

# Introduction to High-Performance Computing

Session 03

HPC Environment, Modules, Compiler,  
and Toolchains



# HPC User Environment

the user environment on a HPC cluster consists of:

- the operating system (OS)
  - e.g. RHEL Linux (all HPC systems in top500 have Linux-like OS)
  - basic functionality (login, create and edit files, ...)
- data storage
  - one or more file systems
  - temporary, short and long term storage
- software
  - scientific applications
  - libraries
  - compiler
- job scheduler

# File Systems

# HPC File Systems

- typically on a HPC system different file systems are available

[http://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/JUDAC/Filesystems/filesystems\\_node.html](http://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/JUDAC/Filesystems/filesystems_node.html)

Name	Description	Features
<b>\$TMPDIR</b> or <b>/scratch</b>	temporary storage provided on a per job basis, deleted after job often local disk or similar	very fast I/O, up to a few TB, no backup
<b>\$WORK</b>	temporary storage for job data, maybe kept after job, typically parallel file system attached to interconnect	fast, parallel I/O, up to PB, no backup
<b>\$DATA</b>	mid-term storage for job output, parallel filesystem or NFS	up to PB, maybe with backup
<b>\$HOME</b>	NFS storage, long term and secure, for program codes, initial conditions	few 100GB, full backup, snapshots
<b>\$ARCH</b>	permanent archive, storage for finished projects, tape library	few PB, possible slow read

# File Systems

[http://wiki.hpcuser.uni-oldenburg.de/index.php?title=File\\_system\\_and\\_Data\\_Management](http://wiki.hpcuser.uni-oldenburg.de/index.php?title=File_system_and_Data_Management)

- central **Enterprise Spectrum Scale storage (ESS)**
  - used for home, data, group and offsite directories
  - NFS mounted over 2x 10Gb Ethernet
  - full backup and snapshot functionality
  - can be mounted on local workstation using SMB
- shared **parallel storage (GPFS)**
  - used work directory only
  - data transfer over FDR Infiniband
  - no backup
  - can also be mounted on local workstation using SMB
- **local disks or SSDs for scratch**
  - CARL compute nodes have local storage (1-2TB per node)
  - EDDY compute nodes have 1GB RAM disk (for compatibility)
  - usable during job run time

# Directory Structure

- on every filesystem (**\$HOME**, **\$DATA**, **\$WORK**) users will have their own subdirectory

- e.g. for **\$HOME**

```
drwx----- abcd1234 agsomegroup /user/abcd1234
```

- default permissions prevent other users from seeing the contents of their directory
- user can give permissions to others to access files or subdirectory as needed (**user's responsibility**)
- file and directory access can be based on primary (the working group) and secondary (e.g. the institute) Unix groups
- **recommendation**: keep access restricted on **\$HOME** and if needed share files/dirs. on **\$DATA** or **\$WORK**

[https://wiki.hpcuser.uni-oldenburg.de/index.php?title=File\\_system\\_and\\_Data\\_Management#Managing\\_access\\_rights\\_of\\_your\\_folders](https://wiki.hpcuser.uni-oldenburg.de/index.php?title=File_system_and_Data_Management#Managing_access_rights_of_your_folders)

# File Systems

File System	Env. Variable	Path	Used for
Home	<b>\$HOME</b>	<b>/user/abcd1234</b>	critical data that cannot easily be reproduced (program codes, initial conditions, results from data analysis)
Data	<b>\$DATA</b>	<b>/nfs/data/abcd1234</b>	important data from simulations for on-going analysis and mid term (project duration) storage
Work	<b>\$WORK</b>	<b>/gss/work/abcd1234</b>	data storage for simulation runtime, pre- and post-processing, short term (weeks) storage
Scratch	<b>\$TMPDIR</b>	<b>/scratch/&lt;job-dir&gt;</b>	temporary data storage during job runtime
Offsite	<b>\$OFFSITE</b>	<b>/nfs/offsite/user/abcd1234</b>	long term storage for inactive data, only available on login nodes

- **\$HOME**, **\$DATA** and **\$OFFSITE** have backup for disaster recovery and daily snapshots for file recovery
- quotas are use on all file systems to limit the amount of data that can be stored by a user

# Quotas

[https://wiki.hpcuser.uni-oldenburg.de/index.php?title=File\\_system\\_and\\_Data\\_Management#Quotas](https://wiki.hpcuser.uni-oldenburg.de/index.php?title=File_system_and_Data_Management#Quotas)

- on every file system default quotas are in place
    - **\$HOME**, **\$DATA** and **\$OFFSITE** have 1TB, 20TB and 12.5TB, respectively
    - the number of files is also limited (**\$HOME**: 500k, **\$DATA**: 1M, **\$OFFSITE**: 250k)
    - **\$WORK** has 25TB and no limit on number of files
    - maybe increased upon request (if resources are available)
  - soft and hard quotas
    - in addition to the soft limit above, there is also a higher hard limit
    - if usage is over soft quota a grace period of 30 days is triggered
    - after grace period no data can be written to the affected directory by user
- check your usage with **lastquota** and clean up your data on work regularly



# Group Directories

- group directories are available upon request
  - storage on the ESS
  - can be mounted via SMB (only version 2 or better)
  - path: **\$GROUP** or **/nfs/group/agyourgroup**
  - should be used for data shared among members of the same group, in particular to avoid multiple copies of the same file
  - group leader is owner of directory
  - default rights are set so that anyone in group can read and write to group directory

# File System Shares

[https://wiki.hpcuser.uni-oldenburg.de/index.php?title=Local\\_Mounting\\_of\\_File\\_Systems](https://wiki.hpcuser.uni-oldenburg.de/index.php?title=Local_Mounting_of_File_Systems)

- you can mount your **\$HOME**, **\$DATA** and **\$WORK** as well as **\$OFFSITE** and **\$GROUP** directories on your local workstation

- server address for mounting are

**\$HOME**            `//smb.uni-oldenburg.de/hpc_home`

**\$DATA**            `//smb.uni-oldenburg.de/hpc_data`

**\$WORK**            `//smb.hpc.uni-oldenburg.de/hpc_work`

**\$OFFSITE**        `//smb.uni-oldenburg.de/hpc_offsite`

**\$GROUP**          `//smb.uni-oldenburg.de/<groupname>`

- for Windows connect a network drive (and replace “/” with “\”)
- for Linux add information in `/etc/fstab`

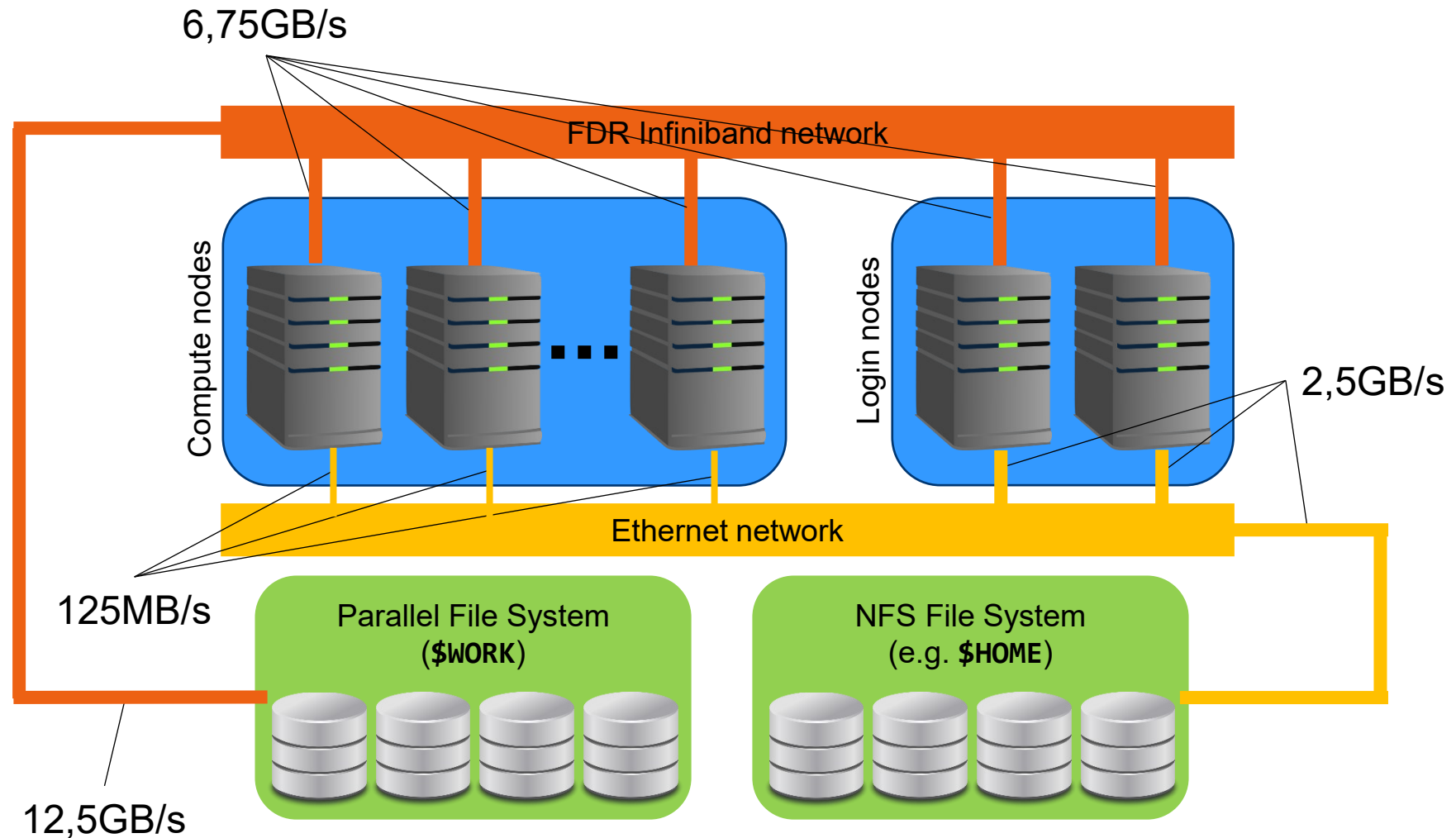
# File System Use

- applications with high I/O demands can put a lot of stress on the used file system
- I/O-performance depends on the I/O profile
  - I/O with few but large files is better than many small files
  - sequential I/O is better than random access
- pick the right file system for your I/O profile
  - local disks or SSDs are best for I/O with small block sizes
  - parallel files system (**\$WORK**) is best for large files and parallel I/O
  - **\$HOME** and **\$DATA** (and all NFS mounted directories) should be avoided for I/O at runtime

simple I/O performance tests can be done with **dd**

[https://www.thomas-krenn.com/de/wiki/Linux\\_I/O\\_Performance\\_Tests\\_mit\\_dd](https://www.thomas-krenn.com/de/wiki/Linux_I/O_Performance_Tests_mit_dd)

# File System Bandwidth Limits



Note, that the maximum bandwidth is shared for the whole node/cluster

# Best Practices for File System Use

- if your job is doing heavy I/O use **\$WORK** or **\$TMPDIR**
  - I/O bandwidth to **\$WORK** is more than 10GB/s (shared for the whole cluster), compared to 125MB/s at most to **\$HOME** and **\$DATA**
  - try to use parallel I/O and avoid using many small files
  - **\$TMPDIR** is best for small files and random access (in particular in the partitions **mpcb.p** and **mpcp.p**)
- keep your data on **\$WORK** while it is being processed
  - data that is currently not needed can be moved to **\$DATA**
  - consider creating compressed archives and organise your data
  - only keep important data and delete as much as possible when a project is finished
  - use **\$GROUP** if you frequently need to share data within your group to avoid unnecessary copies of data

# Final Remarks File Systems

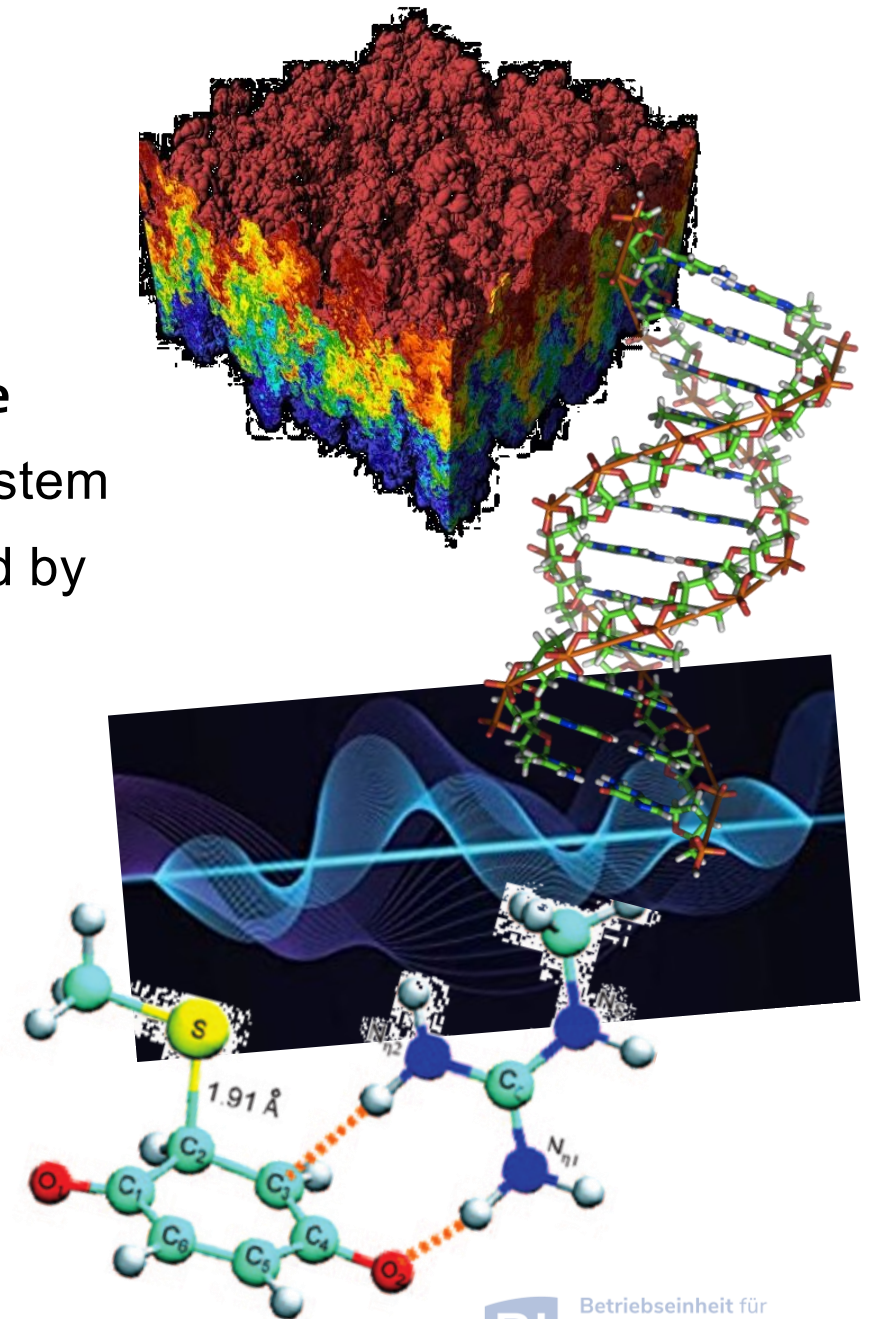
- setting file permissions
  - add execute (**x**) permission to directories to allow **cd**
  - add read (**r**) permission to directories to allow **ls**
  - avoid adding write (**w**) permission for group or others on directories (you cannot change ownership of files)
- checking quotas
  - use the **lastquota** command to find out how much disk space you are using
  - also weekly e-mails to all users
- reducing your file system footprint

```
$ tar cf project.tar project/      # create tar-file
                                   # to reduce number of files
$ zstd --rm project.tar            # compress files for smaller size
```

# Software and Modules

# Using Installed Software

- software is installed centrally on the cluster
  - main path: `/cm/shared/uniol/software`
  - installed applications are optimized for system
  - can be used by all users (unless restricted by license terms)
  - own software can be installed, too, e.g. in `$HOME`
- installed software includes
  - compilers
  - libraries (MPI, numerical libraries,...)
  - scientific application
  - overview and help in the HPC wiki





# Environment Modules

- Linux settings are defined by environment variables

```
$ echo $HOME          # home directory
/user/abcd1234
$ echo $PATH          # where to look for applications
/cm/shared/apps/slurm/current/sbin:/cm/shared/apps/slurm/current/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/ibutils/bin
$ env                 # full list
HOSTNAME=hpc1002
TERM=xterm
. . .
```

- applications require correct settings of environment variables (e.g. the **PATH**-variable)
- environment modules are used to make predefined changes to the environment using the **module**-command

# The `module`-command

- the environment settings for installed applications are managed using modules
  - example: loading a module for **SAMtools**

```
$ module load SAMtools/1.9-GCC-8.3.0
$ samtools --version
samtools 1.9
Using htlib 1.9
Copyright (C) 2018 Genome Research Ltd.
$ which samtools
/cm/shared/uniol/software/8.3/SAMtools/1.9-GCC-8.3.0/bin/samtools
[abcd1234@carl]$ echo $PATH
/cm/shared/uniol/software/8.3/SAMtools/1.9-GCC-8.3.0/bin:/cm/shared/...
```

- after the module is loaded, the application can be used
- the variable **\$PATH** has been modified (among other things)

# The `module`-command

- available modules can be displayed and searched for
  - displaying all modules (also work with `spider`)

```
$ module available
----- /cm/shared/uniol/modules/8.3/bio -----
. . .
SAMtools/0.1.19-foss-2019b      SAMtools/1.9-GCC-8.3.0 (L,D)
. . .
```

- very long list of available modules
  - modules can be highlighted with (L) for loaded and/or with (D) for default
  - add application name to get shorter list
- show currently loaded modules

```
$ module list
Currently Loaded Modules:
 1) slurm/current  2) hpc-env/8.3  3) GCCcore/8.3.0  ...
```

# The module-command

[https://wiki.hpcuser.uni-oldenburg.de/index.php?title=User\\_environment\\_-\\_The\\_usage\\_of\\_module\\_2016](https://wiki.hpcuser.uni-oldenburg.de/index.php?title=User_environment_-_The_usage_of_module_2016)

- find modules

```
$ module available [module-name]  
$ module spider [module-name]
```

- list all modules [with given module name]
- both commands are case-insensitive and understand regular expressions when using option `-r`

- load/unload

```
$ module load <module-name>  
$ module remove <module-name>
```

- to return to a default state

```
$ module restore
```

- information about modules

```
$ module list  
$ module help <module-name>  
$ module spider <module-name>
```

# The `module`-command

- you can define, save, and restore your own module collections

```
$ module load SAMtools Python      # to load some modules
$ module save mycollection          # to save currently loaded modules
Saved current collection of modules to: "mycollection"
$ module purge                      # unload all modules
$ module restore mycollection        # restore previously saved collection
Restoring modules from user's mycollection
$ module list
Currently Loaded Modules:
  1) slurm/current                7) ncurses/6.1-GCCcore-8.3.0    13) Tcl/8.6.9-GCCcore-8.3.0
  2) hpc-env/8.3                 8) bzip2/1.0.8-GCCcore-8.3.0    14) SQLite/3.29.0-GCCcore-8.3.0
  3) GCCcore/8.3.0              9) XZ/5.2.4-GCCcore-8.3.0      15) GMP/6.1.2-GCCcore-8.3.0
  4) zlib/1.2.11-GCCcore-8.3.0 10) cURL/7.66.0-GCCcore-8.3.0   16) libffi/3.2.1-GCCcore-8.3.0
  5) binutils/2.32-GCCcore-8.3.0 11) SAMtools/1.9-GCC-8.3.0      17) OpenSSL/1.1.1d-GCCcore-8.3.0
  6) GCC/8.3.0                  12) libreadline/8.0-GCCcore-8.3.0 18) Python/3.7.4-GCCcore-8.3.0
```

- if no name is given for `save` or `restore`, the collection `default` is used

# The `m1`-command

- the `module`-command (as well as some subcommands) can be abbreviated
  - any command `module subcmd` can be replaced with `m1 subcmd`
  - the `m1`-command also may have different meanings depending on the context

```
$ m1 # same as module list
Currently Loaded Modules:
  1) slurm/current    2) hpc-env/8.3
$ m1 av # same as module available
. . .
----- /cm/shared/uniol/modules/core -----
  slurm/current (L)    hpc-env/8.3    (L)
. . .
$ m1 SAMtools # same as module load SAMtools
```

# hpc-env Modules

- in the module core-section you can find a number of **hpc-env** modules

```
$ module available
----- /cm/shared/uniol/modules/core -----
hpc-env/6.4 (D) hpc-env/8.3 (L)
hpc-env/8.1 hpc-uniol-env
hpc-env/8.2 hpc-uniol-new-env
```

- these modules provide some basic settings (e.g. `$DATA`, loading the Slurm module) and make a specific module stack available
- the version corresponds to a specific **GCC** version and all modules in the stack are based on this **GCC** version
- the non-version modules are older and not based on a specific **GCC**
- most software is installed in **hpc-uniol-env**, **hpc-env/6.4** and **hpc-env/8.3**
- if you login you will find **hpc-uniol-env** loaded, this can be changed (e.g. with `module save`)
- only one **hpc-env** module can be loaded at any time

# Modules

- why use modules
  - modules allows multiple versions of the same application to be installed
  - modules change all the environment settings as needed
  - modules know about dependencies and conflicts
- modules and jobs
  - modules have to be loaded within a job script (as needed)
  - modules loaded when the job is submitted are remembered by SLURM (**but** you may submit a job later again with different modules loaded)



# Compiler, Libraries and Toolchains

# Compiler

- different compilers available (from vendors and also open-source)

```
----- /cm/shared/uniol/modules/compiler -----  
  CUDA-Toolkit/8.0.44  
  GCC/4.9.4-2.25  
  GCC/5.4.0-2.26  
  GCC/6.2.0-2.27 (D)  
  LLVM/3.8.1-goolf-5.2.01  
  LLVM/3.8.1-intel-2016b  
  LLVM/3.9.0-intel-2016b (D)  
  NAG_Fortran/5.2  
  PGI/12.10  
  PGI/15.10  
  PGI/16.10 (D)  
  icc/2016.3.210  
  ifort/2016.3.210
```

- Intel compiler usually gives very good performance (icc and ifort)
- using different compilers may help to better understand your code
- some compiler support special hardware (e.g. GPUs by PGI)
- always load one compiler (don't use OS GCC)

# Example: RandomWalk.cpp

- download the code `RandomWalk.cpp` (and the other RandomWalk files) from Stud.IP
  - the code simulates a 2d random walk, each step of length one in random direction, prints out distance from start after N steps
  - expected distance is  $\sqrt{N}$
  - compile with GCC or ICS
    - \$ `gcc RandomWalk.cpp -o RandomWalk`
    - \$ `icpc RandomWalk.cpp -o RandomWalk`
  - run with one argument for seed, e.g.
    - \$ `./RandomWalk 12345`
  - timing with
    - \$ `time ./RandomWalk 12345`

# Libraries

- libraries are available as modules
  - numerical libraries provide optimized solutions of general problems

```
----- /cm/shared/uniol/modules/numlib -----  
ATLAS/3.10.2           Octave/4.0.3  
Armadillo/7.500.1     OpenBLAS/0.2.19  
CLHEP/2.2.0.4-intel-2016b  Qhull/2015.2  
Eigen/3.2.9           ScaLAPACK/2.0.2  
FFTW/3.3.5-gompi-5.2.01  SuiteSparse/4.5.3  
FIAT/1.6.0-intel-2016b  cuDNN/5.1-CUDA-8.0.44  
GMP/6.1.1 (D)         cvx/2.1  
GSL/2.1               imkl/11.3.3.210  
Hypre/2.11.1         leda/6.3  
LinBox/1.4.0         maple/18  
MATLAB/2016b        maple/2016 (D)  
MPFR/3.1.4          stata/13  
NTL/9.8.1
```

# Example: Matrix-Matrix Multiplication

- basic linear algebra is available in many different numerical libraries
  - OpenBLAS, Lapack, MKL, ...
  - Basic Linear Algebra Subprograms (BLAS) contain e.g. a General Matrix Multiplication (gemm) of the form:
$$C = \alpha A \cdot B + \beta C$$
  - original version written in Fortran
  - used in the `mm.cpp` example (`cblas_dgemm` is the C-interface for double precision `gemm`)

```
// A, B, and C are objects of class SqMatrix but A[0] etc. are  
// pointers to first element in matrix which is what dgemm expects  
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,  
            n, n, n, alpha, A[0], n, B[0], n, beta, C[0], n);
```

# Toolchains

[http://easybuild.readthedocs.io/en/latest/eb\\_list\\_toolchains.html](http://easybuild.readthedocs.io/en/latest/eb_list_toolchains.html)

- some modules are called toolchains
  - provide a collection of compiler, MPI, and/or numerical libraries

```
----- /cm/shared/uniol/modules/toolchain -----  
foss/2016b      gomp/5.2.01      iimpi/2013b     intel/2016b (D)  
gimpi/6.2016   gomp/6.2.01 (D)  iimpi/2016b (D)  
gomp/4.1.10    goolf/5.2.01     intel/2013b
```

- examples:
  - foss: free and open-source software, currently GCC, OpenMPI, OpenBLAS, ScaLAPACK, FFTW
  - fosscuda: same as foss with CUDA support
  - gomp: GCC, OpenMPI
  - intel: Intel compilers, MPI, MKL

# Example: Matrix-Matrix Multiplication

- the code `mm.cpp` uses **OpenBLAS** which is included in the `foss-toolchain`

```
$ ml restore
Resetting modules to system default
$ make clean
rm mm mm.o
$ make
g++ -O2 -c mm.cpp
mm.cpp:7:19: fatal error: cblas.h: No such file or directory
  #include "cblas.h"
                ^
compilation terminated.
make: *** [mm.o] Error 1
$ ml foss
$ make
g++ -O2 -c mm.cpp
g++ -O2 -o mm mm.o -lopenblas
```

# Example: Matrix-Matrix Multiplication

- alternatively, the code can be compiled with Intel MKL
  - requires some code change (different header file)
  - requires changes to Makefile (different libraries to link)
  - result: code runs faster by 25%

```
$ sacct -j 2591679 -o JobID,JobName,Partition,Elapsed,MaxRSS,State,ExitCode
      JobID      JobName  Partition  Elapsed      MaxRSS      State  ExitCode
-----
2591679      run_mm.job  car1.p    00:06:21      7336K      COMPLETED  0:0
2591679.bat+  batch      00:06:21      7336K      COMPLETED  0:0
2591679.0      mm        00:00:33      37600K      COMPLETED  0:0
2591679.1      mm        00:00:32      113412K     COMPLETED  0:0
2591679.2      mm        00:00:33      412420K     COMPLETED  0:0
2591679.3      mm        00:00:32      1592064K    COMPLETED  0:0
2591679.4      mm        00:04:09      6310656K    COMPLETED  0:0
```



# Advanced Job Management

# Running Many Jobs

- you may need to run a program on the HPC cluster many times with different parameters
- example: run program **isPrime** several (M) times
  - different input parameter (value to test) every time
  - all input parameters are in file **parameter.dat**
- strategies:
  - simple approach: make M copies of job script, modify the input parameter in every file, could be automatized, **not recommended**
  - loop approach: use a single job script with a loop
  - **job array approach:** use **Slurm's job array functionality**

# Running Many Jobs: Job Arrays

- job or task arrays are defined by Slurm option

```
$ cat array_job.sh
. . .
### settings for job array
#SBATCH --array 1-10:1%4      # define task array
                               # format range:step%tasklimit
. . .
```

- range of tasks can be defined as **from-to:increment**
- multiple ranges with comma-separated list
- limiting the number of parallel tasks is possible with **%tasklimit** (when tasks have high resource requirements)

# Job Arrays

- the same job script is executed for each task in the array
- additional variable **SLURM\_ARRAY\_TASK\_ID** is provided

```
$ cat prime_job.sh
. . .
# get parameter from file for each task
parameter=$(awk "NR==$SLURM_ARRAY_TASK_ID {print \$1}" parameter.dat)
echo -n "Task $SLURM_ARRAY_TASK_ID tested if $parameter is prime? "
./isPrime $parameter
. . .
```

the task-ID can be used

- e.g. to number input or output file
- read specific line from input file (as in the example above)
- computations in bash (limited)

# Job Arrays

- job array are a powerful tool for task parallel jobs
  - to be preferred over submitting many individual jobs
  - each tasks in a job array should be sufficiently long (e.g. > 1h), due to the overhead for a single task
- requires some strategy for post-processing
  - often Linux tools can do the trick, more complex tasks may require post-processing script in e.g. Python
- additional environment variables for first and last task
  - however, tasks may not complete in the correct order
  - alternatively job dependencies can be used

# awk

<http://www.gnu.org/software/gawk/manual/gawk.html>

- powerful Linux tool that searches the lines of a file for patterns and performs an action on that line
  - similar tools are **grep** (pattern matching) and **sed** (streaming edit)
  - works well with data files (tables)
  - uses a **C**-like syntax
- example: **prime.awk**
  - reads all output files from the job array (using **cat** to combine them)
  - counts yes and no answers
  - prints final result

# Job Arrays: Do's and Don'ts

- **do** use job arrays whenever you run many almost identical jobs (e.g. parameter studies)
  - **don't** automatically submit 100s or 1000s of jobs simultaneously
- **do** limit the number of parallel running tasks if individual jobs require a lot of resources
  - there is a setting **MaxJobsPerAccount=250** limiting the maximum number of running jobs for your group
- **don't** parallelize very short jobs in a job array
  - individual tasks should run for minutes at the very least, better for hours
  - group tasks for longer job run time and parallelize for groups
- **do** test
- **don't** run tasks if you do not need to

# Running Many Jobs

- you may need to run a program on the HPC cluster many times with different parameters
- example: run program **isPrime** several (M) times
  - different input parameter (value to test) every time
  - all input parameters are in file **parameter.dat**
- strategies:
  - simple approach: make M copies of job script, modify the input parameter in every file, could be automatized, **not recommended**
  - loop approach: use a single job script with a loop
  - job array approach: use Slurm's job array functionality
  - **parallel approach: use the Linux command **parallel****



# The `parallel` Command

<https://www.gnu.org/software/parallel/>

- the `parallel` command is a shell tool for executing command in parallel
  - available on the cluster as module

```
$ module load parallel
```

- example: run `RandomWalk_task.sh` ten times in parallel

```
$ parallel -N 1 -j 4 --joblog parallel.log ./RandomWalk_task.sh {1} ::: {1..10}
Running RandomWalk with seed 2000 on hpc1001
Seed = 2000
Running RandomWalk with seed 4683 on hpc1001
Seed = 4683
```

# The `parallel` Command

<https://www.gnu.org/software/parallel/>

- the `parallel` command can be used in many different ways
  - in the example

```
$ parallel -N 1 -j 4 --joblog parallel.log ./RandomWalk_task.sh {1} ::: {1..10}
```

- a range is given with `::: {1..10}`, alternatively use `::: $(seq 10)`
- with `{}` or `{n}` the value of the argument is passed to the task
- the option `-N` defines how many arguments are passed to the task
- the option `-j` defines how many tasks can run in parallel
- an additional logfile is created with the option `--joblog <logfile>`

The use of the `parallel` command should be cited.

# Running Many Jobs

- several approaches can be used to run many tasks on the cluster

**loop approach:** single job, post-processing could be included, only serial processing, best used if tasks are short (minutes) and total runtime not too long

**job arrays:** single **sbatch**, one job per task, parallel processing on available resources, tasks should run >1h, limit maximum number of tasks running, overhead for starting tasks

**parallel approach:** single job, distributed resources can be used, better control over used resources, little overhead for starting tasks, scripts can be adapted easily

also see [https://wiki.hpcuser.uni-oldenburg.de/index.php?title=How to Manage Many Jobs](https://wiki.hpcuser.uni-oldenburg.de/index.php?title=How_to_Manage_Many_Jobs)

# Handling Many Output Files

- jobs or job arrays with many tasks in general also generate many output files
  - may degrade performance, use `$WORK` or `$TMPDIR`
  - you may hit your file number quota
- if possible try to generate single output file
  - might be difficult at job runtime but can be done afterwards

```
$ tar zcf RandomWalk.data.tar.gz RandomWalk*.data # create a compressed tar-file
$ rm RandomWalk_*.data # delete small files
$ tar -zxOf RandomWalk.data.tar.gz | awk -f RandomWalk.awk | sort -g
#steps  expected          mean          std
10      3.162278                2.089732    1.232151
100     10.000000               9.985299    6.881550
. . .
```

# Job Dependencies

[https://wiki.hpcuser.uni-oldenburg.de/index.php?title=How\\_to\\_Use\\_Job\\_Dependencies](https://wiki.hpcuser.uni-oldenburg.de/index.php?title=How_to_Use_Job_Dependencies)

- jobs can have a dependency on another job
  - option: **--dependency** or short **-d**
  - format: **--dependency <type>:<jobID>[,<jobID>...]**  
where **<type>** can be one of: **afterany**, **afterok**, **afternotok**
- a job with a dependency will not start until the predecessors have completed with the demanded status
  - careful: make sure exit status is correct for your needs
  - additional type **after**: jobs starts once predecessors have started
- a special dependency type is **singleton**
  - all jobs with the same job name and from the same user have to complete first, can be used to collect results

# Exercises

# Exercises

1. Try to compile and run the **mm-code**
  - Try to use different toolchains
2. Try to run a job script for an application
  - See next slide for specific example Orca
3. Try to run and compile the **RandomWalk-code**
  - Try different compilers
  - Run multiple times as job array
  - Run multiple time using the Linux **parallel** command

# Example: ORCA Job

- examples for using installed software on the cluster can be found in the HPC wiki
  - e.g. ORCA (chemistry)  
[http://wiki.hpcuser.uni-oldenburg.de/index.php?title=ORCA\\_2016](http://wiki.hpcuser.uni-oldenburg.de/index.php?title=ORCA_2016)
  - download the files for serial runs and submit job
  - use ORCA 3.0.3
  - the job script is rather complex
    - module is loaded
    - files are copied to \$TMPDIR
    - application is started from \$TMPDIR
    - output is copied to \$SLURM\_SUBMIT\_DIR



# Example: Many Random Walks

- task: run RandomWalk several ( $M=10$ ) times to get the average distance after  $N$  steps from multiple runs
  - different seed every time, provided in a file
  - write job script run as one or more SLURM jobs
  - think how to analyse data from  $M$  completed runs
    - how to combine the output of  $M$  tasks
    - maybe with awk script?