Using Maali to Efficiently Recompile Software Post-CLE Updates on a CRAY XC System

R. Christopher Bording, Christopher Harris and David Schibeci
Pawsey Supercomputing Centre
26 Dick Perry Avenue
Kensington, WA , Australia
Email: chris.bording\chris.harris\david.schibeci@pawsey.org.au

***Abstract*—One of the main operational challenges of HPC centres is maintaining numerous scientific applications in order to support a large and diverse user community. At the Pawsey Supercomputing Centre we have developed Maali, a lightweight automated system for managing a diverse set of optimised scientific libraries and applications on our HPC resources. Maali is a set of BASH scripts that reads a template file that contains the all the information to necessary to download a specific version of an application or library, configure and compile it. This paper will present how we recently used Maali after the latest CLE update and the hardware changes of Magnus, a Cray XC40, to recompile a large portion of our scientific software stack. This includes the changes to Maali that were needed for both the CLE and hardware updates to differentiate between Magnus and Galaxy, our Cray XC30 system.**

***Keywords*-software stack management; automated build systems**

I. Introduction

One of the main operational challenges of High Performance Computing centres is maintaining numerous scientific applications to support a large and diverse user community. At the Pawsey Supercomputing Centre (previously known as iVEC) we recognised the need to have a systematic approach to maintaining the application software stack. This has led to the development of Maali[1], is a light-weight automated system for managing a diverse set of reproducible, optimised scientific libraries and applications on our HPC resources.

Initially, Maali was known as the iVEC Build System and was developed in 2012 to support our commodity cluster systems Epic and Fornax. From an operational point of view these systems were nearly identical, both systems had similar hardware (Qlogic Infiniband and Intel Xeon CPUs) although Fornax also has NVIDIA GPUs. Prior to Fornax being put into production the decision was made to ensure that the Operating Systems, Compilers and MPI libraries would stay in lock step with Epic. To achieve this it would also require updating the hand-built software stack on Epic that had been in production for over a year. We saw this as a unique opportunity for a project to allow us to use Fornax as a test bed to try and develope a more automated software provisioning system. The initial work on Fornax allowed us to understand and refine the software installation process. Then using the iVEC Build System we were able to rebuild the entire software stack on both Epic and Fornax.

With the iVEC Build System we have been able to expand the extensive library of software packages that we can support. With further refinements we have been able to continuously improve on what has now become known as Maali. The development of Maali has also been done with an eye towards the procurement of our Peta-scale system: we understood that, regardless of what system was selected, it would be subject to vendor-specific system updates for example, updates to the Cray Linux Environment (CLE), in the case of a Cray system. These updates can involve major changes, such as changing the default compiler versions or changes to the MPI libraries, for which we need to have a systematic approach to quickly and reliably reproduce our scientific application software stack.

Maali at its core is a set of BASH scripts that are designed to allow for the automation of the Autoconf process of *configure, make and make-install* commonly used by many HPC applications. Maali works from a set of system-level configuration files that defines a set of default environment variables. This allows us to control various aspects of the installation; such as the default build directory location, the installation directory, the location for the original package source code to be download to and stored, the location for the log files, and the compilers used to configure and compile an application.

There is a separate Maali build file for each application that, amongst other things, defines which Cray programming environment(s) are to be used. In our case, this is usually a selection from PrgEnv-cray, PrgEnv-intel, and PrgEnv-gnu. A key function of Maali is to capture what dependent environment modules are used, as well as the configuration flags and the compiler optimisations that are selected and used to build the application. Maali additionally determines what is required in the application or library environment module file and automatically generates a new environment module

[1]*Maali* is the Noongar word for the **Black Swans** that can be regularly seen on the Swan River here in Perth. The Noongar are the indigenous Australian people who have lived in South-West Australia for 45,000 years.

file. Maali also captures log files from the configuration and compilation outputs useful as reference and for debugging. More recently, Maali has been modified to handle CMake and Python builds.

This paper will highlight the main benefits and challenges we observed when using Maali to rebuild our application-software stack following an upgrade of our Magnus system hardware from XC30 (*Intel SandyBridge*) to XC40 (*Intel Haswell*) and the associated major update to the CLE environment (for Haswell support). Maali allowed us to recompile a large portion of our scientific software stack as we transitioned Magnus from early-adopters into production. A detailed overview will be presented of the enhancements that have been made to Maali as a result of our experience from completing the Magnus upgrade, plus the need to differentiate between Magnus and a separate Cray XC30 system named Galaxy.

II. Development Background

The Pawsey Supercomputing Centre has grown in terms of compute capacity, from approximately 6 TFLOPS in 2010 to having approximately 1.5 PetaFLOPS capacity as of the beginning of 2015, nearly a 250X increase in compute.

Year	Name	System	TFLOPs
2010	Cognac	SGI Altix 3700	4
2010	XE	Cray XT3	2
2011	Epic	HP	115
2012	Fornax	SGI	100
2013	Zythos	SGI UV2000	20
2013	Zeus	SGI cluster	30
2014	Galaxy	Cray XC30	200
2014	Magnus	Cray XC40	1100

Figure 1. History of Pawsey compute resources. The first block of systems have since been decommissioned, while the latter block of systems are still in operation.

In addition to this growth in computational processing capability, there has been an increase in the number of research projects and researchers. In turn, this has created a need to support more software applications and libraries for the researchers. Fortunately, we have also been able to increase the staffing levels on the Operations and Supercomputing teams to help meet the researcher needs and the Centre obligations.

The increase in staffing allowed for the work of supporting software applications and libraries to be shared across the Operations and Supercomputing teams, however it requires greater organisation and the establishing conventions and policies that support those conventions. Unfortunately these policies and conventions where not in place from prior to Epic going into production. The initial software stack that was originally installed on Epic, although meeting the researchers requirements, had several major shortcomings:

1) Compiled by hand for each compiler and MPI library combination.
2) Lack of documentation on how software was compiled.
3) Environment modules in a flat directory.
4) None of the work was backed up.

It was clear a set of policies and conventions were needed to make the staff more efficient and organised in order to deal with multiple systems. In August 2012 with the Fornax cluster being prepared for production to the research community, and we realised that we had an opportunity to run a test project on Fornax, as the entire software stack was to be rebuilt in order to update and synchronise the operating systems, Lustre, compilers and MPI of the Epic[2] and Fornax[3] clusters to improve the ease of operation and maintenance of the two systems from different vendors.

Epic	Fornax
HP POD cluster	SGI Linux cluster
800 nodes	100 nodes
Intel Xeon X5660 (x2)	Intel Xeon X5660 (x2)
24 Gbyte memory	72 Gbyte memory
QDR Infiniband	dual-rail QDR Infiniband
CentOS6.3	CentOS6.3
lustre 1.8.8	lustre 1.8.8
	7 TB local disk
	NVIDIA Tesla C2075 GPU

Figure 2. Epic and Fornax System Configuration.

As a team we realised that we needed a better way of supporting maintaining an increasing number of software packages with ever increase dependencies. The test project was implemented that used templates to help standardise compiling and installing software. This allowed us to create a initial repository of applications build scripts, source files and a set of build logs. However, this approach had two severe short-comings. Firstly, we still needed to create the environment module file for each application and we had to manually run the build scripts for each compiler and MPI library combination that we supported on Fornax. The results of this initial test project on Fornax led directly to the development of the iVEC Build System (iBS) by David Schibeci. Subsequently, iBS has undergone numerous improvements and added functionality through its use at iVEC and the Pawsey Supercomputing Centre, to reach its present form as Maali.

III. Automation and Modules

Maali is a lightweight tool consisting of a set of BASH scripts. For the purpose of this paper we will define the individual who uses or writes the build scripts as the

[2]Epic was decommissioned December 2014

[3]Fornax is scheduled to be decommissioned in July 2015.

Maali_builder. The Maali tool attempts to automate the following steps:

1) Download the software
2) Unpack the software
3) Compile the software (configure, make and make install)
4) Create a environment module file.
5) Document the procedure

A. System Level Configuration Requirements

With the arrival of the Cray XC30s to the Pawsey Supercomputing Centre we realised that the iBS would need modifying as we would need to be able define additional requirements but at a high level so we could maintain the generic build scripts. To provide the flexibility required to handle multiple systems Maali uses a system configuration script that defines the key-pair values shown in Figure 3.

Of the fifteen different keypair values, thirteen are used with the Cray systems as MAALI_DEFAULT_MPI and MAALI_DEFAULT_INTEL_MKL are not required. The MAALI_ROOT keypair variable is set to where you want to install based on what the local policy and naming conventions are. As you see in *magnus.config*4 file or as the MAALI_CONFIG file that we use at the Pawsey Supercomputing Center the follow key-pair values 4 with the legacy name *ivec* name as the root directory. Where the build, src and modulefiles subdirectories.

B. Maali build script

The Maali build script provide the definitions for the four of the five basic tasks we are attempting to automate. We will briefly describe what Maali does for each of these basic tasks. Beginning with downloading the software and unpacking the software. Followed by the configure, make and make install of the software package. Lastly we will go over creating the environment modules files .

1) Download and unpacking the software: Maali uses GNU Wget [1] as it provides a non-interactive method for downloading packages from FTP and HTTP servers. The build script sets the MAALI_URL key-pair variable to the correct URL for the application to be installed. To download zlib the full URL would be:

MAALI_URL="http://sourceforge.net/projects/libpng/files/
$MAALI_TOOL_NAME/$MAALI_TOOL_VERSION/
$MAALI_TOOL_NAME-$MAALI_TOOL_VERSION.tar.gz/download".

This brings up an import point when defining the MAALI_URL is the dot extensions (i.e. .tar.gz) and that is that the package maintainers often change their compression tools version to version. The Maali command script is able reconcile a majority of the compression formats (e.g. .bz2, .tgz, .xz), but the extension are "hard-coded' in the build scripts. The packages are downloaded to the MAALI_DST (destination) path for which the root directory is $MAALI_SRC as defined in the system $MAALI_CONFIG file. The full MAALI_DST variable then becomes $MAALI_SRC/$MAALI_TOOL_NAME/$MAALI_TOOL_VERSION continuing with the zlib example MAALI_DST= $MAALI_SRC/$MAALI_TOOL_NAME-$MAALI_TOOL_VERSION.tar.gz. Maali automatically checks if the application is already in the MAALI_DST directory and will skip the download if it is. Storing the source packages is a good practice for a couple of reasons, Maali recompiles an application form source every time. It forces a clean install by deleting any previous builds in the MAALI_BUILD_DIR, then copies and uncompress the

MAALI_OS
: Operating system version.

MAALI_SYSTEM
: Name of system on which the software is being installed.

MAALI_ROOT
: Directory under which everything else sits "/'*path to Maali* '/$MAALI_OS/$MAALI_SYSTEM"

MAALI_BUILD_DIR
: The is the build directory, a subdirectory of the MAALI_ROOT.

MAALI_MODULE_DIR
: Location where the environment modules are installed, a subdirectory of the MAALI_ROOT.

MAALI_SRC
: Where the software source packages is cached, a subdirectory of the MAALI_ROOT.

MAALI_SYSTEM_BUILD
: A *yes* indicates that this file is for systems builds and will generate a wiki page.

MAALI_DEFAULT_COMPILERS
: System compilers that should be used by default when compiling software. $ie.PrgEnv-cray, PrgEnv-intel, PrgEnv-gcc$

MAALI_DEFAULT_GCC_COMPILERS
: PrgEnv-gnu

MAALI_DEFAULT_INTEL_COMPILERS
: PrgEnv-intel

MAALI_DEFAULT_CRAY_COMPILERS
: PrgEnv-cray

MAALI_DEFAULT_MPI
: Not used with cray

MAALI_DEFAULT_PYTHON
: Default Python to use

MAALI_DEFAULT_INTEL_MKL
: Not used with cray

MAALI_EXTRA_CRAY
: craype-haswell, craype-sandybridge, craype-ivybridge

Figure 3. Maali system level configuration keypair variables.

```
MAALI_OS=cle52
MAALI_SYSTEM=magnus
MAALI_ROOT="/ivec/$MAALI_OS/$MAALI_SYSTEM"
MAALI_BUILD_DIR="$MAALI_ROOT/build"
MAALI_MODULE_DIR="$MAALI_ROOT/modulefiles"
MAALI_SRC="$MAALI_ROOT/src"
MAALI_SYSTEM_BUILD="YES"
MAALI_DEFAULT_COMPILERS="PrgEnv-gnu/5.2.25
  PrgEnv-intel/5.2.25 PrgEnv-cray/5.2.25"
MAALI_DEFAULT_GCC_COMPILERS="PrgEnv-gnu/5.2.25"
MAALI_DEFAULT_INTEL_COMPILERS="PrgEnv-intel/5.2.25"
MAALI_DEFAULT_PYTHON="python/2.6.8"
MAALI_EXTRA_CRAY="craype-haswell"
```

Figure 4. Pawsey Supercomputing Centre system configuration directories.

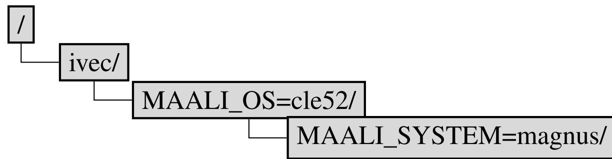

Figure 5. The basic MAALI_ROOT file tree structure

package source in the MAALI_BUILD_DIR. Maintaining a repository of the original source helps to maintain the researchers workflow provenance; allows the Maali_builder to speed up the process of reinstalling a package as needed when optimising the application.

2) Compilation and Installation process: The Maali copies the 'tar-ball' file from the MAALI_DST to the MAALI_BUILD_DIR where it uncompressed and extracted. The heart of the of Maali is that it automates the "configure, make and 'make install"' process from the GNU Autotools framework whose components are GNU Autoconf [3], GNU Automake [4] and GNU Libtool [5]. The purpose of these frame tools is to make it easy for users to compile the packages and make the software more portable. Using templates Autoconf is able to generate the configure file and Automake creates the Makefile.in template. In the simplest of terms running the configure command uses the Makefile.in template to create the Makefile and libtool to help make things more portable. Again Maali attempts to provide a means to automate the results of the Autotool process of running *configure*, *make* and *make install*.

The configure command accepts a large number of common command arguments i.e.–prefix='path to install directory', which allows for installation of the package out of source, Maali tries to leverage that by abstracting those common details away. The remaining configuration flags are typically used to select different features, libraries-paths and code optimisation that are system dependent. They are defined in Maali using the MAALI_TOOL_CONFIGURE key-pair value for the data format library here is the HDF5 configuration, e.g.

MAALI_TOOL_CONFIGURE='–enable-silex –enable-fortran –enable-shared –with-hdf5=$HDF5_DIR/include,$HDF5_DIR/lib'

The *make* and *make install* are remain relatively unmodified it is possible however to add in the MAALI_CONFIG file MAALI_CORES which can be used to compiles an application in parallel using "make -j$MAKE_CORES". That however is application and library dependent and may not be suitable for using on the login nodes on systems. In keeping with what was the original idea of Maali is to increase the automation of installing software, in the MAALI_CONFIG file we define the MAALI_DEFAULT_COMPILERS, which is a simple list. Maali will iterate over this list and build each version without any other effort of the "Maali_builder". So for the simple codes that use the GNU Autotools and do not have any complex dependency the "Maali_builder" can compile on *magnus* three versions from a single command. These are installed in the "apps" directory, a subdirectory of the MAALI_ROOT as shown in figure 6. We use the compiler names defined in the MAALI_CONFIG file MAALI_DEFAULT_COMPILERS and the system environment variable CRAYOS_VERSION is set in the PrgEnv-* modules. To define a unique directory so all libraries and applications compiled with the respective programming environment are in that directory.

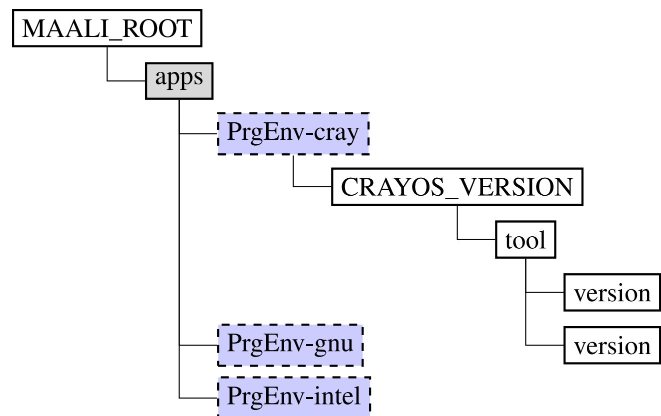

Figure 6. The applications tree structure

3) Environment modules: is used on the Cray XC systems it enforces a convention on the users that they must select the programming environment. Which mirrors the convention we have enforced on our other HPC resources. This allows for setting a up a simple tree structure for the installation path of software on the Pawsey Supercomputing Centre HPC resources as shown in Figure 6. We have only one module file per application regardless of the number of different compilers. Environment module files are templates used to dynamically modify the environment adding or removing packages from the PATH, LD_LIBRARY_PATH and other system variables. There is a manageable number of Maali key-pair values needed for most application or libraries to set the applications environment correctly.

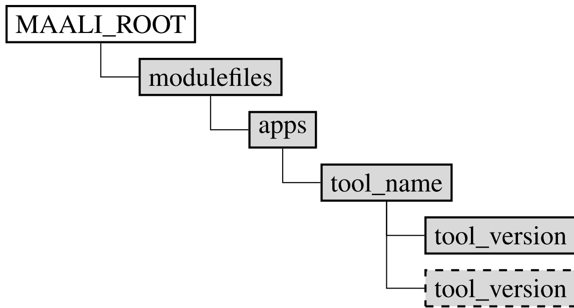

Figure 7. The modules tree structure

```
## These are boolean options set to 1 to add to module
MAALI_MODULE_SET_PATH=1
#MAALI_MODULE_SET_PKG_CONFIG_PATH=
#MAALI_MODULE_SET_LD_LIBRARY_PATH=
#MAALI_MODULE_SET_MANPATH=
#MAALI_MODULE_SET_INCLUDE_PATH=

#MAALI_MODULE_SET_CPLUS_INCLUDE_PATH=
#MAALI_MODULE_SET_C_INCLUDE_PATH=
#MAALI_MODULE_SET_LD_PRELOAD=
#MAALI_MODULE_SET_CPATH=
#MAALI_MODULE_SET_FPATH=
#MAALI_MODULE_SET_FCPATH=

#MAALI_MODULE_SET_PERLLIB=
# for Cray system modules
#MAALI_MODULE_SET_CRAY_LIB_LIBRARY_PATH=
#MAALI_MODULE_PE_PKGCONFIG_NAME=
#MAALI_MODULE_PE_PKGCONFIG_CFLAGS=
#MAALI_MODULE_PE_PKGCONFIG_LIBS=

# To create exported variable in the module files
#
#MAALI_MODULE_SET_SETENV="path or license server= "

# Descriptions of what the package/tool is.
#MAALI_MODULE_WHATIS=" "
```

Figure 8. Maali keypair values to create a modulefile.

C. *The Maali Command Script*

The **Maali** script is a lightweight set of bash script that reads a Maali build script and attempts to manage the build process. The Maali command expects a minimum of three arguments the first one is the name of the package you wish to install the second argument is the version number of that application followed by the name of the system configuration file.“ maali -t MAALI_TOOL_NAME -v MAALI_TOOL_VERSION -c MAALI_CONFIG” for example, to install zlib-1.2.7 on the Pawsey Supercomputing Centre’s Cray XC40 system *Magnus* the command would be:

maali -t zlib -v 1.2.7 -c magnus

Below is a complete listing of the Maali command options.

The list of Maali command options in figure9 shows that what the “Maali_builder” is able to do; re-run the Maali command in a debug mode that produces verbose text to standard out; re-run Maali but have it only update the module file; and/or build with a specified compiler/programming environment module.

-h	show this message
-t	tool name
-v	tool version
-c	maali configuration file
-d	run in debug mode
-m	just create the module file
-l	use the options from the last run
-r	build with the specified compiler (and no others)
-b	build the tool, but don’t create a module file

Figure 9. Maali Command Options

IV. Maali Use

The previous section provided some details of the internals of Maali to give insight on it’s workings. The Operations Team and Supercomputing Team at the Pawsey use Maali regularly to manage building software on the different systems. We quickly realised having a large number of people with root access was not practical so a new user account was created *stapops* that had limited sudo access.

However this is intended for software applications that used by multiple groups and the MAALI_CONFIG file is located in the *stapops* home directory. So that only build scripts that have been tested are then installed as *stapops* we can then add those new Maali build scripts to the Maali git repository.

While the operations and supercomputing teams have grown in numbers we are still vastly outnumbered by the research community. As such we have a number of packages that are only used by a single researcher or group. It quickly becomes very expensive in terms of staff time to support them. So we are showing these groups how to maintain their own software in their project “group” directories. This limited effort has shown good results and is easier for the Pawsey staff to manage.

In summary, where Maali really excels is at simplifying the build process for many applications that use the GNU Autotools build system. However many applications are still difficult to build but Maali does help speed up the iterative nature of the installation process. The number of packages that build scripts have been created for is now over 380+. The large number of applications is a good indicator of how easy Maali is to use.

V. Haswell Migration - Sprint

As part of the second phase upgrade to Magnus in 2014, the CPU architecture was updated to Haswell. Prior to this update, both Magnus and Galaxy had a ”Sandybridge’ CPU architecture. Consequently, the Pawsey software stack was

shared between both machines to minimise installations. However, with the architecture change to Magnus, it was necessary to split the software stack. In order to take full advantage of the new instructions offered by Haswell, the entire software stack on Magnus would need to be reinstalled. As the Cray Linux Environment was also changing on both Magnus and Galaxy to version 5.2, the Galaxy stack was also reinstalled. For the purpose of upgrade a list of "must install" software was constructed. That list consisted of 30 packages plus iBS.

- python-2.6.8
- exabayes1.4.1
- qbox-1.60.0
- gts120708
- gromacs-5.0.2
- mrbayes-3.2.2
- intel-mkl
- gsl-1.15l
- xerces-c3.1.1
- glib-2.42.0
- zlib-1.2.7
- scons-2.2.0
- scipy-0.11.0
- distribute-0.6.49
- d2to1-0.2.11p
- siesta3.2
- gamess-20130501
- ncview-2.1.2
- hypre-2.9.0b
- amber14
- beagle-lib-2.1.2
- cmake2.8.12.1
- lapack3.4.2
- libffi-3.1
- udunits2.2.17
- szip-2.1
- numpy-1.6.2
- mecurial-3.1.1
- astropy-0.4.1
- ephem-3.7.5.3

Figure 10. List of applications for the Magnus Haswell sprint.

Without an automated build system, each package would have required manual configuration and compilation, including customised compilation flags and patches that would have been made with reference to previous installation notes. However, by using Maali and the scripts from the previous installations, the majority of packages simply required a one line change updating the Cray architecture module dependency to Haswell on Magnus and running a single Maali installation command. It also was necessary to implement temporary module paths for team members to allow for dependencies without interfering with the old software stack that were in use at the time.

An installation sprint was organised for Haswell and attended by seven members of the Pawsey supercomputing team. A wiki page was used to keep track of the packages to be installed, which were marked first as in progress and then complete by team members. For most of the packages, the automated installation had a wall time from a few minutes to one to two hours, depending on the complexity of the package. Some team members used the time waiting on installation to start on other packages, or to work on updating the scripts that needed changes to work with Haswell.

The majority of the packages were installed on both systems in a couple of hours, with one or two of the longer installs lasting into the afternoon. The summary of the installation was uploaded to the Pawsey documentation portal after each installation. Following the sprint, the module paths were updated to point to the new software stacks at the subsequent scheduled maintenance.

A. *Haswell Sprint Outcome*

The purpose of the Haswell sprint was to re-compile these applications so that they could make use of the new architecture on the Intel Haswell CPUs and resolve any dependency issues from the CLE upgrade. Overall the Haswell sprint on Magnus was very successful in that we were able to install a large number of packages very quickly and to insure Magnus was available to researchers very quickly. There were several issues that were identified post-sprint that should be investigated to try and help improve future upgrades. The build script documentation should contain more details with regard to the package dependencies. A technical review would have also shown that the list of packages in figure **??** could have been split up into four distinct groups Python, Library independent applications, application with simple dependencies and applications with strong dependencies.

Installing applications on the Cray XC is not without challenges. The login nodes on Magnus are not Haswell so we must cross-compile applications on the login nodes so that they will run optimally on the compute nodes and that requires extra attention to do correctly. Having a large number of people trying to compile codes on the login nodes will cause the process to slow down as well. One way to get around this difficulty is to compile on the compute nodes as there many time more compute nodes than login nodes and it allows you to avoid having to cross-compile the applications. This is not always possible when the system is in production, it is possible to create batch jobs that run in the debug nodes. The real hurdle to compiling a new application on the compute nodes in either the workq or the debugq is that they are on an internal network and hard to access. This prevents Maali from being able to download the required source package as needed. To iterate the Haswell sprint was to *re-compile* a set of applications previously installed which meant we should have the source already downloaded we just needed to point the MAALI_SRC to the right directory with the source packages. Then using the compute nodes for re-building/re-compiling the code should be fairly easy. It was apparent that there is an issue with Maali when trying to download applications using *GIT* or *SVN* on the compute nodes as well. The build script needs to modified to take advantage of codes that have already been cloned using GIT or SVN.

VI. MAALI'S FUTURE

Maali demonstrated great flexibility and ease of use during the Haswell sprint. Some the issues identified during the Haswell sprint will be addressed and as well as continued effort to improve Maali. The next milestone in the development of Maali is to make it open source project and build a user community around it. This will require improved user documentation and development of a rigorous testing scheme. Other areas for future work are listed in figure 11.

- module files prereqs and conflicts need more attention.
- NeCTAR research cloud use.
- bioinformatics pipeline construction.
- regression testing of build scripts.
- wiki page creation.

Figure 11. Future Work

VII. CONCLUSION

As with any High Performance Computing system, making changes to a production system involves a certain amount of risk. However, such updates are necessary to maintain the stability, security and performance of the system, and so the risks cannot be avoided, only controlled. In the Pawsey Supercomputing Centre, we are responsible for the largest research supercomputer in the Southern Hemisphere (Magnus) and the supercomputer that provides real-time processing capabilities for both Australian SKA precursor telescopes (Galaxy). Having either system unavailable for an extensive period of times (to reinstall software) carries negative financial, operational and research cost, and must be avoided. Using Maali, we are now able to plan effectively for a CLE upgrade in order to reduce the length of system downtime across a diverse set of Cray XC systems, minimise the staff effort required to reinstall the software and control the impact on researchers and their projects.

ACKNOWLEDGMENT

The authors would like to thank Ashley Chew and Mark O'Shea from the Operations team at the Pawsey Supercomputing Centre for their work during the initial development of Maali, nee iBS. We also would like to the members of the Supercomputing Team Rebecca Hartman-Baker, Daniel Grimwood, Mohsin Shaikh, Brian Skjerven, Kevin Stratford and Charlene Yang at the Pawsey Supercomputing Centre also, Paul Ryan who is seconded from the Commonwealth Scientific and Industrial Research Organisation(CSIRO) to the Supercomputing team for all their efforts in the Haswell sprint success and their effort in the continued development of Maali. Lastly would like to thank George Beckett who is the Head of the Supercomputing Team for his support for the entire Maali project.

REFERENCES

[1] G. Scrivano. (2015, Jan.) Gnu wget 1.16.2. wget.pdf. [Online]. Available: http://www.gnu.org/software/wget/manual/

[2] P. W. O. John L. Furlani. (2011) Environment modules. [Online]. Available: modules.sourceforge.net

[3] (2012, Apr.) Autoconf - creating automatic configuration scripts for version 2.69. autoconf.pdf. [Online]. Available: http://www.gnu.org/software/autoconf/manual/

[4] (2014, Dec.) Gnu automake - for version 1.15. automake.pdf. [Online]. Available: http://www.gnu.org/software/automake/manual/

[5] (2015, Jan.) Gnu libtool - for version 2.4.6. Libtool.pdf. [Online]. Available: http://www.gnu.org/software/libtool/manual/