Using the GWDG Scientific Compute Cluster - An Introduction

by Azat Khuziyakhmetov and Marcus Boden

Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

Am Fassberg, 37077 Göttingen

Fon: 0551 201-1510 Fax: 0551 201-2150
gwdg@gwdg.de www.gwdg.de

Outline

1. Connecting to the frontends
2. The most important Linux commands
3. Preparing the compilation environment with “modules”
4. Compiling Software
5. Efficiently Submitting Jobs to the Cluster
6. Getting Help

Section 1

Connecting to the frontends

Frontends

- gwdu101 (and transfer-scc): Abu-Dhabi AMD Opteron 6220
  - ➥ processor features identical to gwdaXXX
  - ➥ older nodes in fat-partition
  - ➥ access to /scratch
- gwdu102: Sandy-Bridge Intel E5-2670 v1
  - ➥ processor features identical to gwddXXX
  - ➥ older nodes in medium-partition
  - ➥ access to /scratch
- gwdu103: Broadwell Intel E5-2650 v4
  - ➥ processor features identical to dfaXXX, dmpXXX, dgeXXX, dteXXX
  - ➥ new nodes in fat and medium partition (and gpu partition)
  - ➥ access to /scratch2

ssh to the frontends

- Linux or OS X: “ssh gwdu101.gwdg.de -l {GWDG-USERID}”
- Windows: Download *putty.exe* from https://www.chiark.greenend.org.uk/∼sgtatham/putty
  - ➥ Run it. Enter “gwdu101.gwdg.de” in *hostname* and click open
  - ➥ Select “Yes” to trust the connection
  - ➥ Login as: {GWDG-USERID}
  - ➥ Enter password

```
The authenticity of host 'gwdu101.gwdg.de (134.76.8.101)' can't...
ECDSA key fingerprint is SHA256:sIJNEepmILeEq/7Zqq4HCtpTM8L98ar...or
ECDSA key fingerprint is 7c:52:2b:17:f8:ba:29:bd:c5:45:d1:1a:9e...or
RSA key fingerprint is b9:f9:46:0f:23:c8:8d:76:b9:83:b9:1b:f6:5...or
ED25519 256 key fingerprint is e3:ef:39:f5:df:4f:c2:e2:c4:d0:28...
Are you sure you want to continue connecting (yes/no)?
```

Section 2

The most important Linux commands

Listing files and directories

- List the current directory you are in, “`ls`”
  - ➥ List the “hidden” files (beginning with “.”) too, “`ls -a`”
  - ➥ All files in an extended manner, “`ls -la`” or just type “`l`”
- Let’s look at three lines of the output

```
drwxrwxrwx   3 tehlers users    4096  4. Apr 17:29 test
-rw-r--r--   1 tehlers users     283 24. Sep 2003  Info.txt
lrwxrwxrwx   1 root root          23  Jul 22 12:10 passwd -> /etc/passwd
```

1	2	3	4	5	6	7	8	9	10

ten permission flags:

1 directory flag, “`d`”: directory, “`-`”: normal file, “`l`”: symlink

2,3,4 read, write, execute permission for **U**ser (Owner of the file)

5,6,7 read, write, execute permission for **G**roup

8,9,10 read, write, execute permission for **O**thers

Changing the language, what if I don’t undestand German

```
> echo $LANG
de_DE.UTF-8
> rm test
rm: reguläre leere Datei "test" entfernen?
```

```
> export LANG=en_US.UTF-8
> rm test
rm: remove regular empty file ‘test’?
```

For persistent English language, put it in your “.profile”:

```
echo ’export LANG=en_US.UTF-8’ >> ∼/.profile
```

File operations, processes and file system

cd change directory

top display Linux processes, sorted list

ps display current processes, imp. opt. a [all sessions], u [owner], x [all], w [wide], ww [even wider]

touch create file / update timestamps

more... `cp`, `rm`, `mv`, `mkdir`, `rmdir`, `ln`

df display filesystem usage, `df -h`, `df -hl`

chmod

- Files attributes (mode bits) can be changed with `chmod`
- chmod can be used in two ways:
  - ➥ user friendly form:
    u (user) g (group) o (others) a (all)
    `chmod a+r {file}, chmod g=rwx,o+r {file}`
  - ➥ tell the mode bits:
    `chmod 744 {file}`

chmod (2)

- 0-7 are 3 bits: 111 → 7
- same order, like in dir listing: r,w,x

000 0 → `---` no read write or execute allowed

001 1 → `--x` (last bit is set)

010 2 → `-w-` (middle bit is set)

011 3 → `-wx` (last 2 bits are set)

100 4 → `r--` (first bit is set)

101 5 → `r-x` (first and last bits are set)

110 6 → `rw-` (first and second bits are set)

111 7 → `rwx` (all 3 bits are set)

chmod (3)

- In sum we have 9 bits now in 3 groups (user, group, others)
- But there is a 4th group: SUID/SGID/sticky-bits
- SUID/SGID means that the called program will run with the UID or GID of the owner
  - ➥ e.g. if the program owns root and has SUID set, you run the program as root
  - ➥ `chmod u+s {file}`, or `chmod g+s {file}`, `chmod a+s {file}` would set both
  - ➥ Since we are normal users on the system, this is very seldom needed.
- sticky-bit is more relevant for you, if you open a directory for colleagues to write (`chmod g=rwx {dir}`)
  - ➥ the stick-bit prevents others from deleting files, they do not own. (`chmod +t {dir}`)
  - ➥ e.g. if you create a file, others cannot delete it, even though they have write permission to the directory.

umask

- standard umask is “022” or “`u=rwx,g=rx,o=rx`”
- umask is the inversion (mask) of default file attributes, when creating a file
    - ➥ But you can use it like chmod with u=XXX, g=XXX or o=XXX, to display write “umask -S”
    - ➥ e.g. `umask u=rwx,g=rx,o=`

Editors, help and Bash

- vi/vim, mcedit, joe, nano

For most commands you can read the manual pages, just type “`man {COMMAND}`”.

The prompt is a so called “Shell” with built-in commands and functions. We are using the “bash”. Type “man bash” to get an impression about the power and flexibility of that shell.

Editors, help and Bash

- vi/vim, mcedit, joe, nano

For most commands you can read the manual pages, just type “`man {COMMAND}`”.
The prompt is a so called “Shell” with built-in commands and functions. We are using the “bash”. Type “man bash” to get an impression about the power and flexibility of that shell.

nice and renice

- Test your program before submission
- You can do it on frontends (only short tests!)
- And please be nice on gwdu101, gwdu102 and gwdu103
  - ➥ `nice -n 19 {COMMAND}`
- If you forgot to nice and don't want to restart the program
  - ➥ open a new terminal:
  - ➥ `renice -n 19 {PROCESS ID}`
- For heavy programs use "short" partitions (part II)

Environment variables

Where the system gets all the commands we learned today?

Bash searches all paths in the environment variable **PATH**.

```
gwdu101:84 15:03:22 ~ > echo -e ${PATH//:/:\\n}
/opt/slurm/bin:
/usr/lib64/qt-3.3/bin:
/opt/lsf/10.1/linux2.6-glibc2.3-x86_64/etc:
/opt/lsf/10.1/linux2.6-glibc2.3-x86_64/bin:
/usr/local/bin:
/usr/bin:
/usr/local/sbin:
/usr/sbin:
/sbin:
/usr/sbin:
/cm/local/apps/environment-modules/3.2.10/bin
```

The first Shell-Script

For our first Shell script we need additional information

- *grep* gets the input and only outputs matching lines, command "`grep akhuziy`" outputs only lines containing "akhuziy"
- A Pipe "|" puts the output stream (stdout) into the input stream (stdin) of another program:
  - ➥ "`ls -la | grep akhuziy`" shows only files owned by akhuziy or if the filename contains "akhuziy".
- "`mktemp -d /scratch/${USER}/XXXXXXXX`" will create a unique directory, e.g. /scratch/akhuziy/XymeK4nq and echo it to stdout
- To store an output of a program in a variable, we write "`TEMPDIR=$(mktemp -d /scratch/${USER}/XXXXXXXX)`"

Let's write a little Shell script...

The first Shell-Script

For our first Shell script we need additional information

- *grep* gets the input and only outputs matching lines, command "`grep akhuziy`" outputs only lines containing "akhuziy"
- A Pipe "|" puts the output stream (stdout) into the input stream (stdin) of another program:
  - ➥ "`ls -la | grep akhuziy`" shows only files owned by akhuziy or if the filename contains "akhuziy".
- "`mktemp -d /scratch/${USER}/XXXXXXXX`" will create a unique directory, e.g. /scratch/akhuziy/XymeK4nq and echo it to stdout
- To store an output of a program in a variable, we write "`TEMPDIR=$(mktemp -d /scratch/${USER}/XXXXXXXX)`"

Let's write a little Shell script...

Reformatting Data

```
~ > cat file1
column1      column2      column3
1            2            3
4            5            6
```

We just want column number 2.

```
~ > cat file1 | (while read a b c; do echo $b; done)
column2
2
5
```

Reformatting Data II

```
~ > cat file2
column1,column2,column3
1,2,3
4,5,6
```

We still want column 2, but the separator is “,”.

```
~ > cat file2 | sed "s/,/ /g" |
 (while read a b c; do echo $b; done)
column2
2
5
```

Reformatting Data III

We only need line number 2 and column number 2 from *file2*.

```
~ > cat file2
column1,column2,column3
1,2,3
4,5,6
```

```
~ > cat file2 | sed "s/,/ /g" |
(count=0; while read a b c;
do let count=$count+1; if [ "$count" = "2" ];
then echo $b; fi; done)
2
```

Reformatting Data IV

The comma separated list has empty values.

```
∼ > cat file3
column1,column2,column3
1,2,3
4,5,6
7,,9
```

With “ ” as a separator we get:

```
∼ > cat file3 | sed "s/,/ /g" | (while read a b c; do echo $c; done)
colum3
3
6
```

We set the bash-variable “IFS”

```
∼ > IFS=","
∼ > cat file3 | (while read a b c; do echo $c; done)
colum3
3
6
9
```

Stageout from /scratch (not for /scratch2)

- We have a stageout mechanism from /scratch to your `HOME`
- All data you want to have copied into your `HOME` should be located under `/scratch/${USER}/scc_backup`
- It will be copied during the night to your `HOME` (`${HOME}/scc_backup`)
- You will get a mail about this process to your GWDG-Account
- If you want to get the mail to another mail address, put the address in `${HOME}/scc_backup/.mailaddress`

Section 3

Preparing the compilation environment with “modules”

The modules system

- “`module avail`” find a list of installed modules
- “`module list`” list of currently loaded modules
- “`module load software/version`”
- “`module purge`” unload all modules
- “`module unload software`” unload a single module
- Most of the modules just append or prepend a path to PATH and MANPATH variables.
- Or default variables to be found by compiler/configure scripts at compile time.

Section 4

Compiling Software

Why Compiling?

- GWDG cannot install all software required by users (see modules for what is available)
- Scientific software is often only available as source code
- Compiling means to create an executable – or a library – from the source code
- Compiling on the target system often yields better performance
- Prepackaged software typically requires administrator (root) privileges ...
  - ➥ (`sudo` or `su` won't work)
  - ➥ but you can use Singularity containers!

Getting and Unpacking the Source Code

- Source code is usually packaged as "tarball"
  - ➥ Look for file extensions "`tar.gz`", "`tar.bz2`", "`tgz`"
  - ➥ Naming convention is often `{NAME}-{VERSION}.tar.gz`
- If the tarball is available on the web use "`wget`" to download
- Use "`tar`" to unpack the tarball
  - ➥ Use "`tar xvzf`" for '`tar.gz`", "`tgz`"
  - ➥ Use "`tar xvjf`" for "`tar.bz2`"

Recipe: wget and tar

Using wget and tar to prepare the source code

```
> mkdir $HOME/build
> cd $HOME/build
> wget <tarball URL>
> tar xvzf <name-version>.tar.gz
> cd <name-version>
```

Compiling (or “Building”) the Software

- Standard method: “`./configure; make; [make check; make install]`”
- Without root privileges: “`--prefix`” at configuration
- For better performance: Use Intel compilers and MKL
- For MPI (distributed parallel) applications: Use Intel MPI

About “`--prefix`”

- “`--prefix`” is used to specify the base diretory for your software
- use “`./configure --prefix=DIR`” to install directly in DIR.
- e.g. “`./configure --prefix=$HOME/software/<name-version>`” to install into a software specific directory.

Recipe: Basic Building and Installing

Building and installing software into a specific directory

```
> cd $HOME; mkdir software
> cd $HOME/build/<name-version>
> ./configure --prefix=$HOME/software/<name-version>
> make -j 4; make check
> make install
> ln -s $HOME/software/<name-version>/bin/* $HOME/bin
> ln -s $HOME/software/<name-version>/lib/* $HOME/lib
> ln -s $HOME/software/<name-version>/include/* $HOME/include
```

Compilers

- The GNU compilers (`gcc`, `gfortran`) are the standard compilers in Linux
- Other compilers are often faster, especially for Fortran code
- Recommended for overall performance: Intel compilers (`icc`, `ifort`)
- Other compilers available at GWDG: PGI, Open64
  - ➥ For special cases and users willing to try several approaches for best performance

Recipe: Using Intel Compilers

Building and installing software with Intel compilers

```
> module load intel/compiler
> CC=icc; CXX=icpc; FC=ifort; F77=ifort; F90=ifort
> export CC CXX FC F77 F90
> ./configure --prefix=$HOME/software/<name-version>
> make -j 4; make check
> make install
```

Intel Math Kernel Library (MKL)

- A (shared) library is a collection of thematically related subroutines ready to use in a program
- The process of connecting a library to the (compiled) program is called linking
- Intel's Math Kernel Library provides performance optimized linear algebra and Fourier transform functions

Recipe: Using the MKL

Example: linking programs to MKL

```
> module load intel/compiler
> CC=icc; CXX=icpc; FC=ifort; F77=ifort; F90=ifort
> export CC CXX FC F77 F90
> module load intel/mkl
> export CPPFLAGS="-I${MKLROOT}/include -I${MKLROOT}/include/fftw"
> export LDFLAGS="-L${MKLROOT}/lib/intel64 -lmkl_intel_lp64\
> -lmkl_sequential -lmkl_core -lpthread -lm"
> ./configure --prefix=$HOME/software/<name-version>
> make -j 4; make check
> make install
```

Use Intel MKL Link Line Advisor!

`https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor`

MPI programs

- MPI programs are meant to run distributed across several computers
- They require to be linked to an MPI library
- The recommended MPI library at GWDG is Intel MPI
- Others available are OpenMPI (tested), MVAPICH, and MVAPICH2

Recipe: Building MPI programs

Building MPI programs with Intel MPI

```
> module load intel/compiler
> module load intel/mpi
> CC=mpiicc; CXX=mpiicpc; FC=mpiifort; F77=mpiifort; F90=mpiifort
> export CC CXX FC F77 F90
> module load intel/mkl
> export CPPFLAGS="-I${MKLROOT}/include -I${MKLROOT}/include/fftw"
> export LDFLAGS="-L${MKLROOT}/lib/intel64 -lmkl_intel_lp64\
> -lmkl_sequential -lmkl_core -lpthread -lm"
> ./configure --prefix=$HOME/software/<name-version>
> make -j 4; make check
> make install
```

Recipe: Building Rmpi for R

Preparation

```
> module load openmpi/gcc
> export OMPI_MCA_mtl=^psm
> echo $MPI_HOME
/cm/shared/apps/openmpi/gcc/64/1.10.1
> R
```

R command line

```
> install.packages("Rmpi", dependencies=TRUE,
    configure.args=c("--with-mpi=/cm/shared/apps/openmpi/gcc/64/1.10.1"
    ))
> install.packages(c("foreach", "doMPI"))
```

Table of Contents, Part II

1. Connecting to the frontends
2. The most important Linux commands
3. Preparing the compilation environment with “modules”
4. Compiling Software
5. Efficiently Submitting Jobs to the Cluster
6. Getting Help

Section 5

Efficiently Submitting Jobs to the Cluster

Using Slurm

Basic Concepts 1

Cluster A collection of networked computers intended to provide compute capabilities.

Node One of these computers, also called host or server.

frontend Special node provided to interact with the cluster. gwdu101, gwdu102, and gwdu103 in our case.

Job Program consisting of one or several parallel tasks.

Partition A group of nodes on which a job is intended to run

Batch System Management system distributing job tasks across job slots. We are changing from LSF to Slurm.

Submitting Job to the Cluser

- `srun/sbatch` submits information on your job to the batch system
  - ➥ What is to be done? (path to your program and required parameters)
  - ➥ What are its requirements? (e.g. partition, number of tasks, maximum runtime)
- Slurm matches the job's requirements against the capabilities of available job slots
- When suitable job slots are found the job is started
- Slurm prioritizes the jobs based on a number of factors.

Available Partitions

General purpose *meta*-partitions:

medium General purpose queue, well suited for large MPI jobs. Up to 1024 tasks, up to 48 hours runtime.

fat For SMP jobs. Up to 512 GB in one host. Otherwise as `mpi`.

fat+ For extreme memory requirements. Up to 2048GB per host and 120 hours, max 40 tasks.

Special purpose partitions:

gpu For jobs using GPU acceleration.

int For interactive jobs, i.e. jobs which require a shell or a GUI.

Available Partitions

Meta-partitions just resubmit into:

medium-fmz	medium nodes in the FMZ
medium-fas	medium nodes at the Faßbeg
fat-fmz	fat nodes in the FMZ
fat-fas	fat nodes at the Faßbeg
fat-fas+	fat+ nodes in the FMZ
fat-fmz+	fat+ nodes at the Faßbeg

Submitting a job

```
srun <parameters> <program>
```

common parameters

-p <partition>	partition.
-t <hh:mm:ss>	Maximum runtime. If this is exceeded the job is killed.

Interactive Jobs

srun: Interactive jobs

--x11 Adds X11 (GUI) forwarding. This requires that you connect to the frontend with `ssh -Y` and your local machine supports X-Windows.

-p int Use the interactive partition. In `int` the nodes have no slot limit. They will take jobs until their load crosses a specified threshold, so jobs start immediately.

--pty interactive mode

Try it!

Interactive X11 Job

Running Matlab

```
> ssh -Y gwdu101.gwdg.de
> module load matlab/2015a
> srun --x11 -p medium matlab
```

- The job will be dispatched and as soon as an available node is found and the Matlab interface will start.
- If you have your own license for Matlab then you need to place your `license.lic` file in `$HOME/.matlab/R2015a_licenses` directory (dependent on the version you are using).

Interactive Console Job

Running R interactively

```
> ssh gwdu101.gwdg.de
> srun --pty -p medium R
```

Non interactive Jobs

Problem

- if you have big jobs, you queue time will be long
- `srun` needs you to stay logged in
- jobs can run for days

Non interactive Jobs

Solution

```
sbatch <slurm options> jobscript
```

--mail-type=<TYPE> get mail notifications (type: `BEGIN`, `END`, etc.)

--mail-user=<address> Default: ${USER}@gwdg.de

-o/-e <file> Store job output in file (slurm-<jobid>.out by default). `%J` in the filename stands for the jobid.

sbatch: Using Job Scripts

A job script is a shell script with a special comment section.
The #SBATCH lines have to come first!

sbatch: Basic job script example

```
#!/bin/bash
#SBATCH -p medium
#SBATCH -t 10:00
#SBATCH -o job-%J.out

hostname
```

Submit with:

```
sbatch <script name>
```

Download examples

http://wwwuser.gwdg.de/~mboden/pkurs.tar.gz

LSF → Slurm cheat sheet

Table: Basic submission options

Description	LSF	Slurm
Submit job	bsub <job.sh	sbatch job.sh
Scheduler Comment	#BSUB -...	#SBATCH -...
Queue/Partition	-q <queue>	-p <partition>
Walltime	-W 48:00	-t 2-00:00:00
Stdout	-o <outfile>	-o <outfile>
Stderr	-e <errfile>	-e <errfile>
Interactive	-ISs /bin/bash	srun [...] --pty bash

Basic Concepts 2

Serial job Job consisting of one task using one job slot.

SMP job Job with shared memory parallelization (often realized with OpenMP), meaning that all tasks need access to the memory of the same node. Consequently uses several job slots **on the same node**.

MPI job Job with distributed memory parallelization, realized with MPI. Can use several job slots on several nodes and needs to be started with a helper program, e.g., `mpirun` or `srun`.

Resource selection

sbatch options for parallel (SMP or MPI) jobs.

-N <min>-<max>, --nodes=<min>-<max>
Minimum and maximum node count. You can also specify the exact number.

-n,--ntasks=<n>
Number of tasks (not equally distributed!)

--tasks-per-node=<n>
Tasks per node. If used with `-n` it denotes the maximum number of tasks per node.

-c,--cpu-per-task=<n>
CPUs per tasks. Useful for hybrid jobs

A note on -n vs. -c

Rule of thumb

- `-c` for single node jobs
- `-n` for MPI jobs

Rule of thumb 2

If you are unsure if your program uses MPI, then it does not.

A note on -n vs. -c

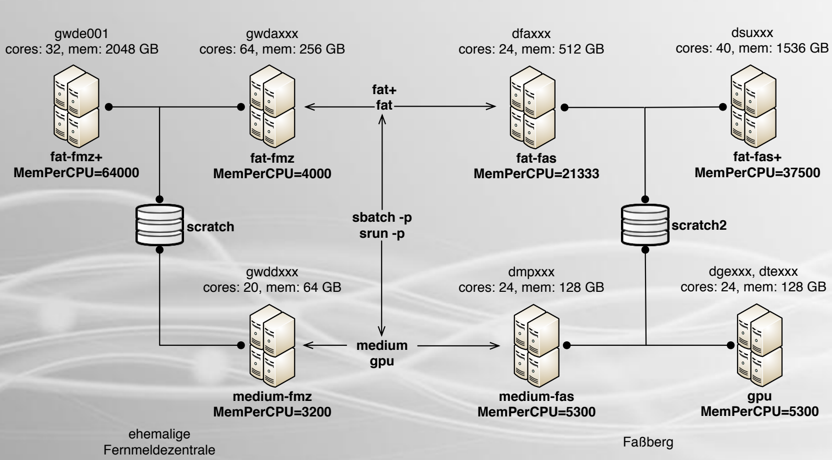
Rule of thumb

- `-c` for single node jobs
- `-n` for MPI jobs

Rule of thumb 2

If you are unsure if your program uses MPI, then it does not.

The GWDG Scientific Compute Cluster



Recipe: Submitting an MPI job

Distributing tasks in the medium partition

```
#SBATCH -p medium
#SBATCH -n 240
#SBATCH -o job-%J.out

module purge
module load intel/compiler intel/mkl intel/mpi namd

srun namd2 +setcpuaffinity apoa1.namd
```

Recipe: Submitting an MPI job

Distributing tasks in the `medium` partition

```
#SBATCH -p medium
#SBATCH -N 10
#SBATCH --ntasks-per-node 24
#SBATCH -o job-%J.out

module purge
module load intel/compiler intel/mkl intel/mpi namd

srun namd2 +setcpuaffinity apoa1.namd
```

Try it!

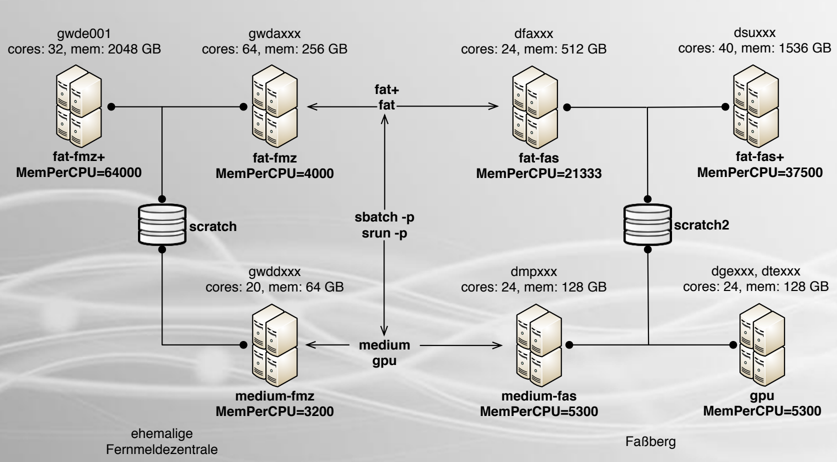
Job Disk Space Usage Options

/local Local hard disk of the node. SSD based on almost all nodes, therefore a very fast option for storing temporary data. Automatic file deletion. A temporary directory is created on all nodes at `$TMP_LOCAL`.

/scratch Shared scratch space, available on most nodes, but there are two instances (use `-C scratch` or `-C scratch2`). Very fast, no automatic file deletion, but also no backup! Files may have to be deleted manually when we run out of space.

$HOME Available everywhere, permanent, with backup. Personal disk space can be increased. Comparably slow.

The GWDG Scientific Compute Cluster



Recipe: Using /scratch

```
#!/bin/bash
#SBATCH -p fat
#SBATCH -n 64
#SBATCH -N 1
#SBATCH -C scratch
#SBATCH -t 1-00:00:00

export g09root="/usr/product/gaussian/g09/d01"
source $g09root/g09/bsd/g09.profile

MYSCRATCH=‘mktemp -d /scratch/${USER}/g09.XXXXXXXX‘
if [ ${MYSCRATCH} -a -d ${MYSCRATCH} ]; then
  export GAUSS_SCRDIR=${MYSCRATCH}
else
  export GAUSS_SCRDIR=/local
fi

g09 myjob.com myjob.log

if [ ${MYSCRATCH} -a -d ${MYSCRATCH} ]; then
  rm -rf ${MYSCRATCH};
fi
```

Memory management

sbatch options

--mem <size[K|M|G|T] > Memory per node.

--mem-per-cpu <size[K|M|G|T] > Memory per task.

- without options:
  - ➥ each partition has a `DefMemPerCPU` option
  - ➥ can be retrieved via `scontrol show partition <name>`

General remark on resources

Resource limitation

- Resources are limited to what you specify
- If you exceed the memory you specified, your job is automatically killed
- Your available cores are limited to the amount you specified

Partition selection

- only use fat and fat+ if you really need it
- you can directly submit to the underlying partitions

Exclusive jobs

- `#SBATCH --exclusive` in a job script denotes an exclusive job.
- An exclusive job uses all job slots (cores) of all its nodes.
- Using `--exclusive` together with `-N 1` reserves one complete node, independent of `-n`.
- You automatically get all the memory. Do not use `--mem` as that might limit you available memory.
- Disadvantage: Jobs with many nodes may wait longer, compared to those with exact `-n`.

LSF → Slurm cheat sheet

Table: Resources

Description	LSF	Slurm
Processes	-n #	-n #
One Host	-R "span[hosts=1]"	-N 1
Process Distribution	-R "span[ptile=<x>]"	--ntasks-per-node x
Exclusive Node	-x	--exclusive
Scratch	-R scratch[2]	-C scratch[2]

Recipe: MPI jobs with --exclusive

Using exclusive jobs to get full nodes

```
#SBATCH -p medium
#SBATCH -N 4
#SBATCH --ntasks-per-node=4
#SBATCH -o job-%J.out
#SBATCH --exclusive

module purge
module load intel/compiler intel/mpi

srun big_mpi
```

Recipe: Combine shared memory and MPI

Running hybrid jobs

```
#SBATCH -p medium
#SBATCH -N 5
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=6
#SBATCH -o job-%J.out

module purge
module load openmpi/gcc

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

srun hybrid_job
```

Longer or shorter jobs

The `--qos` parameter

- Default maximum runtime: 2 days
- `--qos= <qos>` can select a QoS
- Two extra QoS available:

short for shorter jobs (max. 2h), has higher priority, limited job slots

long longer jobs (max. 7d), limited job slots.

Miscellaneous Slurm Commands

sinfo Info about the system and partitions.
`-p <partition>, -t <state>`

squeue Show the job queue.
`-p <partition>, -u $USER`

scontrol show [partition|node|job] <x> where x should be a node name, jobID or partition name.

ssprio Priority information about pending jobs

sacct Get information about a job after it finished

```
-j <jobid>
--format=JobID,User,JobName,MaxRSS,Elapsed,Timelimit
```

sview GUI system and queue view (needs X11 forwarding)

`scancel`: Terminate your jobs

- Two use modes:
  1. `scancel <jobid>`: Kill job with specific jobid.
  2. `scancel <select options>`: Kill all jobs fitting the selection.
     Select option examples:
     - `-p <partition>`
     - `-u <$USER>`
     - `-s <state>`

Using the gpu partition

GPU parameters

--gres:gpu:<n> requests `n` GPUs of any kind

--gres:gpu:<type>:<n> requests `n` GPUs of `type`

- CPUs are evenly distributed for every GPU
- Available types are:
  - gtx980
  - gtx1080
  - k40
- See: `sinfo -p gpu --format=%N,%G`

Comments

- use jobarrays where possible (don't sbatch in a for loop)
- ignore BadConstraints Warning
- set a reasonable time limit (not just 2 days)
- use the short QOS where applicable

Recipe: Iterators in R

Using the foreach package

```
library(foreach)

ls<-foreach(i=1:100) %do% {
      norm=rnorm(100000)
      summ=summary(norm)
      summ
      }
ls
```

Recipe: Parallelization in R with doMPI

Using doMPI as backend for foreach

```
library(doMPI)

cl <- startMPIcluster()
registerDoMPI(cl)

ls<-foreach(i=1:100) %dopar% {
    norm=rnorm(100000)
    summ=summary(norm)
    summ
    }
ls

closeCluster(cl)
mpi.quit()
```

Recipe: MPI and R

Using R with doMPI in a batch job

```
#SBATCH -p medium
#SBATCH -n 20
#SBATCH -o job-%J.out

module load openmpi/gcc

srun Rscript "doMPI_script.R"
```

Task parallelization with GNU parallel

- GNU parallel distributes a set of tasks to a set of cores
- Requirement: No dependencies and side effects between tasks (*embarrassingly parallel*)

Using parallel to run a program with multiple input files

```
parallel ’cp {} .; g09 {/} {/.}.log’ \
::: $(find /usr/product/gaussian/g09/tests -name *.com -type f)

parallel ’cp {} .; if (eval "g09 {/} {/.}.log");
then echo {/} >> ok; else echo {/} >> failed; fi’ \
::: $(find /usr/product/gaussian/g09/tests -name *.com -type f)
```

Recipe: GNU parallel in a batch job

Multiple input files with parallel in a batch job

```
#!/bin/bash

#SBATCH -p medium
#SBATCH --qos=short
#SBATCH -n 20
#SBATCH -N 1
#SBATCH -t 02:00:00
#SBATCH -C scratch|scratch2

module load gaussian
mkdir /scratch/${USER}/g09_ptest
cd /scratch/${USER}/g09_ptest

parallel \
  ’cp {} .;
  if (eval "g09 {/} {/.}.log");
    then echo {/} >> ok;
    else echo {/} >> failed;
  fi’ \
::: $(find /usr/product/gaussian/g09/tests -name *.com -type f)
```

LSF → Slurm cheat sheet

Table: Basic submission options

Description	LSF	Slurm
Submit job	bsub <job.sh	sbatch job.sh
Scheduler Comment	#BSUB -...	#SBATCH -...
Queue/Partition	-q <queue>	-p <partition>
Walltime	-W 48:00	-t 2-00:00:00
Stdout	-o <outfile>	-o <outfile>
Stderr	-e <errfile>	-e <errfile>
Interactive	-ISs /bin/bash	srun [...] --pty bash

LSF → Slurm cheat sheet

Table: Resources

Description	LSF	Slurm
Processes	-n #	-n #
One Host	-R “span[hosts=1]”	-N 1
Process Distribution	-R “span[ptile=<x>]”	--ntasks-per-node x
Exclusive Node	-x	--exclusive
Scratch	-R scratch[2]	-C scratch[2]

LSF → Slurm cheat sheet

Table: Queues and Paritions

LSF	Slurm
-q mpi	-p medium
-q mpi-short	-p medium --qos=short
-q mpi-long	-p medium --qos=long
-q fat	-p fat
-q fat-short	-p fat --qos=short
-q fat-long	-p fat --qos=long
-q fat+	-p fat+
-q int	-p int
-q gpu	-p gpu

Section 6

Getting Help

Information sources

- man pages
- Slurm online help
  - ➥ For example: `sbatch --help`
- GWDG scientific compute cluster documentation
  - ➥ `https://info.gwdg.de/docs/doku.php?id=en:services:application_services:high_performance_computing:start`
- GWDG scientific compute cluster user wiki
  - ➥ `https://info.gwdg.de/wiki/doku.php?id=wiki:hpc:start`
- HPC announce mailing list
  - ➥ `https://listserv.gwdg.de/mailman/listinfo/hpc-announce`

Adding to the Wiki

- Everyone with a cluster account can add to the Wiki!
- Please inform us of all changes and new articles at parallel@gwdg.de.
- Please add the category "*Scientific Computing*" to all contributions regarding the cluster.

Using the GWDG Support Ticket System

- Write an email to *hpc@gwdg.de*
- State your user id (`$USER`)
- If you have a problem with jobs, **always** include:
  - ➥ Job IDs
  - ➥ standard output (`-o <file>`)
  - ➥ standard output (`-e <file>`)
- If you have a lot of failed jobs send at least two outputs. You may also list the jobid's of all failed jobs.
- If you don't mind us looking at your files, please state this in your request
  - ➥ You may limit your permission to specific directories or files

Digression: Directory Structure 1

- Convention: Executables are stored in “`bin`”, shared libraries in “`lib`” directories
- Directories in “`$PATH`” are searched for binaries, directories in “`$LD_LIBRARY_PATH`” for libraries
- Two strategies:
    1. Put everything directly under `$HOME/bin`, `$HOME/lib`
        - Easy to setup search paths
        - Difficult to remove software packages
    2. Put each software into its own subdirectory
        - Easy to remove software (with “`rm -rf` <subdirectory>”)
        - Difficult to setup search paths

Digression: Directory Structure 2

- Or combine both strategies:
  - ➥ Put each software in its own subdirectory
  - ➥ Use "`ln -s`" to link everything to $HOME/bin and $HOME/lib, respectively
  - ➥ Use "`export LD_LIBRARY_PATH=$HOME/lib:$LD_LIBRARY_PATH; export PATH=$HOME/bin:$PATH`" in your shell and scripts
  - ➥ Use "`find $HOME/bin $HOME/lib -xtype l -delete`" after removing software