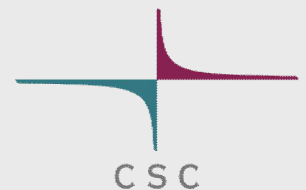
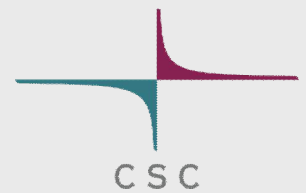


# **Bioinformatics with large data-sets and the database service of CSC**

**Kimmo Mattila**  
**Ari-Matti Sarén**



# CSC Environment



# Louhi

## ➤ **Cray XT4/XT5 Massively Parallel Processor (MPP) supercomputer**

- quad-core 2.3-GHz AMD Opteron 64-bit processors
- 9424 cores
- 1 GB or 2 GB memory/core
- SeaStar2 interconnects

## ➤ **Meant for jobs that parallelize well**

- project resources only after scalability test
- normally 64-512 cores/job
- can be increased for Grand Challenge projects

## ➤ **Louhi user's guide**

- [http://www.csc.fi/english/pages/louhi\\_guide](http://www.csc.fi/english/pages/louhi_guide)



# Murska

## ➤ **HP CP4000 BL ProLiant supercluster**

- dual-core 2.6 GHz AMD Opteron 64-bit processors
- 2176 cores
- 128 cores 8 GB, 512 cores 4 GB, 512 cores 2 GB, 1024 cores 1 GB
- InfiniBand (4x DDR) network

## ➤ **Meant for serial and mid-size parallel jobs**

- 1-256 cores/job (1-32 typical)

## ➤ **Murska user's guide:**

- [http://www.csc.fi/english/pages/murska\\_guide](http://www.csc.fi/english/pages/murska_guide)



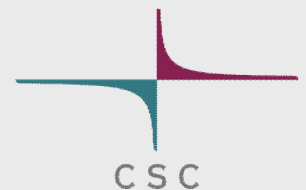
# Hippu

- **2x HP ProLiant DL785 G5**
  - quad-core 2.5 GHz AMD Opteron 8360 SE 64-bit processors
  - 64 cores
  - 512 GB shared memory (hippu1), 256 GB shared memory (hippu2)
- **Meant for interactive jobs**
  - job length not limited
  - no queue system installed
- **Hippu user's guide:**
  - [http://www.csc.fi/english/pages/hippu\\_guide](http://www.csc.fi/english/pages/hippu_guide)

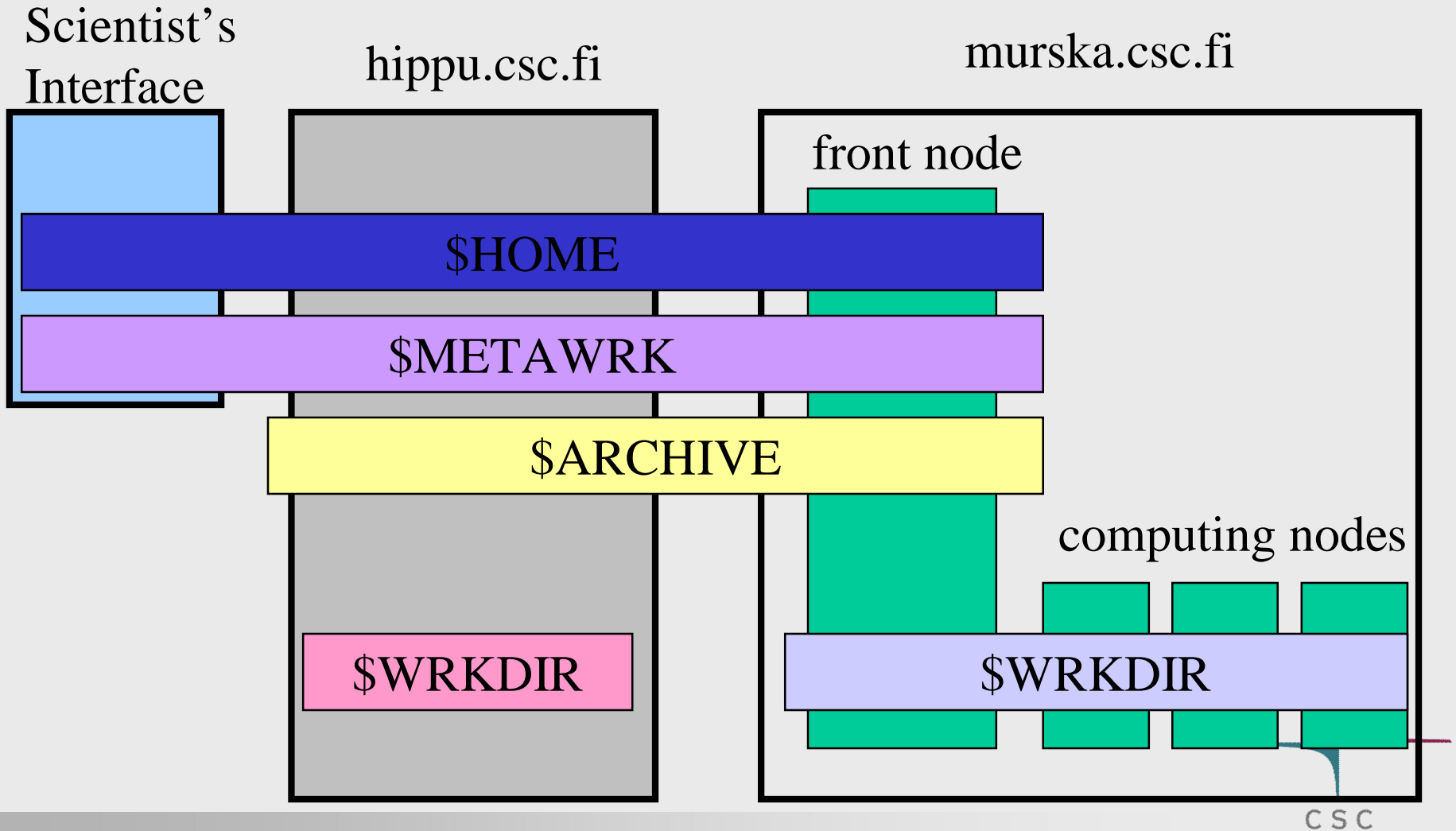


# File systems and directories

<b>Directory</b>	<b>Intended use</b>	<b>Quota</b>	<b>Backup</b>	<b>Shared</b>
<b>home</b> <b>\$HOME</b>	initialization scripts, source codes, etc.	Limited	Yes	Yes
<b>tmp</b> <b>\$TMPDIR</b>	Run-time, temporary files, stored for 1 day	Unlimited	No	Node specific
<b>wrk</b> <b>\$WRKDIR</b>	temporary files, stored for 7 days	Unlimited	No	Server specific
<b>metawrk</b> <b>\$METAWRK</b>	Program development, analysis of results, stored for 30 days	Unlimited	No	Yes
<b>userappl</b> <b>\$USERAPPL</b>	Your own frequently used program installations	Limited	Yes	Server specific
<b>archive</b> <b>\$ARCHIVE</b>	Long-term storage, 22 months Note! Tape archive	Unlimited	Two tape copies	Yes



# Visibility of personal directories at CSC



# Using the archive server

## ➤ **Tape based storage**

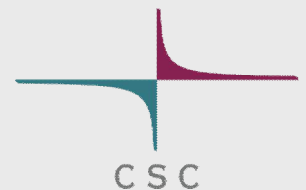
- very large capacity (currently 560 TB)
- retrieving the files may take a few minutes

## ➤ **Can be accessed with normal unix commands: `cp`, `mv`, `rm` etc.**

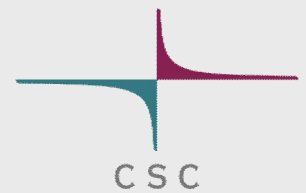
- mounted as `$ARCHIVE` on Hippu and on log-in nodes of Murska and Louhi

## ➤ **Avoid archiving small individual files on the server**

- If you have to archive small files, you should first combine them to tar format and compress

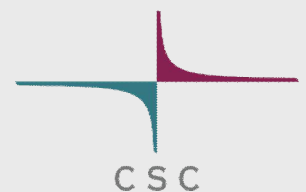


# Data handling



# Some brief generalizations:

- **It's usually faster to move one large file than many small ones**
- **On the other hand you should avoid too large files**
  - it's nicer to re-send one 1 GB chunk than the whole 100 GB file
- **Consider compression**
- **Prefer file formats that have checksums or other verification mechanisms**
- **Data should also be packaged for saving in \$ARCHIVE**



## ➤ tar

- concatenates files but does not compress
- preserves directory structure
  - many compression programs don't handle directories well/at all
  - answer: first tar, then compress
- making a tar package:

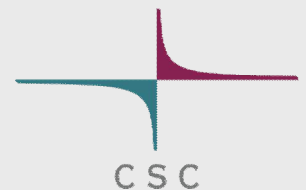
```
tar cf myfolder.tar myfolde
```
- opening a tar package:

```
tar xf myfolder.tar
```
- checking tar file contents

```
tar tf myfolder.tar
```

# File compression

- **File compression/decompression takes time, but saves time on upload/download**
  - net gain depends on data size
- **Files used in bioinformatics (sequences, pedigree files etc) are often text-based and compress well (to ~30% of original size)**
- **Compressed file formats typically include checksums**
  - if you can uncompress the file without error messages you know your data is intact
- **Commonly used compression programs:**
  - zip
  - gzip
  - bzip



➤ **zip**

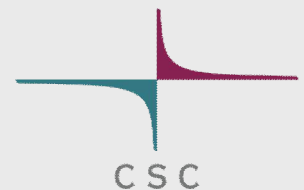
- compressing  
`zip myfiles.zip file1 file2`
- uncompressing  
`unzip myfiles.zip`
- leaves original file intact

➤ **gzip**

- compressing  
`gzip myfile`
- uncompressing  
`gunzip myfile.gz`
- replaces original file with the compressed file

➤ **bzip2**

- slightly better compression ratio than zip/gzip
- mostly linux specific
- compressing  
`bzip2 myfile`
- uncompressing  
`bunzip2 myfile.bz2`
- replaces original file with the compressed file



➤ **Linux**

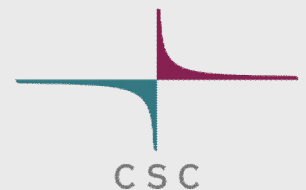
- tar, zip, gzip, bzip2 part of most standard distributions

➤ **Windows**

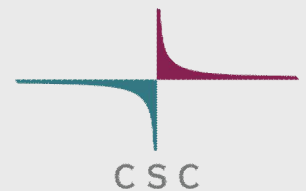
- 7-Zip  
free  
makes and opens tar, zip, gzip, bzip2  
<http://www.7-zip.org/>

➤ **Mac**

- tar, zip, gzip available on standard installation



# Moving data to and from CSC



## ➤ **Scientist's Interface**

- not recommended for large files
- will be updated in 2009

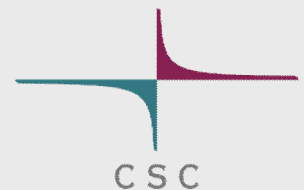
## ➤ **Linux and Mac**

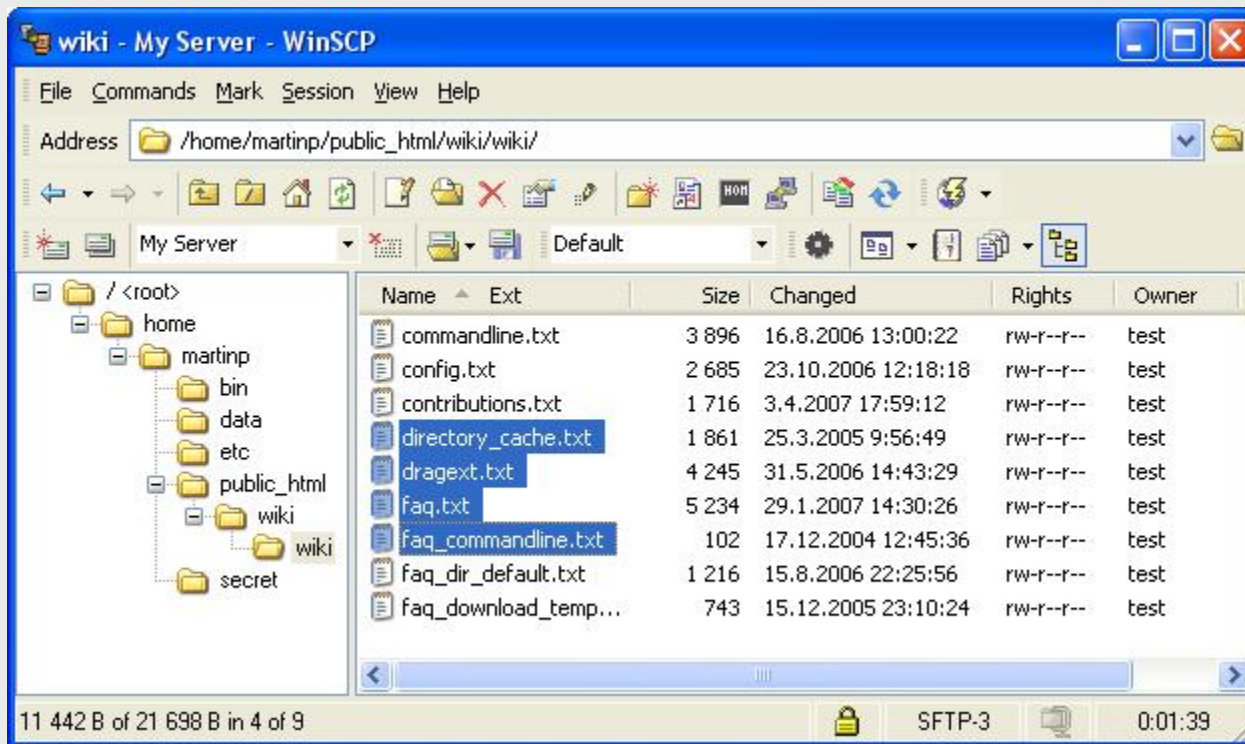
- scp is standard and works well

```
scp myfiles.tar.gz 'user1@murska.csc.fi:$WRKDIR'  
scp 'user1@murska.csc.fi:$WRKDIR/myfiles.tar.gz' .
```

## ➤ **Windows**

- most commercial ssh programs have graphical file moving interfaces  
commonly used PuTTY does not (it does have command line based scp and sftp)
- winSCP is good free option  
<http://winscp.net/eng/index.php>

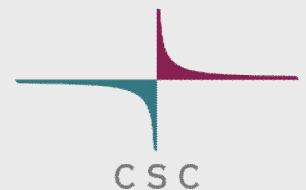




WinSCP interface

# How to run you jobs

- **Interactive jobs**
- **serial patch jobs**
- **array jobs**
- **parallel jobs**



# Interactive jobs

## ➤ Interactive jobs are best run on Hippu

- no time limit on jobs
- no need to reserve a node
- if job takes long, it should be run as background job

```
myprogram &  
or use screen
```

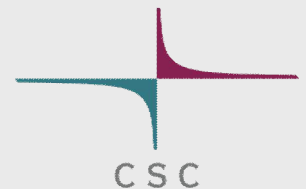
## ➤ Interactive jobs can be run on Murska by reserving a node

- time limit max 4 h
- reserve a node for interactive work:
- reserve 1 core with 1 GB memory for 2 hours for interactive work

```
bsub -Ip $SHELL -i
```

- if you can run xterm applications, its best to run xterm

```
bsub -Ip xterm
```



# screen

## ➤ **screen is a virtual window manager**

- available on Murska and Hippu
- your session stays "as is" even if you disconnect

## ➤ **screen is machine specific**

- make note whether you are connected to hippu1 or hippu2 and connect to the same machine when re-attaching

## ➤ **Basic commands**

- open a new screen

```
screen
```

- list open screens

```
screen -ls
```

- re-attach to a screen (if only one open)

```
screen -r
```

- re-attach to screen with id 12345 (as shown by `screen -ls`)

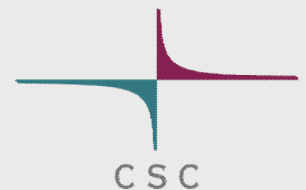
```
screen -r 12345
```

- detach from screen

```
screen -d
```

- screen exits when all processes (including the shell) exit. Or type

```
Ctrl+a Shift+k
```



# Serial patch jobs

- **Best suited for single, long jobs**
  - typical examples: gene mapping, clustering
  
- **Steps for running a serial patch job in Murska**
  1. write a patch job script
  
  2. make sure all the necessary files are in \$WRKDIR
    - Murska computing nodes can not see \$HOME, \$METAWRK, etc.
  
  3. Submit your job

```
bsub < myscript
```



Example serial patch job script:

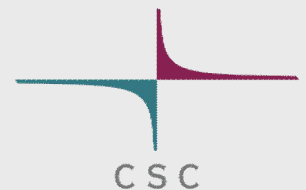
```
#!/bin/tcsh
#BSUB -L /bin/tcsh
#BSUB -J myjob
#BSUB -e myjob_err_%J
#BSUB -o myjob_output_%J
#BSUB -N
#BSUB -u my.address@foo.net
#BSUB -n 1
#BSUB -M 4194304
#BSUB -W 3:00

cd $WRKDIR
module load myprog
myprog -option1 -option2
```



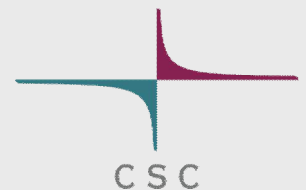
```
#!/bin/tcsh
```

- **Tells the computer this is a script that should be run using tcsh shell**
- **Everything starting with "#BSUB" is passed on to the batch job system**
- **Everything starting with "# " is considered a comment**
- **Everything else is executed as a command**
- **Course examples use tcsh, but you can use other shells if you wish**



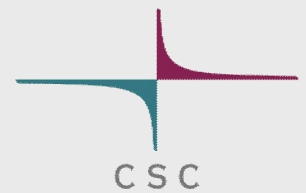
```
#BSUB -L /bin/tcsh
```

- **Sets the execution shell for the batch job**
- **Course examples use tcsh, but you can use other shells if you wish**



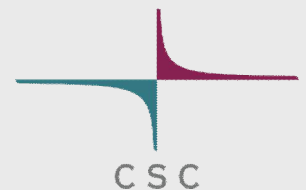
```
#BSUB -J myjob
```

- **Sets the name of the job**



```
#BSUB -e myjob_err_%J  
#BSUB -o myjob_output_%J
```

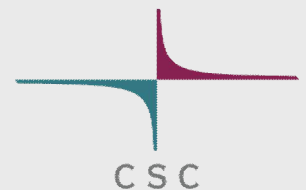
- **Option `-e` sets the name of the file where possible error messages (stderr) are written**
- **Option `-o` sets the name of the file where the standard output (stdout) is written**
- **When running the program interactively these would be written in the command prompt**
- **`%J` is replaced with the job id number in the actual file name**



```
#BSUB -N
```

```
#BSUB -u my.address@foo.net
```

- **Option `-N` = send email**
- **Option `-u` = your email address.**
- **If these are selected you get a email message when the job is done. This message also has a resource usage summary that can help in setting batch script parameters.**

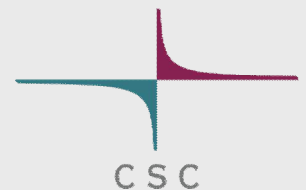


```
#BSUB -n 1
```

➤ **Number of cores to use**

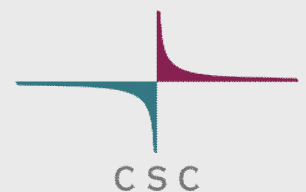
➤ **Check software documentation**

- most bioinformatics software can not utilize more than one core per process



```
#BSUB -M 4194304
```

- **The amount of memory reserved for the job in kB**
  - 1 GB = 1048576
  - 2 GB = 2097152
  - 4 GB = 4194304
  - 8 GB = 8388608
  
- **If you reserve too little memory the job will use swap disk and become very slow**
  
- **If you reserve too much memory your job will spend much longer in queue**
  - on Murska there are 2716 cores able to run 1 GB jobs, but only 128 cores able to run 8 GB jobs



```
#BSUB -W 3:00
```

➤ **time reserved for the job in hours**

- you can also give it in minutes, *e.g.*:

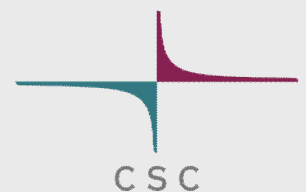
```
#BSUB -W 150
```

➤ **when the time runs out the job will be terminated!**

➤ **with longer reservations the job might spend longer in the queue**

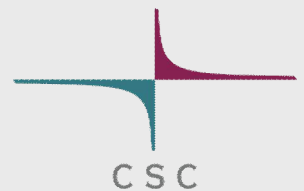
➤ **limit for normal serial jobs is 7d (168 h)**

- if you reserve longer time, the job will go to "longrun" queue (limit 21d)
- In the longrun queue you run at your own risk. If a batch job in that queue stops prematurely no compensation is given for lost cpu time!



```
cd $WRKDIR  
module load myprog  
myprog -option1 -option2
```

- **your commands**
- **also remember to load modules if necessary**



# Array jobs

- **Best suited for running the same analysis for large number of files**
  - typical examples: running BLAST on many sequences
- **Array jobs are set up and run the same as serial jobs.**
- **Some slight modifications to the patch job script**
  - Job name given in format:  
`#BSUB -J my_job[1-500]`
  - Take care result files are not overwritten
- **Please submit your jobs in arrays of max 1000 jobs**
  - it is also polite to limit the number of jobs run simultaneously
- **More info:**
  - [http://www.csc.fi/english/pages/murska\\_guide/batch\\_jobs/serial\\_batch\\_jobs/index\\_html/](http://www.csc.fi/english/pages/murska_guide/batch_jobs/serial_batch_jobs/index_html/)

➤ **Controlling the array behavior**

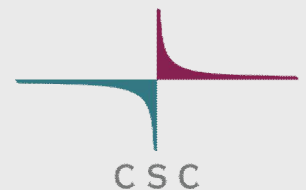
- array of 50 jobs, numbered 1-50:  
`#BSUB -J myjob[1-50]`
- array of 50 jobs, numbered 51-100:  
`#BSUB -J myjob[51-100]`
- array of 100 jobs, 10 jobs run at a time  
`#BSUB -J myjob[1-100]%10`

➤ **When running array jobs, you need to differentiate between the output files of jobs**

- Variable `$LSB_JOBINDEX` points to current job

➤ **Remember that all the jobs will run the batch script**

- don't put in commands that should only be run once



## Example array job script

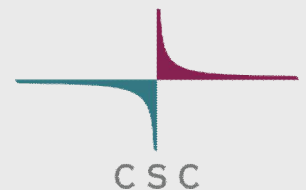
```
#!/bin/tcsh
#BSUB -L /bin/tcsh
#BSUB -J my_job[1-50]
#BSUB -o jobout
#BSUB -e job_err
#BSUB -M 1048576
#BSUB -W 3:00
#BSUB -n 1

# move to the directory where the data files locate
cd data_dir

# run the analysis command
my_prog data_"$LSB_JOBINDEX".inp data_"$LSB_JOBINDEX".out
```

Here it is assumed your input data is files named "data\_1.inp", "data\_2.inp" etc.

Results will be in files named "data\_1.out", "data\_2.out" etc.



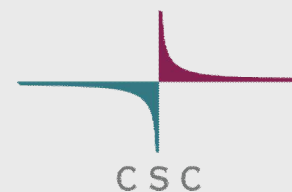
# Parallel batch jobs

- **Only applicable if your program supports parallel running**
- **Mostly identical to running serial batch jobs**
  - main difference is the actual run command
- **See server manuals for specifics**
  - [http://www.csc.fi/english/pages/murska\\_guide/batch\\_jobs/parallel\\_jobs/index\\_html](http://www.csc.fi/english/pages/murska_guide/batch_jobs/parallel_jobs/index_html)
  - [http://www.csc.fi/english/pages/louhi\\_guide/batch\\_jobs/parallel\\_jobs/index\\_html](http://www.csc.fi/english/pages/louhi_guide/batch_jobs/parallel_jobs/index_html)

## Example parallel job patch script for Murska

```
#!/bin/tcsh
#BSUB -n 64
#BSUB -W 2:30
#BSUB -o mpi_prog.out.%J
#BSUB -M 1048576
#BSUB -N
#BSUB -u user1@univ2.fi

mpirun -srun ./mpi_prog
```



# Managing batch jobs in Murska

## The script file is submitted with command:

```
bsub < batch_job.file
```

## The job can be followed with commands

```
bjobs                (shows your jobs)
bjobs -u all         (shows all jobs)
busers all           (shows all jobs)
```

## You can delete a jobs with command

```
bkill job_id_number
```

**or**

```
bkill -r job_id_number
```

## more information:

[http://www.csc.fi/english/pages/murska\\_guide/batch\\_jobs/monitoring/index\\_html](http://www.csc.fi/english/pages/murska_guide/batch_jobs/monitoring/index_html)



# Managing array jobs

➤ **Kill the whole array job**

```
bkill job_id_number
```

➤ **Kill subjobs 100-150 from array job with jobid 1234**

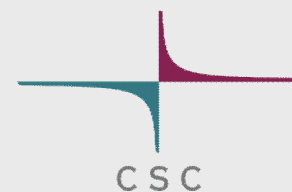
```
bkill "1234[100-150]"
```

➤ **Kill subjobs 100, 113 and 213 from array job with jobid 1234**

```
bkill "1234[100,113,213]"
```

**Res-submit the jobs above**

```
#BSUB -J "my_job[100, 113, 213]"
```



**Display the stdout and stderr output of an unfinished job**

`bpeek`

**Displays the output of the job using the command `tail -f`**

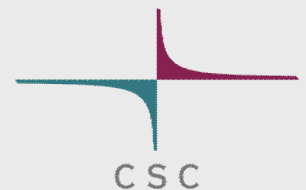
`bpeek -f`

**Statistics about recent finished jobs**

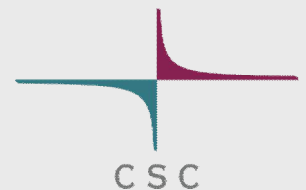
`bacct`

**More specific information on resource use etc.**

`bacct -l <job_id>`

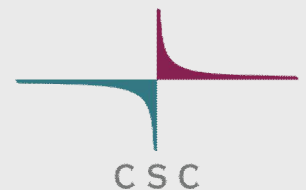


# Automating analysis: Introduction to shell scripting



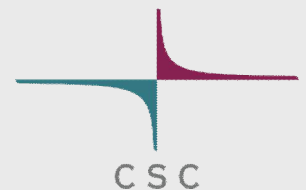
# Scripting

- **For more complex operations it is advisable to use a full-fledged programming language (e.g. python, perl)**
- **Shell scripting can be very usefull for automating analysis, handling large number of files, doing repetitive tasks etc.**
- **You can get a lot done with just a handful of commands**
- **Course examples are in tcsh, but other common shells available**



## Getting started:

- **Write & save the script using a text editor**
- **Every tcsh script should start with**  
`#!/bin/tcsh`
- **Remember to give yourself execution rights to the script**  
`chmod u+x filename`
- **If the scripts are not in your \$PATH, remember to give path in command (same as with all other programs)**
  - to run a script in your current directory  
`./myscript`
- **Remember: `man` command and Google are your friends**



# Unix commands

## Basic syntax:

*comand -option argument*

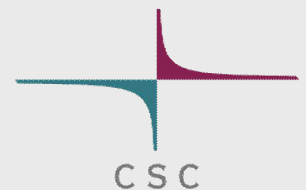
**ls**

**ls -l**

**ls -l myDirectory**

Use man command to get information about possible options

**man ls**



## Commands for directories:

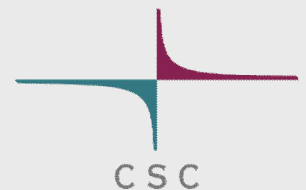
`cd` change directory

`ls` list the contents of a directory

`pwd` print (=show) working directory

`mkdir` make directory

`rmdir` remove directory



## Commands for files:

cat print file to screen

cp copy

less view text file

rm remove

mv move/rename a file

head show beginning of a file

tail show end of a file

grep find lines containing given text

wc count number of words or lines



# Special characters:

\*(asterisk), wild card, means any text

```
ls *.fasta
```

| (pipe) guides output of a command to an input of another commands

```
ls *.fasta | less
```

> Writes output to a new file

```
ls > files_of_the_directory.txt
```

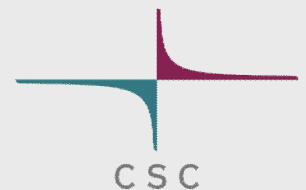
~ (tilde) means your home directory as does \$HOME

```
cp test.txt ~/file.txt
```

```
cp text.txt $HOME
```

& runs command in background

```
gzip my_big_file.tar &
```



# Quotes

- The tcsh shell is very picky about quotes.
  - '' Take text enclosed within quotes literally
  - ` ` Take text enclosed within quotes as command and replace with output
  - """ Take text within quotes literally after substituting any variables
- Compare the results of these commands:

```
set var = "test"; echo 'echo $var'  
set var = "test"; echo `echo $var`  
set var = "test"; echo "echo $var"
```

# Variables

```
set variable = (value)  
$variable
```

```
set array = (a b c)  
$array[1], $array[2], $array[3]
```

## Note:

`$argv[ ]` holds command line arguments

## Example:

```
set len = (1)  
echo $len
```



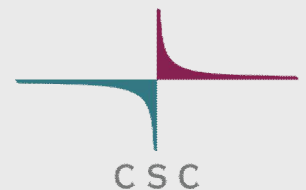
# Conditional structures

```
if (condition) command
```

```
if (condition) then  
    commands  
else  
    commands  
endif
```

## Example:

```
if ($n < 10) then  
    echo "small"  
else  
    echo "large"  
endif
```

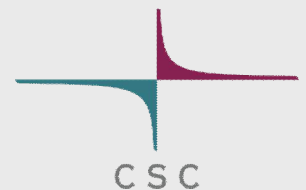


# while loop

```
while (condition)
    commands
end
```

## Example:

```
set n = 1
while ( $n < 10)
    echo $n
    set n = (`expr $n + 1`)
end
```



# Logical operators

➤ **You can use the normal conditional operands**

== equal to

!= not equal to

=~ similar to (allows wildcards)

!~ not similar to (allows wildcards)

> greater than

< smaller than

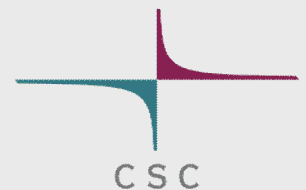
>= greater or equal to

<= smaller or equal to

&& AND

|| OR

! NOT



# File-test operators

- **There are a number of operators you can use to test different attributes of a file:**

`-e file`    **true if file exists**

`-z file`    **true if file exists and is zero size**

`-d dir`     **true if dir exists and is a directory**

`-o file`    **true if file exists and is owned by the user**

`-r file`    **true if file exists and is readable**

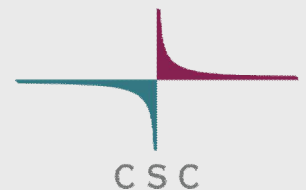
`-w file`    **true if file exists and is writable**

`-x file`    **true if file exists and is executable**

- **Examples**

```
if (-e file) echo "file exists"
```

```
if !(-e file) echo "file does not exist"
```



# Foreach structure

```
foreach variable (value_list)
  commands
end
```

## Examples:

```
foreach gap ( 5 10 15 20 25 )
```

```
foreach sequence (`ls *.seq`)
```

```
foreach sequence (`ls | grep .seq$`)
```

```
foreach line ("`cat myfile.txt`")
```



# Some useful commands for parsing lines

Try these to see what they do!

## sed

```
echo "one this two this three" | sed s/this/that/  
echo "one this two this three" | sed s/this/that/g  
sed -n 3p myfile.txt
```

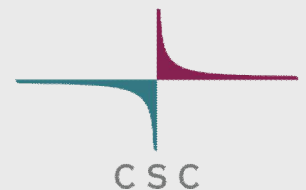
## awk

```
echo "one two three" | awk '{print $2}'  
echo "one;two;three" | awk -F";" '{print $2 $3}'
```

## cut

```
echo "123456789" | cut -c 4  
echo "123456789" | cut -c -4  
echo "123456789" | cut -c 4-  
echo "123456789" | cut -c 4-7  
echo "one_two_three" | cut -d "_" -f 2
```

All of these have much more options. See `man` pages for details.

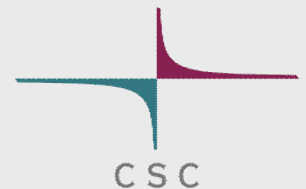


# Example scripts

The following two scripts use two loops to study the gap opening penalty and gap extension penalty for global alignments of sequences `swiss:cas1_human` and `swiss:cas1_sheep`. Gap creation penalty is varied from 5 to 50 with step size of 5 and the gap extension penalty from 1 to 5 with step size 1. The resulting similarity and identity values are sorted according to the similarity and stored to file "sorted.out"

Note that the `expr` command can only handle integers, so if you would like to use step size less than one, you should use different commands.

The first version of the sample scripts uses `foreach` loop and the second version `while` loop. Results are the same in both cases



## Example 1

```
#!/bin/tcsh

use emboss
echo "Testing matcher" > result.out

set gap = (5)
set n = (1)

foreach gap (5 10 15 20 25 30 35 40 45 50)

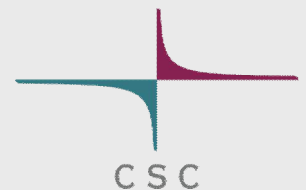
    foreach len (1 2 3 4 5)

        stretcher swiss:cas1_human swiss:cas1_sheep -gapopen $gap -gapextend \
            $len -outfile out.pair -auto

        echo "`grep Similarity out.pair` Len= $len Gap $gap" >> result.out
        rm -f out.pair

    end
end

sort result.out > sorted.out
```



## Example 2

```
#!/bin/tcsh
use emboss
echo "testing matcher" > result.out

set gap = (5)
set n = (1)

while ($gap <= 50)
  set len = (1)

  while ($len <= 5)

    stretcher swiss:cas1_human swiss:cas1_sheep -gapopen $gap \
      -gapextend $len -outfile out.pair -auto

    echo "`grep Similarity out.pair` Len= $len Gap $gap" >> result.out
    rm -f out.pair
    set len = (`expr $len + 1`)
  end

  set gap = (`expr $gap + 5`)
end

sort result.out > sorted.out
```

