Bare Demo of IEEEtran.cls for IEEE Journals

Ozlem Erkilic
Pawsey Supercomputing Centre Curtin University
Perth, Australia
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 diffucult 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 maynot 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 to how to run code 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.**

***Index Terms*—IEEE, IEEEtran, journal, LaTeX, paper, template.**

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 challanging 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 resources with many constraints. The most common differences between their systems are as follow:

- Contrasting operating systems
- Divergent program environments
- Non-idendical compiler options with varying commands
- Various compiler flags and wrappers for varying compiler options
- Module systems and paths that are different
- Disparate module naming conventions
- Various library versions
- Dissimilar personal scratch, group and home directories
- Different 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, 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 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 and vital for their researches. It is also very crucial that these are stored correctly within these sources.

In order to minimise these problems, the getexample was developed to suplly 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.

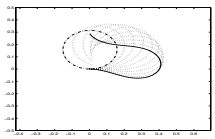

Fig. 1. elastic data set 1

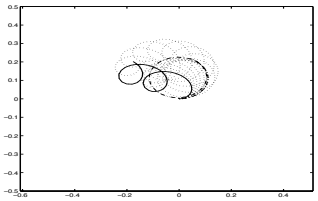

Fig. 2. elastic data set 2

II. Overview of getexample

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:

- It should be easy to use for everyone including 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 other users for their own work. Therefore, it encourages the users to learn how to use the resources given to them effectively.
- The example 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 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. The new directory then can be accessed by the user simply typing:

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

III. Creating Examples on getexample

The examples included in the getexample library were created by internship students at Pawsey with their supervisor while the source codes within the getexample were obtained from some wiki pages and websites and 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 such as 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 detail 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 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 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 HPC resources, there are many design considerations to be held as listed below:

- Before introducing the examples to the users, it should be made sure 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 shell script is used, it should be run as ./README.
- The examples should have enough instructions about 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.
- It must be ensured that the README and SLURM have enough information about how to change certain parts of these files so that the users can run the example how they wish to on the supercomputers. 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

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 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 and 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 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 might 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 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 codes. 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 Magnus, the classified partition is 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 MPI job with fully occupied 2 nodes, the aprun command was used as expressed below:

```
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 as there are 2 nodes.

In the README file, the correct program environment should be included for the codes to run well. 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 world"* MPI Fortan code, hello_mpi.f90 on Cray, GNU and Intel environments:

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

-02 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 user as name. For GNU and Intel, this was called hello_mpi_gnu and hello_mpi_intel.

When the source code in use was the c code, 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_gnu
```

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 from which module to swap, at the end of the SLURM and the README, the program environment was always set back to the default, Cray by:

```
module swap PrgEnv-<name of 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}
```

-n defines the total number of MPI tasks, -N specifies the number tasks per node while -S defines 6 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 different environments on Magnus with the OpenMP tasks. Thus, it is necessary to swap to the right program environment corresponding to the compiler.

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.

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

The initiation of the job by aprun is done by specifying that there is 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 are 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 were expressed as:

```
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 involved 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 6) 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 after 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 compiler 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 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 semicoductors. It also has potential for delicate matter i.e biomolecules and polymers and also 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 as it is the correct partition for Magnus. 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 does not completely run the code and kills the job once the time runs out.

The following commads 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 the lammps module was loaded which is required to run the source code and the data.

```
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 filenames below:

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

Before running the example on the SCRATCH, the files ending with .lmp name which contain the data and code were copied to the SCRACTH 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 and this was implemented with the aprun command to launch:

```
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.

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 workq instead of 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 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 as Magnus. 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 mistakes while running the code.

The two common source codes used for the MPI examples were *hello_mpi.f90* (a FORTRAN 90 source code) and *hello_mpi.c* (a c code). 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, even if the command for compiling the same source code does not for MPI tasks on Zeus.

As mentioned earlier, both codes were run on both Magnus and Zeus on 2 nodes. However, the partition in the SLURM was changed to workq from 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 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 as:

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

where (-n) defines the total number of MPI tasks while (-N) defines the number of 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 FORTRAN 90, *hello_mpi.f90* code and *hello_mpi.c* 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_c
```

2) OMP Examples: The OpenMP examples used for the Zeus are the same source codes as the ones used for Magnus which are *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 mistakes, 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 diccussed in the MPI example, the workq partition was used unlike the debugq partition on Magnus. Therefore, both of the OpenMP examples run one 16-thread OpenMP instance with one node.

To run the *omp_hello.f* and *omp_hello.c* code on Zeus with GNU compiler, the number of OpenMP threads were set to 16 on the SLURM and 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
\begin{Verbatim}
\begin{tcolorbox}

The omp_hello.c code can be compiled with the GNU as shown below:

\begin{tcolorbox}
\begin{Verbatim}[fontsize=\scriptsize]
gcc -O2 -fopenmp omp_hello.c -o hello_omp_gnu
\begin{Verbatim}
\begin{tcolorbox}

As it can be seen, the only difference between compiling the c cod
compiled with Intel and PGI compilers, but the wrapper for them wa

For Intel, compiling the Fortran and c codes respectively:

\begin{tcolorbox}
\begin{Verbatim}[fontsize=\scriptsize]
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 hybrid codes used for Zeus were exactly same as the ones for Magnus such as FORTRAN and C code, 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 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 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
\begin{Verbatim}
\begin{tcolorbox}

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

\begin{tcolorbox}
\begin{Verbatim}[fontsize=\scriptsize]
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 following compiler line in the README file:

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

IV. Conclusion

The conclusion goes here.

Appendix A
Proof of the First Zonklar Equation

Appendix one text goes here.

Appendix B

Appendix two text goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

PLACE PHOTO HERE

Ralph Bording Biography text here.

Jane Doe Biography text here.