getexample: A tool for the Resources of Pawsey Supercomputing Centre

Ranganai Mapondera, and Ozlem Erkilic, *Interns at Pawsey Supercomputing Centre*
Email: ranganai.mapondera@student.curtin.edu.au
Email: ozlem.erkilic@student.curtin.edu.au

***Abstract*—The best way to understand how to compute a task based on some resources, constraints and goals on advanced computing systems is to work through sample codes. A lot of expert users browse through the Internet to find these examples that they can slightly modify to achieve the results they expect. However, this can be very difficult for some users who are new to this environment as there are always so many of these examples which are not always what they specifically need. This brings out the challenge of having to modify the example to suit their requirements on a particular computing resource, mainly the High Performance Computing systems (HPC) since the modified examples mostly do not work due to HPC systems having different set ups. Therefore, it is hard to tell what the issue is for inexperienced users. Quite often, the users may not be able to tell if the example is broken or if the example is correct, but maybe the code requires some adjustments. This is a very common issue encountered by the users of Pawsey Supercomputing Centre. For this reason, this paper provides solutions to how to run and submit examples on different resources such as supercomputers of Pawsey Supercomputing Centre through the use of a simple tool *getexample*. The *getexample* is an effective tool that is developed for the supercomputers of Pawsey that are dependent on a command line interface and aims to help their users learn how to run codes mainly parallel programming examples, submit these examples to different operating systems of Pawsey such as Magnus, Zeus and Zythos. Although, the getexample is limited to these HPC resources, it has the potential to be expanded to other resources and interfaces.**

I. Introduction

EVEN though there are many sample codes on search engines for programming purposes, it is usually very rare that these codes work at the first try on different high performance computing systems without modifying them. This is mainly due to each advanced computing system being built up differently from each other to perform specific tasks for the needs of the users. At Pawsey Supercomputing Centre, there are various HPC resources where each of them are set up with slight variations from each other. Therefore, this becomes very challenging for the users, especially the beginners to change the example codes to display a working task on Pawsey's resources as there are many differences between each resource with many constraints. The most common differences between their systems are as follows:

- The operating systems
- The clusters such as homogeneous or heterogeneouos
- The program environments
- The compiler options with varying commands
- The compiler flags and wrappers for varying compiler options
- The module systems and paths
- The library versions
- The personal scratch, group and home directories
- The scheduling policies

For this reason, when the user runs a sample code found from the Internet for each supercomputers (Magnus, Zeus and Zythos), the code fails to compile as expected and hence, does not run as each individual system has specific operating system and commands set up for them. To rectify these problems, some HPC resources provide their users websites with working examples that aim to assist them how to carry out their tasks step by step. However, sometimes the commands in these examples can be outdated due to the updates in compilers and the operating systems. They also can be challenging to follow and perform on the real systems for the users who are not very experienced with these resources. Although, one may find a well-written Bash script to run these systems, the user may not know how to make this script executable by changing the permissions using the chmod commands. Furthermore, a user may use a sample source code such as a basic MPI code which includes some MPI libraries to perform a particular job on these supercomputers but may fail when compiled due to these libraries being no longer valid or upgraded to a different version. Or one may complete their task, but may not know in which filesystem to store their results, for example in scratch or their group because some supercomputers have policies on how long to keep the data on certain filesystems such as the scratch. If they are not moved from that specific file system within a certain amount of time, the results get deleted and thus, the user loses their work. This is a major difficulty for the users, especially researchers and scientist since it takes very long time to collect their data. It is also very crucial that these are stored correctly within these sources.

In order to minimise these problems, the getexample was developed to supply examples which do not require further editing or modification and works when the executable is run. Thus, it aims to teach the users of Pawsey Supercomputing Centre how to run and submit tasks on the supercomputers without encountering issues. The rest of this paper explains how the *getexample* tool works.

II. Getexample Outline

The main aim of the *getexample* is to provide easy access to the examples for the users. For this reason, the *getexample* tool is expected to be have the following design specifications:

```
~>
~> getexample
magnus

 the example directory is /group/interns2016/getexample/magnus_examples on magnus

Download an HPC example:
usage:
   getexample <examplename>

Where <examplename> is the name of the example you want to
download.  This will create a directory using the example name
which you can cd into.  The directory should contain an
executable README file (if one is avaliable) or just submit
the batch file.

For Example:
  getexample helloworld

Possible example names:
MKL_Example          fortranMPI_gnu             helloC_cray       hello_hybrid_c_cray   myParallelApp.c                task_placement_intel
MKL_FFTW             fortranMPI_intel           helloC_gnu        hello_hybrid_c_gnu    mySerialApp.c                  test_run.sh
MKL_benchmark        fortran_helloOpenMP_cray   helloC_intel      hello_hybrid_c_intel  mySerialApp.optrpt             thread_placement_cray
MKL_c_eigenvalues    fortran_helloOpenMP_gnu    helloOmp_crayC    hello_mpi_c_cray      myVectorizedApp.c              thread_placement_gnu
MKL_parallel         fortran_helloOpenMP_intel  helloOmp_gnuC     hello_mpi_c_gnu       partially_occupied_nodes_cray  thread_placement_intel
fortranHybrid_cray   fortran_helloworld_cray    helloOmp_intelC   hello_mpi_c_intel     partially_occupied_nodes_gnu
fortranHybrid_gnu    fortran_helloworld_gnu     hello_NUMA_cray   hello_omp_c_ranga     partially_occupied_nodes_intel
fortranHybrid_intel  fortran_helloworld_intel   hello_NUMA_gnu    lammps_mpi            task_placement_cray
fortranMPI_cray      gromacs_mpi                hello_NUMA_intel  mom_node              task_placement_gnu
```

Fig. 1. The list of examples when typed *getexample* on Magnus

```
> getexample
zeus

 the example directory is /group/interns2016/getexample/zeus_examples on zeus

Download an HPC example:
usage:
   getexample <examplename>

Where <examplename> is the name of the example you want to
download.  This will create a directory using the example name
which you can cd into.  The directory should contain an
executable README file (if one is avaliable) or just submit
the batch file.

For Example:
  getexample helloworld

Possible example names:
fortran_helloOpenMP_gnu    fortran_helloworld_intel  fortranHybrid_pgi  helloC_gnu      hello_hybrid_c_gnu    hello_mpi_c_intel  helloOmpC_pgi
fortran_helloOpenMP_intel  fortran_helloworld_pgi    fortranMPI_gnu     helloC_intel    hello_hybrid_c_intel  hello_mpi_c_pgi
fortran_helloOpenMP_pgi    fortranHybrid_gnu         fortranMPI_intel   helloC_pgi      hello_hybrid_c_pgi    helloOmpC_gnu
fortran_helloworld_gnu     fortranHybrid_intel       fortranMPI_pgi     hello_cuda_gnu  hello_mpi_c_gnu       helloOmpC_intel
```

Fig. 2. The list of examples when typed *getexample* on Zeus

```
~> getexample fortranMPI_intel
magnus

 the example directory is /group/interns2016/getexample/magnus_examples on magnus

`/group/interns2016/getexample/magnus_examples/fortranMPI_intel' -> `./fortranMPI_intel'
`/group/interns2016/getexample/magnus_examples/fortranMPI_intel/README' -> `./fortranMPI_intel/README'
`/group/interns2016/getexample/magnus_examples/fortranMPI_intel/hello_mpi_intel.slurm' -> `./fortranMPI_intel/hello_mpi_intel.slurm'
`/group/interns2016/getexample/magnus_examples/fortranMPI_intel/hello_mpi.f90' -> `./fortranMPI_intel/hello_mpi.f90'
```

Fig. 3. Feedback given to the user while downloading the example

```
~> cd fortranMPI_intel/
~> ls
README  hello_mpi.f90  hello_mpi_intel.slurm
```

Fig. 4. The list of files in the fortranHybrid_intel directory

- It should be easy to use for everyone including the beginners. The user should be able to download the examples and run them with one command.
- It should provide practical examples which can be modified or updated by the users for their own work. Therefore, it encourages the users to learn how to use the resources given to them effectively.
- The examples should be clear and completely detailed with steps to help the users understand and apply it.

The *getexample* tool is designed in a way such that when *getexample* command is typed from any directory on the resources of Pawsey Centre, it lists all the examples within the *getexample* library as shown in Fig. 1. on the previous page. As the *getexample* displays many examples which aim to perform various tasks for each individual resource at Pawsey, it ensures that when the user logins from a specific supercomputer such as Magnus, it only lists the examples provided for Magnus rather than showing all of the examples within the library. To obtain a copy of any example, the user simply types *getexample* followed by the name of the example as shown below:

```
getexample <name of the example>
```

This creates a new directory with same name as the example requested in wherever the *getexample* tool is accessed from and it downloads all the files of the example to the new directory created and gives a feedback to the user about what the command is doing as shown in Fig. 2. The new directory then can be accessed by the user simply typing:

```
cd <name of the example>
```

As mentioned earlier, each example consists of 3 files which are the README and SLURM scripts, and the source code as shown in Fig. 3. on the previous page.

III. Creating Examples for the Getexample Library

The examples included in the *getexample* library were created by the internship students at Pawsey with their supervisor while the source codes within the *getexample* were obtained from some wiki pages and websites which were acknowledged in the examples. Some of the examples were also created due to the needs of research students for their internship projects at Pawsey. The examples obtained in the *getexample* are mainly for teaching the users how to work with parallel programming including MPI, OpenMP and hybrid jobs consisting of OpenMP/MPI which utilise basic source codes similar to *"Hello world"* program and submit these jobs to different supercomputers such as Magnus, Zeus and Zythos using different program environments and compiler modules.

Each individual example on the *getexample* occupies a directory that is reserved for them and each example consists of three files listed below:

- SLURM (Simple Linux Utility for Resource Management): This allows the users to submit batch jobs to the supercomputers, check on their status and cancel them if needed. It contains the necessary information about the name of the executable, the results directory, the name of the output file and the jobID. It also allows the users to edit how many nodes the code requires to run on the HPC systems, the duration of the task that it takes, which partition to be used and their account name. The SLURM initially creates a scracth directory for the example to run in and the results are outputted to a log file. Then, it creates a group directory in which the results directory is located for that example. Once the output file is completed within the scratch, the output file is then moved to the results directory located in the group directory and the scratch directory then gets removed.
- Source Code: This is usually a source code in C or FORTRAN and taken from some wiki pages to run the example.
- README: This file is an executable Bash script which can be read and run by the users. It provides details about what the source code does, how to compile the source code depending on the program environment such as Cray, Intel, GNU or compilers like PGI, what can be modified in the SLURM directives, and a set of instruction on how to submit the SLURM to the supercomputers including which specific commands to use for particular supercomputers. It can be executed by simply typing ./README which then compiles the source code and submits the batch job to the chosen supercomputer.

For the examples in the *getexample* tool to be user friendly, helpful and efficient for working with Pawsey Centre resources, there are many design considerations to be held as listed below:

- Before introducing the examples to the users, it should be ensured that the examples run without encountering any errors. For example, they should be suitable for the current operating systems with the use of correct commands for the different compiler modules such as Intel, GNU, and PGI or different program environments such as Cray.
- All the files to run the example should be included with the example.
- To minimise confusion on how to perform the example on the supercomputers, the example should be executed with a single command. For example, if a simple Bash script is used, it should be run as ./README.
- The examples should have enough instructions on what each command does and which parts of the SLURM and the README file can be modified so that all the files and the scripts should be able to edited easily by the users for their preferences. These instructions should be understandable by the new users who are not very experienced with these systems. For example, if the user wishes to use more nodes on the supercomputers, one should be able to know where to change it from.

Even though, the examples in the getexample tool are designed in a such way that once they are downloaded, the user should be able to run them without having to modify them. However, this is not always the case as some of the

supercomputers in Pawsey Supercomputing Centre such as Zythos due to different set ups, require the account name which is customized for each user and without the correct account name in the SLURM, the code fails to run. Therefore, the examples in the *getexample* tool ensures that the users are informed on how to change their account name located in the SLURM file. Furthermore, they assist the users on how to change number of nodes used within the supercomputers and increase or reduce the number of cores used based on the user's preferences.

A. Magnus

Magnus is described as a Cray X40 supercomputer that consists of many nodes that are bound by a high speed network. For the compute nodes, each one of them has 2 sockets and each of these has 12 cores. Magnus is also specified to have 24 cores per node and in total these sum up to 35,712 cores across the 1488 nodes.

On Magnus, jobs run on the back-end of the system with the help of SLURM and ALPS (the Cray Application Level Placement Scheduler). A batch job is submitted to the queue system on the front-end from the sbatch command. When it runs, it executes the launch command *aprun* on the MOM nodes which are the login nodes of Magnus. The *aprun* keeps running on these login nodes until the application gets completed. Once *aprun* finishes, the SLURM job also completes.

As mentioned previously, the focus of the *getexample* tool was not only to provide working examples to the users, but also assist them on how to run jobs on their scratch directories. Therefore, whenever the Batch job was submitted to Magnus, it was ran on the scracth directory and once the job was completed, the results were carried to the group directory. In the end, the scracth was removed. The examples used for Magnus were mainly parallel programming examples such as MPI, OpenMP and hybrid codes which is a combination of OpenMP and MPI tasks and some applications such as LAMMPS and GROMACS. The source codes used for these examples were written in C or FORTRAN which were basic *"Hello world"* codes with the exception of the application codes. Each example was displayed in different environments on Magnus such as GNU, Intel and more importantly the default environment, Cray with the compiler options of *ftn, mpif90* and *cc*. When using these environments, it was important to load the actual programming environment before compiling the source codes and running them as each environment has specific compiler commands with distinct wrappers.

1) MPI Examples: MPI is known as a Message Passing Interface application that is a communication model for moving data between processors. For MPI examples, FORTRAN and C codes were mainly used and ran on different environments including Cray, GNU and Intel. On Magnus, the default program environment is Cray. Therefore, it was necessary to load each program environment that was intended to work on by doing a:

```
module swap PrgEnv-cray PrgEnv-<name of the environment>
```

It is also very important to notice that if the right program environment is not loaded, the code will fail as each environment have different commands for compilers. However, on Magnus, swapping from one environment to the other may be very challenging for the new users. As the *getexample* tool is desired to be as automated as possible to minimise failures on tasks, it was made sure that after running each job on the Intel and GNU environments, the environment was set back to the default environment, Cray. This prevents the users to manually list the module every time they run something on Magnus and modify the codes to swap from one module to the other.

The very first MPI example performed on Magnus ran on 2 nodes with a total of 48 tasks with a basic *"Hello world"* source code in both C and FORTRAN. This example consisted of three files as mentioned previously which were README, SLURM and the source code. The SLURM script included information about the choice in the number of nodes, the partition, the duration of the time it takes to run the job, where to run the task such as on scratch and where to store the results, for example the group. It also specified the name of the executable, the results directory and the output file.

For the *getexample* on Magnus, the classified partition was the *debugq*. Since, there were 2 nodes used, the SLURM directives were given as:

```
#!/bin/bash -l
#SBATCH --job-name=GE-hostname
#SBATCH --partition=debugq
#SBATCH --nodes=2
#SBATCH --time=00:05:00
#SBATCH --export=NONE
```

The generic variables such as the executable, scratch, the results directory and the output file were defined as shown below:

```
EXECUTABLE=hello_mpi_cray
SCRATCH=$MYSCRATCH/run_hostname/$SLURM_JOBID
RESULTS=$MYGROUP/mpifortran_cray_results/$SLURM_JOBID

OUTPUT=mpifortran_cray.log
```

To launch the MPI job with fully occupied 2 nodes, the *aprun* command was used as expressed:

```
aprun -n 48 -N 24 ./$EXECUTABLE >> ${OUTPUT}
```

where -n defines the total number of MPI tasks while -N specifies the number of MPI tasks per node.

In the README file, the correct program environment should be included for the codes to run correctly. For compiling on Cray, there was no need to load this environment as the default module was Cray. However, to run on GNU and Intel, the program environment was changed from Cray to GNU or Intel as shown below:

```
module swap PrgEnv-cray PrgEnv-gnu
module swap PrgEnv-cray PrgEnv-intel
```

However, the compiler commands do not change for Cray, GNU and Intel for MPI codes on Magnus, whereas they have

distinct differences for other parallel programming tasks such as OpenMP.

To compile the *hello_mpi.f90* on Cray, GNU and Intel environments:

```
ftn -O2 hello_mpi.f90 -o hello_mpi_cray
```

-O2 is the optimization method, while *-o* placed after the source code directs to an executable called *hello_mpi_cray* which was previously defined in the SLURM and chosen by the technical staff as a name. For GNU and Intel, this was called *hello_mpi_gnu* and emphhello_mpi_intel.

When the source code was in C, the SLURM was not different from that of FORTRAN. However, the README did change. To compile the *hello_mpi.c code*, the following command was used:

```
cc -O2 hello_mpi.c -o hello_mpi_cray
```

Once the codes were compiled, the SLURM was then submitted to Magnus by:

```
sbatch hello_mpi_cray.slurm
```

As mentioned earlier, to prevent the users from manually listing the modules to see which modules are loaded and which module to swap from, at the end of the SLURM and the README, the program environment was always set back to the default by:

```
module swap PrgEnv-<name of the environment> PrgEnv-cray
```

Another example on MPI was to run the same sources but on partially occupied nodes which means that instead of having 24 tasks per node, each node could only have 12 tasks. When running this example on Magnus, the README remained unchanged, but there were minor changes to the SLURM.

The number of OpenMP threads was set to 1 to prevent from inadvertent OpenMP threading and the *aprun* command was changed to:

```
aprun -n 24 -N 12 -S 6 ./$EXECUTABLE >> ${OUTPUT}
```

where -n defines the total number of MPI tasks, -N specifies the number tasks per node while -S defines the number of MPI tasks per socket.

2) OMP Examples: OpenMP (Open Multi-Processing) is known as a common shared memory model in which the threads work in parallel and access all shared memory. Therefore, it can be thought of one task because of their restrictions to only one node. Unlike the MPI examples, the compiler commands and the wrappers do change for the different environments on Magnus with the OpenMP tasks. Thus, it is necessary to swap to the right program environment corresponding to the compiler in use.

As mentioned previously, the OpenMP jobs use only one node and this node can be fully occupied node or a node occupying a single NUMA region. When it was fully occupied, the SLURM script had a choice of only one node with the *debugq* partition as explained previously.

```
#SBATCH --partition=debugq
#SBATCH --nodes=1
#SBATCH --time=00:05:00
#SBATCH --export=NONE
```

The initiation of the job by *aprun* was done by specifying that there was one task and 24 threads. The number of threads is also called a ”depth” and hence, the number of threads, *OMP_NUM_THREADS*, was set to 24 giving an expression of:

```
export OMP_NUM_THREADS=24
aprun -n 1 -d 24 ./$EXECUTABLE >> ${OUTPUT}
```

For a node occupying a single NUMA region, the code is more efficient when threads are limited to one NUMA region containing 12 cores. The expression -d 12 ensures that threads are bound correctly and this can be clearly done by specifying this via -cc which specifies the cores used. This is all illustrated below as:

```
export OMP_NUM_THREADS=12
aprun -n 1 -d 12 -cc 0-11 ./$EXECUTABLE >> ${OUTPUT}
```

The source codes used for OpenMP jobs were *omp_hello.f* and *omp_hello.c*. To run these source codes respectively on Cray, the README included the following compiling commands:

```
ftn -O2 -h omp omp_hello.f -o hello_omp_cray
cc -O2 -h omp omp_hello.c -o hello_omp_cray
```

Then, the job was also submitted to Magnus by using the same sbatch command in the README file of the MPI examples except replacing the name of the SLURM with the correct name for the OpenMP task.

To compile with the GNU environment, the compiler for FORTRAN and C codes respectively changed to:

```
ftn -O2 -fopenmp omp_hello.f -o omp_hello_gnu
cc -O2 -fopenmp omp_hello.c -o omp_hello_gnu
```

To run the OpenMP code correctly on Cray with the Intel environment, the affinity should be disabled. If it is not disabled, the code runs on only one thread rather than running on multiple threads and this was done by adding the following command before the *aprun:*

```
export KMP=AFFINITY=disabled
```

For the Intel environment, the compilers used are expressed as below:

```
ftn -O2 -openmp omp_hello.f -o omp_hello_intel
cc -O2 -openmp omp_hello.c -o omp_hello_intel
```

3) Hybrid Examples: A hybrid job is a mixed job that is a combination of OpenMP and MPI tasks. It aims to take advantage of the OpenMP in NUMA region which is also known as a socket. The NUMA region was in this case included one 12-core chip and ran on 6 threads on each of 8 MPI tasks which dispersed evenly between the NUMA regions.

A sum of 2 nodes was required which accomodated 8 MPI tasks and 6 threads. Besides specifying the number of nodes, the time was also shown in the SLURM script as done in the previous examples. The partition used was also *debugq* for this task.

```
#SBATCH --partition=debugq
#SBATCH --nodes=2
#SBATCH --time=00:05:00
#SBATCH --export=NONE
```

8 MPI tasks (-n 8) were specified to *aprun* to launch the job with 4 MPI tasks per node (-N 4), 2 MPI tasks per socket (-S 2) and each of the 8 MPI tasks had 6 OpenMP threads (-d 6). Therefore, the number of OpenMP threads, *OMP_NUM_THREADS*, was set to 6. This full expression is shown as:

```
export OMP_NUM_THREADS=6
aprun -n 8 -N 4 -S 2 -d 6 ./$EXECUTABLE >> ${OUTPUT}
```

Once again, to run on multiple threads with the Intel environment on Cray, AFFINITY should be disabled with the same command used in the OpenMP examples just before the *aprun* command.

The source codes used for the hybrid examples were also consisted of both FORTRAN and C which were *hybrid_hello.f90* and *hello_hybrid.c*. The README script contained the compiling commands to compile on Cray as shown below:

```
ftn -O2 -h omp hybrid_hello.f90 -o hello_hybrid_cray
cc -O2 -h omp hybrid_hello.c -o hello_hybrid_cray
```

To compile with the GNU environment for FORTRAN and C codes respectively, the README had the following compilers:

```
ftn -O2 -fopenmp hybrid_hello.f90 -o hello_hybrid_gnu
cc -O2 -fopenmp hybrid_hello.c -o hello_hybrid_gnu
```

For the Intel environment, the compiler code changed to:

```
ftn -O2 -openmp hybrid_hello.f90 -o hello_hybrid_intel
cc -O2 -openmp hybrid_hello.c -o hello_hybrid_intel
```

The same sbatch command as the OpenMP and MPI tasks was used for the hybrid codes to submit them to Magnus. The only difference was the name of the SLURM scripts.

4) LAMMPS: LAMMPS is a traditional molecular dynamics code. It is also an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator. It has potentials for solid state materials which include metals and semiconductors. It also has potential for delicate matter i.e biomolecules and polymers and coarse-grained or mesoscopic systems.

LAMMPS is an easy tool used to model atoms or rather, as a parallel molecule test simulator at the atomic, meso, or continuum scale. This tool runs on single processors or in parallel, making the use of message-passing systems (MPI) and a spatial-decomposition of the simulator domain. A significant number of its models have forms that give accelerated performance on CPUs, GPUs, and Intel Xeon Phi processors.

The LAMMPS example done in the *getexample* tool was specifically run on Magnus. The source code had been provided with a large number of atoms and their properties and potentials. The existing SLURM file of the MPI tasks of Magnus was utilized and modified to request a total of 2 nodes. The partition remained as *debugq*. The time was changed to 20 minutes as an increase in time would be better because it gives sufficient time to run the code. If the time is not long enough, it kills the job once the time runs out.

The following commands show the SLURM directives needed for the LAMMPS task:

```
#!/bin/bash -l
#SBATCH --job-name=hostname
#SBATCH --partition=debugq
#SBATCH --nodes=2
#SBATCH --time=00:20:00
#SBATCH --export=NONE
```

The program environment used for this task was GNU, hence the module was swapped from Cray to GNU and and lammps module was loaded which was required to compile the source code.

```
module swap PrgEnv-cray PrgEnv-gnu
module load lammps
```

In the template of the SLURM, an executable, a results directory and an output were declared to store the results of the LAMMPS example in the GROUP with the filenames below:

```
EXECUTABLE=lmp_mpi
SCRATCH=$MYSCRATCH/run_lammps/$SLURM_JOBID
RESULTS=$MYGROUP/lmp_mpi_results/$SLURM_JOBID

OUTPUT=lammps_mpi.log
```

Before running the example on the scratch, the files ending with .lmp name which contain the input data were copied to the scratch and this was obtained in the SLURM as:

```
cp *.lmp $SCRATCH
```

As this LAMMPS example is an MPI task, it was run on Magnus on 2 nodes with 24 MPI tasks per node giving a total of 48 tasks with the executable working on the data set called *emp2.lmp* and this was implemented with the *aprun* command to launch the job:

```
aprun -n 48 -N 24 $EXECUTABLE < epm2.lmp >> ${OUTPUT}
```

The SLURM was submitted to Magnus by the identical sbatch command used in the README files of all the Magnus examples.

5) GROMACS: GROMACS is also known as GROningen MAchine for Chemical Simulations which is a molecular dynamics package. It is designed for simulations of proteins, lipids, and nucleic acids. It contains a script to convert molecular coordinates from Protein Data Bank (PDB) files into formats. The simulation starts running when a configuration file for the simulation of the molecules is created. This produces a trajectory file which then can be analysed with various tools.

GROMACS can run on Central Processing Units known as CPUs as well as Graphics Processing Units called as GPUs. It can be executed in parallel, using Message Passing Interface (MPI) or threads. For the *getexample* tool, GROMACS example was run utilizing MPI similar to LAMMPS example, but it was more complicated to implement.

This example requested 2 nodes with 24 MPI tasks per node with a time allocation of 40 minutes, as it is a relatively big task to run. These changes were shown in the SLURM directives as:

```
#!/bin/bash -l
#SBATCH --job-name=hostname
#SBATCH --partition=debugq
#SBATCH --nodes=2
#SBATCH --time=00:40:00
#SBATCH --export=NONE
```

This code was also run on the GNU environment and hence, it was necessary to change the program environment from Cray to the GNU and load the gromacs module. This module was needed to compile the source code and access the original gromacs data set.

```
module swap PrgEnv-cray PrgEnv-gnu
module load gromacs/5.1.1
```

The generic variables for this job were defined in the SLURM just like the other examples, but with the addition of the input and GROMACS data directories as shown below:

```
EXECUTABLE=gmx
SCRATCH=$MYSCRATCH/run_gromacs/$SLURM_JOBID
RESULTS=$MYGROUP/gromacs_results/$SLURM_JOBID
INPUT_DATA_DIR=${SLURM_SUBMIT_DIR}/gromacs_mpi
GROMACS_DATA_DIR=${SLURM_SUBMIT_DIR}
OUTFILE=1AKI_processed.gro
PDB_FILE=1AKI.pdb
```

For this example, all of the gromacs and input data files were copied to the scratch.

```
cp $GROMACS_DATA_DIR/*.pdb $SCRATCH
cp $INPUT_DATA_DIR/posre.itp $SCRATCH
cp $INPUT_DATA_DIR/1AKI_processed.gro $SCRATCH
cp $INPUT_DATA_DIR/topol.top $SCRATCH
cp $INPUT_DATA_DIR/*.mdp $SCRATCH
```

Unlike the LAMMPS example, GROMACS consisted of many serial tasks which had to be performed first, before running the actual MPI task. These serial tasks were basically the pre-processing data for the MPI task and this was done as demonstrated below:

```
aprun -n 1 -N 1 $EXECUTABLE pdb2gmx -f ${PDB_FILE} -o
                        $OUTFILE -water spce << EOF
29
0
EOF
aprun -n 1 -N 1 gmx_d editconf -f 1AKI_processed.gro -o
        1AKI_newbox.gro -c -d 1.0 -bt cubic  > ${OUTPUT}

aprun -n 1 -N 1 gmx_d solvate -cp 1AKI_newbox.gro -cs
spc216.gro -o 1AKI_solv.gro -p topol.top >> ${OUTPUT}

#Adding Ions
aprun -n 1 -N 1 gmx_d grompp -f ions.mdp -c
1AKI_solv.gro -p topol.top -o ions.tpr >> ${OUTPUT}

aprun -n 1 -N 1 gmx_d genion -s ions.tpr -o
1AKI_solv_ions.gro -p topol.top  -pname NA
                -nname CL -nn 8 << EOF
13
EOF

# Energy Minimization
aprun -n 1 -N 1 gmx_d grompp -f minim.mdp -c
1AKI_solv_ions.gro -p topol.top -o
                em.tpr >> ${OUTPUT}

aprun -n 48 -N 24 mdrun_mpi_d  -v -deffnm em -g
                        energy.log >> ${OUTPUT}
```

The README script of GROMACS was very similar to the previous examples. However, the only difference was that it included many links to download the mdp files that the SLURM script required for the input data sets. Therefore when ./README was typed, the README automatically downloaded these files with the help of *wget* as shown below:

```
wget http://www.bevanlab.biochem.vt.edu/Pages/
Personal/justin/gmx-tutorials/lysozyme/Files/ions.mdp

wget http://www.bevanlab.biochem.vt.edu/Pages/
Personal/justin/gmx-tutorials/lysozyme/Files/minim.mdp

wget http://www.bevanlab.biochem.vt.edu/Pages/
Personal/justin/gmx-tutorials/lysozyme/Files/nvt.mdp
```

B. Zeus

According to Pawsey system descriptions, Zeus is an SGI Linux cluster that is principally utilized for pre and post-preparing of data, extensive shared memory calculations and remote visualization work. It shares the same /home, /scratch and /group file systems with other Pawsey Centre systems such as Magnus. Zeus is a heterogeneous cluster with a large number of back-end nodes with both CPU an GPU hardware which makes Zeus an excellent resource to run jobs like CUDA codes. To access the back-end nodes, a SLURM script is used just like Magnus except the default partition queue is the *workq* instead of the *debugq*. There are other nodes on Zeus such as *copyq* which is another partition queue that can be only used for specific purposes. Zeus also contains one large-memory node known as Zythos, which is an SGI UV2000 system and Zythos can be accessed from the front end login node of Zeus.

The *getexample* tool also provided some examples for Zeus which were also mainly based on parallel programming tasks. The SLURM and README scripts contained the same design as the ones for Magnus with only differing commands for running the task on Zeus, compiling the source codes and loading the correct compiler modules.

1) MPI Jobs: The *getexample* tool also includes MPI examples for Zeus with the GNU, Intel and PGI compilers because both Zeus and Magnus have different operating systems and hence, the same MPI example needs to handled differently on Zeus. It is necessary to realise that the MPI source codes used for Zeus are the same as Magnus, but the commands to run the same source codes are different on Zeus as compared to Magnus. Additionally, when the same code is run with different compilers such as GNU, Intel and PGI, the compiler module for the required compiler must be loaded. These descriptions were included in the README and SLURM files of each example within the *getexample* tool and the required module for the specific compiler was already loaded within these files so that the user is not asked to download them manually to prevent any errors while running the code.

The two common source codes used for the MPI examples were *hello_mpi.f90* and *hello_mpi.c*. These codes were used with GNU, Intel and PGI compiler options on Zeus to assist the users on how to run the same code on Zeus with different modules.

As mentioned earlier, both codes were run on both Magnus and Zeus on 2 nodes. However, the partition in the SLURM was changed to the *workq* from the *debugq*. These were defined in the SLURM script of the MPI examples as shown:

```
#SBATCH --partition=workq
#SBATCH --nodes=2
```

Unlike Magnus, when –export=NONE is included in the SLURM for Zeus, the code does not compile and it gives many errors. For this reason, this line was excluded in the SLURM files of Zeus to compile the codes correctly.

To run the code with a given executable on scratch, and store the results in a directory within the group directory to an output file, the generic variables were created in the SLURM as shown:

```
EXECUTABLE=hello_mpi_gnu
SCRATCH=$MYSCRATCH/run_hostname/$SLURM_JOBID
RESULTS=$MYGROUP/hellompi_gnu_results_zeus/$SLURM_JOBID

OUTPUT=hello_mpi_gnu.log
```

On Magnus, a total of 48 MPI tasks could be run whereas this was too much for Zeus and thus, the number of MPI tasks was reduced to 32. To run the job on Zeus, instead of using *aprun* command used in Magnus, *srun* command was used as srun is specific to Zeus while *aprun* is designed for Magnus. Therefore, to run the code, the following command was used:

```
srun -n 32 -N 2 --mpi=pmi2 ./$EXECUTABLE >> ${OUTPUT}
```

where -n defines the total number of MPI tasks while -N defines the number of nodes used on Zeus which is different from Magnus as -N defines the number of MPI tasks per node. Once again, in comparison to Magnus –mpi=pmi2 was added to *srun* which wasn't used in *aprun* where –mpi=pmi2 is the MPI implementation and must specified to *srun* for the correct operation.

In the README file, depending on which compiler the source code would be compiled with, the compiler module was included and loaded in the README file when it ran. For example, if the GNU compiler module was to be used, the gcc module was loaded with the mpt module as shown below:

```
module load gcc
module load mpt
```

If the code was to be compiled with Intel, the gcc module needs to be unloaded and the Intel compiler module needs to be loaded. This applies to the PGI compiler as well, after the gcc module is unloaded, the pgi module should be loaded as shown below:

For Intel:

```
module unload gcc
module load intel
module load mpt
```

For PGI:

```
module unload gcc
module load pgi
module load mpt
```

The mpt module needs to be loaded whenever an MPI or OpenMP/MPI task is submitted to Zeus as it provides the SGI message to access the MPI library.

Additionally, the README file contained the command to compile the *hello_mpi.f90* code and *hello_mpi.c* code with the GNU or Intel compiler as shown:

```
mpif90 hello_mpi.f90 -o hello_mpi_gnu
mpicc hello_mpi.c -o hello_mpi_gnu
```

For MPI tasks as mentioned earlier, the command to compile the code does not change for GNU and Intel compilers, whereas it differs for PGI compiler. Hence to compile the basic *hello_mpi.f90* and *hello_mpi.c* code, the following commands were used:

```
pgf90 -Mmpi=sgimpi hello_mpi.f90 -o hello_mpi_pgi
pgcc -Mmpi=sgimpi hello_mpi.c -o hello_mpi_pgi
```

2) OMP Examples: The OpenMP examples used for Zeus had the same source codes as the ones used for Magnus which were *omp_hello.c* and *omp_hello.f* However, as mentioned earlier due to the different setups between Magnus and Zeus, the SLURM and the README for these codes are different.

The main difference between the MPI and OpenMP jobs for Zeus is that the compiling command changes for all the different compiler options with OpenMP commands as the wrappers are not the same for GNU, Intel and PGI compilers. It is vital to notice that the commands to compile *hello_mpi.f90* and *hello_mpi.c* were the same for GNU and Intel compilers for the MPI jobs, whereas this is not the same case for OpenMP jobs anymore. However, everytime the compiler is swapped from one to the other, the correct compiler module should be loaded because the default compiler module is gcc. If not, the system will fail to recognize the compiler commands and end up giving many errors, even if the source codes and the SLURM files work.

In the *getexample* tool, to run an OpenMP job on Zeus, only one node was used just as Magnus, but as discussed in the MPI example, the *workq* partition was used unlike the *debugq* partition. Therefore, both of the OpenMP examples ran one 16-thread OpenMP instance with one node.

To run the *omp_hello.f* and *omp_hello.c* codes on Zeus with the GNU compiler, the number of OpenMP threads was set to 16 on the SLURM and the *srun* command was used to run it as shown:

```
export OMP_NUM_THREADS=16
srun -n 1 -c $OMP_NUM_THREADS ./$EXECUTABLE >> ${OUTPUT}
```

An alternative way of running the same job could be using the *omplace* command.

```
omplace -nt $OMP_NUM_THREADS -tm open64 ./$EXECUTABLE
                                        >> ${OUTPUT}
```

The *srun* command above can be used for both Intel and PGI compilers without requiring modification. However, the *omplace* command would be different for the other compilers. It is essential to note that *-tm open64* should be included above in the *omplace* command because when compiling with GNU, this command will not be identified, as the default thread model is intel. Hence, *-tm open64* tells that the compiler module is GNU. Therefore, to run this job with the Intel compiler, the *omplace* command in the SLURM would look like:

```
omplace -nt $OMP_NUM_THREADS ./$EXECUTABLE >> ${OUTPUT}
```

To compile the *omp_hello.f* with the GNU compiler, the following command was used in the README file:

```
gfortran -O2 -fopenmp omp_hello.f -o hello_omp_gnu
```

The *omp_hello.c* code was compiled with the GNU as shown:

```
gcc -O2 -fopenmp omp_hello.c -o hello_omp_gnu
```

As it can be seen, the only difference between compiling the C code and the FORTRAN code was the gcc and gfortran commands. Both of these codes can be compiled with Intel and PGI compilers, but the wrappers for them are different.

For Intel, compiling the FORTRAN and C codes respectively:

```
ifort -O2 -qopenmp omp_hello.f -o hello_omp_intel
icc -O2 -qopenmp omp_hello.c -o hello_omp_intel
```

For PGI, compiling the FORTRAN and C codes respectively:

```
pgfortran -O2 -mp omp_hello.f -o hello_omp_pgi
pgcc -mp omp_hello.c -o hello_omp_pgi
```

3) Hybrid Examples: The OpenMP/MPI codes used for Zeus were exactly the same as the ones for Magnus such as the FORTRAN and C codes, but they both had distinct compiler commands specific to each compiler module. Therefore, it was also necessary to include which module to be loaded in the README files of Zeus in the *getexample* tool.

This hybrid job requires 2 nodes and runs 1 MPI process with 16 OpenMP threads on each compiled executable. In order to launch the job to Zeus for both of the source codes, the number of OpenMP threads was set to 16 and the srun command was used.

```
export OMP_NUM_THREADS=16
srun --mpi=pmi2 -n 2 -N 2 ./$EXECUTABLE >> ${OUTPUT}
```

This command was used for all of the SLURM files with different compilers without making any changes. To compile the *hybrid_hello.f90* and *hello_hybrid.c* respectively with various compilers, the following commands were used:

For GNU:

```
mpif90 -O2 -fopenmp hybrid_hello.f90
                    -o hello_hybrid_gnu
mpicc -O2 -fopenmp -O2 hello_hybrid.c
                    -o hello_hybrid_gnu
```

For Intel:

```
mpif90 -O2 -qopenmp hybrid_hello.f90
                     -o hello_hybrid_intel
mpicc -O2 -qopenmp hello_hybrid.c
                     -o hello_hybrid_intel
```

For PGI:

```
pgf90 -Mmpi=sgimpi -mp hybrid_hello.f90
                       -o hello_hybrid_pgi
pgcc -Mmpi=sgimpi -mp hello_hybrid.c
                       -o hello_hybrid_pgi
```

4) CUDA Examples: The CUDA programming is a heterogeneous model where both CPU and GPU nodes are used. In CUDA, the host refers to the CPU and its memory, whereas the device refers to the GPU an its memory. Therefore, a CUDA code running on the host have access to the memory on the host as well as the device. It also executes kernel functions on the device which are executed by GPU threads in parallel. A basic CUDA works by declaring and allocating the host and device memory. Then, it initializes the host data and transfers the data from the host to the device. Once it executes the kernel function, it transfers the results from the device to the host.

The *getexample* tool includes a basic *"Hello world"* CUDA code, *hello_cuda.cu,* for Zeus. It uses 1 node with any generic GPU card, and this was defined in the SLURM file as:

```
#SBATCH --partition=workq
#SBATCH --nodes=1
#SBATCH --gres=gpu:1
```

To compile the CUDA code correctly, the cuda module should be loaded and this was done in both README and the SLURM file by using the following line:

```
module load cuda
```

Then, to submit this task to Zeus, the following command was used:

```
./$EXECUTABLE >> ${OUTPUT}
```

The CUDA code was compiled by using the NVIDIA compiler in the README file as:

```
nvcc hello_cuda.cu -o hello_cuda_gnu
```

C. Zythos

As mentioned earlier, Zythos is a large-memory node of Zeus and can be accessed through the front end login node of Zeus. Unlike Zeus, Zythos is a restricted asset and liable to a strict qualification criteria. One of the following criteria must be met for the task to be performed on Zythos:

- A vast detailed data should be held in shared memory, and be larger than 512GB.
- Gigantic thread-level parallelism, for example, utilizing a large number of CPU cores.

It is more preferrable that the work meets both criteria. If the work does not satisfy either one of these criteria, then Zeus is a better source to use. In comparison to Magnus and Zeus, there is no direct access to Zythos, all work is facillitated via Zeus. However, the partition queue is no longer the workq and it changes to zythos. Additionally, the codes on Zythos does not run properly if the account name is not specified or entered correctly as specific accounts are authorized to run on Zythos. For this reason, it is necessary to change account name and put the authorized account, otherwise the job does not run. Therefore, this brings out the challenge of having to modify the account name in the SLURM scripts of the *getexample* tool which means that the *getexample* is not completely automated for Zythos. The users who are interested in running these examples on this resource will not be simply typing ./README to execute the example but instead will have to replace the account name entered in the examples with their own. The examples provided for Zythos were also parallel programming codes such as MPI, OpenMP and hybrid tasks and these jobs were also run with GNU, Intel and PGI compilers which utilized exactly the same compiler commands as the Zeus examples. However, to run the jobs on Zythos, the SLURM scripts included different commands from *srun* such as *omplace* and *mpirun.*

1) MPI Examples: Similarly, the MPI examples used for Zythos also utilized the same source codes as Zeus and Magnus which were either FORTRAN or C. To compile with the correct modules, the compiler modules were also needed to be loaded by using the same commads as Zeus in both the README and SLURM scripts. Unlike Zeus, the partition in the SLURM for Zythos was *zythos* instead of the *workq.* The number of nodes was also specified with the use of *–ntasks* rather than the command *–nodes* as compared to the other Pawsey resources. In addition to the SLURM scripts, the number of cores per node was defined with *–cpus-per-task.* These were included in the SLURM as shown below:

```
#SBATCH --job-name=GE-hostname
#SBATCH --partition=zythos
#SBATCH --natasks=4
#SBATCH --cpus-per-task=6
#SBATCH --account=pawsey0001
#SBATCH --time=00:10:00
```

As mentioned previously, the authorised account name was inputted to the SLURM to access Zythos and to run the codes successfully. However, if the users decide to run the same examples, the only modification required is to change the name of the account.

When clients run a job on Zythos, for a better memory perfomance they are recommended to ask for the entire cores on the node as a whole instead of making the SLURM allocate cpus at irregular intervals on Zythos. The basic approach to do this is by occupying all of the 6 cores per node at all times. In so doing, the SLURM will be able to allocate the cpu cores successively, rather than having to pick up available cpu cores which may be scattered all over the framework.

Therefore to run the MPI task on Zythos, 4 nodes with 6 cores per node were required giving a total of 24 cores. This was implemented in the SLURM as shown below:

```
mpirun -np 24 ./$EXECUTABLE >> ${OUTPUT}
```

The *mpirun* command is an MPI launcher provided by the SGI message passing toolkit.

The source codes, *hello_mpi.f90* and *hello_mpi.c* were compiled with the default compiler GNU, Intel and PGI respectively by the using the compiler commands of Zeus with no modification as the compilers and wrappers are the same since Zythos is a node of Zeus. The only difference was the name of the executables as they are special to each example within the *getexample* library.

2) OMP Examples: For the OpenMP examples, the source codes used were *omp_hello.f90* and *omp_hello.c* which were compiled with GNU, Intel and PGI that used similar compiling commands as the OpenMP examples of Zeus. The SLURM script for the OpenMP task requests 2 nodes in comparison to the MPI tasks and 6 cores per node with 128 GB of memory each which gives a total of 12 cores. As mentioned previoulsy in the MPI examples, the 2 nodes were specified as *–ntasks* rather than *–nodes* in the SLURM directives as shown:

```
#SBATCH --partition=zythos
#SBATCH --ntasks=2
#SBATCH --cpus-per-task=6
#SBATCH --account=pawsey0001
#SBATCH --time=00:10:00
```

To run the job on Zythos, omplace was used for the OpenMP to control thread placement with a default of 6 threads per node which gave a total of 12 threads. This was written in the SLURM as:

```
export OMP_NUM_THREADS=12
omplace -nt $OMP_NUM_THREADS ./$EXECUTABLE >> ${OUTPUT}
```

Since Zythos is a node of Zeus, to compile the source codes and submitting the SLURM script to the Zythos, the same commands as Zeus were used and thus, the README and the SLURM scripts for the OpenMP jobs were similar to ones of Zeus.

3) Hybrid Examples: The OpenMP/MPI source codes used for Zythos were *hybrid_hello.f90* and *hello_hybrid.c* which were exactly the same as Zeus. These codes were compiled with all the compilers available on Zythos and before compiling, it was made sure that the right module was loaded by using the same commands in the MPI examples of Zeus. Similarly, the compiler commands for Zythos with the hybrid jobs were exactly the same as the ones for the hybrid jobs of Zeus. The only difference between the OpenMP/MPI examples of Zythos and Zeus was running the task on Zyhtos, as *srun* was no longer used, but instead *mpirun* with *omplace* was implemented for these examples on Zythos.

The characteristic approach to run the hybrid jobs on Zythos is to run one MPI task for per 6-core NUMA region. This job script contains a request for 4 nodes and begins 4 MPI tasks by using *mpirun* that is combined with *omplace* to place threads.

```
#SBATCH --partition=zythos
#SBATCH --account=pawsey0001
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=6
#SBATCH --time=00:10:00
```

To run the codes on Zythos, the number of OpenMP threads were given and passed the message through the *mpirun* as:

```
export OMP_NUM_THREADS=6
mpirun -np 4 omplace -nt $OMP_NUM_THREADS ./$EXECUTABLE
                                          >> ${OUTPUT}
```

The rest of the SLURM and the README files remained the same as the hybrid examples of Zeus, since the compilers and wrappers do not require any changes. The only modification made to the SLURM and the README was changing the names of the executables, results directories and output files to make them more specific to the hybrid jobs of Zythos.

IV. How to Implement the Getexample Command

Due to the different operating systems of the Pawsey supercomputers, each individual supercomputer had separate examples that could display the same function, but with different commands. Therefore when implementing the *getexample* command, the key principal was to ensure that it only displayed the examples related to the supercomputer in use. For example, if the user logins from Zeus, when one types the command *getexample*, it should only show the examples that can be performed on Zeus. This prevents the users from forcing to run examples that are not applicable for that particular supercomputer. For example, if the user logins from Magnus and tries to run an example taken from Zeus that compiles with GNU, the code will fail to run on Magnus with the GNU environment. Even though the program environment on Magnus was set to GNU, the compiler commands as well as the commands to run the task on Magnus differ from the ones of Zeus.

In order to follow this design specification, the *getexample* command was developed by using the following Bash script that determines which examples to list based on the *hostid* of the supercomputer.

```
#!/bin/bash
#DESCRIPTION Download user examples
#LABEL Files
hostid=`echo $HOST | cut -d "." -f 1`
echo ${hostid}
if [ ${hostid} == "magnus" ]; then
  location=/group/interns2016/getexample/magnus_examples
elif [ ${hostid} == "zeus" ]; then
  location=/group/interns2016/getexample/zeus_examples
fi

#TODO Make a list of users downloading examples?

#Display help message if no example is given
if [ "$1" == "" ]
then
    echo "Download an HPC example:"
    echo "usage:"
    echo "   getexample <examplename>"
    echo ""
    echo "Where <examplename> is the name of the example
    you want to "
    echo "download.  This will create a directory
    named examplename which"
    echo "you can cd into and hopefully read the
    README file (if one is"
    echo "avaliable) or just submit the *.qsub file."
    echo ""
    echo "For Example:"
    echo "  getexample helloworld"
    echo ""
    echo "Possible example names:"
    ls $location
    exit 0
fi

#TODO Check to see if folder already exists
cp -r -v -u ${location}/$1 .
```

V. Maintanence and Future Work

Even though the *getexample* tool is designed for the latest operating systems of the Pawsey Supercomputing Centre, it may not work properly in the future due to the changes and updates made to the advanced computing resources. This means that a previously working example may fail to work and resulting in inconvenience for the users and lose their trust for the *getexample.* This brings out the issue of how to maintain and curate the *getexample* so that it continues to be helpful and beneficial for the users. Therefore, in order to prevent the *getexample* becoming outdated, it is the technical staff's duty to ensure that all the examples run correctly which requires them to check for updates regularly, change the files in the *getexample* if necessary and modify the codes to suit the current systems.

However, the *getexample* tool is a large library with many examples. Thus, it can be really frustrating and hard for the system administrators to run all the examples one by one and having to fix these problems. In order to make this procedure easier for them, a basic Bash executable script can be used which runs all the examples at once, checks their status and prints out whether the examples were succesful or not to an output file based on their status.

Even though, the script shows which examples manage to run successfully or failed, the challenge for the system administrators is having to fix these problems as the *getexample*

consists a large number of examples. Therefore, it crucial to make this less complicated for the staff members and more time efficient as the less time it takes to resolve the issues, the more convenient it is for the users.

When revising the examples, it was noted that most of the examples use the same source codes such as *hello_world.f90, helloworld.c, hello_mpi.f90, hello_mpi.c, omp_hello.f, omp_hello.c, hybrid_hello.f90,* and *hello_hybrid.c*. This means that if one of these sample codes is broken, any examples that rely on this sample code will fail to work and this demands the staff members to look at all the examples that use the source code and fix the same errors one by one. One suggestion to make this more effective can be to have all the source codes in one directory rather them including them in the example directories initially. This suggests that each example directory will only contain the README and the SLURM scripts where the README will include the path to the source codes. When the user requests to use the example and run it by typing ./README, it will automatically copy the source code from the directory in which all codes are stored.

It is also significant to make sure that the *getexample* tool can be easily used to help the users create their own tasks without having to modify them much. When reviewing the *getexample,* it was noticed that for Magnus, all the examples use the *debugq* partition. This partition is mainly used for quick and small jobs. Thus, it is not a major problem for the examples in the *getexample* as they are basic codes which take less than 10 seconds to execute. However, when the users decide to utilize the SLURM directives for their own tasks which may be quite large and slow to run on Magnus, the *debugq* is not a suitable option. For this reason, it is a better idea to use the *workq* partition in the SLURM scripts for Magnus and thus, this requires modifications in all of the scripts.

Once again, as there are many examples located in the *getexample,* editing the partition manually is quite time consuming for the technical staff. To do this procedure in a faster way, a Bash script can be written to go through each README and the SLURM scritps of the Magnus examples and modify the partition from the *debugq* to *workq* automatically. This script can be even used for other purposes especially curating the *getexample* whenever the examples need any modification.

Thus, for the future development of the *getxample*, the key principal that needs to be followed is always to keep this tool as user friendly as possible and make it easy to maintain. It is also essential to keep adding more examples such as applications to the library to make it more comprehensive for the needs of the users.

VI. Conclusion

Throughout this journal, various ways of decreasing the obstructions faced for the new users of the resources at Pawsey Supercomputing Centre were discussed. Pawsey Centre has many clients who work with all sorts of resources provided for them. Therefore, it is helpful to build a tool that can supply a list of working examples for the users to show and teach them how to use these resources effectively. This inevitably not only reduces the amount of time it takes for the new users to get to know these resources better, but also encourages them to apply what they learnt to their own projects. In order to perform these actions, the *getexample* tool was developed which displays a comprehensive list of examples for the users. It is believed that the *getexample* can enhance supercomputing training, eliminate the problems faced by the new users of having to modify the codes to make them work and decrease the time of getting familiar with the resources at Pawsey.

Acknowledgment

The authors would like to thank to their supervisor, Chris Bording for his patience and endless help for this project.

References

[1]
- H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.
- Dirk Colbry, *getexample: Reducing Barriers to Entry on HPC Resources*, Michigan State University, 2016
- Blaise Barney, *website:https://computing.llnl.gov/tutorials/openMP/*, Lawrence Livermore National Laboratory
- Magnus User Guide, *https://support.pawsey.org.au/documentation/display/US/Magnus+User+Guide*, Pawsey Supercomputing Centre
- Compiling on a Cray, *https://support.pawsey.org.au/documentation/display/US/Compiling+on+a+Cray*, Pawsey Supercomputing Centre
- Running on a Cray, *https://support.pawsey.org.au/documentation/display/US/Running+on+a+Cray*, Pawsey Supercomputing Centre
- Zeus/Zythos User Guide, *https://support.pawsey.org.au/documentation/pages/viewpage.action?pageId=2162999*, Pawsey Supercomputing Centre
- MPI FORTRAN Examples, *https://people.sc.fsu.edu/~jburkardt/f_src/hello_mpi/hello_mpi.f90*, Oct 23, 2011
- MPI C Examples, *https://people.sc.fsu.edu/~jburkardt/c_src/mpi/mpi.html*, Oct 24, 2011
- Mixing MPI and OpenMP, *http://www.slac.stanford.edu/comp/unix/farm/mpi_and_openmp.html* Mar 22, 2006
- LAMMPS, *http://www.nersc.gov/users/software/applications/materials-science/lammps/*, Jan 31, 2017
- GROMACS, *http://www.gromacs.org/*, Nov 4, 2016