Universiteit Leiden
Opleiding Informatica

Deploying Single Particle Analysis

on the LLSC

Name: Neal van Veen
Studentnr: s0718971

Date: Thursday 28th August, 2014

1st supervisor: dr. ir. Fons Verbeek
2nd supervisor: dr. Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The LIACS Life Sciences Cluster (LLSC) is a Beowulf-style[4] cluster of a heterogeneous collection of Dell PowerEdge 2950 server nodes. A lot of techniques used in processing the image data produced by large-scaled electron microscopes and higher resolution nanoscopes, called micrographs, can be applied to clusters composed of older cluster architectures. The separate steps in processing these micrographs is largely done on large scale, multi-core computational clusters, as they can be done in parallel, but are also very expensive in terms of computational power. In 2012 the Leiden Institute for Advanced Computer Science received a large number of Dell PowerEdge 2950 server nodes and a smaller number of Dell PowerEdge 1950 server nodes from an insurance company, called CZ. The decision was made to construct a Beowulf-style cluster to provide such a system for big CPU-intensive tasks. What this means is that the nanoscope operators and all affiliated research groups can use the hardware provided to them by us in a manner that is intuitive to learn and creates a minimum of obstacles in processing their data.
After setup, an investigation is needed to determine the performance increases, and how these effect the execution time of the software that is supposed to run on the cluster. What this research intends to find out, is the size and manner of performance gains and any related factors that might impact the eventual results. However, each additional parameter adds a new layer of complexity, and as such, we will not be able to do an exhaustive analysis, although we hope to answer the primary questions.

Chapter 2

Background

2.1 Single Particle Analysis

To run the experiments and determine if our implementation was successful with regards to the end goals we have to delve a little deeper into the techniques used by EMAN. EMAN is a collection of image analysis tools focused on particle analysis and reconstruction and is the software used in collaborations on the cluster.

Single particle analysis is an analysis technique that is used to create 3D representational structures of small biological structures, such as proteins and viruses. While electron microscopy allows for small-scale microscopy up to a certain level, analysis techniques for ultra-high resolution in nanoscopy are used to correct for errors that arise when trying to capture nano-scale particles. In this regard, single particle analysis allows a researcher to cross over into the nano-scale domain by using the principles of tomography. More specifically, the different orientations of particles are the starting points in creating a 3D model of the particles and single particle analysis is the collection of techniques used to do this.
Cryo-electron microscopy in particular has been invaluable in obtaining the resolution necessary for imaging these tiny samples[28]. The structures of viruses and proteins are incredible complex and to do a full-scale analysis on them practically requires a resolution on the atomic scale. This is so that researchers can model the interactions that these biological micro-structures can do.
An important aspect to note is that cryo-electron microscopy provides a better opportunity to model these structures if they are too large to be investigated by X-ray crystallography or nuclear magnetic resonance imaging.

To obtain proper image samples, specimen are frozen with either liquid helium[8] or liquid nitrogen[26][27]. Explaining the differences between liquid nitrogen and liquid helium cooling is outside of the scope of this project. The next step is to image the samples into micrographs, the raw images of the particles produced by the microscope. Because biological specimen are very sensitive to radiation exposure, only a very small current can be used. As such, each imaged micrograph has a very low

signal-to-noise ratio and different techniques must be employed to alleviate this issue. The first step in these techniques is to classify each particle (bacterial, viral, protein or otherwise) into multiple different orientations in the different dimensions. Particles with the same orientations are translated and/or rotated so the images can be superimposed on each other, such that the features of that particle and its orientation are enhanced. This is done using cross-correlation. In most data sets, including the one we have used for the experiments, there are a large number of micrographs to be used.
The second step is the use of image filtering techniques to remove outlier frequencies that can obfuscate the image and cause a loss of quality when classifying and in subsequent steps. Such techniques generally use Fast Fourier Transforms. To improve contrast and resolution by analyzing the contrast transfer function, several techniques such as phase flipping and amplitude correction are used to correct for the CTF and resolution is improved.
Following these different steps, we end up with a classified database of corrected images of much better resolution and contrast. This database can then be used to create a 3D model of the used particle by in essence combining all these two-dimensional images of the particles in various orientations. In a given sample, the particles are all in different orientations with respect to the detector. This gives us the opportunity to construct a fairly accurate model. Algorithms such as filtered back projection are used to this extent.

2.2 EMAN

2.2.1 Description

In 1999, the software project EMAN[21][10] was first released. It was designed to allow for scientific image processing in the field of transmission electron microscopy: single particle reconstruction in particular. EMAN provides a large collection of tools written in C++ and Python, so that a researcher can have easy access to programs that implement the needed algorithms and techniques used in the field. It uses an image format which contains grey scale information stored in floating-point format and no micrographs, intermediary results or end results are compressed.
Where EMAN was first designed to aid in the 3D reconstruction of particles, it has since expanded to include tools for cryo-electron tomography, crystallography and other types of tomography. In 2010, EMAN was upgraded to version 2. The entire code base was rewritten from C++ and Python to almost exclusively Python. This rewrite meant that the code base was easier to maintain for the team behind EMAN2 and any people who needed to (re)write a certain module, as Python is more often seen as a more productively writable programming language, albeit a bit slower in general. Indeed, Tang et al. [21] tell us that C/C++ have an average speed advantage of factor 5-10 for the 80 different tests used in the paper. However, while EMAN is designed to process data fast, it is also important for the code base to remain clear and concise and the same paper tells us that designing and writing the test programs in a

scripting language takes no more than half the time it does to write the programming in the compiled languages. It is also mentioned that the resulting program is only half as long. However, this slowdown does not cause any issues when comparing the two different versions of EMAN. The project maintainers even tell us that compared to the modules that were written in C++, EMAN2 processes the data sets much faster; although no reference is given, the maintainers claim EMAN2 is up to twenty times faster than EMAN1.

2.2.2 Practical

A conclusive list of EMAN2 subprograms can be found in Appendix C. Although most work in EMAN2 can be done using these subprograms, the designers have created a Graphical User Interface (GUI) to interact with these programs in an easy and organized manner. This GUI program is nothing more than a graphical shell on top of these programs using the Qt library[13].
This program, *e2projectmanager.py*, presents the different aspects of single particle analysis and allows the user to adjust the specific command line flags for each step and program along the way. A lot of work is being done in providing documentation for these options, as most are transliterations of the command line flags and thus provide little clarification as to their meaning. Because each step in the analysis corresponds to an EMAN-program it is sometimes necessary to edit the commands themselves instead of adjusting the presented flags. This can be achieved by a secondary tab in the window that shows the entire command that will be run. One specific issue that will come forward in later steps when continuing the deployment of EMAN2 on the LLSC is how to provide a clear and easy method of submitting jobs to the PBS/TORQUE job scheduler for clusters. While a few subprograms provide a flag for parallel behavior in executing the commands, one must submit a special script to the scheduler for it to be distributed intelligently across the nodes in a cluster. For example, the *e2refine2d.py* program has the command line options set like so:

```
e2refine2d.py --input=sets/all__ctf_flip_hp.lst
--ncls=24 --normproj --fastseed
--iter=6 --nbasisfp=12 --naliref=3
--simalign=rotate_translate_flip
--simaligncmp=ccc --simraligncmp=dot
--simcmp=ccc --classkeep=0.85 --classiter=5
--classalign=rotate_translate_flip
--classaligncmp=ccc --classraligncmp=ccc
--classaverager=mean --classcmp=ccc
--classnormproc=normalize.edgemean
--parallel=mpi:40:/scratch
```

This command is inserted into the TORQUE script and submitted with the *qsub* command, after which the job scheduler takes over. To thread the application across

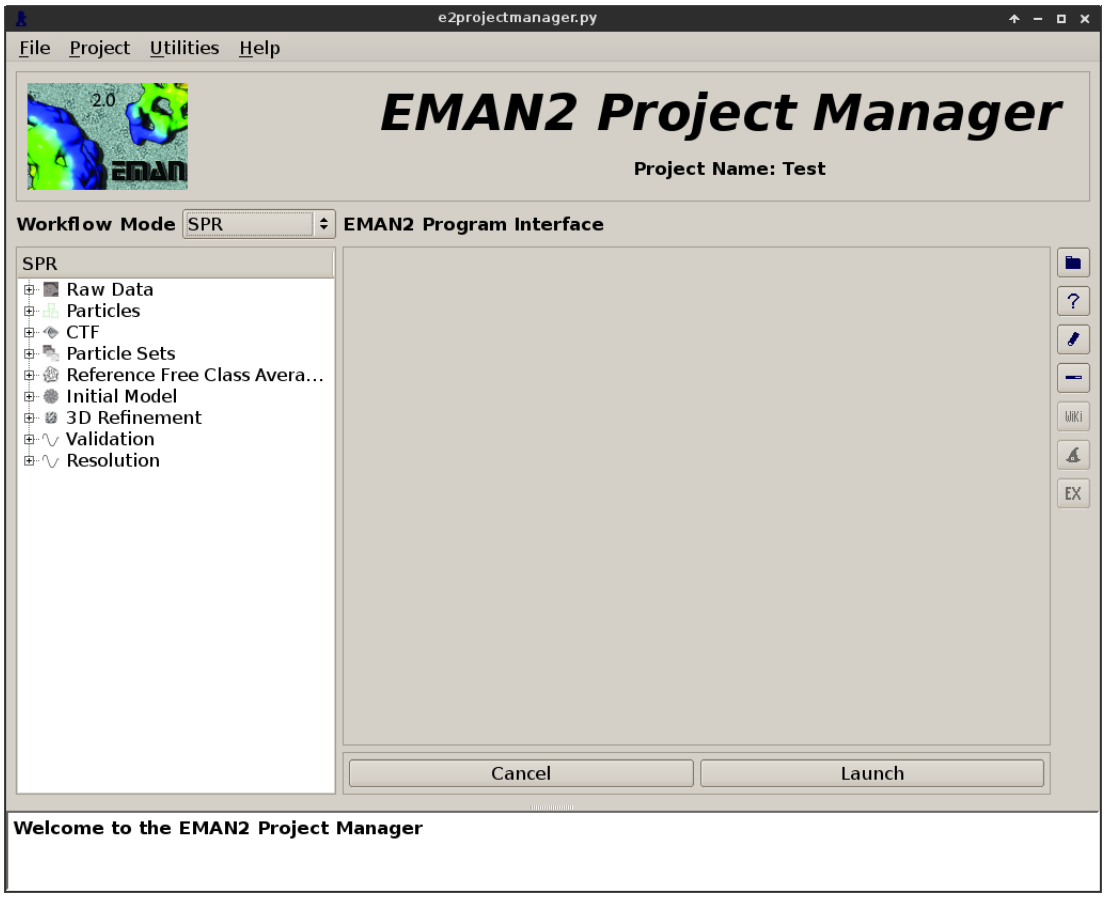

Figure 2.1: The main window of the EMAN2 Project Manager.

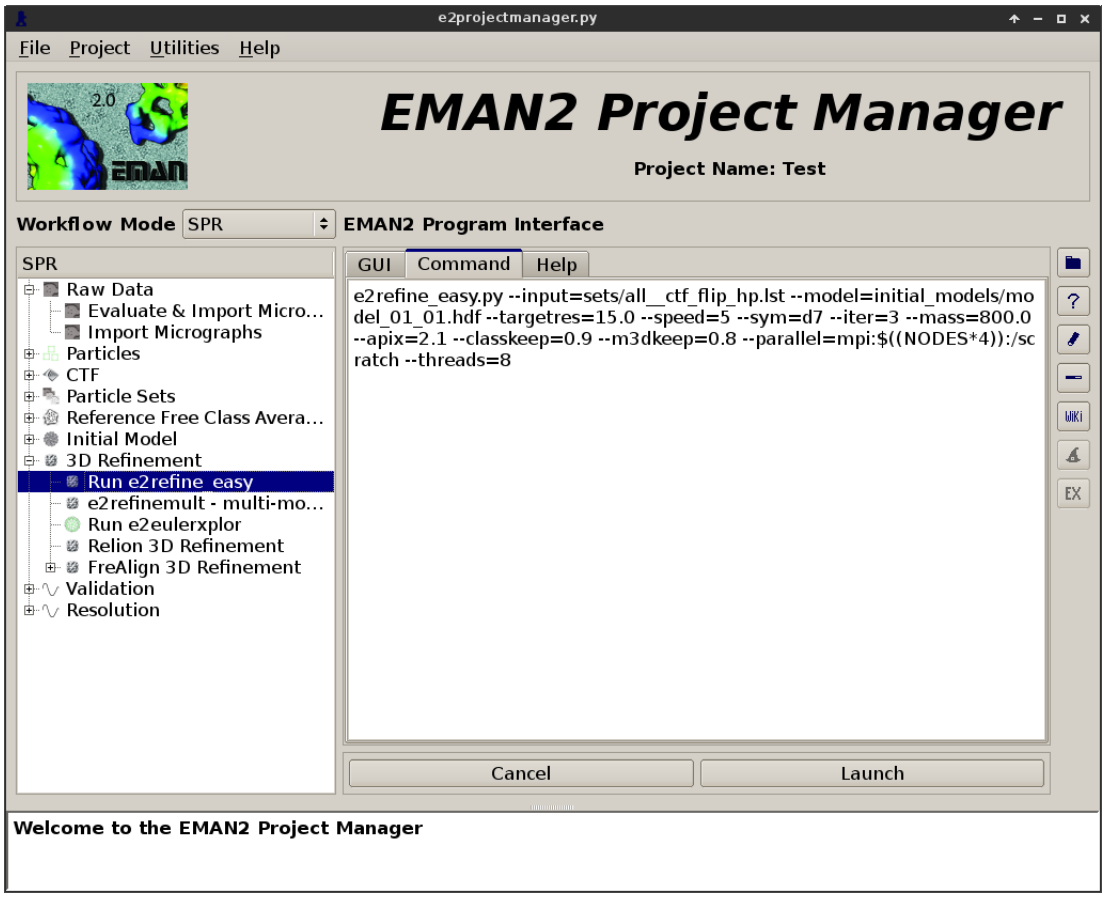

Figure 2.2: The internal command that will be run with the selected parameters.

a CPU:

```
e2refine2d.py --input=sets/all__ctf_flip_hp.lst
--ncls=24 --normproj --fastseed
--iter=6 --nbasisfp=12 --naliref=3
--simalign=rotate_translate_flip
--simaligncmp=ccc --simraligncmp=dot
--simcmp=ccc --classkeep=0.85 --classiter=5
--classalign=rotate_translate_flip
--classaligncmp=ccc --classraligncmp=ccc
--classaverager=mean --classcmp=ccc
--classnormproc=normalize.edgemean
--parallel=threads:8
```

In EMAN2.1 (see Section 2.2.3 for a clarification of the differences between EMAN2 and EMAN2.1), the *e2refine.py* program has been mostly supplanted by *e2refine_easy.py*, that determines a lot of the necessary parameters by an iterative process, so that the user does not have to define them all. Some parts of the program are thus suited for mostly true parallelism across cluster nodes and some are mostly suited for threaded execution on a single node. This is especially true if fast data access is required and the network performance is a bottleneck. As such, the *e2refine_easy.py* program takes two separate parameters for both phases of execution, like so:

```
e2refine_easy.py
--input=sets/all__ctf_flip_hp.lst
--model=initial_models/model_01_01.hdf
--targetres=15.0 --speed=5 --sym=d7
--iter=3 --mass=800.0 --apix=2.1
--classkeep=0.9 --m3dkeep=0.8
--parallel=mpi:40:/scratch --threads=8
```

But, as mentioned, PBS/TORQUE requires a specific type of Bash-script which provides the necessary parameters, such as the number of nodes, the number of CPUs, the amount of virtual memory and other (optional) parameters. When these options are defined, the specific EMAN2-command (an example can be found in the listing of used scripts, Appendix D) is called with the parameters necessary for threaded or parallel execution.
The job is submitted using the program qsub and the status of a job can be queried using showq (for Maui; see also Section 3.3.1).

2.2.3 EMAN 2.1

The next major iteration for EMAN2 has been version 2.1. With it comes a host of changes, such as the removal of the BDB database system from all programs, so that

less time is spent on the parsing of the database and performance gains are made.
As mentioned above, *e2refine_easy.py* was added to simplify the calling of 3D-refinement tools such as *e2refine.py*. The aim of this rewrite was to add automatic parameter selection and return slightly improved 3D structures. Other added features include a new MPI-subsystem and a more refined 3D-viewer[24].
While performing experiments in Chapter 4, we used the work-flow from the EMAN2 tutorial originally demonstrated in Beijing in 2013. This version and its datagram files only works with EMAN2.1, because of the changes in database administration.

2.3 LLSC

2.3.1 Hardware

The LIACS Life Sciences Cluster (LLSC) is a Beowulf-style cluster with the following configuration:

- Approximately fifty Dell PowerEdge 2950 units[6], with either two dual-core Intel Xeon 5150 2.66 GHz processors or two quad-core Intel Xeon 5150 2.66 GHz processors. Each node contains 16 GB of RAM.
- Two file servers, nicknamed Rosalind and Franklin. Each contains 32 GB of RAM, 8 CPU cores and 7.5 TB in a RAID-5 configuration.
- A Foundry Fastron IronEdge 9604 switch. It has 4 1 Gbit/s and 96 10/100 Mbit/s network ports.

A number of servers are not in use [1], either due to faulty hardware or due to a current lack of hard disk drives or RAM banks.
The intended setup used to be 10 nodes in use for user login and application usage, while the remaining nodes would be used for computation. However, after various experiments were done and actual work was simulated, it was determined that so many user nodes is not optimal for computation. For now, this means that the cluster is running a single user node, while the rest is used for computation (bar one). If needed, the cluster can be extended to allow for more users to log in or provide a single dedicated node for web server or database applications by configuring one of the computation nodes for these scenarios.
A primary server is used for cluster administration. It contains the preseed needed to install the OS to the other nodes as either a user node or computation node. It is also the node from which certain administrative commands must be run from. For more information on the administrative commands, refer to Section 2.3.3.
Before each node was selected and prepared for work, a few weeks was spent on organizing the hardware and making an inventory, after which an update was done on the hardware. The reason for the update was a bug in the firmware that did not allow the network interface controllers to run in trunk-mode; a prerequisite for the user nodes in the first iteration of the system's setup.

[1] Servers that were acquired from CZ

2.3.2 Basic software

Each node runs a minimal Debian Wheezy [5] preseed installation as operating system. Each user account resides on the primary file server (Rosalind) and is mounted at login with NFS [9]. User login and authentication is done with LDAP[15].
A lot of cluster-specific tools need password-less access to the nodes in a system. For instance, the *clustertool* program is setup in such a way that to do an update, the tool iterates over all nodes in a list, logs in, and does the manual Debian update command `apt-get update && apt-get upgrade`.
To achieve this, each user account uses password-less SSH keys. The Message Passing Interface-library (MPI) and certain parts of the EMAN2 software suite also use this system.
While the *clustertool* tracks which additional repository packages are installed on the cluster, another system is used for larger projects, including EMAN1, EMAN2, IMOD, and more. This system has separate user accounts for each software suite and thus the binaries and data files reside on the file server. An advantage to this is that we can provide applications that are not provided by the Debian repositories and specific data and project files can be organized in each application's home directory without clouding local directories. A possible disadvantage could be file server performance that can slow down the applications, but as can be seen in Chapter 4, this does not affect EMAN2.
When a user is granted an account on the cluster to be able to use the different programs, local environment variables and overrides are most likely needed to be able to start the programs. This is usually done by sourcing the scripts that contain these variables and are located in the respective directories.

2.3.3 *clustertool*

Main system administration on the cluster is done with a tool simply called *clustertool*. It was originally written by dr. Floris Jan Sicking. It is a bash script containing the functions needed for updating software, running commands, restart nodes, getting information on the cluster status and more. See Section 2.3.3 for a list of commands.

Command	Description
clustertool createsshkeys	When a user is created, create a new password-less key for this user.
clustertool gethome	Print the home directory.
clustertool getload	Print the CPU load.
clustertool getnode	(not used)
clustertool listcores	List the number of CPU cores.
clustertool listnodes	List the number of nodes.
clustertool listprocesses	List the processes.
clustertool listusers	List the users residing in LDAP.
clustertool listgroups	List the groups residing in LDAP.
clustertool listpackages	List the clustertool-installed packages.
clustertool pingnodes	Ping the nodes to check for activity.
clustertool runcommand	Run the supplied command(s)
clustertool runscript	Run the supplied shellscript.
clustertool setpassword	Set the LDAP-password for the user.
clustertool setloginshell	Set the default login shell for the user.
clustertool addpackages	Add the packages with aptitude to the nodes and add them to the preseed.
clustertool createuser	Create a new LDAP-user.
clustertool deleteuser	Delete an old LDAP-user.
clustertool installnodes	Copy the preseed to the new node and add it to the list.
clustertool poweroffnodes	Shutdown the nodes.
clustertool rebootnodes	Reboot the nodes.
clustertool removepackages	Remove the packages with aptitude from the nodes and delete it from the preseed.
clustertool rsyncnodes	Run the *rsync* command on all the requested nodes.
clustertool setpassword	Change the LDAP-password for the user.
clustertool upgradenodes	Update all installed packages with aptitude.

Table 2.1: A list of *clustertool* commands. Most commands require a list of nodes in the format of their type name or hostnames.

Chapter 3

Materials and Methods

An introduction is given to present the various software packages used in parallelizing the work done with EMAN2. We also examine the different alternatives and give the reasons for not choosing those particular packages. Alternatives to the chosen software are many, and we will only explain our direct alternatives that we have not ultimately picked.
As mentioned in Section 2.2, EMAN2 has native support for a number of parallelism methods. Its threaded parallelism functionality is used by certain algorithms when it is not desirable to distribute the work across nodes and incur a performance penalty that arises when transferring data across the local network. The two other parallelism scenarios are for true distributed work on a cluster. An older method that is not being actively developed as of EMAN version 2.1 is the native program that uses only MPI. The user starts a daemon-type program on the main node and the same program is started on other nodes to listen for work and accept. It is a cruder form of distributing the work, because it does not allow for job scheduling, policy management and other required aspects of managing work on a cluster.
However, this method can be selected if the cluster is not as big or when performance overhead by a job scheduler and its constituent modules starts to impact the work. Because of this and with older versions of EMAN, this form of doing parallelism work has been used on a series of ordinary desktop computers in a LAN for quick and 'dirty' distribution of work.
The other, more supported, form of parallelism in EMAN2 involves a group of libraries and applications that EMAN2 uses. Because certain programs in the EMAN2 toolset are programmed to use MPI (see Section 3.1), this means that the work can be efficiently scheduled and controlled using higher level job administration tools such as PBS/TORQUE (Section 3.2) and MOAB/Maui (Section 3.3.1).

3.1 MPI

The Message Passing Interface (MPI) is a system that allows for a standardized and portable method of communication in a parallel system, such as a Beowulf-cluster. It is implemented as a library that a developer calls using the exposed function calls.

It was designed to be portable, efficient and scalable. It is the de facto standard on Linux parallel computation clusters. It functions by passing messages in a point-to-point and collective manner, and is mostly used for parallelism where non-shared memory is the dominant form of memory usage. Written in C, MPI can be called using other languages that can interface with C code, such as C++, Python, Go, C#, Fortran and many more.
A developer is expected to program an application using MPI and then run the compiled program with the *mpirun* executable, so that the application accesses the resources and executes in parallel. The library provides the different methods of communication and the developer uses the memory structures and parallelism concepts while the library "spreads" the work.
EMAN2 is designed in such a way that when certain parameters are given, it uses these structures and functions in MPI, instead of "normal" execution that is at most multithreaded, if at all.
One problem that occurs is to blame on the *libopenmpi1.4* package that can be found in the Debian-repository. This default package is not configured with the `--disable-dlopen` flag. Without this flag, EMAN 2.1 will complain about missing symbols that occur when linking the library with the software. However, we did not see the need to upgrade *libopenmpi1.4* to the 1.8 version it is at now, so we compiled the package with similar flags as to how the Debian-package is compiled for the repository. The biggest reason for this is that some other packages may need the older version (without the `--disable-dlopen`) flag that are still provided by the repository. Only deviating for that specific flag seemed most prudent to avoid breaking anything else, as MPI is used by more than just our programs.

```
./configure --prefix=/opt/openmpi/usr --with-tm \
  --with-devel-headers --sysconfdir=/opt/openmpi/etc \
  --with-threads=posix --without-slurm --disable-dlopen
make
make install
```

As you can see, the package is installed in an */opt/$PACKAGE* location: its own home directory (as mentioned in Section 2.3.2). The file in Appendix A shows how a user that sources the file can then call onto this library instead of the standard package, if it is installed.

3.2 Batch Queuing System

3.2.1 PBS

PBS was started as a Batch Queuing System (BQS), initially developed by MRJ Technology Solutions for the National Aeronautics and Space Administration (NASA) of the United States of America as a joint project between Numerical Aerospace Simulation (NAS) Systems Division of NASA Ames Research Center and the National Energy Research Supercomputer Center (NERSC) of the Lawrence Liver-

more National Library [3]. The company Altair Engineering [2] acquired the PBS technologies and intellectual property from Veridian after that company acquired MRJ in the late 1990s.

3.2.2 TORQUE

The Terascale Open-source Resource and Queue Manager (TORQUE) was forked from PBS [1] by the company Adaptive Computing [1] in 2003. As the name suggests, TORQUE is a resource manager that maintains queues for jobs that are submitted with the utilities TORQUE provides. As said in Section 3.1, programs that use MPI need to be distributed across a cluster. TORQUE is the backbone of this infrastructure in that it maintains controls for handling these specific batch queues and users use its utilities for submission.
TORQUE is the most popular of the free Batch Queuing Systems and is widely used by superclusters at major institutions and corporations, but it is Adaptive Computing that remains the end-maintainer.
While TORQUE also includes its own scheduler, suggestions by dr. F.J. Sicking and personal observations suggested this scheduler is not very refined. It also provides only rudimentary policy support, something that is desirable on our cluster as it will be used for multiple purposes/applications. We arrived at the conclusion that a more advanced scheduler is the better option. Section 3.3.1 shows the different types and which one we used. However, as with MPI, Debian provided an older version of TORQUE, and as such, we needed to compile the newest version, because the combination of EMAN 2.1 and MPI did not play well with this older version of TORQUE. What this eventually amounted to was that we compiled all constituent package at the newest version to ensure no problems with EMAN2, which is being developed with newer versions in mind anyway. For more information on TORQUE, refer to the TORQUE website[23].

Installation and usage The source package TORQUE was acquired from the Adaptive Computing website[1]. The current stable version is 4.2.0, with the snapshot from February 4th, 2013. It contains another Autotools set of scripts, and to ensure minimal changes from older Debian-repository packages, we compiled with similar flags, which we also did for MPI.

```
./configure --prefix=/usr/local \
  --with-server-home=/var/spool/torque --disable-static \
  --with-tcl=/usr/lib/tcl8.5 --with-tk=/usr/lib/tk8.5/ \
  --enable-syslog --with-rcp=scp
make packages
```

Note that this installation is not done in its own home-directory, like MPI. The reason for this is that TORQUE will supplant older versions, while not overwriting the version of TORQUE that ca not be decoupled from other Debian-repository packages.

[1] The terms PBS and TORQUE will be used interchangeably.

Avoiding this overwriting is done by installing to */usr/local*, while almost all Debian packages are installed to */usr*.
As can be seen, the installation is not done with the normal *make install*, but rather a special command is used that builds separate scripts. These scripts are for the different TORQUE versions: server, scheduler, client tools, GUI tools, and the machine oriented miniservers (MOMs). Each script contains the commands to install the tools that are zipped and those archives are appended to the installation script. This way, one only has to copy the respective installation scripts to each node and run to install. The server, client tools and GUI packages are all installed on node-001, which acts as the TORQUE server, and all other nodes, on which jobs can be scheduled, have the MOM package installed. This MOM package contains the libraries necessary for TORQUE and Maui to schedule jobs on the computing nodes.
TORQUE also provides Debian-specialized daemon scripts that are placed in */etc/init.d*.

3.2.3 Other versions and alternatives

Before MRJ was acquired by Viridian, it opensourced PBS into OpenPBS in 1998[16]. However, the last update on the OpenPBS website is from more than 10 years ago, in 2004, and OpenPBS is no longer maintained.
Altair Engineering transformed the acquired property into a non-free version, called PBS Pro[18] when the registration and non-commercial use clauses in the license expired.

3.3 Scheduler

3.3.1 Moab/Maui

The Maui Cluster Scheduler is a set of tools to manage cluster workload. It takes care of logging, policy, job and group control. It can be used in conjunction with TORQUE and, in our case, EMAN2.
The software suite started as an open source software project, like TORQUE. The technologies and intellectual properties were registered by Adaptive Computing (then called Cluster Resources, Inc.). Because of its license, it cannot be used for commercial purposes. This would also impact Adaptive Computing's own commercial successor, the Moab Cluster Suite. While the company advises the end-user to switch to Moab[12], it still provides the Maui set, currently stable at version 3.3.1. We decided to use the somewhat older Maui scheduler, the decision being consistent with our previous intention to focus on open source software.
The Maui scheduler is run instead of the TORQUE *pbs_sched* scheduler and works out of the box. In the future, we can focus on providing advanced job controls and containing job queues and users with policies for a more in-depth cluster controller.
The Maui scheduler should not be confused with the Maui Scheduler Molokini Edition[20], which is an entirely separate (open source) project. The decision not to

pick this scheduler while it is open source under the GNU Lesser General Public License[7] is because the last update was in 2005, while the original Maui scheduler is still supported.

3.3.2 Alternative Batch Schedulers

Oracle Grid Engine is a BQS/Scheduler that started as the Sun Grid Engine before Sun was acquired by Oracle. [17]. On October 22, 2013, the Univa Corporation announced it had acquired the commercial Oracle Grid Engine. Multiple forks of the open source product maintained by Oracle have emerged, including Son of Grid Engine[19] and Open Grid Scheduler[14].
However, none of these successors enjoy as much support as TORQUE and Maui. Another problem is the fragmentation of all these efforts, which means it is harder to decide which version to choose. Apart from the Open Grid Scheduler, our efforts are also focused on open source software.
EMAN2 does provide support for SGE, however. What this means between the different versions we have not looked into, because our focus was already on TORQUE/-Maui. However, considering the fact that they refer to the *Sun* Grid Engine tells us that they use an older version, before Sun was acquired by Oracle or the Grid Engine being adopted by Univa.
The third major alternative we considered is the Simple Linux Utility for Resource Management (SLURM)[25] Like TORQUE and SGE, it is also a job scheduler and resource manager and can be used in conjunction with Maui. It is free and open source and development began as a collaborative effort in the Lawrence Livermore National Laboratory, also like PBS.
SLURM is used by many supercomputers worldwide, including the IBM Sequoia at the LLNL[22], at the time the fastest supercomputer in the world.
While this support and the active development on the project sound appealing, we had already chosen TORQUE/Maui when we came across this alternative. It should also be noted that EMAN2 does not provide proper support for running its jobs on SLURM.

Chapter 4

Experimental setup and Results

With everything installed, we needed a way to determine the answer to our research question. More specifically, we wanted to find out if the cluster provides a performance increase in the computations EMAN2 does compared to a normal desktop execution and how this increase grows (if it does grow). We also wanted to compare a few extra parameters, such as the effects of networked versus local file storage (to a certain extent).
To answer these questions, a work flow had to be established. One problem that arose was the lack of knowledge about single particle analysis and how to operate EMAN2. However, the EMAN2 maintainer has demonstrated the program to colleagues in the field on multiple occasions by designing a tutorial on the usage of EMAN2. The most recent tutorial[11], for 2013, prohibits the use of any version of EMAN2 before 2.1 if the user wants to perform particle box coordinate importing. Another difference is the use of the *e2refine_easy.py* program, which is a new addition to EMAN2.
A simpler metric to test performance increases is to iterate over the amount of nodes while performing the same jobs over and over; in this case those same jobs are the different steps in the tutorial repeated. As we found out by walking through the work flow, the 2D-refinement and 3D-refinement were the 'easiest' steps to parallelize. Each experiment was repeated at least 5 times to average the results, where an experiment is the execution of either 2D-refinement or 3D-refinement on the given data set (the data set from the tutorial).
A few scripts were written to create a number of directories for each amount of nodes used. In the beginning we only had 12 nodes available, so most repeated experiments only iterate until that number. As a reference, we only used 1 single node to get a baseline. One important side note is that running the experiment on a single node was done in two different ways: first by calling EMAN2 directly without using a PBS-script and second by using a PBS-script that calls just the one node. The parameter to specify the number of threads on a single node was forgotten for the primary run and as such only used a single thread, whereas TORQUE automatically parallelizes the job, because you specify the amount of CPUs, even if only one node is selected. As such, the experiments reflect this result in the performance increase by multithreading.
So, to sum the parameters:

- The first parameter is the amount of nodes that run the specified programs.
- The second parameter for experimentation is the program used, *e2refine2d.py* or *e2refine_easy.py*. This parameter was only tested for the first 10 nodes, as this is the maximum number of nodes used on both programs.
- The third parameter is a simpler one: the effect of the networked file system as opposed to local storage. This was done on 16 nodes (see below) and just once.
- The fourth parameter is the the size of the used data set. The 2013 workshop tutorial provides a few data sets with a size that is smaller than the usual data set. A simple method of increasing or decreasing the size of the data set is to double or halve the amount of files. All this does to the result is decrease or increase fitting of the data, something we're not interested in. It is assumed that doing this does not increase or decrease performance. The normal data set size is 1.5GB for the original micrographs.

Just these four different parameters already make the number of experiments balloon until the amount of CPU hours spent is unreasonably large. It makes for a limiting factor in performing more extensive experiments. After these initial runs, more nodes were turned on, so 16 nodes were available. More experiments were performed with these new nodes with the normal and large data sets while iterating in steps of 2 on the amount of nodes.

Already an interesting effect can be seen in Figures 4.1 and 4.2 and Table 4.1. Instead of a linear increase in performance (and thus decrease in execution time), no big increase can be noted for the used data set size after 3-4 nodes. A possible explanation for this result can be attributed to the fact that the *e2refine2d.py* program only partially uses the parallelism functionality and about half of the work is done on a single machine. Another interesting occurrence can be seen in the data for the single node across the different runs. The first value has an execution time that is approximately 75% lower than the other four. A likely explanation for this discrepancy is the way in which the different runs were executed. There have been two methods of calling *e2refine2d.py*: the normal command line method and incorporating the execution commands in a PBS-script. The latter was done once because the entire run was part of a series of commands where each script was created by iterating over the number of nodes in a Bash-script. As such, a difference in execution time was not expected, because the amount of overhead was expected to be negligible. However, after the data was collected and a series of experiments was already performed, we came across the command line flag for enabling threaded single node execution, meaning we did not have this enabled on previous runs. Also, PBS allows one to define the number of CPUs on a single node and because the standardized method of generating the script of the single node, this also included the 4 CPUs as

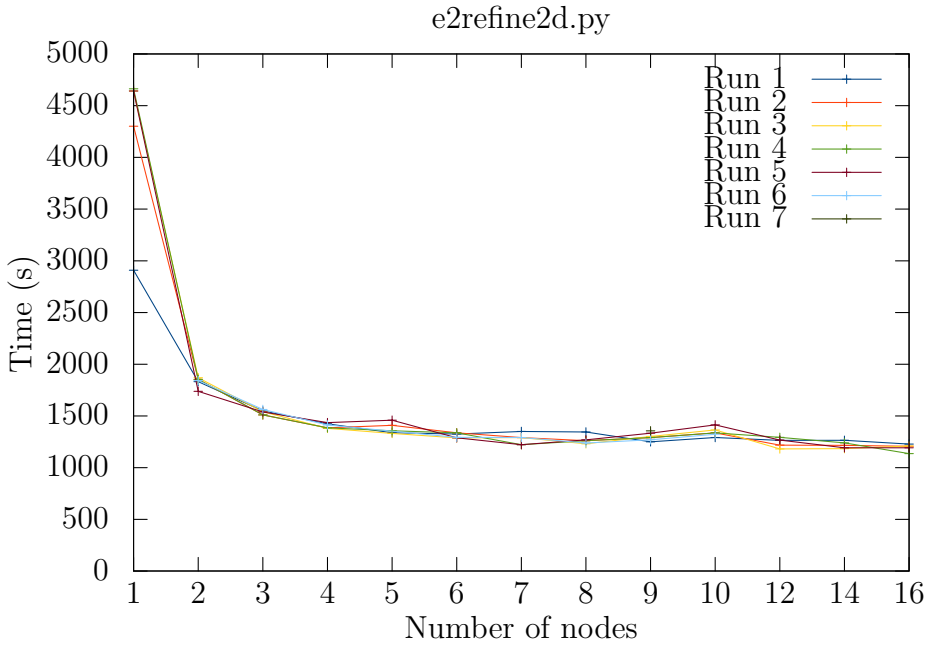

Figure 4.1: A set of experiments run with *e2refine2d.py*.

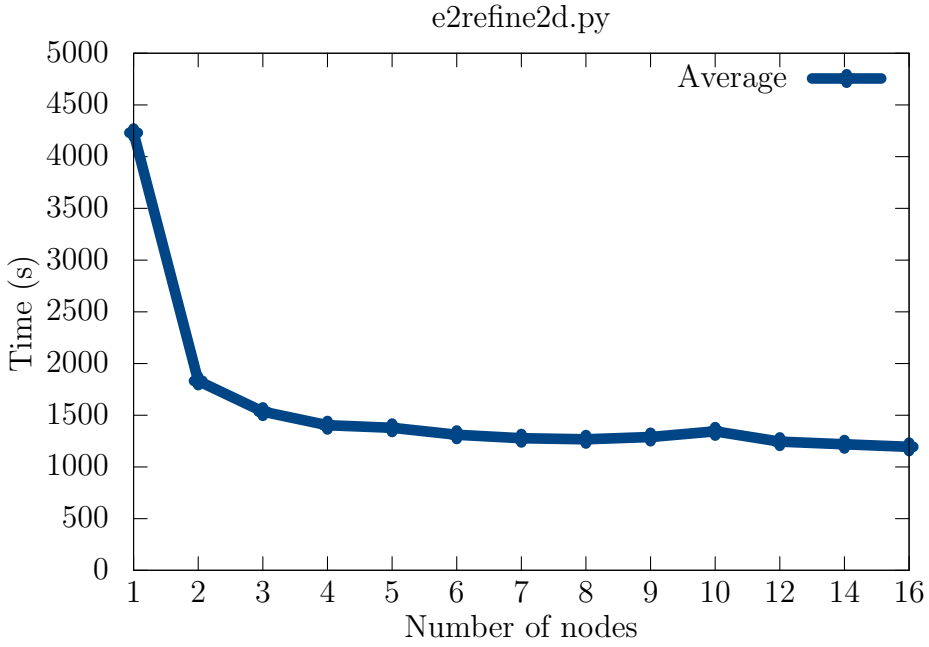

Figure 4.2: An average of the experiments from Figure 4.1.

a parameter. In short, PBS emulated the threading functionality and speeded up the run significantly, whereas the other runs came up short. We decided to incorporate this value to show this difference.
For any other run, the single node threaded execution is a parameter of which we decided against using it for the other runs, as this created another level of parameter complexity. That said, threaded execution provides a large speed increase that should always be used on top of the multi-core parallelism by PBS.
The table for *e2refine_easy.py* (Table 4.2) shows a somewhat different picture, especially after eight nodes. At first, there is a spike in execution time, indicating performance decreases, at two nodes, as opposed to the single node. The most likely explanation for this is the overhead caused by the scheduler, as it is assumed *e2refine_easy.py* can and does create more jobs. It is then preferable to stick to one node as using two creates an overhead that only adds to the total execution time.
After two nodes, a performance gain and limit that is similar to *e2refine2d.py* can be seen, up until the experiment for 8 nodes. Major drops in execution time can be seen: these are so big that one can only assume they represent an invalid run. As noted before a large amount of simple scripts were used that iterate over the number of nodes and do a simple query to PBS as to the state of submitted jobs. However, if, for example, a job has been blocked but not deleted, because there are not enough CPUs to allocate, the script will fail. Other scenarios for failure are the scripts used for extracting execution times from the logfiles. If a job has been running, and is killed by something else but still executes correctly, it logs an endtime and these values are also extracted.
We decided to include these values so that incongruent data is also represented. The general scope of the experiment is not affected too much, especially when looking at the averaged data in Figure 4.4. Another parameter for experimentation is the difference in data set size. As mentioned before, using the same data set, the amount of micrographs and particle coordinate data was duplicated for the large set and half the micrographs were discarded for the small set. Both experiments were run on 8 and 10 nodes to be able to more disregard the amount of nodes as a parameter.
As can be seen in Table 4.3, the speedup for 8 and 10 nodes improves slightly with more nodes, as can be expected. However, the ratio for both set sizes is vastly more than the ratio of 25% which can naively expected, as the small set is 25% of the size of the large set (remember that the large set is double the normal set, and the small set half the normal set). This is another indication for the effect the results from the first experiment showed: namely that the increase in performance is not linear with the number of nodes added and the decrease in size of the data set only offers a smaller performance gain.

After a number of nodes were added to increase the total amount to 16 at the time of doing the experiments, it was decided to do a repeat experiment similar to the previous experiment concerning data set sizes. As Table 4.4 shows, only the normal and large data sets were used, for 2^n nodes. Figures 4.7 and 4.8 indicate a negligible decrease in execution time between 8 and 10 nodes for the large average and a slightly larger one for the normal average.
While the larger data set is exactly twice as large as the normal data set, again there is

Nodes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Average
1	2909	4301	4642	4663	4642			4231.4
2	1832	1854	1870	1854	1738	1842		1831.67
3	1549	1510	1541	1510	1539	1563		1535.33
4	1424	1385	1380	1385	1436	1413		1403.83
5	1338	1409	1331	1357	1459			1378.8
6	1325	1337	1288	1337	1289	1293		1311.5
7	1350	1291	1291	1222	1222	1292		1278
8	1345	1261	1231	1261	1269	1238		1267.5
9	1250	1291	1302	1291	1333	1270	1356	1289.5
10	1292	1336	1365	1336	1414	1323		1344.33
12	1264	1218	1181	1293	1267			1244.6
14	1265	1216	1184	1240	1192			1219.4
16	1228	1207	1210	1136	1192			1194.6

Table 4.1: The data set result from the first experiment (Figures 4.1 and 4.2).

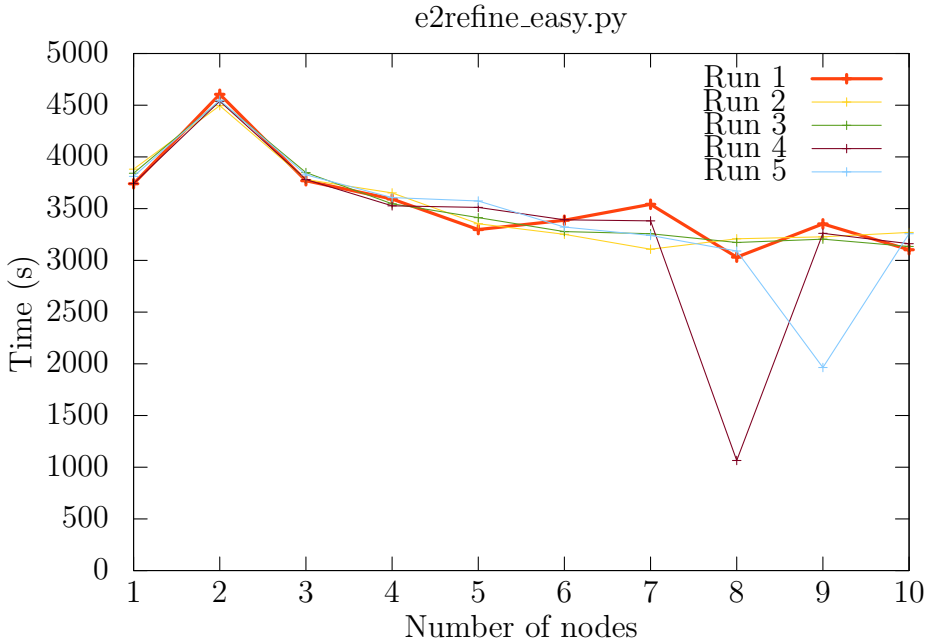

Figure 4.3: A set of experiments run with *e2refine_easy.py*.

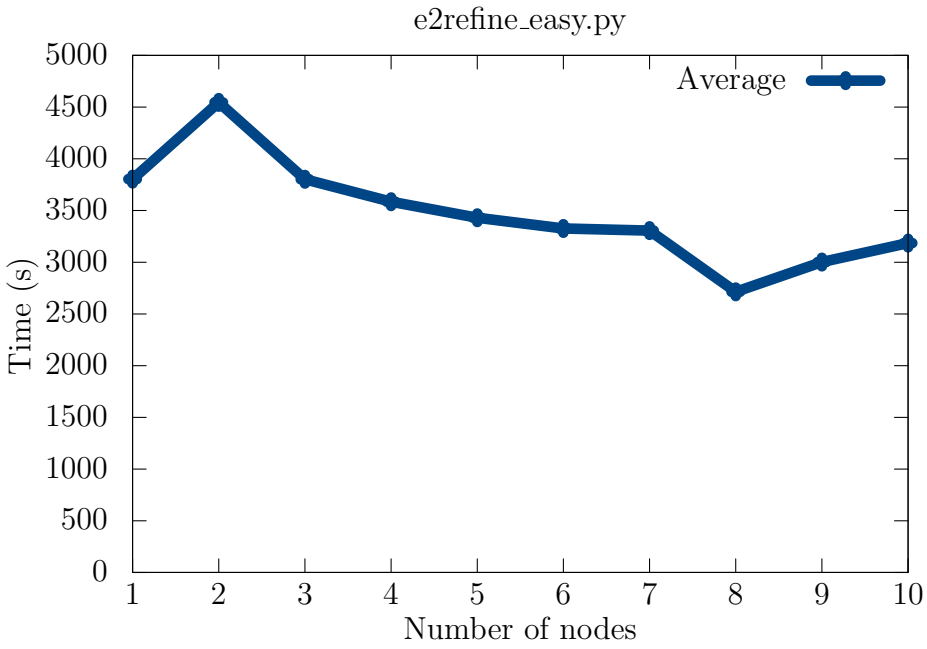

Figure 4.4: An average of the experiments from Figure 4.3.

Nodes	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1	3741	3882	3842	3742	3812	3803.8
2	4605	4496	4540	4539	4548	4545.6
3	3770	3782	3849	3782	3827	3802
4	3593	3652	3547	3527	3607	3585.2
5	3298	3354	3413	3512	3574	3430.2
6	3388	3252	3278	3392	3322	3326.4
7	3542	3108	3258	3382	3241	3306.2
8	3031	3209	3173	1065	3090	2713.6
9	3353	3228	3205	3261	1964	3002.2
10	3103	3270	3134	3161	3258	3185.2

Table 4.2: The data set result from the second experiment (Figures 4.3 and 4.4).

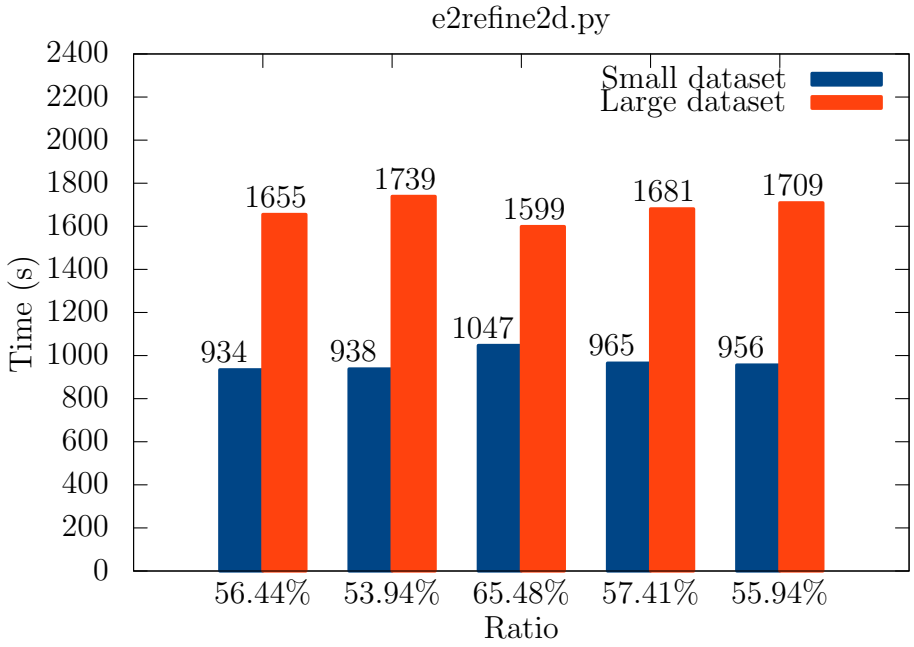

Figure 4.5: A comparison of data set sizes for *e2refine2d.py* with 8 nodes.

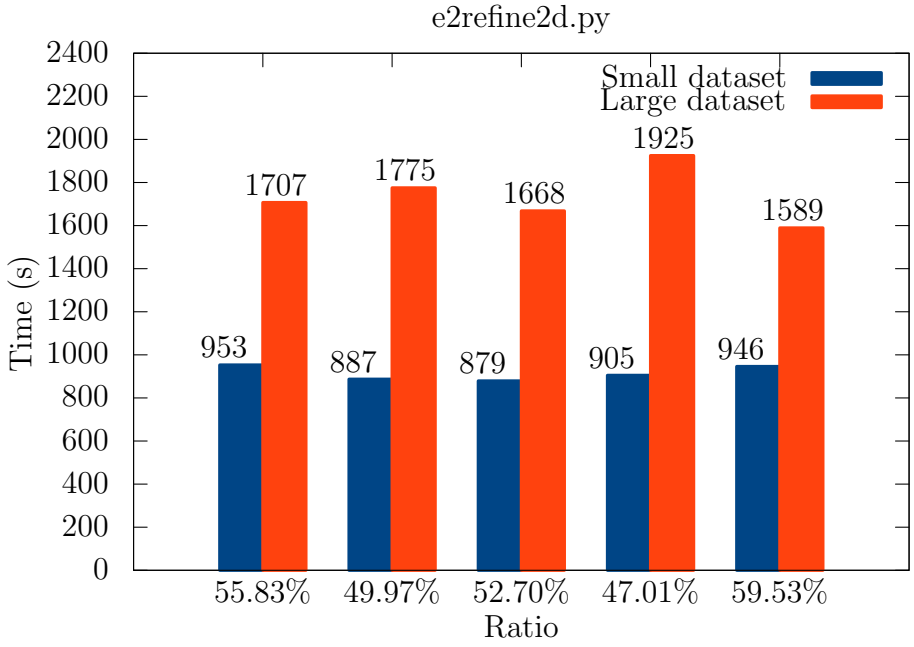

Figure 4.6: A comparison of data set sizes for *e2refine2d.py* with 10 nodes.

8 nodes			
	Small	Large	Speedup
	934	1655	56.44%
	938	1739	53.94%
	1047	1599	65.48%
	965	1681	57.41%
	956	1709	55.94%
Average	968	1676.6	57.84%
10 nodes			
	Small	Large	Speedup
	953	1707	55.83%
	887	1775	49.97%
	879	1668	52.70%
	905	1925	47.01%
	946	1589	59.53%
Average	914	1732.8	53.01%

Table 4.3: The results for the data set size-experiment Figures 4.5 and 4.6.

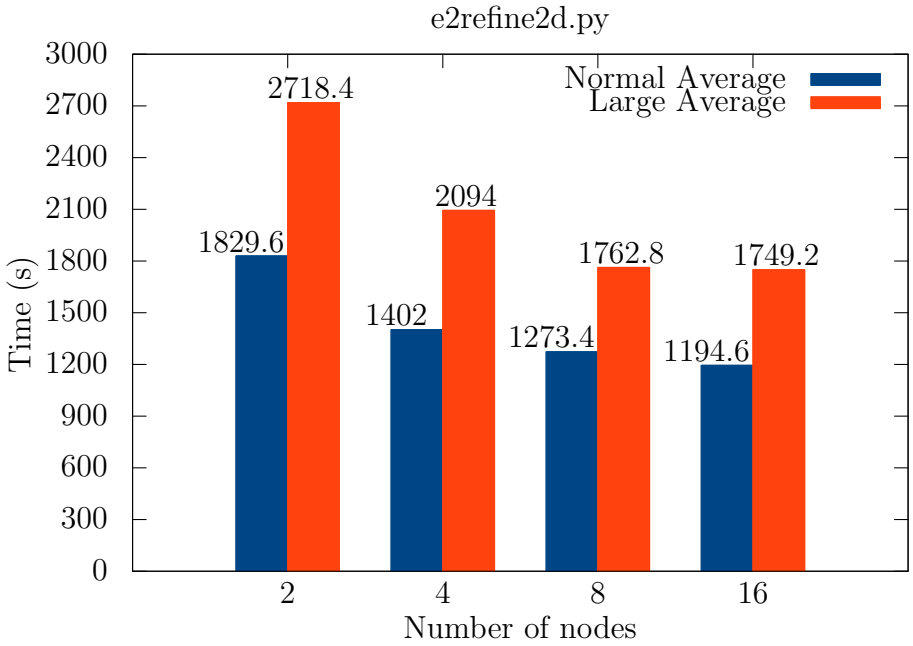

Figure 4.7: A comparison of the average of normal and large data set size runs over 2^n nodes.

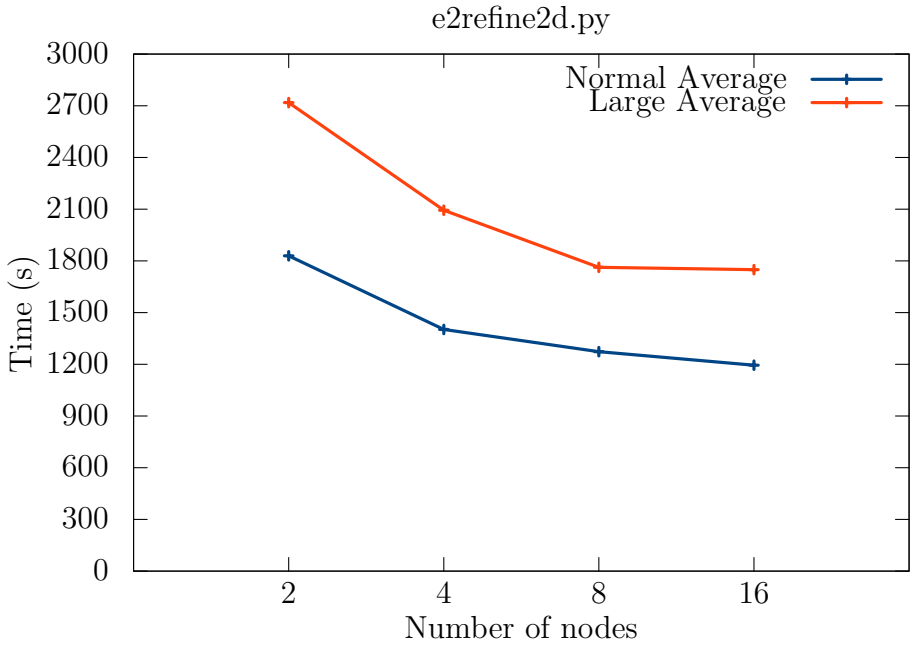

Figure 4.8: An indication of the performance increase from Figure 4.7.

Large							
Nodes	Run 1	Run 2	Run 3	Run 4	Run 5	Large Average	
2	2935	2639	2685	2683	2650	2718.4	
4	2206	2356	1951	1976	1981	2094	
8	1798	1853	1764	1633	1766	1762.8	
16	1771	1792	1748	1703	1732	1749.2	
Normal							
Nodes	Run 1	Run 2	Run 3	Run 4	Run 5	Normal Average	Ratio to large
2	1832	1854	1870	1854	1738	1829.6	67.30%
4	1424	1385	1380	1385	1436	1402	66.95%
8	1345	1261	1231	1261	1269	1273.4	72.24%
16	1228	1207	1210	1136	1192	1194.6	68.29%

Table 4.4: The results for the data set size-experiment Figures 4.7 and 4.8.

no doubling of execution time, but rather a value that hovers around 67%, regardless of the number of nodes. Sadly, more data is not available due to time constraints and the aforementioned ballooning of testable parameters.

An easier parameter that is testable and fast is to compare local storage and file server storage. The data locality experiment was set up without the use of TORQUE and EMAN2's parallelism functionality. Two identical scripts were made and run on a single node, with the only difference being that one had its scratch directory on a local disk and the other remained in network storage (more specifically, the user's home directory). With an experiment like this, we might be able to say more about communication between nodes impacting the execution times of jobs. If at a certain point this sort of data transfer between nodes takes too much time compared to the actual work of the jobs, it could cause a slowdown seen in the first experiments with an increased number of nodes.

As Figure 4.9 and Table 4.6 show us, the file server storage for that experiment is faster by 20 seconds. However, this means a performance increase of 0.66% on this scale. We believe this difference to be negligible, as the file server storage method being faster would not necessarily be true as network transfer speeds are simply not as fast as local transfer speeds.

Of course, this does not mean that network performance for temporary data is taken into account in this experiment, as most parallel programs also need to do quick communication between nodes and store this data locally. For instance, EMAN2 requires a scratch directory on each node for this temporary sort of data. The transfer of this information can cause overhead in a real life scenario, and it is not something we have tested for. Rather, this experiments was done to exclude the differences in containing the scratch directory in different locations, networked or not. The biggest problem with trying to test for networked storage and its impact on multiple nodes is that a scratch directory must be unique to each node. We could make separate directories for each node on the file server and use a symbolic link to this directory from the node. This is done because each scratch directory on each node must also use the same name, as it is a parameter in the parallelism-flag in *e2refine2d.py* and *e2refine_easy.py*. This would perhaps show us a noticeable difference when compared to a normal run with the same amount of nodes, but it takes a fairly involved setup and ultimately was a parameter that we didn't test for.

Small	Normal	Large
57.84%	72.24%	100%

Table 4.5: A summary of the speedup of execution time compared to the large data set (for 8 nodes).

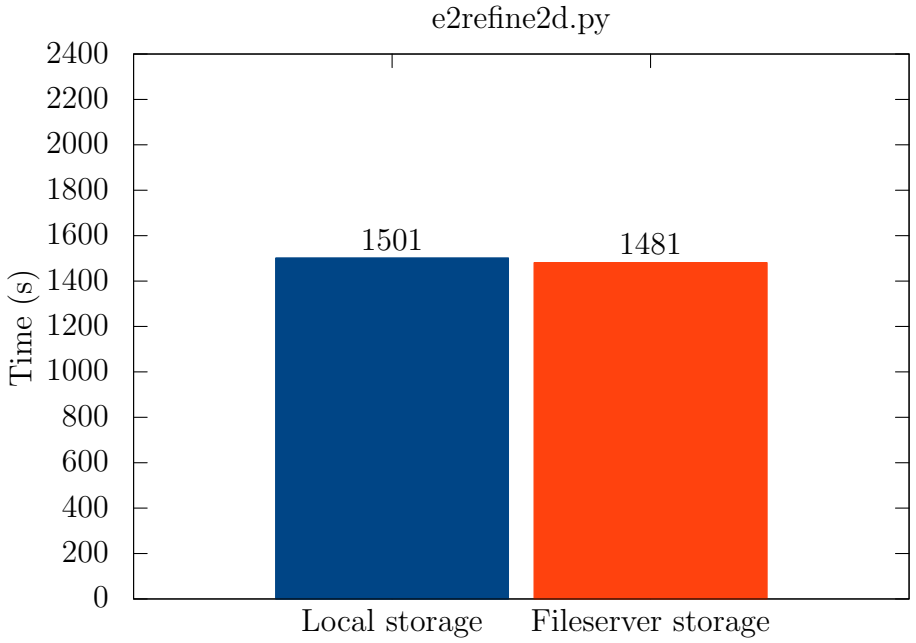

Figure 4.9: A comparison of file server and local storage performance.

Local storage	File server storage
1501 s.	1481 s.

Table 4.6: A summary of the speedup of execution time compared to the large data set (for 8 nodes).

Chapter 5

Discussion and Conclusion

The generated data is very useful as it shows a non-linear, non-trivial amount of performance gains and because of that, a user must be careful in assigning nodes and take into account the size of the data set and leave out nodes if performance increase for similar sized data sets do not show increases. While the cluster is used more by researchers for actual work, more data, especially about workload analysis, will be generated and a clearer picture will emerge about what the preferred assignment of nodes should be.
The first set of experiments shown in Figures 4.1 to 4.4 immediately showed us that a simple increase of nodes doesn't necessarily mean a linear increase in performance and in this regard, with more data being collected after actual work is done on the cluster, we can more accurately determine how many nodes a specific job is assigned. Policy management on the cluster might even allow us to standardize assignment based on this data.
However, we do know that there is a drop in performance increase after a certain threshold (about four to eight nodes for this size of data set) that seems to be partly related to data set size. The cause for this is future work. It is expected that computation must be able to scale beyond four to eight cores.
There are a few differences between the programs used in refinement of the image data, namely in the assignment of two nodes: *e2refine_easy.py* shows a marked slowdown and this will have to be taken into account. Our research question being the analysis of the cluster's performance and how it relates to the usage of EMAN2 gave us an opportunity to show the advantages of this type of cluster and the gains we can make by using it. In this regard, our experiments show a significant boost in performance with diminished returns, answering the question in which manner the execution times are reduced. In most cases, EMAN2 and image analysis software of this kind benefits greatly from CPU clusters as the work done is high-intensity and complex. Adapting EMAN2 has already provided us with results.
The LLSC was based on an older design of a CPU cluster and the *clustertool* program was made so that we could manage the system. However, our differing views on the setup and changing requirements have prompted us to adapt existing software packages and reduce the number of user nodes and keep the root node as a backup that will not be used while only the single user node is used. As it stands now, the

cluster is a very valuable tool in single particle analysis and a lot of mileage can be gained. TORQUE and its related tools, MPI and Maui, will help us in that regard.

5.1 Conclusion

In summary, after the construction of the cluster, we needed to execute a set of experiments to show us if our assumption about the performance gains was correct. The results are somewhat surprising in that the gains don't scale with the number of nodes after about four to eight nodes.
It is necessary to determine baselines for normal sized data sets that are used in actual research projects. Such efforts are already underway and as time goes on this will improve our understanding of the scaling of nodes.
The most work done in this project was the installation of the software. Determining which versions would work properly, finding the right setups (locations to install, etc.), and determining how to run everything cleanly and without interruptions was a daunting task. While we were involved in making an inventory of the nodes and updating all firmware, the most time in this project was spent in compiling the various packages and ensuring they worked correctly. After this work was done, experiment setup was done. A system so complex as a cluster has a lot of testable parameters to determine what the best course of action is for node assignment when working on a real project. As such, our collected data is not conclusive by any means and only provides an indication for the cluster's behavior when running a single software package, since we could not test every case.
Also, while it may seem obvious, we have found a sharp increase in performance for using the cluster as opposed to single desktop computers that were used previously. This means that already the cluster has proven its worth and we believe that with larger data sets this increase will be all the more noticeable. When parts of the cluster are going to be upgraded (see Section 5.2), we can expect to see larger improvements.
Different parts of EMAN2 (the ones experimented with, *e2refine2d.py* and *e2refine_easy.py*) show us a slight difference in execution, but ultimately one with the same behavior, so the same factors need to be taken into account.
All in all, the cluster provides the LIACS and its partners with a useful tool to be used for nanoscopy research projects.

5.2 Next steps

- The diminishing returns shown in the experiments are cause for more research into this behavior.

- The first major improvement we can make to the workflows of researchers using the cluster is something that has already been requested, although we can not provide it yet. This is the addition of graphics processing units to allow for certain steps to be run on these cards when they are optimized for this type of

work. GPUs are becoming increasingly popular for work that is highly parallelizable and consists of small tasks. EMAN2 contains basic support for a few types of GPUs and other software that is to be run on the cluster might also use this functionality. Because of its nature, GPU-based work will most likely provide us with gains that complement the gains in performance we have seen with the CPU-based parallelization across nodes.

- Other steps to be taken next are mostly administrative ones. The *clustertool* program is slightly outdated with respect to the current architecture of the cluster and some parts will need to be changed.

- Furthermore, we will turn on more nodes iteratively and replace nodes that have already been broken. After all, because of the age of the hardware, some things are bound to break relatively fast.

- The cluster's switch will need be updated so we can use more gigabit Ethernet connections. The file servers will probably also need to be replaced, as they are already showing signs of malfunction at times. In conjunction with this work we will need to enable more redundancy in terms of data storage, as only limited forms of data backup is used. However, this is the first step in that this is already being worked on.

- In terms of EMAN2, we will need to find new ways to simplify the usage of batch scripts so that inexperienced researchers do not need to receive a crash course in how to operate TORQUE and other software related to the work. This includes ways to automate the generation of these scripts and software adjustments to EMAN2 that may be submitted to the EMAN2 maintainers. Solutions to this might include the usage of a web interface.

- In time, we will also look into setting up a few Dell PowerEdge 1950 1U servers that were also purchased and use them for more dedicated independent work, such as a web server and other functionalities.

Appendices

Appendix A

openmpi.bashrc

Listing A.1: *openmpi.bashrc* (10)

```
export MPIDIR=/opt/openmpi
export PATH=${MPIDIR}/usr/bin:${PATH}
export LD_LIBRARY_PATH=${MPIDIR}/usr/lib:${LD_LIBRARY_PATH}
```

Appendix B

eman2.bashrc

Listing B.1: *eman2.bashrc*

```
export EMAN2DIR=/opt/eman21/EMAN2
export PATH=${EMAN2DIR}/bin:${EMAN2DIR}/extlib/bin:$PATH
export PYTHONPATH=${EMAN2DIR}/lib:${EMAN2DIR}/bin:${PYTHONPATH}
alias sparx=sx.py
```

Appendix C

EMAN2 programs

e2.py
e22.py
e2RCTboxer.py
e2_real.py
e2align2d.py
e2align3d.py
e2basis.py
e2bdb.py
e2boxer.py
e2boxer21.py
e2buildsets.py
e2buildstacks.py
e2classaverage.py
e2classaveragebysym.py
e2classextract.py
e2classify.py
e2classifykmeans.py
e2classifyligand.py
e2classptcl.py
e2classvsproj.py
e2cmmtomrc.py
e2cmpxplor.py
e2ctf.py
e2ctf2eman1.py
e2ctfsim.py
e2ddd_movie.py
e2display.py
e2emx.py
e2eotest.py
e2eulerxplor.py
e2evalimage.py
e2evalparticles.py
e2evalrefine.py
e2fhstat.py
e2filtertool.py
e2fsc.py
e2helixboxer.py
e2help.py
e2history.py
e2iminfo.py
e2import.py
e2initialmodel.py
e2make3d.py
e2make3dpar.py
e2markbadparticles.py
e2montecarlorecon.py
e2motion.py
e2msa.py
e2parallel.py
e2pathwalker.py
e2pdb2em.py
e2pdb2mrc.py
e2plotEulers.py
e2plotFSC.py
e2proc2d.py
e2proc2dmulti.py
e2proc3d.py
e2procpdb.py
e2project3d.py
e2projectmanager.py
e2projectupdate21.py
e2ptcltrace.py
e2rawdata.py
e2rct.py
e2refine.py
e2refine2d.py
e2refine_easy.py
e2refine_evenodd.py
e2refine_postprocess.py
e2refinefromfrealign.py
e2refinemulti.py
e2refinemultinoali.py
e2refinetofrealign.py
e2refinetorelion2d.py
e2refinetorelion3d.py
e2refinevariance.py
e2refmerge.py
e2reliontoeman.py
e2resolution.py
e2runfrealign.py
e2scannereval.py
e2segment3d.py
e2seq2pdb.py
e2simmx.py
e2simmx2stage.py
e2simmxxplor.py
e2skelpath.py
e2speedtest.py
e2spt_autoboxer.py
e2spt_boxer.py
e2spt_classaverage.py
e2spt_fftamp.py
e2spt_hac.py

e2spt_refinemsa.py
e2spt_refinemulti.py
e2spt_refprep.py
e2spt_resolutionplot.py
e2spt_scramblestack.py
e2spt_simulation.py
e2spt_subtilt.py
e2spt_tiltstacker.py
e2spt_wedge.py
e2ssehunter.py
e2ssematch.py
e2stackanim.py
e2stacksort.py
e2symbest.py
e2symsearch3d.py
e2tilefile.py
e2tiltvalidate.py
e2tomogram.py
e2tomoresolution.py
e2unwrap3d.py
e2version.py
e2version.pyc
e2workflow.py
sx.py
sx3dvariability.py
sx_real.py
sxair.py
sxali2d.py
sxali2d_mref.py
sxali3d.py
sxali_vol.py
sxave_ali.py
sxbmask.py
sxcopyfromtif.py
sxcpy.py
sxcter.py
sxctf.py
sxfactcoords.py
sxfilrecons3d.py
sxfilterlocal.py
sxfind_struct.py
sxfit_error.py
sxgenbuf.py
sxgui.py
sxhac_averages.py
sxhac_clustering.py
sxheader.py
sxhelical_demo.py
sxhelicon.py
sxhelicon_utils.py
sxheliconlocal.py
sxhelixboxer.py
sxihrsr.py
sximgstat.py
sxisac.py
sxk_means.py
sxk_means_groups.py
sxk_means_stable.py
sxlocal_ali2d.py
sxlocal_ali3d.py
sxlocal_ali3dm.py
sxlocres.py
sxmontage.py
sxmref_ali2d.py
sxmref_ali3d.py
sxmref_alignment.py
sxmulti_ali2d.py
sxmulti_assign.py
sxparams_2D_to_3D.py
sxparams_3D_to_2D.py
sxpca.py
sxpdb2em.py
sxplot_projs_distrib.py
sxprepare_2d_forPCA.py
sxprocess.py
sxproj_stability.py
sxproject3d.py
sxrealignment.py
sxrecons3d_f.py
sxrecons3d_n.py
sxreproducibility.py
sxresample.py
sxrot_sym.py
sxshiftali.py
sxspliteigen.py
sxssnr3d.py
sxstability.py
sxtransform2d.py
sxvar.py
sxvarimax.py
sxviper.py
sxwindow.py

Appendix D

Scripts

While only six scripts are listed in this appendix, a much larger number was used to execute the different experiments. However, most scripts are just variations of *e2refine2d.pbs*, *e2refine_easy* and *runexperiments.sh*. As such, those are not listed, but it is easy to see where in the scripts the differences lie.
An important thing to note is that none of these scripts are optimal, but rather used as is and copied all over the place.

Listing D.1: *e2refine2d.pbs* (10)

```
#!/bin/bash
#
#PBS -l nodes=10:ppn=4
#PBS -l walltime=3:00:00
#PBS -l vmem=80gb
#PBS -m be
#PBS -m a
#PBS -d /home/nvveen/ws2013/e2refine2d_experiments/10/

NODES=10
echo "number of nodes: $NODES"
echo "number of cpus per node: 4"
echo "starting job at $(date)"
cd /home/nvveen/ws2013/workshop_beijing/

# a default e2refine2d.py run with included parameters for automatization of PBS
# scripts per nr. of nodes each variable is copy-pasted into the scripts.  also,
# output for the program is piped to logfiles, as opposed to the logfiles TORQUE
# generates (i.e., where the echo-commands in this script go to)
e2refine2d.py --input=sets/all__ctf_flip_hp.lst --ncls=24 --normproj --fastseed
  --iter=6 --nbasisfp=12 --naliref=3 --simalign=rotate_translate_flip
  --simaligncmp=ccc --simraligncmp=dot --simcmp=ccc --classkeep=0.85
  --classiter=5 --classalign=rotate_translate_flip --classaligncmp=ccc
  --classraligncmp=ccc --classaverager=mean --classcmp=ccc
  --classnormproc=normalize.edgemean --parallel=mpi:$((NODES*4)):/scratch
  1>../e2refine2d_experiments/$NODES/e2results.out
  2>../e2refine2d_experiments/$NODES/e2results.err


# reset database
e2bdb.py -c
echo "Job finished at $(date)"
```

Listing D.2: *e2refine_easy.pbs* (10)

```
#!/bin/bash
#
#PBS -l nodes=10:ppn=4
#PBS -l walltime=3:00:00
#PBS -l vmem=80gb
#PBS -m be
#PBS -m a
#PBS -d /home/nvveen/ws2013/e2refine_easy_experiments1/10/

NODES=10
RUN=1
echo "number of nodes: $NODES"
echo "number of cpus per node: 4"
echo "starting job at $(date)"
cd /home/nvveen/ws2013/workshop_beijing/

# a default e2refine_easy.py run with included parameters for automatization of
# PBS scripts per nr. of nodes each variable is copy-pasted into the scripts.
# also, output for the program is piped to logfiles, as opposed to the logfiles
# TORQUE generates (i.e., where the echo-commands in this script go to)
e2refine_easy.py --input=sets/all__ctf_flip_hp.lst
  --model=initial_models/model_01_01.hdf --targetres=15.0 --speed=5 --sym=d7
  --iter=3 --mass=800.0 --apix=2.1 --classkeep=0.9 --m3dkeep=0.8
  --parallel=mpi:40:/scratch --threads=8
  1>../e2refine_easy_experiments$RUN/$NODES/e2results.out
  2>../e2refine_easy_experiments$RUN/$NODES/e2results.err

# reset database
e2bdb.py -c
echo "Job finished at $(date)"
```

Listing D.3: *getresults.sh*

```
#!/bin/bash

# a convoluted script that scrolls through the output files given as
# parametersand extracts run times

output=$(find ${*:3} | grep -P 'pbs.o[0-9]+$' | while read FILE; do
  # iterate through all pbs output files and grep lines in date format
  start=$(grep starting $FILE | awk '{print $4" "$5" "$6" "$7}');
  end=$(grep finished $FILE | awk '{print $4" "$5" "$6" "$7}');
  if [[ "$end" == "" ]]; then
    continue;
  fi
  diff=$(($(date -d "$end" +%s) - $(date -d "$start" +%s)));
  printf "%d,%d\n" \
    $(grep "number of nodes" $FILE | awk '{print $4}') $diff;
  # output everything to a variable
done)

# again, iterate over output variable from min to max
# print for each run the amount of seconds in CSV format
min=$1;
max=$2;
if [[ "$max" -gt 10 ]]; then
  max=10;
fi
for i in $(seq $min $max); do
  printf "$i,";
  echo "$output" | grep "^$i" | cut -d',' -f2- | while read LINE; do
    printf "%d," $LINE
  done
  printf "\n";
done
if [[ $2 -gt 10 ]]; then
  for i in 12 14 16; do
    printf "$i,";
    echo "$output" | grep "^$i" | cut -d',' -f2- | while read LINE; do
      printf "%d," $LINE
    done
    printf "\n";
  done
fi
```

Listing D.4: *runboth.sh*

```
#!/bin/bash
# a simpler script that just calls e2refine2d.py twice in two different
# locations: one on the fileserver, and one local.
echo "starting job at $(date)" >>
  /home/nvveen/ws2013/fs_experiment/e2refine2d.sh.out
cd /home/nvveen/ws2013/workshop_beijing/

e2refine2d.py --input=sets/all__ctf_flip_hp.lst --ncls=24 --normproj --fastseed
  --iter=6 --nbasisfp=12 --naliref=3 --simalign=rotate_translate_flip
  --simaligncmp=ccc --simraligncmp=dot --simcmp=ccc --classkeep=0.85
  --classiter=5 --classalign=rotate_translate_flip --classaligncmp=ccc
  --classraligncmp=ccc --classaverager=mean --classcmp=ccc
  --classnormproc=normalize.edgemean --parallel=thread:8
  1>/home/nvveen/ws2013/fs_experiment/e2results.out
  2>/home/nvveen/ws2013/fs_experiment/e2results.err &
pid=$!

while sleep 20; do
  kill -0 $pid
  if [[ $? -eq 0 ]]; then
    echo "still running";
  else
    break;
  fi;
done


e2bdb.py -c
echo "Job finished at $(date)" >>
  /home/nvveen/ws2013/fs_experiment/e2refine2d.sh.out

echo "starting job at $(date)" >>
  /scratch/workshop_beijing/experiment/e2refine2d.sh.out
cd /scratch/workshop_beijing/

e2refine2d.py --input=sets/all__ctf_flip_hp.lst --ncls=24 --normproj --fastseed
  --iter=6 --nbasisfp=12 --naliref=3 --simalign=rotate_translate_flip
  --simaligncmp=ccc --simraligncmp=dot --simcmp=ccc --classkeep=0.85
  --classiter=5 --classalign=rotate_translate_flip --classaligncmp=ccc
  --classraligncmp=ccc --classaverager=mean --classcmp=ccc
  --classnormproc=normalize.edgemean --parallel=thread:8
  1>/scratch/workshop_beijing/experiment/e2results.out
  2>/scratch/workshop_beijing/experiment/e2results.err &
pid=$!

while sleep 20; do
  kill -0 $pid
  if [[ $? -eq 0 ]]; then
    echo "still running";
  else
    break;
  fi;
done


e2bdb.py -c
echo "Job finished at $(date)" >>
  /scratch/workshop_beijing/experiment/e2refine2d.sh.out
```

Listing D.5: *runexperiments.sh*

```
#!/bin/bash

# this script is generated 5 or more times, including the directory structure,
# for the purpose of executing one run of 10 experiments for 10 nodes.

# iterate over number of experiments
for i in $(seq 2 10); do
  cd /home/nvveen/ws2013/e2refine2d_experiments1/$i;
  # reset database access
  e2bdb.py -c
  echo "starting job $i";
  # execute job and retrieve job id
  JOB=$(qsub e2refine2d.pbs)
  JOBID=${JOB%%.*}
  echo "Job id: $JOBID";
  # keep polling for job status as Running
  STATUS=$(showq | grep $JOBID | awk '{print $3}')
  while sleep 20; do
    STATUS=$(showq | grep $JOBID | awk '{print $3}')
    if [[ "$STATUS" != "Running" ]]; then
      echo "stopped running";
      break;
    fi
    echo "still running";
  done
  echo "finished job $i";
done
```

Listing D.6: *updatescreen.sh*

```
#!/bin/bash

# a simple script that keeps looping and printing something while the job is
# running so the SSH connection doesn't time out. This only happened on a few
# terminals, but jobs were long enough to want to keep the same terminal alive
JOB_STATUS=$(showq | grep "$1" | awk '{print $3}')
while [ "$JOB_STATUS" == "Running" ]; do
  echo "$JOB_STATUS";
  sleep 20;
done
```

Bibliography

[1] *Adaptive Computing - Intelligent Workload Management for Cloud and HPC Environments*. URL: http://www.adaptivecomputing.com/.

[2] *Altair: Innovation Intelligence*. URL: http://www.altair.com/.

[3] Albeaus Bayucanand et al. *Portable Batch System Software License*. URL: http://www.mcs.anl.gov/research/projects/openpbs/docs/v2_2_ids.pdf.

[4] Donald J Becker et al. "BEOWULF: A parallel workstation for scientific computation". In: *Proceedings, International Conference on Parallel Processing*. Vol. 95. 1995.

[5] *Debian - The Universal Operating System*. URL: https://www.debian.org/.

[6] *Dell Poweredge 2950 Server*. URL: http://www.dell.com/downloads/global/products/pedge/en/2950_specs.pdf.

[7] *GNU Lesser General Public License*. URL: https://www.gnu.org/licenses/lgpl.html.

[8] Wen Jiang et al. "Backbone structure of the infectious ε 15 virus capsid revealed by electron cryomicroscopy". In: *Nature* 451.7182 (2008), pp. 1130–1134.

[9] *Linux NFS faq*. URL: http://nfs.sourceforge.net/.

[10] Steve Ludtke. *EMAN2 - Wiki*. URL: http://blake.bcm.edu/emanwiki/EMAN2.

[11] Steve Ludtke. *EMAN2/Tutorials*. URL: http://blake.bcm.edu/emanwiki/EMAN2/Tutorials.

[12] *Maui - Adaptive Computing*. URL: http://www.adaptivecomputing.com/products/open-source/maui/.

[13] Haavard Nord and Eirik Chambe-Eng. *QT Project*. URL: http://www.qt-project.org.

[14] *Open Grid Scheduler*. URL: http://gridscheduler.sourceforge.net/.

[15] *OpenLDAP, Main Page*. URL: http://www.openldap.org/.

[16] *OpenPBS Public Home*. URL: http://www.mcs.anl.gov/research/projects/openpbs/.

[17] *Oracle — Hardware and Software, Engineered to Work Together*. URL: http://www.oracle.com/index.html.

[18] *PBS Professional*. URL: http://www.pbsworks.com/.

[19] *Son of Grid Engine*. URL: https://arc.liv.ac.uk/trac/SGE.

[20] *SOURCEFOURGE: Maui Scheduler*. URL: http://mauischeduler.sourceforge.net/.

[21] Guang Tang et al. "EMAN2: an extensible image processing suite for electron microscopy". In: *Journal of structural biology* 157.1 (2007), pp. 38–46.

[22] *Texas Advanced Computing Center*. URL: https://www.tacc.utexas.edu/home.

[23] *Torque Download - Adaptive Computing*. URL: http://www.adaptivecomputing.com/support/download-center/torque-download/.

[24] *Version EMAN 2.1 alpha-1 NCMI - Baylor College of Medicine - Houston, TX*. URL: http://ncmi.bcm.edu/ncmi/software/counter_222/software_123.

[25] AndyB. Yoo, MorrisA. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". English. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Vol. 2862. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-20405-3. DOI: 10.1007/10968987_3. URL: http://dx.doi.org/10.1007/10968987_3.

[26] Xuekui Yu, Lei Jin, and Z Hong Zhou. "3.88 Å structure of cytoplasmic polyhedrosis virus by cryo-electron microscopy". In: *Nature* 453.7193 (2008), pp. 415–419.

[27] Xing Zhang et al. "Near-atomic resolution using electron cryomicroscopy and single-particle reconstruction". In: *Proceedings of the National Academy of Sciences* 105.6 (2008), pp. 1867–1872.

[28] Z Hong Zhou. "Towards atomic resolution structural determination by single-particle cryo-electron microscopy". In: *Current opinion in structural biology* 18.2 (2008), pp. 218–228.