Program development basics on Linux
Exercises
March 22, 2010

{anmey|bierbaum|iwainsky|kapinos|reichstein|schmidl|wienke}@rz.rwth-aachen.de

1. Login to Cluster

In this exercise you will log in to the Linux part of the cluster.
There are two possible ways to login: the console login and the graphical login via NX.

The *text console login* over SSH is possible to all Linux frontend nodes, e.g. **cluster.rz.rwth-aachen.de**. You must have a cluster account and be in the RWTH network. Use the hpclabXZ from you badge as login name; the password will be announced during the lab.

Linux: open a console and type (where Z is the rightmost digit on your badge):

```
ssh -Y -l hpclabXZ labZ.rz.rwth-aachen.de
```

Windows: open an SSH session by double-clicking on an icon on the desktop and perform adjustments in the opened window.

The *graphical login using NX* is possible on a few frontends only:

cluster-x.rz.rwth-aachen.de
cluster-x2.rz.rwth-aachen.de

The NoMachine (NX) software can be downloaded from www.nomachine.com. On the lab laptops a NX client is already installed.

Open a new connection by double-click on the NX icon on the Desktop and enter the hostname, user name (login) and password. Click on *Configure* to perform some advanced tuning. You should specify the expected bandwidth of your network connection (*ADSL*). On Windows we recommend to disable the direct draw option: *Configure* --> *Advanced* --> *Disable direct draw for screen rendering*
The ultimate benefit of NX graphical login is the survival of your session in case of network problems. After you started an NX session, you should open a terminal and log in to the lab machine (where *Z* is again the rightmost digit on your badge):

```
ssh labZ
```

2. The Linux environment

After you logged in, you may get to know the Linux environment.

2.1 The **module** system helps to manage all the software installed in the cluster. To get some experience with the module system, try all commands out which are described below.

Use the command **module help** to see some hints about the usage of modules.

There are some modules loaded by default; use **module list** to see them.

To see which modules are available, use the **module avail** command.

The modules are organised in *categories*, only the DEVELOP category (containing compilers, MPIs and some tools) is loaded by default. The category containing a module must be loaded before the module can be loaded, e.g. CHEMISTRY for some chemistry software:

module load CHEMISTRY

Use **module avail** command again and compare the output with the first completion.

The distribution of modules into categories is historically grown and not always intuitive. You may use the **module apropos** command to figure out in which category a module is located: e.g. for finding out where the Matlab module is, use **module apropos matlab** command.

Non-needed modules can be unloaded by **module unload** command:

module unload CHEMISTRY

unloads the CHEMISTRY category loaded previously.

Some modules can depend on other modules, e.g. MPI modules depend on certain compilers. So the order of loaded modules is crucial, and unloading a compiler and then loading another compiler by *module (un)load* will lead to a broken environment (because of a wrong order of modules). 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, unload 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 known that the value of the LD_LIBRARY_PATH environment variable vanishes after login which can be repaired by reloading the modules.

2.2 The **ulimit** command is a built in command of the shell environment.
Check all set limits with **ulimit -a**

Two limits are known for being an often malfunction source: the *cpu time* (-t) and the *stack size* (-s) limit.

The *cpu time* limit sets a restriction for the runtime a program can run; so if your program has to run for more than a certain time, the cpu time limit must be set accordingly.

The *stack size* limit sets a restriction for a dedicated memory area called “stack”. For Fortran programs and for OpenMP programs in all supported programming languages, the consumption of 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 time limit to an hour and the stack size limit to 250 megabytes using the **ulimit** command.

3. Hello-World-program

3.1 Compiling

First, unpack the TAR archive containing the two very simple programs i.e. hello.c and hello.f source files:

tar -xzf /home/hpclab/PPCES2010/ProgEnv/hello.tgz

You can overview the unpacked files using the **ls** and **less *file*** commands.

Compile the source files to an executable file. The module system sets environment variables to provide easy and conform access to different compilers; of importance are the following environment variables:

$FC – the Fortran compiler
$CC – the C compiler
$CXX – the C++ compiler
$FLAGS_FAST – a set of optimisation flags we recommend to use with loaded compiles.

To compile a Fortran or C HelloWorld program, type:

$FC $FLAGS_FAST hello.f90

or

$CC $FLAGS_FAST hello.c

The compiler should produce an executable file a.out. Check it by

file a.out

3.2 Running

Let's run the binary file produced in the previous step:

./a.out

3.3 The importance of ulimits

As described in Exercise 2, the environment user limits (ulimits) can lead to an unexpected behaviour of your program. Compile the small example program *ulimit_s-test.f90* and let it run with different settings to stack size ulimit: 10 Megabytes and 200 Megabytes.

4. The Linux batch system: (qsub, qstat, qdel)

All long-time computations (more than 15 minutes) should be performed in the batch system (Oracle Grid Engine, formerly known as SGE). A batch script must be submitted using the qsub command (or alternatively using the qmon GUI which however is not considered in this exercise). A batch script is basically a shell script containing all commands which must be executed, and (recommended) needed options of the batch system. It is also possible to provide all options to qsub over the command line. An example batch script is the *submit.sh* file. Please edit it using a text editor (e.g. **vim**, **nano** or **kate**) to provide meaningful values for the batch options. Note, that the batch script can (and should!) be tested interactively before submitting. The interactive test lets you directly find eventual errors omitting the batch system waiting time. To let the batch file run interactively, it must be marked as “executable” by the command:

chmod 755 submit.sh

And then execute it by

./submit.sh

After you have tested your script you can submit it to the batch system:

qsub submit.sh

The batch system tells you the job ID of the submitted job:
Your job xxxxxx ("submit.sh") has been submitted

The status of all your jobs can be controlled by the **qstat** command. Only waiting jobs and jobs in an error state are shown; finished jobs are not listed in.

If a submitted job is not needed to run, it can be deleted by the **qdel** command:
qdel *xxxxxx*
where *xxxxxx* is the job ID.

If a batch job is executed, up to four files will be created in your home dir, 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.

Exercise: Edit the *submit.sh* file to obtain a correct batch file, test it interactively, and submit the batch file 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.

5. make/gmake

The *make* (or *gmake* for GNU make) is an excellent instrument to automatise program development on Unix and Linux. So use the opportunity to practise and write a makefile which allows to execute the exercises 3 and 4 by *make*. The makefile should contain at least the targets *build, run, clean*.

6. Login to Virtual Machine (VM)

To start the VM on the laptop, double-click on the **VMware** labelled icon on the desktop. Click on the *SUS_HPC_Kurs_1* tab and then on the green arrow to power on this virtual machine. Log in with "**hpclab**" as user name (the password will be announced during the lab). Click on the desktop icon labelled with *Terminal* to open a console. To access the folder shared between Windows and the VM, either click on the PPCES2010 folder on the desktop or go to /mnt/hgfs/shared/PPCES2010 in the console.

Then you can examine the exercises 2, 3 and 5 as described above. The batch system is not usable from the VM, so omit the fourth exercise.

Note on 2.1: In the VM only a module for Oracle (Sun) MPI is available, and there are no module categories to load. Anyway, all the module commands work in the same way as on the cluster.
Note on 3.1: In the VM a Sun Studio compilers are installed instead of Intel compilers. There is also no module for the Sun Studio compilers and TotalView debugger in the VM, but the environment is prepared to be the same as the modules would be loaded in the cluster. So the environment variables can be used as described.