To run a python script from a LoadLeveler file a simple command is used:

```
python2.7 python_script.py
```

Note that it is more efficient to have more groups than processors. Especially considering that the nth group contains more stars than the other groups. One processor might complete it's group faster than another group as some stars take a longer time than others for the analysis.

## 2.6 Running the Code

In summary it is a simple starting process that triggers a line of jobs to analyze the set of data. Any set of data could be used so long as the format of the data matches and all the information needed it available. The code is run by using the “llsubmit starting.llfile.ll” command. This runs the file_maker_jeffrey.py script which groups the data, creates a load leveler script for each group and writes a job shell script. The next line in the starting.llfile.ll script runs the job shell script. This submits all of the group load leveler files to the LoadLeveler in one group so that they can be run in parallel. Each of the group LoadLeveler scripts calls Analyzer.py and runs the analysis function over the group's start and stop values corresponding to lines in the input data. The analysis function calculates the temperature, gravity, and chemical abundances for each star and it also plots a graph of the observed and synthetic spectrum. Each of the groups append an output file which holds all the data regarding temperature, gravity, and abundances.

It is useful to make a reservation on the node that will be used. This ensures that the node will be available when you want to use it. The reservation also prevents runaway unattended jobs to continue to run as there is a time limit along with the reservation. Reservations are also a courtesy to other users on BlueFern. The command used to make a reservation is:

```
llmkres -t 12/12/12 12:00 -d 2880 -n 1,
```

where:
- `t` specifies the date and time of the reservation. Note that the format of the date is American, that is, Month/Day/Year
- `d` specifies the time length of the reservation in minutes.
- `n` specifies the number of nodes needed.

Note that there are many other specifications that can be made, but these were all that was necessary for these jobs.

The reservation is then embedded into each LoadLeveler script header to tell the job to use that reservation. A new reservation is needed for each new run.

There are only a few minimal changes to the code necessary for each run. If reservations are being used, the reservation number needs to be changed in the starting.llfile.ll header and at the top of the file_maker_jeffrey.py script the variable ‘res’ is defined with the reservation number. The only other change is optional and this is the value assigned to n, the number of groups.

To transfer the code to another user there are minimal changes that need to be made. The notify user line in the LoadLeveler header needs to be changed directly in the starting.llfile.ll script. Another user change is needed in the file_maker_jeffrey.py script. In the top section of the script the user is defined.
Chapter 3

Results

3.1 Timing and Efficiency

There were two types of runs executed for the analysis. In a normal run the synthetic spectrum and plots from previous runs were preserved. In a fresh run all of the temporary files, synthetic spectrum, and plots were deleted prior to the run.

Table 3.1: Timing results with the use of multiple processors.

<table>
<thead>
<tr>
<th>Type of Run</th>
<th>Original Time</th>
<th>Current Time</th>
<th>Average Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresh</td>
<td>5.5 days</td>
<td>4.1 hours</td>
<td>7.3 minutes/\text{star}</td>
</tr>
<tr>
<td>Normal</td>
<td>3.2 days</td>
<td>2.4 hours</td>
<td>4.3 minutes/\text{star}</td>
</tr>
</tbody>
</table>

The original time is over a single processor and the current time is over 32 processors. As more processors are used the time taken overall decreases. The efficiency is greatly improved when the code is run in parallel rather than on a single processor.

3.2 Convergence

This analysis is an automated procedure that converges to a solution. For repetitive processes it is possible that convergence is an effect of the procedure. Often it is the set initial conditions that continue to yield the same solution. For this reason it is imperative to have random starting points for the analysis. Testing was accomplished over all the 1088 stars and it shows that convergence is not an artifact of the initial conditions.

Table 3.2: Example testing results over 4 stars to prove convergence.

<table>
<thead>
<tr>
<th>LEID</th>
<th>I. C.</th>
<th>[Fe/H]</th>
<th>[N/Fe]</th>
<th>[C/Fe]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>-1.4</td>
<td>-1.1</td>
<td>-0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>-1.7</td>
<td>-1.1</td>
<td>-0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>-1.5</td>
<td>-1.1</td>
<td>-0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>9</td>
<td>-1.7</td>
<td>-1.7</td>
<td>-0.4</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>-1.8</td>
<td>-1.7</td>
<td>-0.4</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>-1.6</td>
<td>-1.7</td>
<td>-0.4</td>
<td>1.2</td>
</tr>
<tr>
<td>5010</td>
<td>-1.3</td>
<td>-2.2</td>
<td>-0.5</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>-1.7</td>
<td>-2.2</td>
<td>-0.5</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>-1.4</td>
<td>-2.2</td>
<td>-0.5</td>
<td>1.6</td>
</tr>
<tr>
<td>13014</td>
<td>-1.7</td>
<td>-2.3</td>
<td>-0.1</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>-1.3</td>
<td>-2.3</td>
<td>-0.1</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>-1.8</td>
<td>-2.3</td>
<td>-0.1</td>
<td>0.9</td>
</tr>
</tbody>
</table>

LEID is the star ID, I.C. is the initial condition, and $[X/\text{Fe}] = \log_{10}(N_X/N_{\text{Fe}})_{\text{star}} - \log_{10}(N_X/N_{\text{Fe}})_{\text{Sun}}$.
3.3 Deleted Stars

From the original group of data there are two sets of unsuitable stars. One set of stars lies on the horizontal branch and are too hot to have any molecular lines. The analysis code is unsuccessful on these stars. The code returns ‘nan’ for the computed values of the gravity and the abundances. A python script was written to determine which stars produced an output of ‘nan’. The python script looks at both the index for gravity and the nitrogen abundance. If these values are both ‘nan’, then that star LEID number, index 0, is written to another file, bad_star.txt. Another python script considers each line of the bad_star.txt file and the original input file. The script then writes a new input file that is without the unsuitable stars.

Once the analysis code was working properly on Beatrice and running in parallel without any errors the original input list was analyzed. The total number of irrelevant, too hot stars, was found to be 296. The resulting input file contained 1090 stars.

The other set of inadequate stars is a small set of only two stars. These two stars, LEID 40376 and 44209, both create errors as the data line for these stars contain an empty 6th index. This index refers to the ‘Bmag_vL07’ value and is necessary for the analysis.
Chapter 4

Conclusion & Future Work

4.1 Conclusion

The massive and old globular clusters contain information about the formation and early evolution of the Galaxy. Through the analysis of the chemical evolution of globular clusters this information can be inferred. The analysis of the chemical abundances was performed by matching synthetic spectrum to an observed spectrum. Running Analyzer.py on a single processor worked well for analysing a single star at a time, but is inefficient for analysing thousands of stars. Once the modifications to the MOOG code were complete the prospect of running the analysis in parallel on the BlueFern system became a reality. Scripts were written that performed the task of grouping the data set accordingly and submitting each of the groups to the LoadLeveler. The automated analysis then started on each of the groups; when the resources for it became available. The advantage of high performance computing lead to a great extension of the possibilities for the analysis of the chemical abundances of the stars in globular clusters. Through the use of the BlueFern system a process that originally needed over 3 days to be completed can now be done in a mere 2.4 hours. This time can continue to be reduced through the utilization of more processors. Understanding the chemical evolution and formation of stars in globular clusters will help us learn about our own Galaxy and the Universe as a whole.

4.2 Future Work

An analysis procedure was used that created a random number as a starting point within a given range. However, a Monte Carlo type process that creates multiple random initial conditions for the analysis would ensure that convergence is not an effect of the procedure. Multiple random starting points would be a great addition to the analysis.

A code that creates a queue of all the stars would be the best method for running the analysis, so that all the stars were submitted as one long list. Then the first available processor would take the first star, next available would take the next, and so on. This would be a better option than having to divide up the data table into multiple groups.

Another extension of the analysis would be to examine higher resolution data. The higher resolution data would allow other elements to be analyzed. This would create a good opportunity to find a relation to the number of elements analyzed and the time taken for the analysis to complete.

Acknowledgements

Thank you to the University of Canterbury for the financial support over the summer. A big thank you to Professor Peter Cottrell, Jeffrey Simpson, and Dr François Bissey (BlueFern) for your ongoing support and advice. Also, thank you to Andrew Ridden-Harper for your teaching and assistance during the first weeks.
Bibliography


Appendix A

Bluefern

A.1 Load Leveler Header

Any line followed by # will be seen as a comment.
Any line starting with # and followed by a @ symbol will be interpreted by LoadLeveler.

# @ group = UC - specifies the group that the user belongs to. The name of the group the user
belongs to will be provided when they register to use the system. Loadleveler recognizes 4 groups only:
NZ, NZ_merit, UC, or UC_merit. If you are unsure of of your group, run the the command "what-
groupami" on the login node.

# @ output = $(job_name).$(schedd_host).$(jobid).out
# @ error= $(job_name).$(schedd_host).$(jobid).err The above lines specify the files to which stdout and
stderr from the job will be redirected. There is no default, so the user must set something here. The use
of $(schedd_host).$(jobid) is recommended as this matches the hostid/jobid reported by the LoadLeveler
command llq.

# @ job_type = serial - informs LoadLeveler that you wish to run a serial job. Can also be set to parallel.

# @ notify_user = user - specifies the user that notification will be sent to. Use your user ID here.

# @ notification = always - email notification of job completion. Can also set to never to suppress
email notification.

# @ class = p7linux - specifies the job is to run on nodes running the Linux Operating System. To run
a job on a node that runs the AIX Operating System specify a class name of p7aix.

# @ wall_clock_limit = 00:20:00 - Specifies a wall clock limit of 20 minutes for the job. The wall
clock limit has the format hh:mm:ss or mm:ss.

# @ rset = rset_mcm_affinity - allows the user to make use of scheduling affinity to improve the perfor-
mance of their program

# @ task_affinity = core(1) - causes LoadLeveler to run each MPI task on a separate core. There
are 32 cores in each of the p755 nodes.

# @ queue - This line tells LoadLeveler that this is the last LoadLeveler command in the job file.

export MP_EAGER_LIMIT=65536 - most applications perform best for a value of 65536, however the
user might want to check this with their application.

For development work set export MP_EAGER_LIMIT=0. If the code doesn’t work for MP_EAGER_LIMIT=0,
there is a problem with the way it uses MPI, which needs fixing.
export MP_SHARED_MEMORY=yes - Use shared memory inside a node.

export MEMORY_AFFINITY=MCM - Use the memory closest to the cpu.

### A.2 Reservations

To allow the code to run on the p1n...-c machine the job class was changed from p7linux_serial to p7linux. Although the job is serial this works properly and is necessary for a reservation to be made. Note that the jobs are running in parallel, over multiple processors, but the way it is being run on the load leveler is classified as serial as scheduling is not required on multiple processors. Therefore all of the jobs can use the same reservation.

### A.3 General Use and Navigation

To connect to the Beatrice system on BlueFern a ssh connection is used directly from a terminal window with:

```
ssh beatrice.canterbury.ac.nz
```

A username can be added to the command with:

```
ssh [username]@beatrice.canterbury.ac.nz
```

Each user has two directories; /hpc/home/user and /hpc/scratch/user. The home directory is backed up to taped with a quota of 1 GB and thus should only be used for important processes. The scratch directory is not backed up and should be used for any testing or creating. The scratch directory also has no quota.

Files can be transferred to and from Beatrice using scp or ftp. This can be run directly from a terminal with commands such as:

To transfer files from a local computer to Beatrice:

```
scp [filename] [username]@beatrice.canterbury.ac.nz:/hpc/home/username
```

To transfer files from Beatrice to a local computer:

```
sudo scp [username]@beatrice.canterbury.ac.nz:/hpc/home/username/filename [destination]
```

Another option is to use a client such as FileZilla to provide a graphical interface for file transfer.

Scripts that load or write files on Beatrice need to be called with the absolute file path. eg. ‘/hpc/scratch/username/folder/filename’

Any output print statements and error messages produced are saved as text files with suffixes .out and .err respectively.

Other useful terminal commands are below:

To change the current directory:

```
cd [new directory]
```

To go up one directory level:

```
cd ..
```

To list all of the files and directories in the current directory:

```
ls
```
Or to list with more detail:

ls -l

To list all of the running, queued and idle jobs:

llq

To list all of the reservations:

llqres

To view evaluations for a job step:

llq -s [jobname]

To hold the top most job(s):

llhold

To release all held jobs:

llhold -r

To cancel any specific job:

llcancel [jobname]

To cancel all pending jobs associated to a username:

llcancel -u [username]

To delete a specific file:

rm [filename]

To delete everything in the current directory:

rm *

To remove all reservations associated to a username:

llrmres -u [username]

To read files within the terminal:

more [filename]

To read, edit, or create new files:

nano [filename]