Tools and more for High-Performance Computing

Overview

- Environment & account setup
- Compilers
- IDEs, Libraries
- Make & Makefiles
- Version control
- Data analysis tools: awk & perl
- Visualization tools
- Resource Managers

The DTU computer system

The central DTU UNIX system

- Application servers – x86_64 based:
  - 14 HP SL165z G7 (2x AMD Opteron 6168 1.9 GHz)
  - 4 Dell PowerEdge FC430 (2x Intel Xeon E5-2670v3 2.3 GHz)
  - Scientific Linux 6.4
- Desktop servers (ThinLinc):
  - 4 servers (4x AMD Opteron 6376, 2.4 GHz)
- 10000+ users (students + employees)

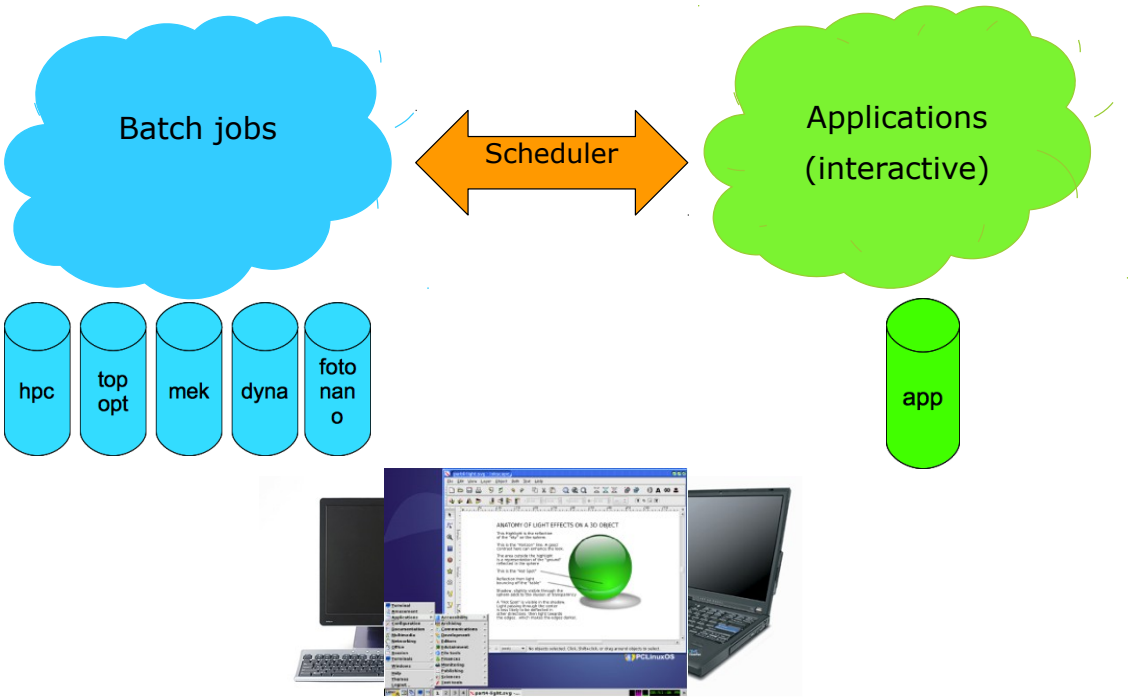
The DTU computer system

- HPC servers:
  - 64 HP SL2x170z (2x Xeon 5550 2.6 GHz, 24 GB memory)
  - 42 IBM NeXtScale nx360 M4 (2x Xeon E5-2680v2 2.8 GHz, 128 GB memory)
- + “private” clusters
  - DTU Compute
  - DTU Nanotech
  - DTU Photonics
  - DTU Chemistry
  - ...

Access to the system

- Remote access:
  - Secure SHell (ssh) connection (login.hpc.dtu.dk or login.gbar.dtu.dk)
  - ThinLinc remote desktop session:
    - download ThinLinc client from www.thinlinc.com
    - connect to thinlinc.gbar.dtu.dk
    - preferred way, if you work a lot with GUIs
    - browser based: https://thinlinc.gbar.dtu.dk/
- On Campus:
  - ThinLinc from Windows computers (Winbar).

Our Setup



Access to Linux machines

- Connected by SSH (login.hpc.dtu.dk)
  - linuxsh – access to Opteron based G-bar nodes
  - qrsh – access to Xeon based HPC nodes
  - Note: X11 forwarding with '-X' option!
- ThinLinc session
  - open a terminal (xterm) from the DTU menu – access to a G-bar node (same as linuxsh)
  - access to interactive HPC nodes with 'qrsh' command

The DTU computer system

Be aware of, that ...

- this is a multi-user system(!)
- (almost) all applications on the system are started by a load-balancing queueing system
- there are different
  - CPU types,
  - clock frequencies,
  - amounts of RAM,
  - etc

The DTU computer system

Comparing performance numbers:

- make sure to be on the same machine type
  - `lscpu` command
  - `echo $CPUTYPE`
- check the load (interactive sessions)
  - `uptime` command
- check the # of CPU cores
  - `cpucount` command

Compilers

- Oracle Studio compilers & tools
  - version 12 upd 4 ('module load studio/12u4')
  - commands: suncc, sunCC, sunf95
  - sets cc, CC and f95, f77, too!
- GNU Compilers (C/C++)
  - gcc 4.4.7 (OS standard)
  - gcc 4.7.2 ('module load gcc')
  - newer versions: check with 'module avail gcc')
- Note: 'cc' depends on the module loaded!!!

More compilers

- Open64 compilers
  - version 5.0 ('module load open64' – default)
  - version 4.2.4 available as well
  - commands: opencc, openCC, openf95
- Portland Group / PGI compilers
  - version 2013 ('module load pgi' – default)
  - version 2012 ('module load pgi/2012')
  - version 2011 ('module load pgi/2011')
  - commands: pgcc, pgCC, pgf95/pgf77

More compilers

- Intel compilers
  - version 13.0.1 ('module load intel')
  - commands: icc, ifort
  - + some extra tools
  - other versions: check with 'module avail intel'

Using modules

- modules help to organize certain Unix environment settings, e.g. PATH, MANPATH, LD_LIBRARY_PATH, etc. for different versions of the same application
- list available modules: `module avail`
- load a module: `module load gurobi`
- swap a version: `module swap gurobi/5.6.3`
- swap to default: `module swap gurobi`
- info: http://gbar.dtu.dk/index.php/faq/83-modules

IDEs

- Oracle Studio (`sunstudio`)
  - Compilers (Fortran, C/C++)
  - Debugger (`dbx`), analysis tools – more later
- Eclipse (`eclipse4`)
- Graphical debuggers:
  - Totalview (`totalview`)
  - Dbxtool (dbxtool)
  - Data Display Debugger (`ddd`)
    - GUI front-end to either dbx or gdb

Libraries

- Available Scientific Libraries:
  - ATLAS
    - BLAS, CBLAS, LAPACK, ...
    - installed on all nodes
  - FFTW3
    - installed on all nodes
  - Solaris Studio Performance Library (optimized)
    - BLAS, LAPACK, FFT, ...
    - part of Oracle Studio
    - installed on all nodes

Make & Makefiles

A tool for building and
maintaing software projects

Make – The ideas behind

- maintain, update and regenerate groups of programs
- useful tool in multi-source file software projects
- can be used for other tasks as well, e.g. typesetting projects, flat-file databases, etc
- in general: every task that involves updating files (i.e. result) from other files (i.e. sources) is a good candidate for make

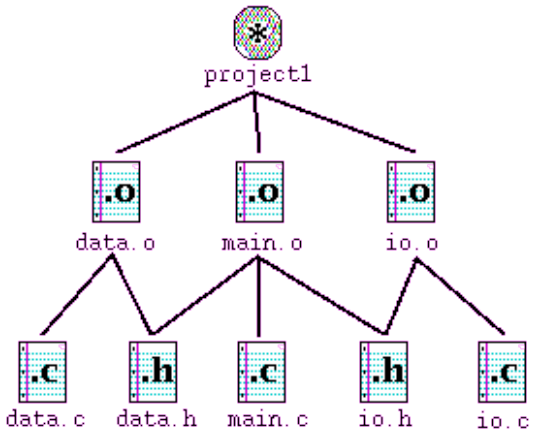
Make – The ideas behind

Dependency graph:

result (executable)

intermediate level

source file level



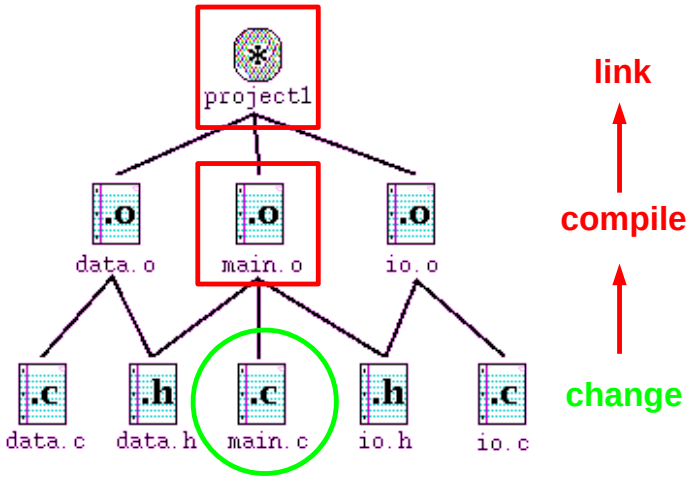
Make – The ideas behind

Dependency graph:

result (executable)

intermediate level

source file level



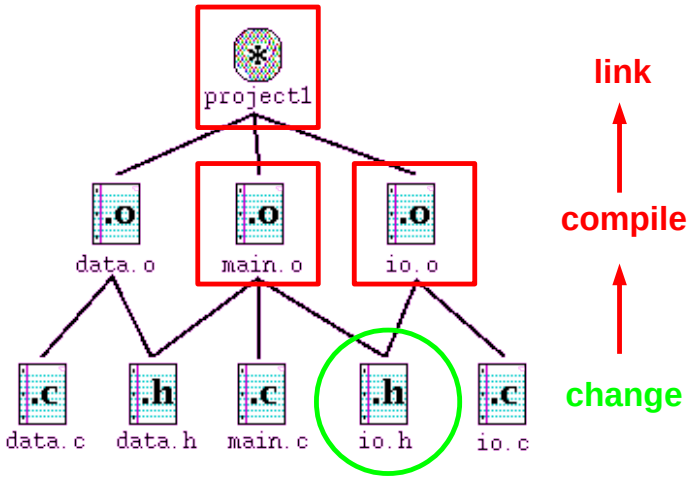
Make – The ideas behind

Dependency graph:

result (executable)

intermediate level

source file level



Make – The ideas behind

- Compiling by hand:
  - error prone
  - easy to forget a file
  - typos on the command line
- There is a tool that can help you:

make

Make – The ideas behind

Things 'make' has to know:

- file status (timestamp)
- file location (source/target directories)
- file dependencies
- file generation rules (compiling/linking)
  - general rules (.c → .o)
  - special rules (io.c → io.o)
- tools (compilers, etc.)

- filesystem - Makefile - environment

Makefile – rulesets...and more

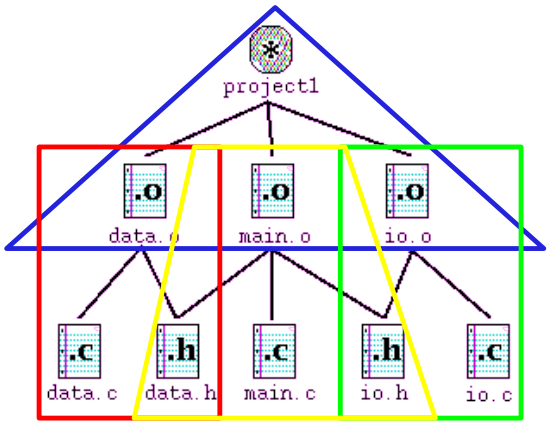
- make needs a set of rules to do its job
- rules are defined in a text file – the *Makefile*
- standard names: Makefile or makefile
- non-standard names can be used with the '-f' option of make: `make -f mymf` ...
- preview/dryrun option: `make -n` ...

Makefile – rulesets...and more

There are two major object types in a Makefile

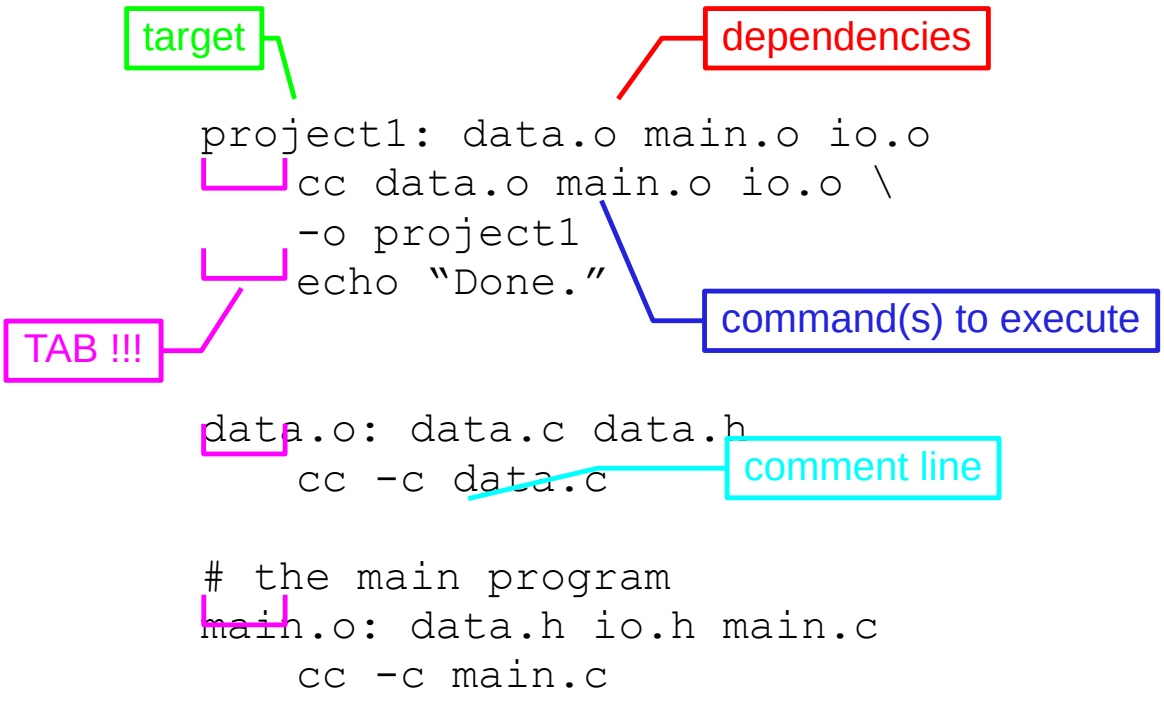
- targets
  - definition by a ":"
  - followed by the dependencies (same line)
  - followed by lines with the commands to execute
- macros
  - definition by "="
  - single line (use "\" to extend lines)
- ... and comments: (lines) starting with #

Makefile – rulesets...and more




```
project1: data.o main.o io.o
	cc data.o main.o io.o -o project1
data.o: data.c data.h
	cc -c data.c
main.o: data.h io.h main.c
	cc -c main.c
io.o: io.h io.c
	cc -c io.c
```

Makefile – rulesets...and more



Makefile – rulesets...and more

```
# Sample Makefile
CC      = gcc
OPT     = -g -O3
WARN    = -Wall
CFLAGS = $(OPT) $(WARN)    # the C compiler flags
OBJECTS   = data.o main.o io.o

project1 : $(OBJECTS)
     $(CC) $(CFLAGS) -o project1 $(OBJECTS)

clean:
     @rm -f *.o core

realclean : clean
     @rm -f project1

# file dependecies
data.o : data.c data.h
main.o : data.h io.h main.c
io.o   : io.h io.c
```

Macro definitions

Macro reference

Where are my rules for compiling the .o files?

Makefile – rulesets...and more

Running make:

```
myhost $ make
gcc -g -O3 -Wall   -c -o data.o data.c
gcc -g -O3 -Wall   -c -o main.o main.c
gcc -g -O3 -Wall   -c -o io.o io.c
gcc -g -O3 -Wall -o project1 data.o main.o io.o
```

How did **make** know how to build data.o, ... ?

Makefile – rulesets...and more

built-in data base of "*standard rules*" and "*standard macros*":

- known rules:
  - compile .o files from a .c/.cpp/.f/... source file
  - link executables from .o files
- pre-defined macros:
  - CC, CFLAGS, FC, FFLAGS, LD, LDFLAGS
- view with `make -p -f /dev/null` (long listing!)

Makefile – rulesets...and more

```
# GNU Make 3.80
# Variables
...
# default
OUTPUT_OPTION = -o $@
# makefile (from `Makefile', line 3)
CC = gcc
# environment
MACHTYPE = i686-suse-linux
# makefile (from `Makefile', line 6)
CFLAGS = $(OPT) $(WARN)
# makefile (from `Makefile', line 4)
OPT = -g -O3
# makefile (from `Makefile', line 5)
WARN = -Wall
# default
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) -c
# makefile (from `Makefile', line 8)
OBJECTS = data.o main.o io.o
```


Makefile – rulesets...and more

```
...

# Implicit Rules
.c.o:
#  commands to execute (built-in):
    $(COMPILE.c) $(OUTPUT_OPTION) $<

...

data.o: data.c data.c data.h
#  Implicit rule search has been done.
#  Implicit/static pattern stem: `data'
#  Last modified 2004-08-27 10:08:56.008831584
#  File has been updated.
#  Successfully updated.
#  commands to execute (built-in):
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Makefile – rulesets...and more

Practical hints:

- preview/dryrun option: `make -n ...`
- switch off built-in rules/macros:
  `make -r ...`
- check the known suffixes (.SUFFIXES) and implicit rules for your source files, e.g. does gmake still fail for .f90/.f95
- add suffixes needed:

```
.SUFFIXES: .f90
```


Makefile – rulesets...and more

Practical hints (cont'd):

- be aware of timestamps (Network-FS)
- override macros on the command line:

```
myhost $ make
gcc -g -O3 -Wall    -c -o data.o data.c
gcc -g -O3 -Wall    -c -o main.o main.c
gcc -g -O3 -Wall    -c -o io.o io.c
gcc -g -O3 -Wall-o project1 data.o main.o io.o

myhost $ make CFLAGS=-g
gcc -g   -c -o data.o data.c
gcc -g   -c -o main.o main.c
gcc -g   -c -o io.o io.c
gcc -g -o project1 data.o main.o io.o
```

Makefile – rulesets...and more

Special variables/targets:

- the first target in Makefile is the one used when you call make without arguments!
- automatic variables:
  - $< - The name of the first prerequisite.
  - $@ - The file name of the target of the rule.
- for more information:
  - man make
  - info make

Makefile – rulesets...and more

Makefile design – Best practice:

- start with the macros/variables
- call your first target “all:” and make it depend on all targets you want to build
- have a target “clean:” for cleaning up
- avoid explicit rules where possible, i.e. use redundancy

Makefile – rulesets...and more

Makefile design – Best practice (cont'd):

- check your dependencies:
  - by hand
  - most C/C++ compilers can generate Makefile dependencies (see compiler documentation)
  - Sun Studio: `suncc -xM1`
  - Gnu C: `gcc -MM`
  - external tool: `makedepend -Y`
  - Note: the options above ignore `/usr/include`

Makefile – rulesets...and more

Common mistakes:

- missing TAB in “command lines”
- wrong variable references:

  $VAR instead of $(VAR)
- missing/wrong dependencies
- remember: each command is carried out in a new sub-shell

Makefile – rulesets...and more

Makefiles – and Makefiles (from IDEs)

- Most IDEs create their own Makefiles
  - ... which are often not very smart
  - ... which are often not compatible

- make and (g)make:
  - Linux: make == gmake (GNU make)
  - Unix: make != gmake
  - if make fails, try gmake

Make and Makefiles: Labs

- There are five short lab exercises
- download from Campusnet
- unzip the file
- the exercises are in the directories lab_N
- read the README files for instructions

Make and Makefiles: Labs

- Hints:
  - M_PI is a definition from <math.h>
  - sin() is a function from libm.so, so you have to link with that library (use -lm the right place)

Version control

- Larger – but also simple – software projects need to keep track of different versions
- This is very useful during development, e.g. to be able to go back to the last working version
- Versioning Tools:
  - RCS – single user, standalone
  - CVS – multi-user, network based
  - Subversion – multi-user, network based
  - git – multi-user, network based

Version control

- DTU has a central CVS server
  - nice tool to share and control source files
  - request access on https://repos.gbar.dtu.dk/
  - basic introduction: http://gbar.dtu.dk/faq/34-cvs
- ... and a Subversion (SVN) server as well
  - request access on https://repos.gbar.dtu.dk/
  - basic introduction: http://gbar.dtu.dk/faq/39-svn
- ... and some info about Git and GitLab:
  - http://gbar.dtu.dk/faq/41-git
  - http://www.gbar.dtu.dk/faq/94-gitlab

Data analysis tools

- Scientific software usually produces lots of data/datafiles
- There are good tools to do (a quick) analysis:
  - awk – standard UNIX/Linux tool
  - perl – (almost) standard on many platforms
- Both tools can be used
  - from the command line
  - with scripts

Data analysis tools – awk

- awk operators:

```
Field reference:                $
 $0: the whole line - $n: the n-th field
Increment or decrement:     ++ --
Exponentiate:                ^
Multiply, divide, modulus:  * / %
Add, subtract:               + -
Concatenation:              (blank space)
Relational:                 < <= > >= != ==
Match regular expression:   ~ !~
Logical:                    && ||
C-style assignment:     = += -= *= /= %= ^=
```


Data analysis tools – awk

Examples:

- Print first two fields in opposite order:

```
awk '{ print $2, $1 }' file
```

- Print column 3 if column 1 > column 2:

```
awk '$1 > $2 {print $3}' file
```

- Print line (default action) if col. 3 > col. 2:

```
awk '$3 > $2' file
```

Data analysis tools – awk

Examples (cont'd):

- Add up first column, print sum and average:

```
awk '{s += $1}; END { print "sum
is", s," avg is", s/NR}' file
```

- Special keywords/variables:

```
BEGIN   do before the first record
END     do after the last record
NR      number of records
NF      number of fields
$NF     the value of the last field
```


Data analysis tools

- Other useful standard Unix tools for data analysis:
  - sort
  - uniq
  - head, tail
  - wc
  - sed
  - ...

Data analysis tools – perl

- Perl is a very powerful tool, that combines the features of awk, grep, sed, sort, and other Unix-tools into one language
- Good tool for more complex data analysis tasks
- Web-site: http://perl.org/
- Archive of perl programs:
  - Comprehensive Perl Archive Network – CPAN
  - http://www.cpan.org/

Data analysis tools – perl

Perl example script:

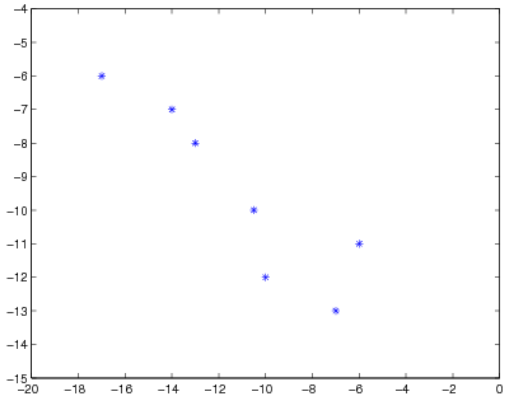
```
#!/usr/bin/perl

while (<>) {

    next if /^#/;       # skip comment lines
    @fields = split();  # split the line

    if ($#fields == 2 ) { # 3(!) elements
        print "$fields[0] $fields[2]\n";
    }
    else {
        print;
    }
}
```

Visualization

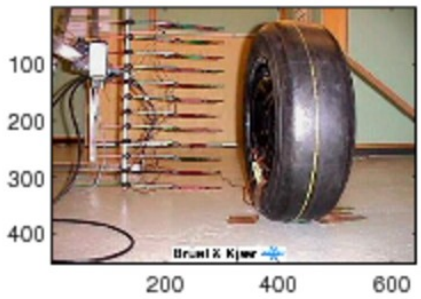
Visualization is an important part of Scientific Computing

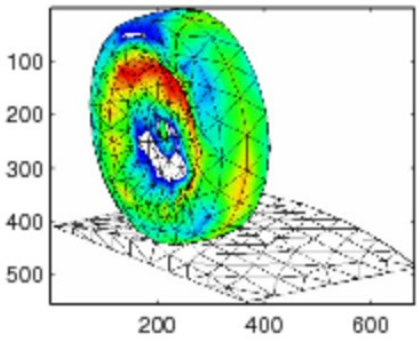
Motivation: What's that?

```
A (  -17,  -6)
B (  -14,  -7)
C (  -13,  -8)
D (-10.5, -10)
E (   -6, -11)
F (   -7, -13)
G (  -10, -12)
```



Visualization





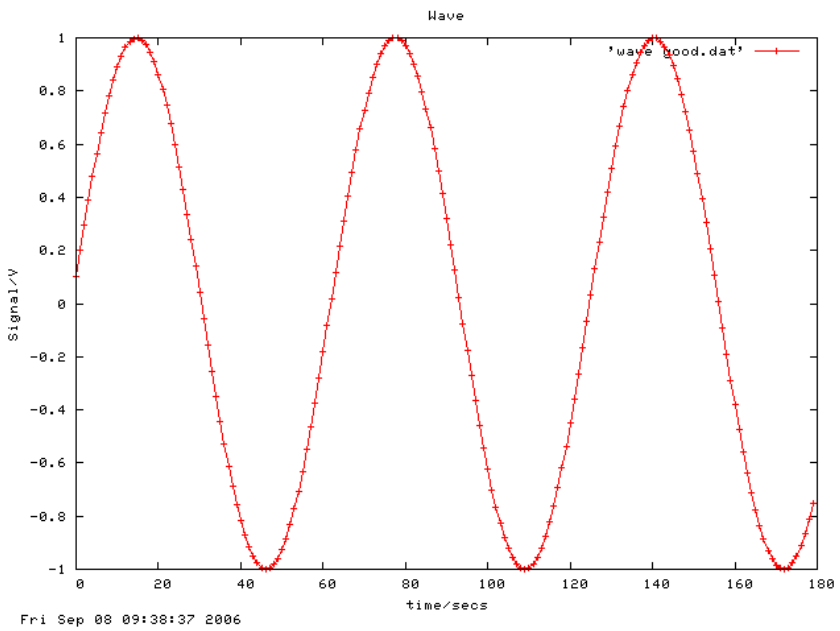
Visualization

- Simple tools to visualize data:
  - Gnuplot (`gnuplot`)
    - command based, flexible
    - good for scripting, batch analysis
    - limited graphics (not always suitable for publishing)
  - Grace (`xmgrace`)
    - GUI-based
    - difficult to do scripting, batch analysis
    - very good graphics (publication-ready)
  - ... or whatever tool you like/prefer

Visualization

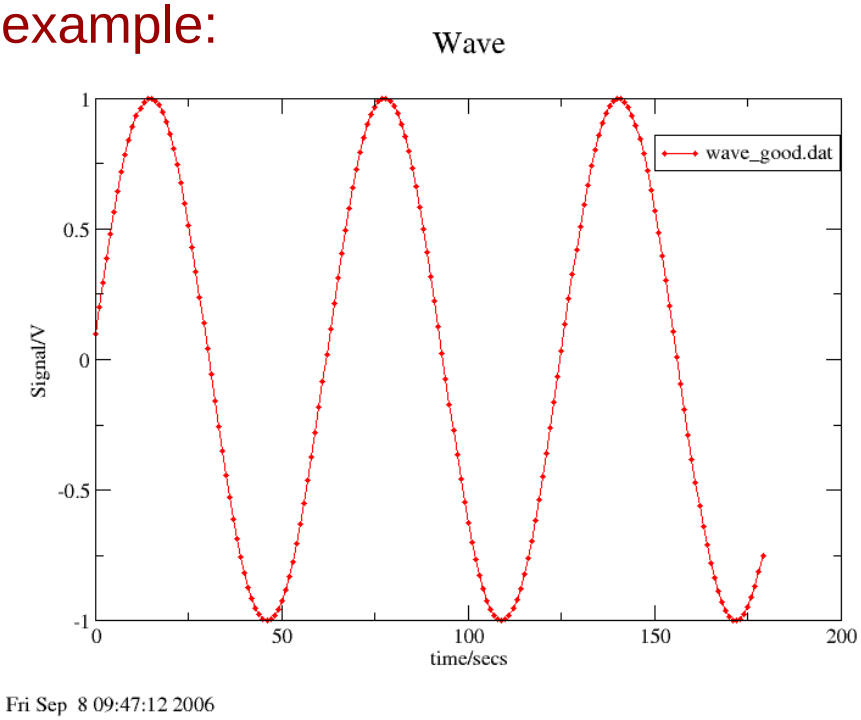
Gnuplot example:

```
gnuplot>
gnuplot>
gnuplot>
gnuplot>
gnuplot>
```



Visualization

Grace example:



Visualization

- Best practice:
  - label the axes
  - use legends (and titles)
  - use the right scaling
    - a plot of a circle should be a circle
  - don't overload figures with information – use more figures instead
  - colors are useful – but can also be confusing

And not to forget ...

... a very powerful tool/language for Scientific Computing:

- Python
  - built-in vector and matrix types (NumPy, SciPy)
  - data plot functionality (matplotlib)
  - interfaces to different languages
  - GPU support (PyOpenCL, PyCUDA)
  - and, and, and ....

Data analysis – lab exercise

- download the file wave.zip from Campusnet
- follow the instructions in wave.readme
- Goal:
  - get used to awk (choose perl, if you like or know it already)
  - get used to either Gnuplot or Grace (or the tool you know/like)

Resource Managers

To handle the workload on an HPC installation, one needs a tool to manage and assign the resources: a Resource Manager – sometimes also called 'batch queue system'

- Most common systems:
  - Torque/PBS (ext. scheduler, like Maui or MOAB)
  - LSF
  - Grid Engine
  - Slurm

Resource Managers

Before submitting a job, one has to specify the resources needed, e.g.

- # of CPUs/cores
- amount of memory
- expected run time (wall-clock time)
- other resources, like disk space, GPUs, etc

This is done in a special job script and is system (RM) dependent – but very similar for all RMs.

Resource Managers

- Examples for the DTU batch system, based on
  - MOAB (scheduler) and
  - Torque (resource manager)
- Note:
  - you cannot submit executables directly, you have to use a job script!
  - don't expect jobs to start immediately – the scheduler has to find free resources first!

Resource Managers

The simplest job script:

```
#!/bin/sh
sleep 60
```

submit.sh

```
$ qsub submit.sh
611064.hnode2
$ qstat
                                                  Req'd   Elap
Job ID          Username Queue    Jobname         Time  S Time
--------------- -------- -------- --------------- ----- - -----
611064.hnode2   gbarbd   hpc      submit.sh         --  R   --
$ ls -g
total 3
-rw-r--r-- 1 gbar 19 Jan  3 17:21 submit.sh
-rw------- 1 gbar  0 Jan  3 17:21 submit.sh.e611064
-rw------- 1 gbar  0 Jan  3 17:21 submit.sh.o611064
```

Resource Managers

A simple job script:

```
#!/bin/sh
#PBS -N sleeper
#PBS -q hpc
#PBS -l walltime=2:00
cd $PBS_O_WORKDIR

sleep 60
```

```
$ qsub submit.sh
611070.hnode2
$ ls -g
total 3
-rw-r--r-- 1 gbar 19 Jan  3 17:31 submit.sh
-rw------- 1 gbar  0 Jan  3 17:34 sleeper.e611070
-rw------- 1 gbar  0 Jan  3 17:34 sleeper.o611070
```


Resource Managers

A simple job script:

```
#!/bin/sh
#PBS -N sleeper
#PBS -o $PBS_JOBNAME.$PBS_JOBID.out
#PBS -e $PBS_JOBNAME.$PBS_JOBID.err
cd $PBS_O_WORKDIR

echo "Just a minute ..."
sleep 60
```

```
$ qsub submit.sh
611075.hnode2
$ ls -g
total 3
-rw-r--r-- 1 19 Jan  3 17:41 submit.sh
-rw------- 1  0 Jan  3 17:45 sleeper.611075.hnode2.err
-rw------- 1 18 Jan  3 17:45 sleeper.611075.hnode2.out
```

Resource Managers

more options and examples:

- see http://www.hpc.dtu.dk/ under
  - HPC Services
    - User Guides

- do the lab exercises
- use 'man qsub', 'man qstat', etc

Resource Managers

Scheduler specific commands:

- showq – shows the complete queue (long output!)
- showclass hpc – shows 'hpc' queue only
- checkjob – checks the status of a job
  - use to find why a job doesn't start, etc
- nodestat – shows the current status of all nodes (use 'nodestat hpc' for the nodes of the 'hpc' queue)

Resource Managers

- There are a few hands-on exercises on CampusNet to get you aquainted with the batch system
- more information can be found on www.hpc.dtu.dk under HPC Services
- we have a special queue for this course, '`hpcintro`', so please use '`-q hpcintro`' instead of '`-q hpc`' in your job scripts!