Getting Started with HPC Clusters

Kai Himstedt, Nathanael Hübbe, and Hinnerk Stüben
Universität Hamburg

December 2019

Introductory remarks

- this set of slides is a result from the PeCoH project
  – Performance Conscious HPC –
  - `https://www.hhcc.uni-hamburg.de/pecoh/`
  - `https://wr.informatik.uni-hamburg.de/research/projects/pecoh/start`
- the slides were auto-generated from *markdown* sources in the framework of our *skill tree* text processing environment
  - `https://www.hhcc.uni-hamburg.de/files/hpccp-concept-paper-180201.pdf` (section 3.2)
- acknowledgement

  This work was supported by the German Research Foundation (DFG) under grants LU 1353/12-1, OL 241/2-1, and RI 1068/7-1.

Overview

- Introduction
- System Architectures
- Hardware Architectures
- I/O Architectures
- Performance Frontiers
- Parallelization Overheads
- Domain Decomposition
- Job Scheduling
- Use of the Command Line Interface
- Using Shell Scripts
- Selecting the Software Environment
- Use of a Workload Manager
- Benchmarking

Getting Started with HPC Clusters (Basic)

Introduction

What is HPC?

- tautological definition
  - "You are doing HPC when you are using HPC hardware."
- traditional definition
  - run *computer simulations in natural sciences and engineering* as fast as possible
  - performance metric: FLOPS or Flop/s (double-precision floating-point operations per second)
- other performance metrics
  - time-to-solution
  - time to get a task done
  - search operations per second
  - ...
- common denominator
  - powerful hardware

Introduction

HPC software environment

- the operating system is GNU/Linux
- interactive access is limited
  - graphical user interfaces are unusual
  - the command line has to be used
- a *batch system* has to be used
  - batch jobs are being prepared and managed from the command line
  - batch jobs have to be formulated as shell scripts
  - job inputs must be prepared beforehand

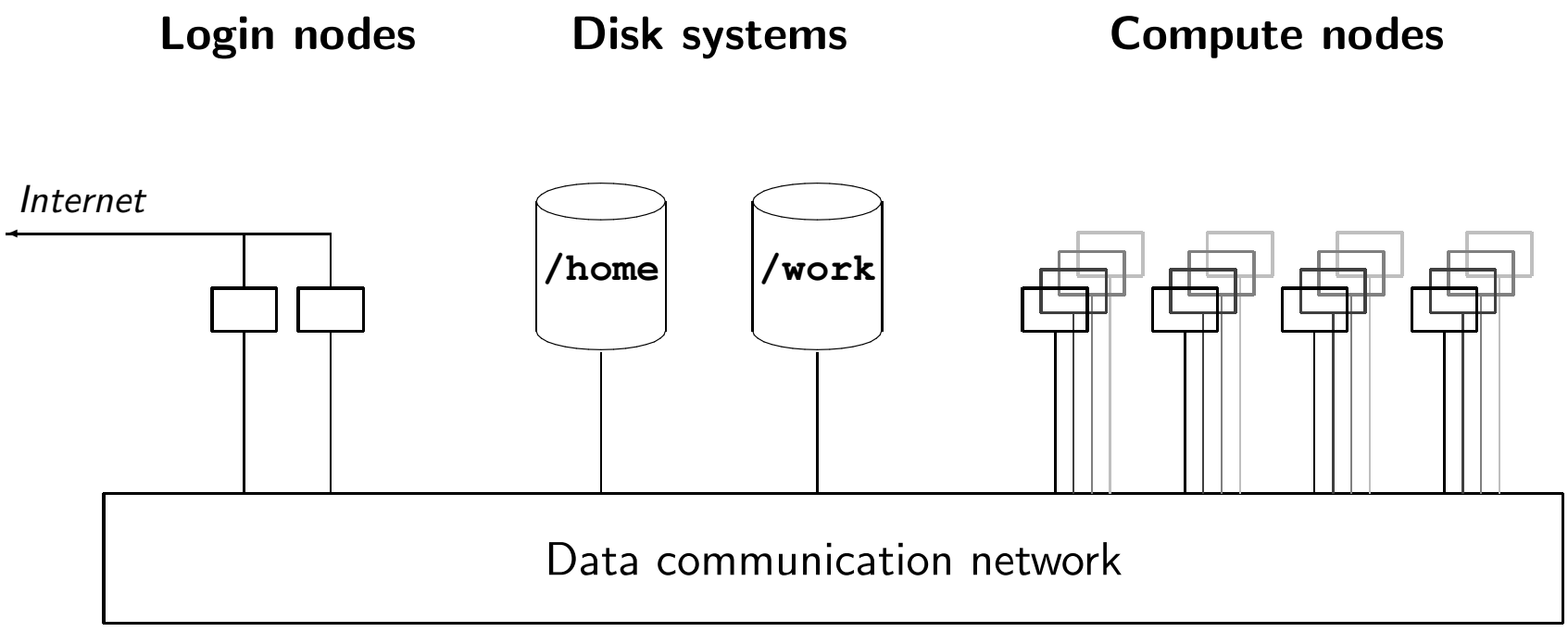
Introduction

Need for parallel processing

- *parallelization* is needed in order to significantly speed up computations
  - the basics of parallel computing must be understood
  - parallel performance needs to be checked: is the runtime (almost) n times shorter when n times as many compute cores are used?

System Architectures (Basic)

HPC cluster architecture



HPC cluster architecture

What the user sees

- login nodes
- compute nodes
- special nodes (e.g. for pre- and post-processing)
- disk systems
- data communication network

Nodes that work in the background

- admin/management nodes
- system services nodes
- disk nodes

Hardware Architectures (Basic)

Parallel computer architectures (1)

Components of a parallel computer

- compute units
- main memory
- high speed network

Compute units

- CPUs
- GPUs / GPGPUs
- FPGAs
- vector computing units

Parallel computer architectures (2)

Main memory architecture

Conceptually, the high speed network connects compute units and main memory.

- shared memory
  - a single computer
  - all compute compute units can access the whole memory
- distributed memory
  - multiple computers (e.g. a cluster)
  - data exchange via the network
- NUMA (Non-Uniform Memory Access)
  - logically shared memory (global address space)
  - physically distributed memory (memory speed depends on the *NUMA distance*)

I/O Architectures (Basic)

I/O architectures (1)

Local file systems

- accessible inside a node

Global file systems

- accessible from all nodes

Object stores

- are typically remote systems
- might only be accessible from the login nodes

I/O architectures (2)

Global file system examples

- distributed (network) file systems
  - no concurrent write to a single file
- parallel (cluster) file systems
  - concurrent writes to a single file
  - provide high I/O bandwidth
- file system with hierarchical storage management (HSM)
  - two (or more) kinds of media: small-fast and large-slow
  - if the slow medium is *tape:* number of files must be kept manageable

Performance Frontiers (Basic)

Floating Point Operations per Second (FLOPS)

FLOPS (also: Flop/s)

- popular way to measure computational power of HPC systems
- in the order of several PetaFLOPS (PFLOPS) for the top HPC systems of 2017
  - peak performance of a powerful PC: $\approx$ 1 TeraFLOPS (TFLOPS)
- $1 PFLOPS = 1000 TFLOPS = 10^{15} FLOPS$
- also measurement for work performed by applications

TOP 500 list[1]

- lists the most powerful machines ranked by FLOPS
- measured using the *Linpack* benchmark
- updated twice a year
- shows past and current trends in HPC

[1] https://www.top500.org/lists/

Pitfalls of FLOPS

There are other critical resources than FLOPS

- memory latency & bandwidth
- network latency & bandwidth
- I/O performance

No clear correlation to real performance

Anything is possible:

- wasteful app with high FLOPS
- wasteful app with low FLOPS
- highly optimized app with high FLOPS
- highly optimized app with no FLOPS

FLOPS cannot tell the wasteful and the optimized apart!

Moore's Law

Moore's law[2] (1965, revised in 1975) states

- the complexity of integrated circuits[3] doubles approximately every two years
  - peak performance of CPU cores for HPC systems doubles too
- true in the past
- this increase in performance gain is no longer achieved
  - no more improvements of *sequential* performance
  - CPU clock rates have settled around 2.5 GHz
- but many cores are used for processing a task in *parallel*
- parallel computing will become increasingly relevant

[2] https://en.wikipedia.org/wiki/Moore%27s_law
[3] https://en.wikipedia.org/wiki/Integrated_circuit

Speedup, efficiency, and scalability

Speedup[4]

- speedup
  - relation between sequential and parallel runtime of a program
  - $S_n = \frac{T_1}{T_n}$
- where
  - T_1 = runtime on a single processor
  - T_n = runtime on n processors
- ideal case ("linear scaling")
  - $S_n = n$
- in practice linear speedup is not achievable due to overheads
  - synchronization
    (e.g. for waiting for partial results)
  - communication
    (e.g. for distributing partial tasks and collecting partial results)

[4] https://en.wikipedia.org/wiki/Speedup

Speedup, efficiency, and scalability

Efficiency[5]

- $E_n = \frac{S_n}{n}$

Scalability

- goal: efficiency remains high when the number of processors is increased
- also called: *good scalability*[6] of a parallel program

[5] https://en.wikipedia.org/wiki/Speedup
[6] https://en.wikipedia.org/wiki/Scalability

Speedup, efficiency, and scalability

Scalability in practice

- some problems can be parallelized trivially
  - e.g. rendering (independent) computer animation images[7]
  - nearly linear speedup also for a larger number of processors
- there are algorithms having a so-called sequential nature
  - e.g. alpha-beta game-tree search[8]
  - these have been notoriously difficult to parallelize
- typical problems in scientific computing[9] are somewhere in-between these extremes

[7] https://en.wikipedia.org/wiki/Render_farm
[8] https://www.chessprogramming.org/Parallel_Search#ParallelAlphaBeta
[9] https://en.wikipedia.org/wiki/Computational_science

Speedup, efficiency, and scalability

In general, the challenge is to achieve

- good speedups
- good efficiencies

Important aspect

- use the best known sequential algorithm for comparisons in order to get fair speedup results

Amdahl's law

Amdahl's law[10] (1967) states

- there is an upper limit for the maximum speedup of a parallel program
- which is determined by its sequential, i.e. non-parallelizable part
  - e.g. for initialization or I/O operations
  - more generally, for synchronization and communication overheads.

[10] https://en.wikipedia.org/wiki/Amdahl%27s_law

Amdahl's law

Example

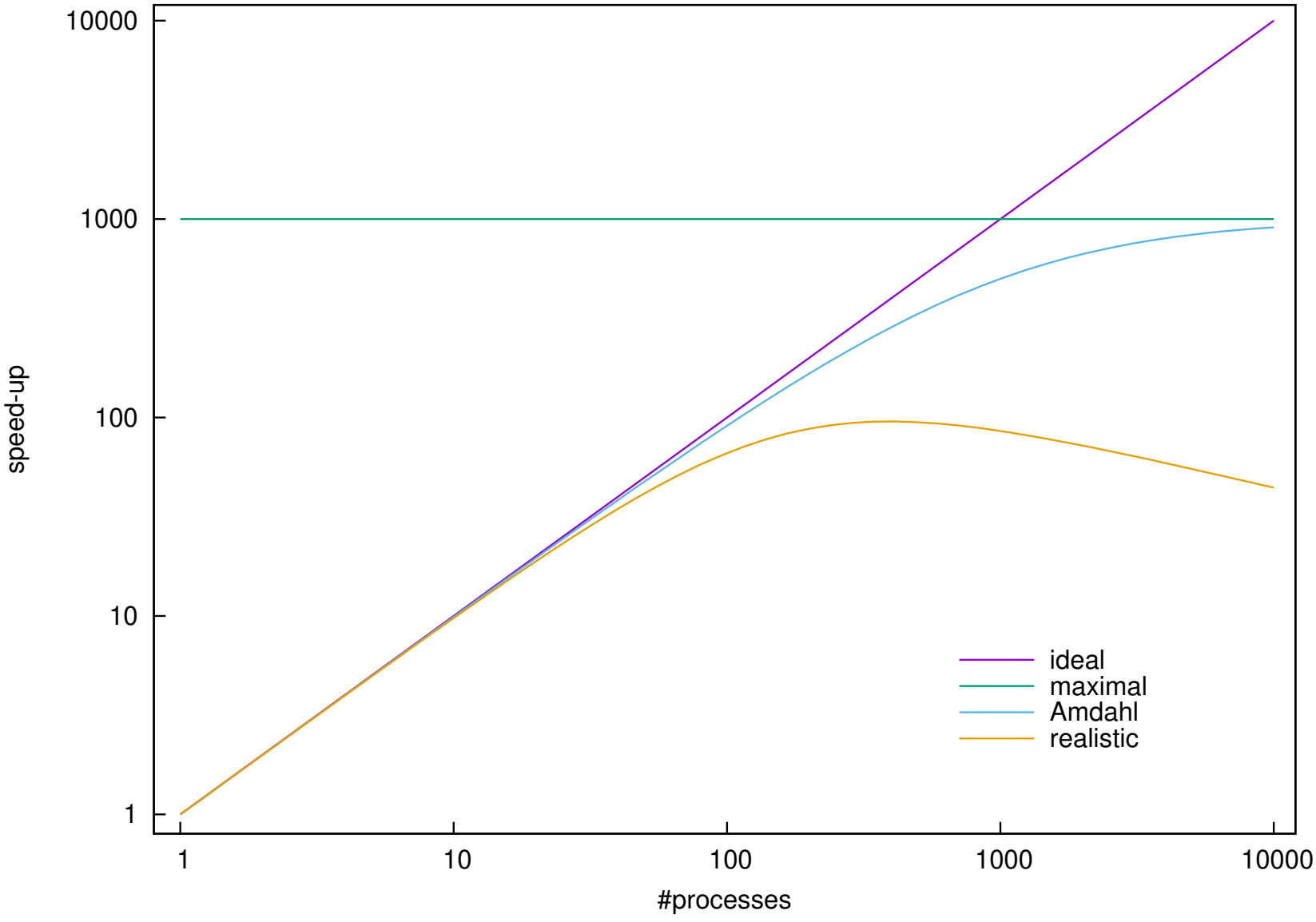
- sequential runtime: 20 hours on a single core
- non-parallelizable part: 10% (2 hours)
  - total runtime would be at least 2 hours
- parallelizable part: 90% (18 hours)
  - maximum speedup is limited by $\frac{20 hours}{2 hours} = 10$

Amdahl's law

Speedup calculation example

- cores used: 32
- runtime of parallelizable part $\geq \frac{18 hours}{32} = 0.56$ hours
- total runtime ≥ 2 hours $+ 0.56$ hours $= 2.56$ hours
- speedup $\leq S_{32} = \frac{20 hours}{2{,}56 hours} = 7.81$
- efficiency $\leq E_{32} = \frac{S_{32}}{32} = \frac{7.81}{32} = 24.41\%$.

Amdahl's law


10000
1000
100
10
1
speed-up
1
10
100
1000
10000
#processes
ideal
maximal
Amdahl
realistic

Parallelization Overheads (Basic)

Parallelization overhead

Parallelization always introduces overhead

- trivial parallelism (many independent tasks)
  - task management
- application parallelism (decomposition of a single application)
  - data communication (between processes)
  - synchronization (of threads)
  - additional operations, e.g.
    - global reduction operations (algorithmic level)
    - address calculations (software level)

Parallelization overhead

Other sources of parallel inefficiency

- the problem itself
  - unbalanced load
- software
  - serial parts (cf. Amdahl's law)
- hardware
  - NUMA
  - *false sharing*

Domain Decomposition (Basic)

Domain decomposition

- a technique for parallelizing programs that perform simulations in engineering or natural sciences
- needed on distributed memory systems
- the model to be simulated is defined in a certain geometric region
- that region is decomposed into domains
  - each process works on one or more domains
- typically domains have *halo regions*
  - data from surfaces of neighbouring domains
  - i.e. data from neigbouring processes

Performance impact (1)

Domain size

- data communication overhead = update of halo regions

$$\propto \frac{surface}{volume}$$

- example: d-dimensional cube
  - linear extension: L
  - volume: L^d
  - surface: $2dL^{d-1}$ (size of halo region)
  - surface / volume $= 2d/L$
- overhead becomes prohibitive if the volume becomes too small

Performance impact (2)

Domain shape

- example: rectangular domains
  - starting point: square
    - linear extension: L
    - volume: L^2
    - surface: $4L$
    - surface / volume: 4/L
  - rectangles with the same volume
    - linear extensions: $Lx \times L/x$
    - volume: L^2
    - surface: $2L(x + 1/x)$
    - $x = 1 \Rightarrow$ surface / volume $= 4/L$
    - $x = 2 \Rightarrow$ surface / volume $= 5/L$
    - ...
    - $x = L \Rightarrow$ surface / volume $= 2 + 2/L^2 \approx 2$
- long narrow domains are disadvantageous

Job Scheduling (Basic)

Motivation

HPC resources can be

- *shared* (e.g. login nodes, global file systems)
- *non-shared* (e.g. compute nodes)

Job scheduler

- manages resources
- goals
  - high resource utilization
  - fairness

Batch systems vs. time sharing systems (1)

Time sharing

- give users that are using the same computer at the same time the impression that the are using a dedicated computer
- is interesting for interactive use, e.g. on a login node

Batch systems vs. time sharing systems (2)

Batch systems

- non-interactive computer use
- processing of *batch jobs*
- batch job
  - a sequence of commands written to a file
- steps
  - job creation (edit job)
  - job submission (put job into a *batch queue*)
  - job monitoring (watch queue for start/completion)
  - job management (delete/cancel job)

Job scheduling

Scheduling

- process of selecting and allocating resources to jobs waiting for execution
- goals
  - maximize resource utilization
  - maximize throughput
  - minimize waiting time
  - minimize turnaround time (waiting time + execution time)

Workload managers

- implement job scheduling
- examples
  - SLURM
  - TORQUE

Scheduling algorithms

First-Come-First-Served (FCFS)

- jobs are executed in the order of submission
- simple algorithm: no optimization, poor performance
- basis for more sophisticated algorithms

Scheduling algorithms

Shortest-Job-First (SJF)

- uses execution time limits
- minimizes average waiting time
- *starvation* problem
  - if short jobs are constantly being submitted, a longer job might never be started

Scheduling algorithms

Priority

- affects the position of a job in the queue
- internal priorities (per batch job)
  - job size
    - number of nodes
    - time limit
    - memory limit
  - job aging
  - other resources, e.g. licenses
- external priorities (per user or group)
  - deadlines (e.g. for weather forecast)
  - amount of funds paid for the computer

Scheduling algorithms

Fair-share

- goal
  - achieve resource utilization that is proportionate to shares
- method
  - take job history into account

Scheduling algorithms

Backfilling

- fill nodes with jobs that
  - have lower priority than bigger jobs waiting for resources
  - fit into holes
    (are completed before the bigger jobs are planned to start)

Use of the Command Line Interface (Basic)

Command line usage

The prompt

- the prompt is defined in the variable PS1
- try: `echo $PS1`

system	definition	example
Bourne shell	`PS1='$ '`	`$`
bash	`PS1='\s-\v\$ '`	`bash-4.4$`
CentOS	`PS1='[\u@\h \W]$ '`	`[user1@host1 ~]$`

- for the root user '#' is used instead of '$'

Facilitate typing

File name completion

key	function
`<tab>`	command and filename completion

Command history

key	function
`<up-arrow>`	go to previous/older command(s)
`<down-arrow>`	go to newer command(s)

Facilitate typing

Command line editing

key	function
`<left-arrow>`	go 1 character to the left
`<right-arrow>`	go 1 character to the right
`<pos1>`	go to beginning of line
`<end>`	go to end of line
`<backspace>`	delete character to the left of the cursor
`<delete>`	delete character below the cursor

Control keys

Unexpected behaviour might occur when pressing control keys

key	function
`<ctrl-c>`	interrupt
`<ctrl-d>`	end of input
`<ctrl-l>`	clear screen
`<ctrl-s>`	pause output
`<ctrl-q>`	resume output
`<ctrl-z>`	pause process (resume with `fg`)

Control-keys known from Windows don't work!

Types of commands

A command can be

- an executable program
- a shell builtin
- a shell function
- an alias

The `type` builtin tells which is which

type examples

```
$ type ls
ls is /usr/bin/ls

$ type pwd
pwd is a shell builtin

$ type module
module is a function
module ()
{
  eval `/usr/share/Modules/$MODULE_VERSION/bin/modulecmd ba
}

$ type ll
ll is aliased to `ls -l'
```

Command line arguments

Arguments can be

- options
- filenames
- other parameters

Typical syntax of most commands

- *command [-options] [filenames]*

Command line syntax

Specifying options

description	example
-letter	`ls -l -R`
-letters	`ls -lR`
-letter value	`ls -I '*.o'`
--keyword	`ls --recursive`
--keyword value	`ls --ignore '*.o'`
--keyword=value	`ls --ignore=*.o`
-keyword	`find . -print`
-keyword value	`find . -name lost.c -print`
keyword=value	`dd if=infile bs=512 count=1`

Specifying filenames

Filenames can be specified with

- *absolute path*
  - absolute paths begin with /
  - all directories starting with the root directory are specified
- *relative path*
  - relative paths do *not* begin with /
  - specification relative to the *current working directory*

example	explanation
`file1`	file1 is in the current working directory
`./file1`	`.` stands for the current working directory
`../file2`	`..` stands for its parent directory
`../dir2/file2`	`../dir2` is a directory in the parent directory

Specifying filenames

Wildcards

character	matches
*	zero a more characters
?	a single character

Escape character \ (backslash)

characters	match
*	a literal *
\?	a literal ?

Getting help

Executable programs

- *man*-pages
  - if the name of the command is known
    - general format: `man` *`command`*
    - example: `man ls`
  - search for keywords in command descriptions
    - general format: `man -k` *`keyword`*
    - example: `man -k pdf`

Shell builtins

- `help` command
  - general format: `help` *`command`*
  - example: `help echo`

How executable programs are found

PATH

- programs are searched in directories specified in the `PATH` environment variable
- `PATH` is a colon separated list of directories

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
```

- the `which` command shows the full path to a command

```
$ which ls
/usr/bin/ls
```

Pitfalls

- There is **no undo**!
  - files can be accidentally deleted
  - files can be accidentally overwritten

- in theses examples file `b` is overwritten
  - `cp a b`
  - `mv a b`
  - `cat a > b`
  - `tar -cf b a`

Pitfalls

`-i` option

- some commands can ask for confirmation (`-i` option)
  - aliases might be predefined that include `-i`
  - this can be dangerous:
    - such aliases might *not* be predefined on a new system

Pitfalls

Starting programs/scripts that are in the working directory

- for security reasons `.` (the current working directory) is not included in `PATH`s
- scripts or programs that are in the current working directory must be started this way:
  - `./my.script`

Frequently used commands

Browsing the directory tree

command	description
`pwd`	print name of working directory
`cd`	change working directory
`ls`	list directory contents

Frequently used commands

Browsing the directory tree

command	description
`cd`	change to the *home* directory
`cd ..`	change to the parent directory
`cd` *`directory`*	change to the specified directory
`cd -`	change to the previous directory
`ls`	list contents of the *current* directory
`ls ..`	list contents of the parent directory
`ls` *`directory`*	list contents of the specified directory
`ls ~`	list contents of the home directory
`ls -l` *`[directory]`*	list contents in long format

Frequently used commands

Looking into text files

command	description
`less`	view file (forward-, backward movement, searching)
`cat`	print (concatenate) files
`head`	print the first lines of a file
`tail`	print the last lines of a file

Frequently used commands

Managing files and directories

command	description
`mkdir`	create (make) a directory
`rmdir`	remove (an empty) directory
`cp`	copy files
`cp -r`	copy recursively
`cp -rv`	copy recursively, print what is being copied
`mv`	move or rename files or directories
`rm`	remove/delete files
`rm -r`	remove files recursively
`rsync`	synchronize directories
`ln -s`	create a *symbolic link*

Frequently used commands

Searching and sorting

command	description
`grep`	search for strings in text files
`find`	search for files
`sort`	sort text files

- search for a string in all `.txt` files under the current working directory

```
find . -name '*.txt' -exec grep SearchText {} \;
```

Frequently used commands

Operations with text files

command	description
`wc`	word count - counts chars, world and lines
`diff`	compares 2 files
`diff3`	compares 3 files
`sed`	stream editor - text transformation

Frequently used commands

(Un)packing and (un)compressing

command	description
`tar`	(un)packing (archiving) files
`gzip`	(un)compressing files (extension `.gz`)
`bzip2`	(un)compressing files (extension `.bz2`)
`xz`	(un)compressing files (extension `.xz`)
`unzip`	extract files from `.zip` archive

Frequently used commands

Calculate and verify checksums

command	description
`cksum`	CRC checksums
`md5sum`	MD5 (128-bit) checksums
`sha256sum`	SHA256 (256-bit) checksums

Frequently used commands

Set execute permission

command	description
`chmod +x`	make a shell script executable

Frequently used commands

Check machine utilization

command	description
`ps`	snapshot report of current processes
`top`	real-time view of a running processes
`free`	print free and used memory
`vmstat`	report I/O *(virtual memory)* statistics
`df`	report disk space usage *(disk free)*
`du`	disk usage of directory hierarchies

- `-h` option
  - human-readable output format
  - available for: `free`, `df`, `du`

Frequently used commands

Remote access and file copy

command	description
`ssh`	secure shell - remote login
`scp`	secure copy - remote copy
`rsync`	remote (and local) synchronization

Frequently used commands

Miscellaneous commands

command	description
`date`	print current date and time
`time`	print resource usage of a command
`kill`	terminate a process by ID
`killall`	kill processes by name
`echo`	*print* command of the shell
`exit`	shell exit - logout

Environment variables

Environment variables are exported to all programs in a calling tree

action	command
definition	`export` *`name=value`*
print value	`echo $`*`name`*
print all values	`export`
print environment	`printenv`

Environment variables

Frequently used environment variables

variable	meaning
`HOME`	home directory (shortcut: `~`)
`LESS`	options for `less` (`-i`: case insensitive search)
`LOGNAME`	username (login name)
`PATH`	command search paths
`PWD`	current working directory
`TMPDIR`	directory for temporary (scratch) files
`USER`	username

Environment variables

Language settings

variable	comment
`LANG`	language and character encoding, e.g. `en_US.UTF-8`
`LC_*`	detailed language settings, cf. `man locale`

I/O redirection and pipes

Output from any command can easily be saved in a file

```
ls > listing1
```

Input can be read from a file (instead of being typed)

```
cat < input2
```

Pipes

- reading long output page by page

  ```
  command-producing-long-output | less
  ```

- filter output for error messages

  ```
  command | grep error-message-pattern
  ```

Remote login

Secure Shell clients

- Linux and MacOS
  - OpenSSH
- Windows
  - OpenSSH
  - *putty*
  - *MobaXterm*

Remote login

Public key authentication

- an alternative to password authentication
  - it is virtually impossible to guess a key
  - entering the password cannot be observed
- should be protected with a **passphrase**
- can be generated with `ssh-keygen`:
  - `ssh-keygen -t rsa -b 4096`
- the *public* key `~/.ssh/id_rsa.pub`
  - has to be appended to `~/.ssh/authorized_keys` on the remote computer
  - or has too be sent/uploaded to the computing center
- `ssh-add` and `ssh-agent` can be used
  - to unlock the *private* keys
  - the passphrase has to be entered only once per local session

Remote login

Agent forwarding

- is a technique to connect to a third computer
- `ssh-agent` is needed

Example

- log into `hpc_1`

```
your_computer$ ssh -A user_1@hpc_1.example.com
```

- from there, log into `hpc_2`

```
hpc_1$ ssh user_2@hpc_2.example.com
```

- copy a file from `hpc_1` to `hpc_2`

```
hpc_1$ scp example.c user_2@hpc_2.example.com:
```

Text editors

- on an HPC cluster one has to work with text files:
  - batch scripts
  - input files

- on the cluster itself
  - *terminal mode* is typical
    (or *text mode* in contrast to a *graphical mode*)
  - text editors are available in text mode

Text editors

Classic Unix/Linux text editors

- `vi`, `vim`
  - is automatically installed on all Linux systems
- `GNU emacs`
  - is probably installed on your HPC cluster as well

Small, more intuitive editor

- `nano`
  - is installed on many systems

Text editors

Least thing to know: key strokes to quit

editor	keys	action
`vi`	`<esc>:q!`	quit without saving
`vi`	`<esc>ZZ`	save and quit
`emacs`	`<cntl-x><cntl-c>`	quit
`nano`	`<cntl-x>`	quit

`emacs` and `nano` ask how to proceed with unsaved files

Text editors

Using a graphical interface

- `vim` and `emacs` have graphical interfaces
- other graphical editors might be installed:
  - `gedit`
  - `kate`
- a graphical editor requires *X11 forwarding*
  - is switched on with `ssh -X`
  - can be slow
- an editor on the local computer can be used
  - copy files back and forth
  - work transparently on the remote system after mounting its file system with *SSHFS*

Using Shell Scripts (Basic)

Using shell scripts

What is a shell script?

- a sequence of commands that is written into a file

```
cd /work/user1/project1
my-simulation-program input1
```

Using shell scripts

More compliated scripts use

- variables
  - `x=foo`
  - `y=$foo`
- arguments from the command line (unusual for batch scripts)
  - `$1 $2 ...`
- execution control
  - `if`
  - `case`
  - `for`

Scripting for batch jobs

Manipulating filenames (character string processing)

action	command	result
initialization	`a=foo`	`a=foo`
	`b=bar`	`b=bar`
concatenation	`c=$a/$b.c`	`c=foo/bar.c`
	`d=${a}_$b.c`	`d=foo_bar.c`
get directory	`dir=$(dirname $c)`	`dir=foo`
get filename	`file=$(basename $c)`	`file=bar.c`
remove suffix	`name=$(basename $c .c)`	`name=bar`
	`name=${file%.c}`	`name=bar`
remove prefix	`ext=${file##*.}`	`ext=c`

Scripting for batch jobs

Recommendation: Never use white space in filenames!

- is error prone
- quoting becomes necessary: `dir=$(dirname "$c")`

Scripting for batch jobs

Temporary files

- choice of the directory/file system
  - `tmp` might be too small
  - `$TMPDIR` is a candidate
  - consider *local* vs. *global* file systems
  - assume that `/scratch` is suited and set
    - `top_tmpdir=/scratch`
- unique filenames
  - `mktemp` generates names from templates
  - a sequence of `X`s is replaced by a unique value
  - a directory with that name is created
  - include `$USER` for easy identification
    - `my_tmpdir=$(mktemp -d "$top_tmpdir/$USER.XXXXXXXX")`

Scripting for batch jobs

Temporary files

- automatic deletion
  - `trap "rm -rf $my_tmpdir" EXIT`
- now the temporary directory is ready
  - `cd $my_tmpdir`
  - *`do some work`*

Scripting for batch jobs

Tracing command execution

- `set -v`
  - print commands as they appear literally in the script
- `set -x`
  - commands are printed as they are being executed (i.e. with variables expanded)

Scripting for batch jobs

Error handling

- `set -e`
  - exit script immediately if a command ends with an error (non-zero) status
  - handling exceptions: *or* operator `||`

```
command_that_could_go_wrong || true
```

- `set -u`
  - exit script exit if an undefined variable is used
  - handling exceptions:

```
if [[ ${variable_that_might_not_be_set-} = test_value ]]
then
    ...
fi
```

Scripting for batch jobs

Trivial parallelization

- starting more than one executable
- example: running on 2 graphics cards:

```
CUDA_VISIBLE_DEVICES=0 cudaBinary1 input1 &
CUDA_VISIBLE_DEVICES=1 cudaBinary2 input2 &

wait
```

- more powerful tool: *GNU Parallel*[1]
  - can start many tasks
  - can process a task queue

[1] https://www.gnu.org/software/parallel

Selecting the Software Environment (Basic)

Environment Modules

Introduction

- a tool for managing environment variables of the shell
- `module load` command
  - extends variables containing search paths (e.g. `PATH`)
- `module unload` command
  - inverse operation
  - removes entries from search paths.
- software can be provided in a modular way

Environment Modules

Initialization

- the `module` command is a *shell function*
- needs to be defined in every instance of the shell
  - interactive environments
    - is typically handled automatically
  - batch environments
    - explicit initialization might be necessary (see documentation of your cluster)

Environment Modules

Naming

- format of Module names
  - `program`
  - `program/version`
- default version
  - might be explicitly defined in your Module system
  - otherwise, Module guesses the latest version
- recommendation
  - *always* specify a version

Environment Modules

Dependences and conflicts

- dependences
  - enforces that other Modules must be loaded first
- conflicts
  - enforces that other Modules must be unloaded first

Environment Modules

Caveats

- *Modules* suggest modularity
  - true for application Modules
  - no longer true for compiler and library modules
- solutions for compilers and libraries
  - `version` is augmented by additional information
  - a toolchain is built
    - a compiler has to be loaded first
    - then MPI Modules becomes visible
    - then libraries and software becomes visible

Environment Modules

Important commands

- `module list`
- `module avail`
- `module load` *`program[/version]`*
- `module unload` *`program`*
- `module switch` *`program program/version`*
- `module [un]use [--append]` *`path`*

Environment Modules

Self-documentation

- `module display` *`program/version`*
- `module whatis` *`[program/version]`*
- `module help` *`program/version`*
- `module help` (help on `module` itself)

See also

- `man module`

Use of a Workload Manager (Basic)

Workload managers

Tasks

- job control
  - submission
  - monitoring
  - cancellation
- scheduling and resource management
  - select waiting jobs for execution
  - allocate and monitor resources
- accounting
  - record resource usage

Workload managers

Popular workload managers

- SLURM
  - *Simple Linux Utility for Resource Management*
  - includes scheduling algorithms
- TORQUE
  - *Terascale Open-source Resource and QUEue Manager*
  - needs a scheduler in addition (e.g. Maui or Moab)

Workload managers

TORQUE

- PBS (Portable Batch System) history
  - TORQUE is an open source implementation of PBS
  - other PBS implementations: OpenPBS, PBS Pro(fessional)
  - PBS started in 1991
- Command syntax
  - command names begin with a `q`
    - `qsub`
    - `qstat`
    - `qdel`

Workload managers

SLURM

- has gained much popularity in the recent past
- is open source
- commercial support since 2010
- command syntax
  - command names begin with an **s**
    - `sbatch`
    - `squeue`
    - `scancel`

Workload manager commands

Job submission

SLURM	PBS/TORQUE
`sbatch [options] [`*`filename`*`]`	`qsub [options] [`*`filename`*`]`

- `options` specify
  - *resource requirements*
  - other job properties
- *`filename`*
  - name of the batch script
  - if not given, script is read from *stdin*
- results
  - job appears in the job queue
  - a *job ID* is assigned

Workload managers

Resource specifications

	SLURM	PBS/TORQUE
number of nodes	`--nodes=`*n*	`-l nodes=`*n*
processes per node	`--tasks-per-node=`*n*	`-l nodes=`*n*`:ppn=`*p*
time limit	`--time=`*hh:mm:ss*	`-l walltime=`*hh:mm:ss*
	`--time=`*minutes*	`-l walltime=`*seconds*
queue/partition	`--partition=`*part*	`-Q` *queue*

Workload managers

Job name and log file names

	SLURM	PBS/TORQUE
job name	`--job-name=`*jobname*	`-N` *jobname*
stdout file	`--output=`*filename*	`-o` *filename*
stdin file	`--error=`*filename*	`-e` *filename*
default names	`slurm-`*jobID*`.out`	*jobname*`.o`*jobID* *jobname*`.e`*jobID*
use *jobID*	`--output=`*file*`.o%j`	
join *stderr* into *stdout*	specify `--output` but not `--error`	`-j oe`

Workload managers

E-mail notification

	SLURM	PBS/TORQUE
e-mail address	`--mail-user=`*`address`*	`-M` *`address`*
notifications	`--mail-type=BEGIN`	`-m b`
	`--mail-type=END`	`-m e`
	`--mail-type=FAIL`	`-m a`
	`--mail-type=ALL`	`-m abe`

Workload managers

Structure of batch scripts

- `options` can be specified on the command line or at the beginning of batch scripts

SLURM	PBS/TORQUE
`#!/bin/bash`	`#!/bin/bash`
`#SBATCH --job-name=job1`	`#PBS -N job1`
`#SBATCH --nodes=2`	`#PBS -l nodes=2`
`#SBATCH --time=00:10:00`	`#PBS -l walltime=00:10:00`
command	*command*
...	...

Workload managers

Environment variables that can be used in batch scripts

	SLURM	PBS/TORQUE
job ID	`$SLURM_JOB_ID`	`$PBS_JOBID`
job name	`$SLURM_JOB_NAME`	`$PBS_JOBNAME`
nodes allocated	`$SLURM_JOB_NODELIST` (a list)	`$PBS_NODEFILE` (a filename)
working directory at submit time	`$SLURM_SUBMIT_DIR`	`$PBS_O_WORKDIR`
default working directory	`$SLURM_SUBMIT_DIR`	`$HOME`

Workload managers

Environment variables

- SLURM provides environment variables that contain resource specifications

	SLURM
number of nodes	`$SLURM_JOB_NUM_NODES`
processes per node	`$SLURM_TASKS_PER_NODE`
CPUs (threads) per process (value from `--cpus-per-task`)	`$SLURM_CPUS_PER_TASK`

Workload manager commands

Show job queue / job status information / job ID

	SLURM	PBS/TORQUE
all jobs	`squeue`	`qstat`
own jobs	`squeue -u $USER`	`qstat -u $USER`
single job	`squeue -j` *`jobID`*	`qstat` *`jobID`*

Workload manager commands

Job status indicators

	SLURM	PBS/TORQUE
pending/queued	P	Q
running	R	R
completed	CD	C
failed	F	
cancelled	CA	

Workload manager commands

Cancel a waiting job / abort a running job

SLURM	PBS/TORQUE
`scancel` *jobID*	`qdel` *jobID*

Workload managers

Starting interactive sessions/batch jobs

SLURM	PBS/TORQUE
`salloc` *`[resources]`*	`qsub -I` *`[resources]`*

Workload managers

SLURM command `srun`

- in batch jobs
  - launches parallel/MPI program
  - replaces `mpirun`/`mpiexec`
- in interactive batch jobs (after `salloc`)
  - is necessary to start *any* program on the allocated node(s)
- in a login session
  - runs a (parallel) program under control of the batch system

Workload managers

Other SLURM commands

- `sinfo`
  - shows information on nodes and partitions
- `sacct -j` *`jobID`*
  - shows accounting information

Benchmarking (Basic)

Benchmarking

Definition

- determination of hard- or software performance by controlled experiments
- *benchmark* can refer to
  - a controlled experiment with a single program
  - a set of programs used for benchmarking

Motivation

- understanding performance of parallel applications
  - is there a speedup?
  - is the speedup reasonably large?

Benchmarking hardware

Linpack and the TOP500 list

- TOP500
  - `https://www.top500.org`
  - list of the 500 fastest computers in the world
- *Linpack* benchmark
  - `http://www.netlib.org/benchmark/hpl`
  - determines the ranking in the TOP500 list

Benchmarking parallel software

Questions that should always be answered

- What is the scalability of my program?
- How many cluster nodes can be maximally used, before the efficiency drops to values which are unacceptable?
- How does the same program perform in different cluster environments?

Benchmarking

General tuning possibilities

- use of hyper-threads
- mapping of processes to nodes
- pinning of processes/threads to CPUs/cores
- choice of compilers
  - e.g. GNU, Intel, PGI
- choice of optimization levels
  - `-O2`, `-O3`, ...
  - PGO (Profile Guided Optimization)
  - IPA/IPO (Inter-Procedural Analyzer/Optimizer)
- choice of libraries
  - BLAS (Basic Linear Algebra Subprograms)
  - FFT (Fast Fourier Transform)

Benchmarking

General questions

- Are the best known algorithms employed?
- Does observed performance persist if the environment changes?

Benchmarking

Benchmarking parallel programs

- MPI programs
  - measure runtimes depending on the number of nodes
- OpenMP programs
  - measure runtimes depending on the number of cores

Benchmarking

Parallel speedup

$$S = \frac{\text{sequential runtime}}{\text{parallel runtime}}$$

Parallel efficiency

$$E = \frac{S}{\text{number of nodes or cores}}$$

Benchmarking

Example: calculation of π

version	runtime [s]	cluster nodes	total cores	speedup	efficiency
OpenMP	2800.0		1	1.00	100%
OpenMP	1414.1		2	1.98	99%
OpenMP	707.1		4	3.96	99%
OpenMP	360.8		8	7.76	97%
MPI	180.5	1	16	1.00	100%
MPI	92.1	2	32	1.96	98%
MPI	47.5	4	64	3.80	95%
MPI	25.1	8	128	7.19	90%

Benchmarking

Runtime measurement

- shell built-in `time` command
  - can be used for any runtime measurement

```
time mpirun ... my-mpi-app
```

- `/usr/bin/time/`
  - reports usage of other resources (memory, I/O) as well
  - interesting for single-process programs (including OpenMP)

```
export OMP_NUM_THREADS=...
/usr/bin/time my-openmp-app
```

Benchmarking

Scaling

- good scalability
  - efficiency remains high when the number of processors is increased

Weak scaling

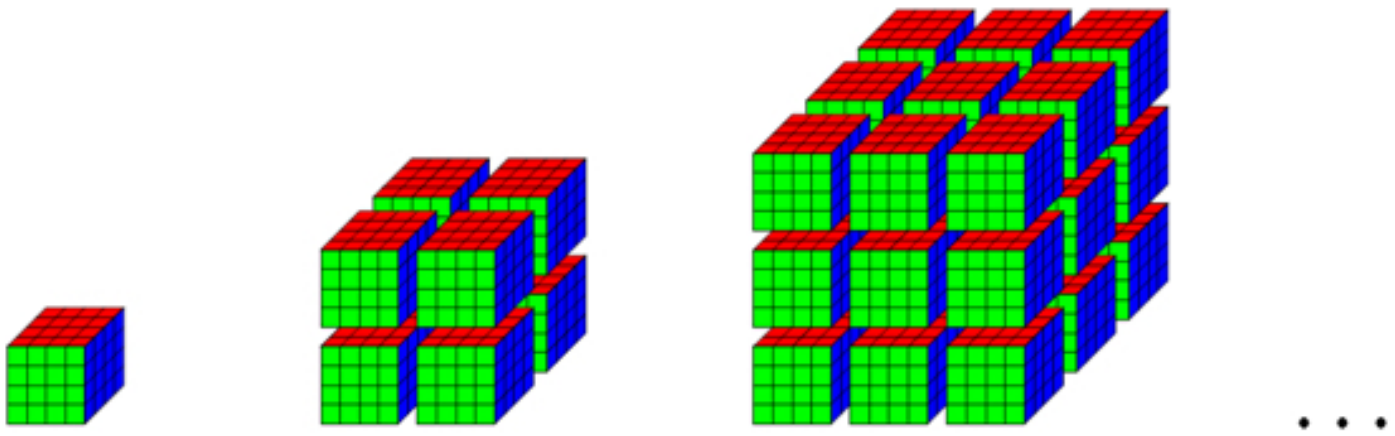
- problem size $\propto$ number of cores
  - "How big may the problems be that I can solve?"

Strong scaling

- problem size $\equiv$ constant
  - "How fast can I solve a problem of a given size?"

Benchmarking

Weak scaling



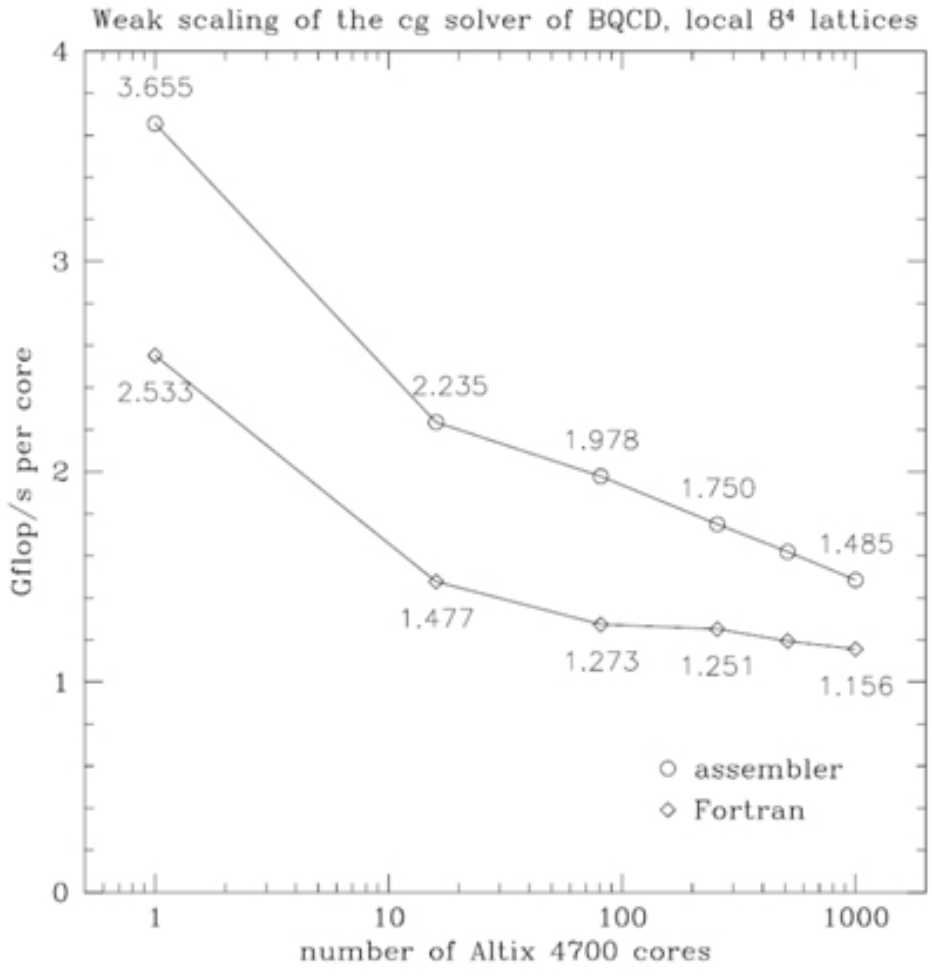
1 process $2^3 = 8$ processes $3^3 = 27$ processes

Benchmarking

Typical weak scaling behaviour

- communication overhead of boundary exchange increases at low process counts
- sustained performance per process is roughly constant at high process counts

Weak scaling plot example



Benchmarking

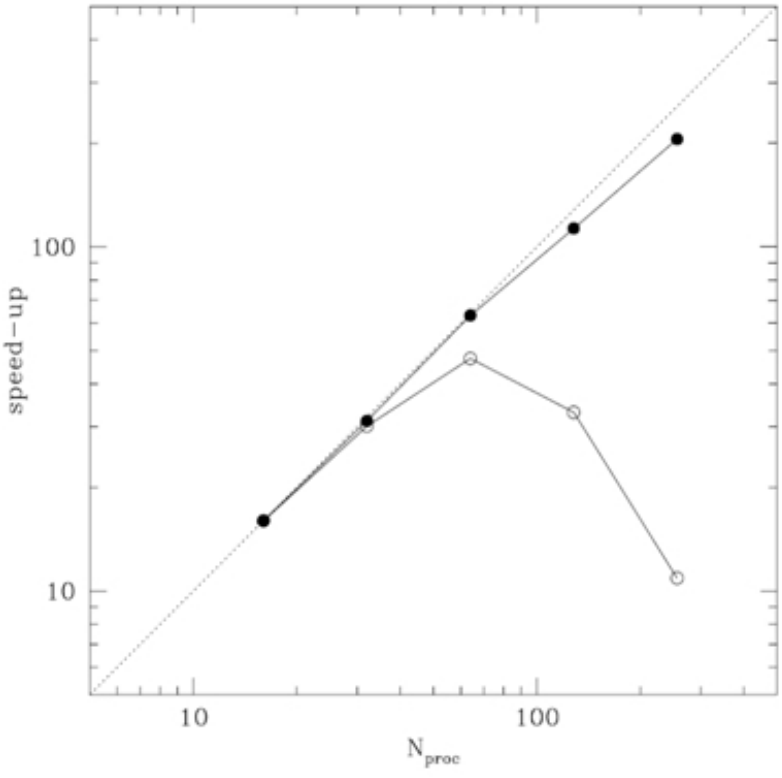
Typical strong scaling behaviour

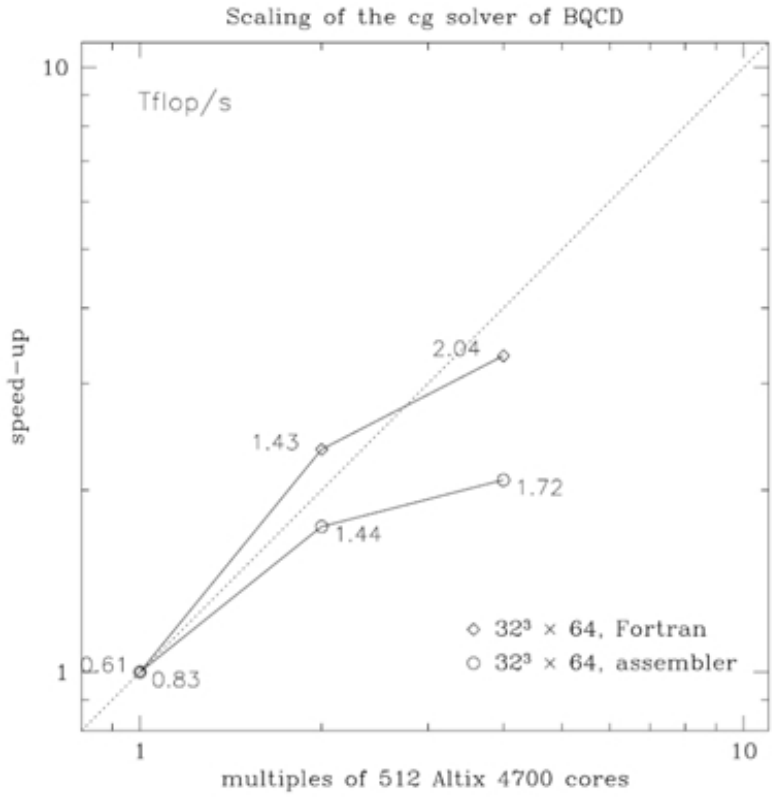
- domain size per process decreases
- communication overhead increases
- sustained performance per process decreases

Goal

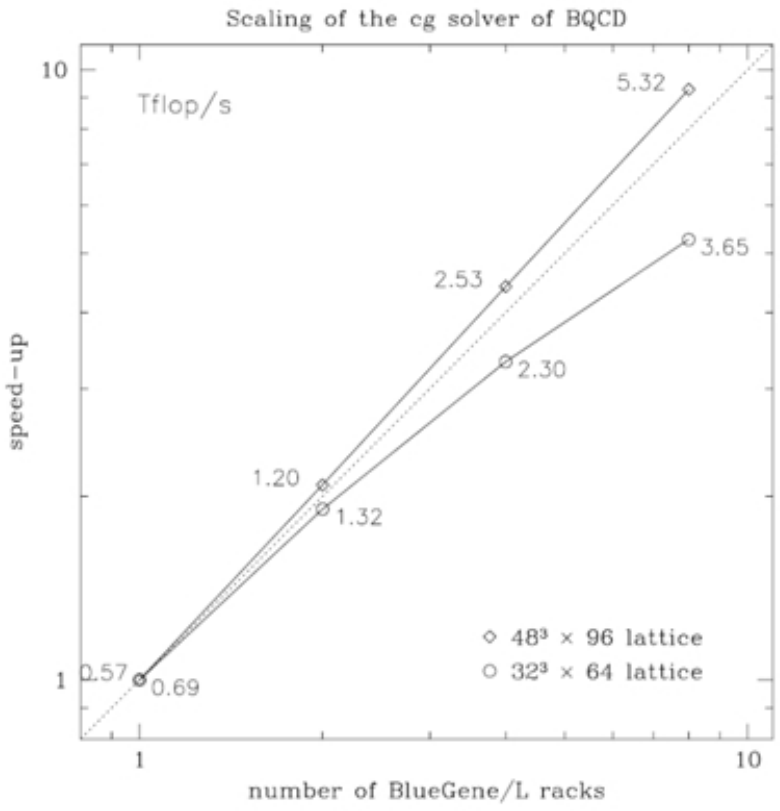
- determination of an optimal number of processes to use

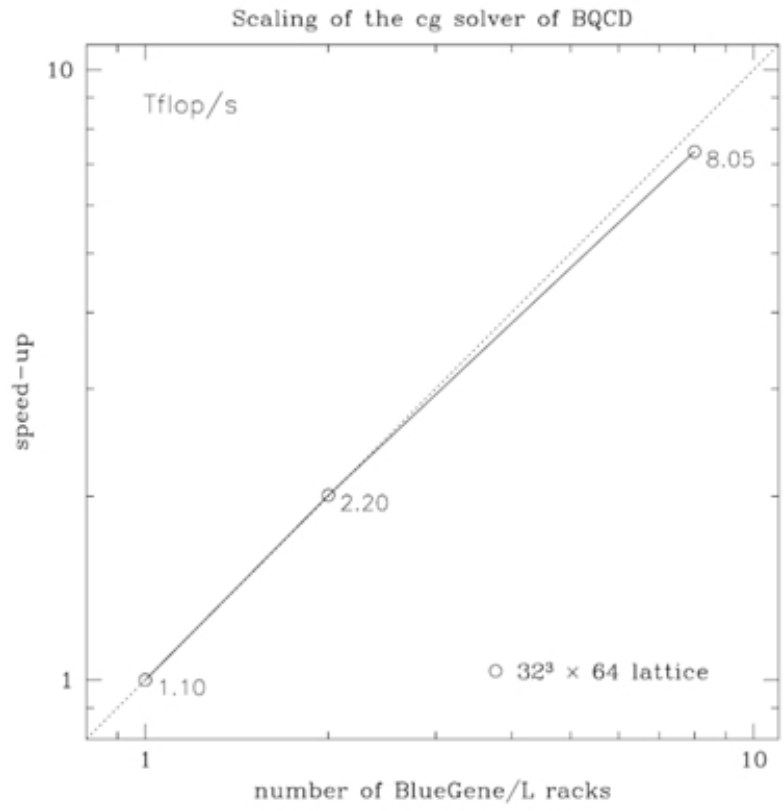
Strong scaling plot examples (1)



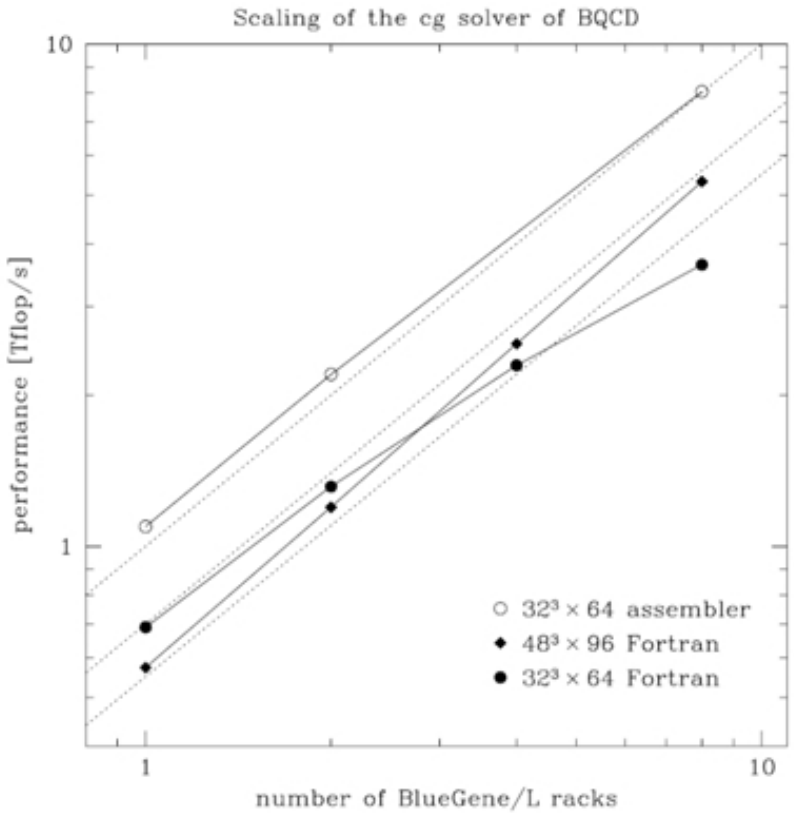


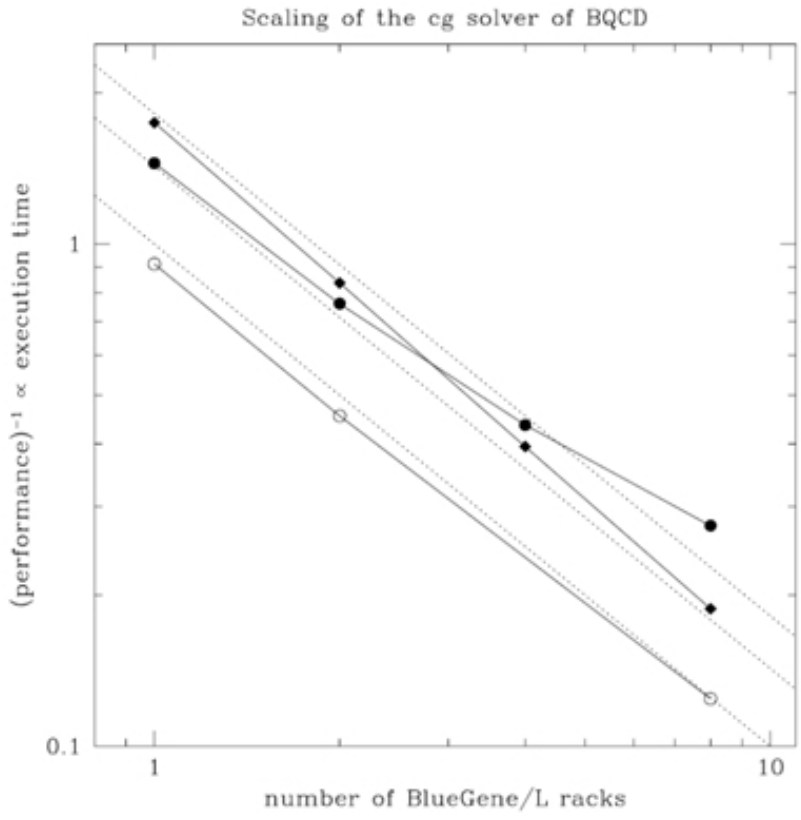
Strong scaling plot examples (2)





Strong scaling plot examples (3)





Benchmarking / tuning

Profile Guided Optimization (PGO)

- step 1
  - run the instrumented (and therefore relatively slow) version of the binary with representative input data
  - collect information about which branches are typically taken and other typical program behavior
- step 2
  - recompile with this information to build a faster program

Benchmarking / tuning

I/O

- choose an adequate file system
  - global file system with HDDs
  - local file systems with SSDs

Benchmarking pitfalls

Break-even considerations

- consider efforts
  - HPC resources explicitly used for that purpose
  - human time

Benchmarking pitfalls

Definition of speedup S

$$S = \frac{T_1}{T_{parallel}}$$

Conventional speedup

- use the same version of an algorithm (the same program) to measure T_1 and $T_{parallel}$

Fair speedup

- use best known sequential algorithm to measure T_1

Benchmarking pitfalls

Features of current CPU architectures

- varying clock rates and *turbo* modes
  - for benchmarking CPUs should be in "thermal equilibrium"
- hardware threads / hyper-threads
  - counted as CPUs by the operation system
  - it might not be clear what counts as a core

Benchmarking pitfalls

Shared resources

- other user's activities can influence runtime
  - I/O on global file systems
  - program execution on shared nodes

Benchmarking pitfalls

Reproducibility

- there are parallel algorithms which may produce non deterministic results and runtimes, due to inherent effects of concurrency
  - some parallel tree-search algorithms
  - event-driven simulations