PPCES 2013: The RWTH Compute Cluster Environment Exercises

March 2013
Tim Cramer, Paul Kapinos, Frank Robel
{cramer,kapinos,robel}@rz.rwth-aachen.de

1. The Linux Cluster

This exercise gives a little overview about some Linux basics and about our cluster environment. Feel free to start with another lab or skip some exercises, if you are already familiar with these topics. All topics are also covered in our HPC User's Guide:

http://www.rz.rwth-aachen.de/hpc/primer

1.1. Login

Please refer to the additional hand out "Access to Lab Machines".

1.2. Environment

1.2.1. The Module System

For representing the different requirements of our users we provide the **module system** which helps you to manage the installed software. To get an overview do the following:

- `$ module help` # This command gives some hints about the usage of the modules
- `$ module list` # Print the loaded modules. Note: Some are loaded by default.
- `$ module avail` # List the available modules.

The modules are organized in categories. Only the DEVELOP category (containing compilers, MPIs and some tools) is loaded by default. To load a module you first have to load the corresponding category (e.g. CHEMISTRY for some chemistry software). Load the module category with

```
$ module load CHEMISTRY
```

and compare the available modules again:

```
$ module avail
```

The distribution of modules into categories is historically grown and not always intuitive. You may use the

```
$ module apropos [keyword | modulename]
```

command to figure out in which category a module is located. Find out where you can find the Matlab module is located this command and load it! After you have done this you can start with

```
$ matlab
```

Some modules can depend on other modules, e.g. MPI modules depend on certain compilers. So the order of loading the modules is crucial. Unloading a compiler and then loading another compiler by `module (un)load` will lead to a broken environment, because of the wrong order of loading. To avoid such scenarios, use the `module switch` command, e.g. for replacing the Intel compiler by the Oracle (Sun) compiler:

```
$ module switch intel studio
```

This will unload all modules from bottom up to the `intel` module, load the `studio` module and then reload all previously unloaded modules. If the environment is damaged the `module reload` command can help. Especially if using the NX software to log in, it is a known problem that the value of the LD_LIBRARY_PATH environment variable vanishes after login which can be repaired by reloading the modules.

1.2.2. User Limits (ulimits)

The `ulimit` command is a built in command for the shell environment. Use

```
$ ulimit -a
```

to check all user limits. Two limits are known for being an often malfunction: the CPU time (-t) and the stack size (-s) limit. The CPU time limit sets a restriction for the runtime of a program. So if your program has to run for more than a certain time, the limit has to be set accordingly. The stack size limit sets a restriction for a dedicated memory area called “stack”. For **Fortran** and for **OpenMP** programs, the consumption of the stack memory is known to be quite high. If stack space is exhausted, the program suffers a segmentation violation error and core dumps. Thus, set the stack size limit to 250 megabytes using the command:

```
$ ulimit -s 250000
```

2. Hello World Program

2.1. Compiling

Change the directory:

```
$ cd serial/environment_lab/source
```

Compile these simple examples. The module system sets different environment variables to provide easy and conform access to different compilers:

- $FC – The Fortran compiler
- $CC – The C compiler
- $CXX – the C++ compiler
- $FLAGS_FAST - a set of optimization flags we recommend to use with the loaded compiler

To compile the Fortran or C Hello World program type:

```
$ $FC $FLAGS_FAST hello.f90
```

or

```
$ $CC $FLAGS_FAST hello.c
```

The compiler should produce an executable file “a.out”. To execute it type:

```
$ ./a.out
```

2.2. The Importance of the Ulimits

As described in exercise 1.2.2, the environment user limits (ulimits) can lead to an unexpected behavior of your program. Compile the small example program `ulimit_s-test.f90` and execute it with different settings of the stack size ulimit: 10 Megabytes and 200 Megabytes.

3. The Linux Batch System (Platform LSF)

All computations that run longer than 15 minutes should be performed in the batch system (Platform LSF). To submit a job to the batch system it is recommended to use a batch script for the `bsub` command. A batch script is basically a shell script containing all commands which you want to be executed and the requirements of the job. It is also possible to provide all options to `bsub` over the command line. An (still incomplete) example batch script is the `submit.sh` file. Please edit it using a text editor (e.g. `vim`, `nano`, `gedit` or `kate`) to the correct values for the batch options. Note that the batch script can (and should!) be tested interactively before submitting. The interactive test helps to identify eventual errors before using the batch system which can have a longer waiting time. To let the batch file run interactively, it must be marked as “executable”:

```
$ chmod 755 submit.sh
```

And then execute with:

```
$ ./submit.sh
```

Read the output carefully and correct all errors. If everything is correct, you can submit it:

```
$ bsub < submit.sh
```

Please note the ”**<**”. This arrow was not needed with the old batch system, with LSF it is. If you do not use it the job will be submitted, but all resource requests will be ignored, because the #BSUB is not interpreted by the workload management. The batch system returns the job ID of the submitted job:

```
Job <xxxxxx> is submitted to default queue <normal>.
```

The status of all your jobs can be controlled with the `bjobs` command. Only waiting jobs and jobs in an error state are shown; finished jobs will not be listed. If a submitted job is not needed to run, it can be deleted with the `bkill` command:

```
$ bkill xxxxxx
```

where *xxxxxx* is the job ID. If a batch job is executed, up to four files will be created in your home directory, containing the output of the batch system and the program itself (standard and error I/O). There are options to merge and rename the output files.

3.1. Exercise 1: Serial Jobs

Edit the `submit.sh` file to obtain a correct batch file, test it interactively and submit the job to the batch system twice. Check the submitted jobs; delete one of the jobs. Wait and see the batch job being executed, check the output of the batch job and the batch system. Note: The minimal time to schedule and execute a batch job may be several minutes, so you can proceed with other exercises and check for output of your batch jobs later.

3.2. Exercise 2: Parallel Jobs

In our wiki you can find several examples for parallel job scripts: https://wiki2.rz.rwth-aachen.de/display/bedoku/Workload+Management+System+LSF. In the lab directory we provide two parallel programs:

1. mpihelloworld.f90: A Fortran example parallelized with MPI
2. openmphelloworld.c: A C example parallelized with OpenMP (Shared Memory)

Look at the wiki2 scripts and modify them to compile and execute the example codes with 4 Cores. The following environment variable might be useful:

- $MPIFC, $MPICC, $MPI - compiler driver for the last-loaded compiler module, which automatically sets the include path and also links the MPI library automatically.
- $MPIEXEC - The MPI command used to start MPI applications, e.g. mprun or mpiexec.
- $FLAGS_MPI_BATCH - MPI options necessary for executing in batch mode.
- $FLAGS_OPENMP - Compiler option to enable OpenMP support.
- $OMP_NUM_THREADS - Sets the number of threads you want to use during the runtime

3.3. Exercise 3: Your own Jobs

You can submit your own jobs with your own scripts now. Feel free to ask questions.