December 2019

GSWHC-B Getting Started with HPC Clusters

Kai Himstedt, Nathanael Hübbe, and Hinnerk Stüben

Universität Hamburg

Preface

This text was written as part of the PeCoH project (Performance Conscious HPC)[1]. One goal of PeCoH was the development of an HPC certification program. In a first step competences in HPC were listed and structured in a *skill tree*[2]. A tree structure was chosen because it can represent a hierarchical nature of competences. The tree is a dependence tree, i.e. each skill depends on the skills (prerequisites) that are listed deeper in tree. The idea is to enable the creation of *views*. A view is a selection of main topics, but includes all prerequisites automatically (analogy: main topics are like targets in Makefiles).

The main sub-trees in the skill tree are:

- general HPC knowledge
- performance engineering
- software engineering
- use of the HPC environment

Each skill (each leaf of the tree) was assigned a *role* of the user and a *level* of expertise. The roles are:

- tester (running programs)
- builder (compiling programs)
- developer (writing programs)

The levels are:

- basic
- intermediate
- expert

In a second step an "HPC-Führerschein" for beginners should be established (literal translation: "HPC driving licence"; the meaning corresponds to a *Golf Proficiency Certificate*

[1] https://www.hhcc.uni-hamburg.de/pecoh/
[2] https://www.hhcc.uni-hamburg.de/files/hpccp-concept-paper-180601.pdf

in Singapore). The text in hand provides learning material for this objective. It is one *view* on the skill tree for *testers* at the *basic* level.

Besides this (pdf) version the same text, starting from “Index”, is available as an online (html) version. Both versions were generated by an automated build process from the same (markdown) source.

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.

Index

K1.1-B System Architectures 5

K1.2-B Hardware Architectures 8

K1.3-B I/O Architectures 11

K2-B Performance Modeling 13

K2.1-B Performance Frontiers 13

K3.3-B Parallelization Overheads 20

K3.4-B Domain Decomposition 22

K4-B Job Scheduling 24

USE1-B Use of the Cluster Operating System 30

USE1.1-B Use of the Command Line Interface 30

USE1.2-B Using Shell Scripts 36

USE1.3-B Selecting the Software Environment 39

USE2.1-B Use of a Workload Manager 42

PE3-B Benchmarking 53

GSWHC-B Getting Started with HPC Clusters

Level: basic

Introduction – What is HPC?

A tautological definition of HPC is: "You are doing HPC when you are using HPC hardware." HPC hardware is needed whenever your personal computer (or workstation) becomes to small and/or to slow to complete your computing tasks. Powerful hardware is the common denominator. However, "there is no free lunch", when it comes to speeding up computations. Because a single core delivers roughly the same compute power on the personal computer and on an HPC system, no significant speedup can be expected without employing some form of *parallel computing*.

HPC stands for *high-performance computing*. The goal of traditional HPC is to run *computer simulations in natural sciences and engineering* as fast as possible. Typically, computer simulations need a lot of aggregated computing power to accomplish a single task. In order to achieve this, parallelization at the task level is a central objective (in addition to achieving high performance at the sequential level). Performance is measured in double-precision floating-point operations per second (abbreviated as FLOPS or Flop/s).

The highest performance can be achieved with supercomputers that have very many powerful compute nodes and a powerful communication network. The communication network is essential to scale a single instance of an HPC application up to many nodes. This kind of computing is sometimes called *capability computing*. In contrast, *capacity computing* refers to the number of instances that can be run at the same time. In practice, HPC projects have capability and capacity aspects.

Most newcomers to HPC systems are attracted by the capacity of these systems. Usually, their overall computing needs are small compared with those in traditional HPC. As a consequence they do not care about the actual performance of their (serial or parallel) programs. While this makes sense, if the overall time of machine usage is small (and the effort to speedup a process is more expensive than just running it), one might not call this kind of machine usage high-performance computing as well. Nonetheless, HPC systems deliver performance to those users in terms of a different metric: shorter time-to-solution (higher throughput) rather than Flop/s. In fact, many applications perform high-performance searching (e.g. in gene sequencing) leading to a third performance metric.

HPC systems are sometimes called *Linux clusters*. They have similar software environments (at least those that are used for academic research) that put some demands on users:

- 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

In addition:

- *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?

The goal of this collection of texts is to provide short introductions to these topics to newcomers to HPC.

K1.1-B System Architectures

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about the hardware components of an HPC cluster and their functions (basic level)

Level: basic

HPC Cluster Architecture

This is an introductory description of HPC cluster hardware and operation. You will probably find that much of this text is valid for the cluster you want to use, too. Since every cluster is different, you should look at the description of the configuration of your cluster in addition.

An HPC cluster is built out of a few to many servers that are connected by a high performance communication network. The servers are called *nodes*. HPC clusters use batch systems for running compute jobs. Interactive access is usually limited to a few nodes.

In the Figure below the architecture of an HPC cluster is sketched. The Figure shows hardware components that basically all clusters have. In the following text the functions of these components, and others, are explained.

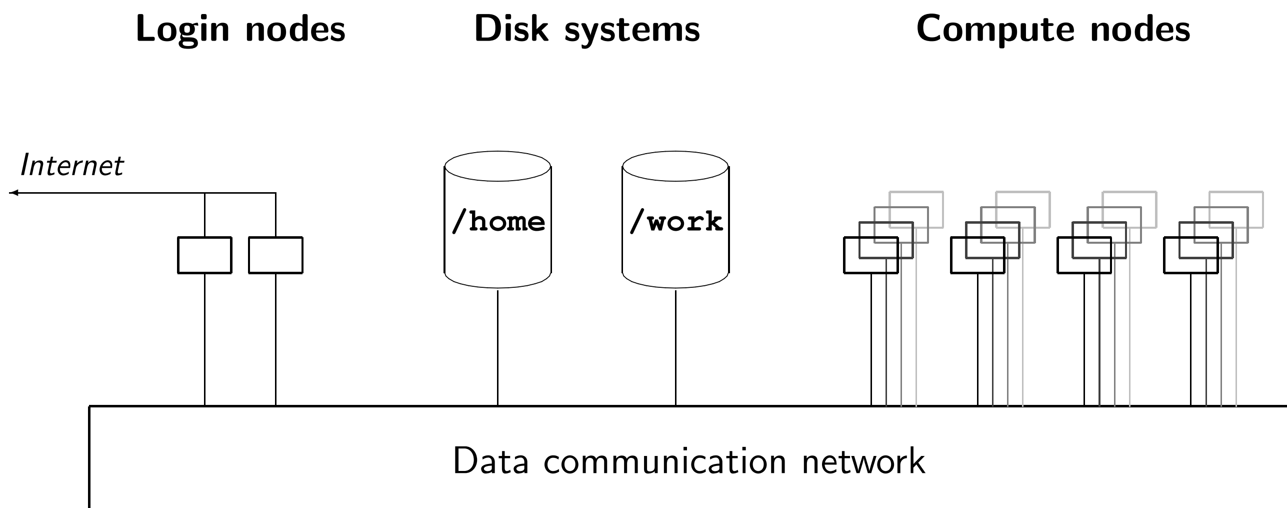

Figure 1: HPC cluster architecture

Nodes

There are nodes with several functions:

Login or gateway nodes. This is the node that you are on after logging into the cluster. There can be more than one of such nodes. On many clusters this is the node on which

you can work interactively, i.e. where you compile programs, prepare and manage batch jobs. It can be that this node is configured only as a gateway. Then you have to hop to (i.e. log into) another node that is configured for interactive work.

Compute nodes are the workhorses of a cluster. Batch runs are being executed there. A standard compute node has CPUs and main memory. It can be equipped with additional or special compute devices (e.g. GPUs, vector cards or FPGAs). The number of CPUs and the main memory size vary. Many clusters run diskless, i.e. their compute nodes have no (local) disk. Local disks are nice to have for heavy scratch I/O. However, spinning disks are components that fail relatively often, and it is not economic to homogeneously put disks into each node if only a few applications need scratch I/O. Compute nodes can be made available exclusively (not-shared) to a batch job or they can be shared by more than one job.

Admin or system nodes. These nodes are mentioned for completeness. They work in the background and are necessary for the operation of the cluster, e.g. for running the batch service, or starting and shutting down compute nodes.

Disk nodes provide global file systems, i.e. file systems that can be used on all other kinds of nodes. Several nodes can provide a single file system. The exact mechanism is hidden from the user. The user just sees the file system and does not need to know of the nodes providing it.

Special nodes. The node types mentioned above are common on any cluster. In addition there can be special nodes, e.g. for data movement, visualization, or pre- and post-processing of large date sets.

Head node. The term head node is not used consistently. It can mean login or admin node. In both cases *head* means *management* (either by a user or an administrator).

Global file systems

Global file systems are available on all nodes of the cluster (in particular on login and compute nodes). Global file systems are convenient because their files can be accessed directly on all nodes. This functionality is known from network file systems. Additional functionality is provided by parallel file systems. Quantitatively, parallel file systems offer higher I/O performance than classic network file systems. Qualitatively, they allow several processes to write into the same file.

Communication network

The communication network has two purposes. It enables high speed data communication for parallel applications running on multiple nodes, and it provides a high speed connection to the disk systems in the cluster. Performance of the communication network is designed according to the demands of the parallel applications running on the machine and the I/O requirements. If performance requirements are low, a cluster can be built without a high performance network, just with an ethernet network. In addition to the communication network, that is used by applications, there can be additional networks that are used for system purposes (e.g. for managing nodes).

Connections to the internet

Usually, only the login/gateway nodes can be reached from the outside. Some clusters have data mover nodes that are reachable as well.

It depends on the policy of the computing center which nodes can connect to the internet from inside the cluster.

K1.2-B Hardware Architectures

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about parallel computer architectures, in particular: the distinction between shared and distributed memory systems (basic level)

Level: basic

Parallel Computer Architectures

HPC computer architectures are parallel computer architectures. A parallel computer is built out of

- compute units,
- main memory,
- and a high speed network.

Compute units

Compute units can be:

Traditional **CPUs.** Although CPU stands for Central Processing Units, there is no central, i.e. single, processing unit any more. Today, all CPUs have multiple compute cores which all have the same functionality.

GPUs (Graphical Processing Units) or **GPGPUs** (General Purpose Graphical Processing Units). Originally, GPUs were used for image processing and displaying images on screens. Then people started to utilize the compute power of GPUs for other purposes and special GPUs were built for computing purposes only, they even cannot be connected to any display. In particular, GPGPUs can do double precision floating point arithmetic in hardware and can be equipped with ECC (Error Correcting) memory.

Special devices like **FPGAs** (Field-Programmable Gate Arrays) or **vector computing units**. FPGAs are devices that have configurable hardware. Configurations are specified by hardware description languages. With this respect configuring FPGAs is closer to designing hardware than to well-known programming. FPGAs are interesting if one uses them to implement hardware features that are not available in CPUs or GPUs. A prominent example is low precision arithmetic that needs only a few bits. Vector units are successors of vector computers (i.e. the first generation of supercomputers). They are supposed to provide higher memory bandwidth than CPUs.

Main-memory architecture

At an abstract level the high speed network connects compute units and main memory. This view leads to three main parallel computer architectures:

- shared memory
- distributed memory
- NUMA (Non-Uniform Memory Access)

Shared memory

In a shared memory system all compute units can directly access the whole main memory. In practice this means that a shared memory system is a single computer. Today this is the standard computer architecture. Desktops, laptops, and even mobile telephones have more that one compute core that share the same memory. Typically, those systems have a single CPU socket. This implies that memory access is *symmetric*, i.e. the time it takes to access any memory address is the same for all compute units. Such systems are called SMPs (Symmetric Multiprocessor systems). In the past SMPs were built with multiple (single-core) CPU sockets. Some years ago two-socket (multi-core) compute nodes were still SMPs. Today NUMA (Non-Uniform Memory Access, see below) is normal. The consequence of NUMA is that in order to achieve best performance these computers must be used in such a way that most memory access is to the local NUMA domain (data locality).

The advantage of a shared memory system is that programming parallel applications in general needs less effort than programming for distributed memory systems. The disadvantage is that scaling is limited to the size of a single shared memory node.

Distributed memory

The prime example of a distributed memory system is a PC cluster: individual computers are connected with a network. In a distributed memory system each compute node can only access its own memory directly. However, two nodes can exchange data via the network. In principle there is no difference between PC and HPC clusters except that nodes and network of an HPC cluster are more powerful. Clusters are built out of individual computer boxes. In the past high end distributed memory systems were higher integrated, i.e. several compute nodes were integrated on a single backplane.

In general, the effort for programming parallel applications for distributed systems is higher than for shared memory systems. Distributed memory systems can be huge and make it possible to use hundreds of thousands of compute cores to run a single application. Sometimes this kind of processing is called *massively parallel processing (MPP)*. However, a classic MPP system is more specialized than a cluster: the operating system is kept simpler (by employing micro kernels) and batch scripts typically do not run on the actual compute nodes but rather on a host.

NUMA

A NUMA (Non-Uniform Memory Access) system combines properties from shared and distributed memory systems. At the hardware level a NUMA system resembles a distributed memory: it is build out of nodes that have their own memory. However, there is additional hardware that enables to directly access (read from and write to) memory of other nodes. In principle there is a global address space that spans the memory of all

nodes. Such systems can be programmed like shared memory systems or like distributed memory systems, or a combination of both.

Meanwhile, big NUMA machines are not being built any more. However, NUMA became a feature of compute nodes that have more than one CPU socket. Even a single socket that has many cores can have NUMA properties. NUMA adds a complication to running parallel applications at the node level: as mentioned above is important to exploit data locality.

K1.3-B I/O Architectures

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about I/O architectures used in HPC environments: local, distributed, parallel and hierarchical file systems (basic level)

Level: basic

I/O Architectures

Before starting to use an HPC system it is important to basically understand file system architectures used in clusters:

Local file systems can be provided on nodes that are equipped with disks. Only the node itself can access its disks. Local file systems are useful for (heavy) scratch I/O. The advantage of local disks is that I/O performance scales perfectly with the number of nodes. The disadvantage is that for an overall view data has to be collected from all the nodes it is stored on. Because disks are a major source of failures many clusters have diskless nodes.

Distributed (or network) file systems are provided by a file server which is integrated into the cluster. These file systems are available on all nodes. A typical example is the /home file system. While all nodes can read concurrently from a distributed file system without interference, care must be taken in concurrent writing. In general, a file should only be written or modified by a single process at a time. A classic distributed file system is *the* Network File System (NFS). Network file systems are not designed for (very) high I/O loads.

Parallel (or cluster) file systems are global file systems like distributed file systems, i.e. they can be used on any node in the cluster. They are designed to deliver high I/O bandwidth and provide large disk space. The parallel or cluster aspect is twofold. Firstly, the hardware is parallel itself (the file system is provided by several servers that operate in a coordinated way). Secondly, parallel I/O is enabled, i.e. more than one process can consistently write to the same file at the same time. The names (mountpoints) of theses file systems vary. There is no standard name for them like /home, although the /home file system can be put on a parallel files system.

Parallel file systems fit well to the classic HPC scenario, i.e. large computer simulations that perform I/O only every one in a while (a few minutes per hour of computation, say). Limitations of I/O performance become noticeable if too many processes are doing I/O and also if very many small files are used (using many small files is a traditional approach under Unix, but this concept becomes a bottleneck if too many processes employ it on a parallel file system). Examples of parallel file systems are Lustre, IBM Spectrum Scale and BeeGFS.

A file system with hierarchical storage management (HSM) has at least two tiers of different storage media. Typically, there are two tiers: a faster, smaller, more expansive

one and a slower, larger, less expensive one. Data is moved automatically between the two tiers. While this is elegant, care should be taken when such a system is heavily used, i.e. it might be necessary to stage data to the fast part or unstage to the slow part manually. At present, probably all HSM systems have spinning disk. Disks can be the slow part if the fast part consists of SSDs, or the fast part if the slow part consists of tapes. With tapes in the background it is important not to generate too many files because it can take a long time to bring them back from tape to disk.

All the file systems mentioned above are *mounted* to (some or all) nodes of the cluster. They can be used via the file handling commands of the operating system. For completeness external storage is mentioned, which needs special (remote copy) commands. An example are **object stores**.

K2-B Performance Modeling

Relevant for: Tester, Builder, and Developer
Short Background:

- HPC systems are parallel computers. Programs running in parallel are required to exploit their performance potential. Hence, parallel performance needs to be understood.

Description:

- You will learn about how the performance of parallel programs may be assessed (basic level)

K2.1-B Performance Frontiers

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about FLOPS (floating point operations per second) which is the key measurement unit for the performance of HPC systems, and its pitfalls (basic level)
- You will learn about Moore's law and its significance for performance frontiers in modern HPC (basic level) (background topic)
- You will learn about the definitions for key terms: speedup, efficiency, and scalability (basic level)
- You will learn about Amdahl's law and its significance for performance frontiers in modern HPC (basic level)

Level: basic

Floating Point Operations per Second (FLOPS)

A key measure for the performance of HPC systems are so-called floating point operations per second (FLOPS or Flop/s), which are in the order of several PetaFLOPS for the top HPC systems of 2017. One PetaFLOPS (PFLOPS) corresponds to 10^{15} FLOPS or 10^3 TeraFlops (TFLOPS), respectively. In comparison, a stand-alone computer, like a powerful PC, achieves a peak performance of about 1 TeraFLOPS.

This measurement is generally used in two distinct ways:

1. To measure the computing power of a computer.

In the context of HPC systems, such measurements are collected in the *TOP 500 list*[3]. This list, which is updated twice per year, lists 500 supercomputers ranked by the FLOPS they achieve in the Linpack benchmark[4]. Because this list also contains a number of additional details of the different machines, like CPU architectures, power consumption, node count, OS, etc., it is a valuable resource to spot trends in the field of high performance computing.

2. To measure the performance of applications.

   Modern CPUs provide the feature of counting every floating point operation that they execute. This can be used to obtain the FLOPS that an application actually uses, and to compare it to its peak performance (the FLOPS that the machine is theoretically able to deliver). The idea is to use this FLOP utilization to assess how well-optimized the application code is.

Pitfalls

While the measurement FLOPS is frequently used to gauge the computing power of a system and of applications, it is also a disputed measurement. Its connection to actual application performance is too loose. For instance, many of the top systems on the TOP 500 list[5] use GPUs to achieve their immense FLOPS, yet many scientific codes perform better on pure CPU based systems.

So, this is a list of the most important pitfalls of using FLOPS to assess computing power:

- Many scientific applications are *memory bound*, not *CPU bound*.

  The arithmetic-logical units that perform the computations are but one of the resources within a CPU. Other resources are important as well, such as caches, instruction decoder slots, and memory buses. If there is not enough computation to be done for each value that is fetched from memory / stored to memory, the memory buses will saturate and become the bottleneck of the application.

  Typical offenders of this include computing the dot product of two long vectors: This will repeatedly load two values (2x8 = 16 bytes) from memory, and then execute a single fused-multiply-and-add operation *(fmadd)*. With vectorization, four of these operations can be performed in a single instruction, but require fetching 4x16 = 64 bytes to work on. A DDR4-3200 memory module can deliver 25.6 GByte/s, which is just enough input data for 400 million arithmetic instructions per second. So, the vectorized *fmadd* instruction is executed with no more than 400 MHz when the CPU hits the memory wall, independent of the CPU clock speed which is usually around 3000 MHz these days.

  This is not a special case. Most codes will do relatively few things with the data before they must store their results and move on to the next data points. The case that the handling of each data point involves lengthy computations is rare. This is the reason why many applications running on supercomputers only use single digit percentages of the peak FLOPS the machine can deliver. The Linpack benchmark,

[3] https://www.top500.org/lists/
[4] https://en.wikipedia.org/wiki/LINPACK_benchmarks
[5] https://www.top500.org/lists/

which is used to measure FLOPS for the TOP 500 list[6], is the exception in its ability to reach FLOP utilizations around 75%. And even that is only possible for pure CPU based systems, on GPU accelerated systems even the Linpack has a utilization of around 65%.

- FLOPS say nothing about the performance of the network or the file system.

  Modern supercomputers are not just the CPUs/GPUs, they also include a high-performance network that connects the individual nodes to a system, and they generally provide a large, parallel file system. And scientific applications make heavy use of both. Of course, the Linpack benchmark also makes some use of the network, but for the TOP 500 list[7] testing, its parameters are tweaked to ensure that the network use does not reduce the performance more than absolutely necessary. And disk I/O is simply not tested by Linpack.

  People are starting to recognize that this is an issue, starting research into more I/O centric evaluations like the IO-500 list[8].

- FLOP utilization says nothing about the quality of code.

  It is easy to produce wasteful code that scores high in FLOP utilization. All that is required is some unnecessary floating point operations that are added to the inner loop. When such inefficient code is optimized, the FLOP utilization goes down. Likewise, it is easy to produce wasteful code that scores low in FLOP utilization. When such inefficient code is optimized, the FLOP utilization goes up. So, when a code change makes FLOP utilization go up, is not immediately clear whether this is good or bad (leads to shorter execution time or not).

  Looking from the perspective of well optimized code, FLOPS seem just as irrelevant: There are highly optimized codes that do score high in FLOP utilization (FFTs for radio-astronomy, for example), while other highly optimized codes do not even use floating point numbers (like applications in bio-chemistry).

  What FLOP utilization is adequate for an application is fully dependent on the problem that the application tries to solve. An application with 5% usage can be great or mediocre, and another application with 50% usage can be great or mediocre. The FLOPS provide no clue which one is well optimized, and which is not.

Moore's Law

In simple terms, Moore's law[9] from 1965, revised in 1975, states that the complexity of integrated circuits[10] and thus the computing power of CPUs for HPC systems, respectively, doubles approximately every two years. In the past that was true. However, for some time it has been observed that this increase in performance gain is no longer achieved through improvements of processor technology in a sequential sense – CPU clock rates, for instance, have not been increased notably for several years –, but rather by using

[6] https://www.top500.org/lists/
[7] https://www.top500.org/lists/
[8] https://www.vi4io.org/io500/start
[9] https://en.wikipedia.org/wiki/Moore%27s_law
[10] https://en.wikipedia.org/wiki/Integrated_circuit

many cores for processing a task in parallel. Therefore parallel computing and HPC systems will become increasingly relevant in the future.

Speedup, Efficiency, and Scalability

Simply put, *speedup* defines the relation between the sequential and parallel runtime of a program. Given a sequential runtime T_1 on a single processor and a parallel runtime T_n for n processors the speedup is given by

$$S_n = \frac{T_1}{T_n}.$$

In the ideal case, the speedup[11] increases linearly with the number of processors, i.e. $S_n = n$

In practice, however, a linear speedup is not achievable due to overheads for synchronization (e.g. for waiting for partial results) and communication (e.g. for distributing partial tasks and collecting partial results).

Efficiency is defined accordingly by

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

If the efficiency[12] remains high when the number of processors is increased, this is also called a good scalability[13] of the parallel program.

For a problem that can be parallelized trivially, e.g. rendering (independent) computer animation images[14], a nearly linear speedup will be achieved also for a larger number of processors. However, there are algorithms having a so-called sequential nature, e.g. alpha-beta game-tree search[15], that have been notoriously difficult to parallelize. Typical problems in the field of scientific computing[16] are somewhere in-between these extremes. In general, the special challenge is to achieve good speedups and good efficiencies. Another important aspect is to use the best known sequential algorithm for comparisons in order to get fair speedup results.

Table 1: Relation of speedup and efficiency

Speedup	Efficiency	Remark
$0 < S_n < n$	$0 < E_n < 1$	Typical Speedup/ Normal Situation
$S_n = n$	$E_n = 1$	Linear/ideal Speedup

[11] https://en.wikipedia.org/wiki/Speedup
[12] https://en.wikipedia.org/wiki/Speedup
[13] https://en.wikipedia.org/wiki/Scalability
[14] https://en.wikipedia.org/wiki/Render_farm
[15] https://www.chessprogramming.org/Parallel_Search#ParallelAlphaBeta
[16] https://en.wikipedia.org/wiki/Computational_science

Speedup	Efficiency	Remark
$S_n > n$	$E_n > 1$	Superlinear Speedup[17] (anomaly originating e.g. from cache usage effects, memory access patterns, not using the best known sequential algorithm for comparison)

Amdahl's Law

In simple terms, Amdahl's law[18] from 1967 states that there is an upper limit for the maximum speedup achievable with a parallel program, which is determined by its sequential, i.e. non-parallelizable part (e.g. for initialization and I/O operations), or more generally, for synchronization (e.g. due to unbalanced load) and communication overheads (e.g. for data exchange).

Here is a simple example to illustrate this:

A program has a sequential runtime of $20h$ on a single core and contains a non-parallelizable part of 10% or $2h$, respectively. Even if the non-sequential part of $18h$ can be parallelized extremely well, the total runtime of the program will be at least $2h$. I.e. the maximum speedup is limited by $\frac{20h}{2h} = 10$, regardless how many cores are available for the parallelization. If 32 cores were used and the parallelizable runtime part reduced to ideally $\frac{18h}{32} = 0,56h$, the total runtime would be at least $2,56h$, i.e. the speedup would be

$$S_{32} = \frac{20h}{2,56h} = 7,81$$

and the efficiency would only be

$$E_{32} = \frac{S_{32}}{32} = \frac{7,81}{32} = 24,41\%.$$

General Formulation

The parallelizable part of a program can be presented as some fraction α.

The non-parallelizable, i.e. sequential, part of the program is thus $(1 - \alpha)$.

Taking T_1 as total runtime of the program on a single core, regardless how many cores n are available, the sequential runtime part will be $(1 - \alpha)T_1$, while the runtime of the parallelizable part of the program will decrease corresponding to the (ideal) speedup $\frac{\alpha T_1}{n}$.

The speedup (neglecting overheads) is therefore expressed as

[17] https://annals-csis.org/Volume_8/pliks/498.pdf
[18] https://en.wikipedia.org/wiki/Amdahl%27s_law

$$S_n \leq \frac{T_1}{(1-\alpha)T_1 + \frac{\alpha T_1}{n}} = \frac{1}{(1-\alpha) + \frac{\alpha}{n}}$$

and the limit for the speedup is given by

$$S_\infty := S_{n\to\infty} = \frac{1}{(1-\alpha)}$$

Table 2: Example: Speedups for a given fraction α of parallelizable work

α	$n=4$	$n=8$	$n=32$	$n=256$	$n=1024$	$n=\infty$
0.9	3.08	4.7	7.8	9.7	9.9	10
0.99	3.88	7.5	24	71	91	100
0.999	3.99	7.9	31	204	506	1000

The efficiency is expressed as

$$E_n = \frac{\frac{1}{(1-\alpha)+\frac{\alpha}{n}}}{n}$$

and the maximal number of cores n_E for achieving a given efficiency E_x is given by

$$n_E = \left\lfloor \frac{\frac{1}{E_x} - \alpha}{1-\alpha} \right\rfloor .$$

Table 3: Example: Maximal number of cores for achieving a given efficiency E_x [%]

α	E_{90}	E_{75}	E_{50}
0.9	2	4	11
0.99	12	34	101
0.999	112	334	1001

Taken the third column in the Table above and the entry for $\alpha = 0.9$ as an example, it is shown that the efficiency will drop below 50% if 12 or more cores are used.

An obvious major goal to increase the potential speedup of a parallel program is to reduce its sequential part as far as possible.

The Figure below shows for a fraction $\alpha = 0.999$ the speedups achievable in relation to the number of processes (or cores) used.

For the curve named “realistic” it is additionally considered that overheads for communication and synchronization will also increase when the number of processes is increased. It is immediately apparent that speedups above 100 are hardly achieved in practice, even if the parallelizable part of a program is 99.9%.

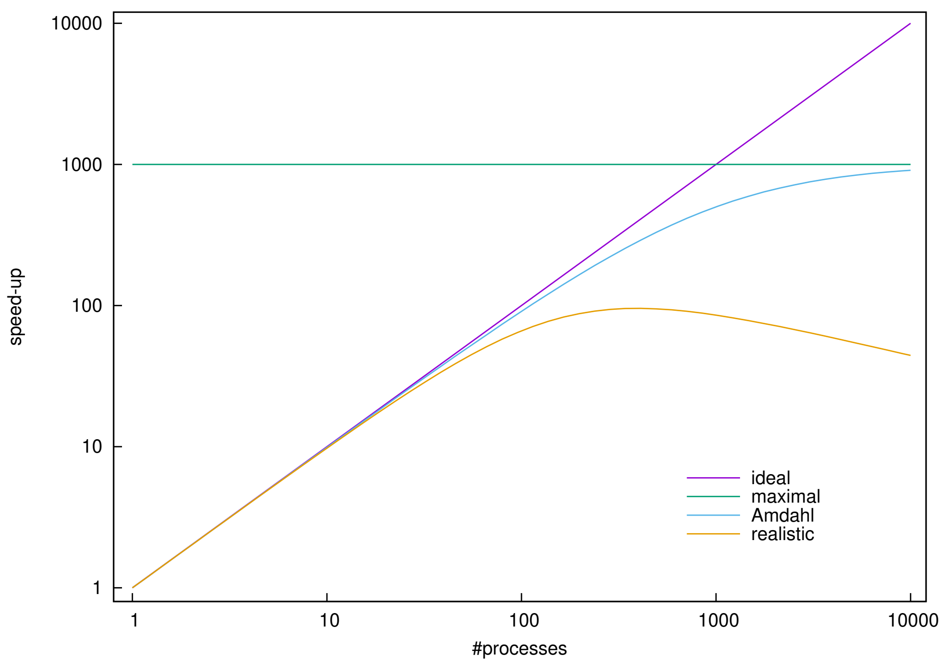

Figure 2: Example: Speedups achievable for parallelizable fraction $\alpha = 0.999$

K3.3-B Parallelization Overheads

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about overheads for communication and synchronization that are introduced by parallelization (basic level)
- You will learn about other sources of parallel inefficiency: load imbalances, hardware effects (basic level)

Level: basic

Parallelization overhead and other sources of parallel inefficiency

Parallelization of a program always introduces some extra work in addition to the work done by the sequential version of the program. The main sources of parallelization overhead are data communication (between processes) and synchronization (of processes and threads). Other sources are additional operations that are introduced at the algorithmic level (for example in global reduction operations) or at a lower software level (for example by address calculations).

Data communication is necessary in programs that are parallelized for distributed memory computers (if data communication is not necessary the program is called *trivially* or *embarrassingly parallel*). The communication effort depends on the communication pattern. Examples for communication patterns are the exchange of data in halo regions (this is typical for simulation programs that are based on discretized partial differential equations and are parallelized by a domain decomposition) and global reduction operation (for example summing up numbers from all processes, or obtaining a minimal or maximal value). In these examples **additional operations at the algorithmic level** (in global reduction operations) **and at the software level** (extra address calculations for accessing data in halo regions) take only little time in comparison to the communication (because the latter involves the network).

For programs running with shared memory parallelization **synchronization** plays an important role. Synchronization means that threads have to wait for the completion of other threads because they need data that is processed by those threads. Overhead is also caused by assigning work to threads (e.g. for executing loops in chunks), and by reduction operations.

Other sources of parallel inefficiency are parts of a program that were not parallelized and still run serially (**serial parts**), and **unbalanced load**. In both cases some hardware is not used for some time, while the goal is to use all hardware all the time.

There are two hardware effects that can reduce the efficiency of the execution of shared memory parallel programs: NUMA and false sharing. **NUMA** can lead to a noticeable performance degradation if data locality is bad (i.e. if too much data that a thread needs is not in its NUMA domain). **False sharing** occurs if threads process data that is in the

same data cache line. Effectively, this can lead to serial execution or even take longer than explicitly serial execution of the affected piece of code.

K3.4-B Domain Decomposition

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about the typical strategy for decomposing work into parallel tasks that is used for parallelizing simulation programs in science end engineering (basic level)
- You will learn about the surface to volume ratio which is important to understand the performance impact of a decomposition (basic level)

Level: basic

Domain Decomposition

Domain decomposition is a technique for parallelizing programs that perform simulations in engineering or natural sciences. Such a technique is needed on distributed memory systems. On distributed memory systems the computational work and, in general, the data have to be decomposed to enable parallel computations that employ several compute processes (which implies the possibility to run on multiple compute nodes).

Many computer simulations are defined on meshes that arise from discretizations of partial differential equations. Another class of simulations are particle simulations (e.g. of many molecules or stars that interact), where the particles can move in a box. The box or the mesh define a geometric region. In a domain decomposition this region is decomposed: a box is split into smaller ones and a mesh is decomposed into smaller parts. The sub-domains are assigned to processes. Each process updates only the data that belongs to its sub-domain.

In order to update a variable that is defined on a site of a mesh, or for a particle, in general data from neighbouring sites or particles is needed. Typically, the neighbour region is small. It is given by the extent of a stencil or the finite range of interactions of molecules (exceptions are gravitational and electrical forces which have infinite range).

The essential point is, that some neighbour regions expand beyond the sub-domain of their process, i.e. neighbour regions are partly stored on remote processes. Those parts must be made available on the local process before updating can begin. Data from remote processes are stored locally in so called *halo regions*. Halo regions store data from remote processes that is needed to perform local operations. The corresponding data communication operation is called *halo exchange* or *exchange of boundary values* (“exchange” because data movement is typically necessary in either direction).

The fact that halo exchange is needed in parallel computer simulations has a performance impact, that everybody, who is running such simulations, should know about. Halo exchange is one kind of parallel overhead. The amount of overhead is proportional to size and shape of the halo region which is approximately the size of the surface of a sub-domain. The size of the surface of a sub-domain has to be related to the volume of the subdomain (which is approximately proportional to the amount of work that needs

to be performed). The relative overhead is approximately proportional to the surface to volume ratio of a sub-domain.

To understand this relative overhead quantitatively it is instructive to consider sub-domains that are d-dimensional cubes that have linear extension L. The size of their surface is $2dL^{d-1}$ and size of their volume is L^d. The surface to volume ratio, i.e. the relative overhead,

$$\frac{2dL^{d-1}}{L^d} = \frac{2d}{L}$$

is inversely proportional to L. In other words this kind of overhead increases at the same rate as sub-domains shrink. This effect limits (strong) scaling: if more processes are used, sub-domains become smaller and the overhead grows. At some point it becomes unacceptably large.

Besides the size of the volume its shape also plays a role. In order to see this one can look at rectangular sub-domains. Let the dimensions of the rectangular be Lx and L/x. Then the size of the volume of the rectangular is L^2 and the size of its surface is $2Lx + 2L/x = 2L(x + 1/x)$. The overhead is proportional to

$$\frac{2L(x + 1/x)}{L^2} = \frac{2(x + 1/x)}{L}.$$

For $x = 1$ the proportionality factor $(x + 1/x) = 2$, for $x = L$ it is $(L + 1/L) \approx L$ for large values of L. Hence, the overhead of a rectangular shape can be up to a factor of $L/2$ larger in comparison to a quadratic shape. The ideal shape is a sphere, because it has the smallest surface to volume ratio. Long narrow sub-domains are disadvantageous.

Optimizing surface to volume ratios is a nested process if the communication hardware is hierarchical. Today this is very often the case: compute nodes can have NUMA domains and the communication networks can be hierarchical. Then good surface to volume ratios are desirable at all hardware levels: at process (core) level, at NUMA domain level, at node level, at the level of nodes connected to same switch, etc.

K4-B Job Scheduling

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about how workload managers control the unattended background execution of programs or jobs, respectively, by the help of job queues (basic level)
- You will learn about typical scheduling principles (e.g. first come first served, shortest job first) to achieve objectives like minimizing the averaged elapsed program runtimes, and maximizing the utilization of the available HPC resources (basic level)

Level: basic

Motivation

Users of compute centers typically compete for the expensive HPC resources of cluster systems.

HPC resources can be distinguished as

- *shared* resources (e.g. a parallel file system that is often shared across all cluster nodes and therefore shared between all users),
- *not-shared* resources (e.g. cluster nodes dedicated to a particular parallel program of an individual user).

The configuration of the cluster system matters as well: a cluster node can also be a resource that is shared between several users.

A major aspect of *job scheduling* is to manage these resources in a way that users are treated fairly. Accounting for users or user groups can additionally support this.

Two questions that are important to users are constantly appearing:

- When will my job (or my parallel program, respectively) start?
- Why is my job *still not running* at the estimated starting time?

Basic knowledge about job scheduling which is provided below shall give an insight into batch systems and enable users to answer such questions.

Batch Systems vs. Time Sharing Systems

Time Sharing Systems

A typical computer system like a PC provides the possibility for interactive communication between the user and the system. The user gives instructions via a keyboard and/or a mouse to the operating system or directly to a program, and the response is immediately displayed on the monitor. Such interactive systems are characterized by the users' desire for a short response time.

The user can easily experiment and see immediate results, e.g. during interactive program development with the typical *edit* $\rightarrow$ *build* $\rightarrow$ *test* cycle (*build* comprises the steps *compile* and *link*). In a typical cluster system such activities are performed simultaneously by a number of users on nodes explicitly provided for that purpose, e.g. *login* or *head nodes*. The Linux operating system (nowadays used in almost all cluster systems) provides a time sharing environment, such that all users logged in to a specific node (e.g. in to the same login or head node) have the impression of using a dedicated computer. Nevertheless, it is important to keep in mind that login or head nodes are *shared resources* and all users are advised to use them carefully. It is e.g. perfectly alright to perform a CPU intensive but short build step, but it is e.g. not alright to start a user-specific parallel program on several cores – not even for testing purposes – on a login or head node.

A cluster node is characterized by running its own instance of the operating system on one or more of its CPUs. Because of the nowadays common multi-core CPU architectures the terms core and processor – in its classical meaning of a physical unit capable of executing a program – are used synonymously here.

Simply put, on a single processor machine the *scheduling mechanism of the operating system* ensures that each of n CPU intensive processes has its own processor running at $1/n$ times the speed of the real processor. For a multi-core cluster node (neglecting for the sake of simplicity hyper-threading[19] and varying CPU clock rates with features like turbo boost[20]) the same applies analogously: for a sufficiently large n on a machine with c cores (i.e. $n > c$) the pseudo processor speed available for a process would be c/n of a real core (if $n \leq c$, each process can run at the speed of a real core). For an in-depth introduction to operating system topics like process management, memory management, and storage management refer e.g. to *Operating Systems Concepts* (ninth edition) by Silberschatz, Galvin, and Gagne. Scheduling implementations for *time sharing systems* shall not be discussed in more detail here. The topic of this learning unit is scheduling in *batch systems*.

Batch Systems

A *batch job* mainly consists of a program or a set of programs being processed by the computer system. One major feature of a batch system is the lack of interaction between the user and the job. The programs to run are typically given by a sequence of operating system commands in a batch file. The user submits the batch file to the batch system for execution. In a cluster system there are (hopefully) more batch jobs submitted than can be executed immediately. Submitted batch jobs are queued for later execution by the *batch system*.

Basically, running jobs on a cluster only begins to be meaningful when a problem cannot be solved using a desktop PC within a few days. Batch systems are used for executing large parallel jobs which need no interaction.

[19] https://en.wikipedia.org/wiki/Hyper-threading
[20] https://en.wikipedia.org/wiki/Intel_Turbo_Boost

Job Scheduling

Scheduling is the process of selecting and allocating resources to a job waiting for execution. Queues are used to handle this. As a consequence, waiting times will typically arise before a job is executed.

Linux itself is not a batch system. Workload managers like SLURM[21] or TORQUE[22] are used to provide this functionality in a Linux cluster system.

Scheduling is fundamental, since almost all computer resources are scheduled before use. Job scheduling deals with the problem of deciding which jobs are allocated in which order to cluster nodes. Normally, job scheduling is heavily influenced by resource allocation considerations as specified at job submission time e.g. by the number of nodes (or processors, respectively) required for a job, estimated job runtime limit, etc.

There are many different scheduling algorithms that can be used to achieve different objectives. A typical goal is to minimize the average waiting time for a given set of jobs.

Before scheduling queues are presented in more detail, a set of terms in the sense of performance criteria shall be introduced:

- *Resource Utilization:* For expensive resources like CPUs utilization should be high.
- *Throughput:* A measure of work being done: the number of jobs completed per time unit. For this consideration the jobs can be additionally grouped into different classes, in particular with regard to their runtimes: for long jobs this rate may be 20 jobs per day, for short jobs throughput might be 40 jobs per hour.
- *Waiting Time:* Delay after job submission before the job starts executing.
- *Execution Time:* The time between job execution start and job completion.
- *Turnaround Time:* Waiting time plus execution time. In other words it is the delay between job submission and job completion. Since scheduling does not really affect the execution time of a job, attention is frequently focused on waiting time rather than turnaround time.

Goals of job scheduling are

- maximization of resource utilization,
- maximization of throughput,
- minimization of waiting time,
- minimization of turnaround time.

Scheduling Algorithms

In order to achieve these goals the following scheduling algorithms may be employed.

First-Come-First-Served (FCFS)

[21] https://www.schedmd.com/
[22] https://en.wikipedia.org/wiki/TORQUE

This very simple algorithm is implemented with a First-In-First-Out (FIFO) queue and as its name indicates, jobs are executed in the order of submission. The performance of FCFS is quite poor and it is not directly suitable for job scheduling. The following example, by analogy, can illustrate this: In a supermarket a customer with a well-filled shopping cart is the first to arrive at the checkout. The next three customers arriving have just one banana in their shopping cart. Obviously, the *average* waiting time with regard to the four customers will be quite high.

However, FCFS is the basis for more sophisticated algorithms.

Shortest-Job-First (SJF)

This algorithm associates an estimated runtime with each job and selects the job with the smallest runtime available (i.e. the shortest job) in the job queue for next execution. To stay with the supermarket example, the waiting time of the customers with just a banana in their shopping cart are greatly reduced if the customer with the very well-filled shopping chart gives them precedence.

SJF is trivially provably optimal with respect to minimizing the average waiting time for a given set of jobs: If a short job is moved before a long job, its waiting time is decreased to a greater extend than the waiting time for the long job is increased.

For the scheduler there is no way of estimating the runtime of a job itself. To reduce this problem, in batch environments the user typically estimates the maximum runtime of his job and specifies this value as a *job time limit* when the job is submitted. A job reaching the time limit and still being executed is aborted by the batch system. The accurate setting of the job time limit is in the user's own interest and already being exploited in early batch systems to increase the throughput.

A problem – named *starvation* – may occur if short jobs, which are constantly being submitted, are giving precedence all the time, so the execution of long jobs will not start. Starvation describes the situation of blocking a job that is ready to be run but waiting indefinitely.

In accordance to SJF it was common practice in early batch systems to use several job queue classes to get high throughput. There might for example be FIFO queues for short jobs, medium jobs, and long jobs, with maximum runtimes of 5 minutes, 30 minutes, or any desired time span. To draw the analogy with the supermarket example once more, the queue for short jobs could be represented by an express checkout with no more than 3 items allowed per customer.

Priority

A Priority is associated with each job and the resources are allocated to the job with the highest priority. SJF can be regarded as a special case of priority scheduling.

In practice, priorities are defined by value ranges whereby internal and external priorities can be distinguished.

- Internal priorities arise e.g. from
  - job size, based e.g. on
    - number of nodes on which to run the program

    * time limits
    * memory limits
  - aging, for which the priority of each job waiting in the system is steadily increased as time goes on
  - other required resources like licenses
- External priorities, as external criteria to the operating system, arise e.g. from
  - (scientific) institution sponsoring the (scientific) work
  - amount of funds being paid for computer use
  - type of funds being paid for computer use
  - other, mostly political, factors

The priority of a job affects its position in the job queue. The higher the priority of a newly submitted job, the nearer will it be inserted to the top of the queue. Further job submissions in combination with possibly changing job priorities over time will lead to a very dynamic scheduling system. Information requested by the user about the status of the system, for example starting time of a job, must therefore be considered as a rough estimate.

Fair-Share

Priorities, as described above, offer very versatile options for ordering waiting jobs in a FIFO-based job queue.

Besides criteria like job size and aging, a technique named *Fair-Share* can be used to take into account the history of previously submitted jobs by a user or, respectively, users belonging to the same group: For calculating the priority for an actually submitted job it will be considered how many of the computing resources (e.g. computing time), which were initially granted to the user or its user group, have already been consumed. The fewer resources have been consumed in the past the higher the job priority is set (and vice-versa). Additionally, the stored history of jobs could be considered in order to decrease the impact of the really distant past. Jobs that are older than some threshold could e.g. be regarded as “forgotten”. The overall goal is to treat all users fairly (i.e. to obtain certain shares of the resources).

Backfilling

What could be observed by a user in a batch system is that there seems to be a sufficient number of nodes available for its own waiting job, but it still does not start. The reason might be that a higher prioritized bigger job at the top of the queue, with demands for a number of nodes not actually available, is also waiting and the scheduler has reserved already some idle nodes for the bigger job.

In order to mitigate this problem a technique named *backfilling* can help to increase resource utilization and to decrease average waiting times by allowing a small job with a low priority to run before of a bigger job with a higher priority.

Simply put, in a situation where the scheduler has already reserved some idle nodes for the next big job at the top of the queue, it calculates a backfilling time window based on the job time limits of still running jobs to estimate the starting time of the big job. If, for example, the big job will not start within the next 15 minutes, small jobs with a job time

limit of at most 15 minutes and for which the number of reserved idle nodes is sufficient can be started immediately in the backfill window without influencing the planned start time of the big job.

USE1-B Use of the Cluster Operating System

Relevant for: Tester, Builder, and Developer
Short Background:

- HPC systems are usually accessed via a command line interface (CLI). The user acquires skills to use a – generally Linux based – CLI to interact with the HPC system.

Description:

- You will learn to use and write shell scripts e.g. to automatically execute several commands in a row that otherwise would have to be entered manually one by one and to automate simple tasks (basic level)
- You will learn to select the right environment setting to build programs with the proper compiler, linker, and libraries versions or to run programs (basic level)

USE1.1-B Use of the Command Line Interface

Relevant for: Tester, Builder, and Developer
Description:

- You will learn about the book "The Linux Command Line" by William Shotts which we recommend for learning the command line (basic level)
- You will learn to login remotely with public key authentication (which is not covered in the book) (basic level)
- You will learn to use text editors (first steps without the need to consult the book) (basic level)

Level: basic

Use of the command line interface

Users interact with an HPC system through a *command line interface (CLI)*, there are no graphical user interfaces. The command line is used for interactive work and for writing shell scripts that run as batch jobs.

Using a command line interface is a good way to interact very efficiently with a computer by just typing on a keyboard. Fortunately this is especially true for Unix-like operating systems like Linux, because for the typically Linux based cluster systems graphical user interfaces (GUIs) are rarely available. In the Unix or Linux world a program named *shell* is used as a command language interpreter. The shell executes commands it reads from the keyboard or – more strictly speaking – from the standard input device (e.g. mapped to a file).

The shell can also run programs named shell scripts, e.g. to automate the execution of several commands in a row.

The command line is much more widely used than in HPC. Accordingly, much more has been written about the command line than on HPC. Instead of trying to write yet another text on the command line we would like to refer the reader to the very nice book *The Linux Command Line*[23] by William Shotts. The book can be read online[24]. A pdf version (here: 19.01)[25] is also available.

As a motivation to start reading it, we quote a few lines from the Introduction of version 19.01 of the book:

- *It's been said that "graphical user interfaces make easy tasks easy, while command line interfaces make difficult tasks possible" and this is still very true today.*
- *. . . , there is no shortcut to Linux enlightenment. Learning the command line is challenging and takes real effort. It's not that it's so hard, but rather it's so vast.*
- *And, unlike many other computer skills, knowledge of the command line is long lasting.*
- *Another goal is to acquaint you with the Unix way of thinking, which is different from the Windows way of thinking.*

The book also introduces the *secure shell* (`ssh`). The secure shell is a prerequisite for using HPC systems, because it is needed to log in, and to copy files from and to the system. Since *The Linux Command Line* does not cover ssh keys, we explain how to use these below. We also mention the very first steps with *text editors*.

Remote login

Login nodes

To get access to a cluster system, a terminal emulator program is used to remotely connect to a cluster node (possibly belonging to a group of such nodes) which authenticates the user and grants access to the actual cluster system. Such login nodes are also named gateway nodes or head nodes.

SSH (Secure Shell) and Windows Equivalents

In the Linux world the `ssh` (Secure Shell) client program is used as a terminal emulator for logging into a remote machine like a login node (running the SSH server counterpart).

The `ssh` program provides a secured and encrypted communication for executing commands on a remote machine like building programs and submitting jobs on the cluster

[23] http://linuxcommand.org/tlcl.php
[24] http://linuxcommand.org/tlcl.php
[25] http://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.pdf/download

system. Windows users can use third-party software like `putty`[26] or `MobaXterm`[27] to establish ssh connections to a cluster system. Meanwhile a quite mature OpenSSH port (beta version)[28], i.e. a collection of client/server ssh utilities[29], is also available for Windows.

Another very useful option is to run Linux (e.g. Ubuntu[30] or openSUSE[31]) on a Windows computer in a virtual machine (VM) using a hypervisor like VMware Workstation Player[32], VirtualBox[33], or Hyper-V[34].

Login example for the cluster at Universität Hamburg

As an example, it is shown how to login to the *Hummel* cluster at Universität Hamburg (there are two login gateways available: `hummel1` and `hummel2`):

```
user@your-pc:~$ ssh yourHummelUsername@hummel1.rrz.uni-hamburg.de
Enter passphrase for key '/home/user/.ssh/id_rsa': ****************
       _                                                                 _
    __/ \   __/ RRZ HPC Login | Zugang nur mit Berechtigung      \__   / \_
         \_/  \ RRZ HPC login | Access for authorized users only /  \_/


  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                             *
 *  Hummel HPC Cluster (2015) am RRZ der Universität Hamburg     ____Ô____     *
 *                                                             /|\  |  /|\     *
 * > Interaktive Knoten / Interactive nodes:                  H U M | M E L    *
 *     front1, front2                                             _/ \_        *
 * > Weitere Informationen / Further information:                              *
 *     http://www.rrz.uni-hamburg.de/de/services/hpc.html                      *
 * > Beratung und Hilfe / Support: mailto:hpc@uni-hamburg.de                   *
 *                                                                             *
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

[yourHummelUsername@login1 14:48:23]~$
```

Public-key cryptography

For the *Hummel* cluster public-key cryptography is used for authentication to increase security in comparison to simple username / password authentication. To access the cluster via `ssh`, as shown above, the user initially generates a private/public keypair. The public key file will then be installed on the cluster by the cluster administrator. The

[26] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
[27] https://mobaxterm.mobatek.net/
[28] https://github.com/PowerShell/Win32-OpenSSH/releases
[29] https://www.openssh.com/
[30] https://www.ubuntu.com/download/desktop
[31] https://software.opensuse.org/
[32] https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html
[33] https://www.virtualbox.org/wiki/Downloads
[34] https://blogs.technet.microsoft.com/schadinio/2010/07/09/installing-hyper-v-manager-on-windows-7/

private key file is kept in home directory on the user's computer (e.g. ~/.ssh/id_rsa). The private key will never leave the user's computer, which is a major concept and advantage of public key cryptography. For more details also see *challenge-response authentication* in Wikipedia[35]. On the user's computer, access to the private key needs to be protected by appropriate file permissions (only the owner of the private key file shall have permission to read the file) and above all by a *passphrase*. Note: Everyone who has access to the private key can gain access to the machine on which the public key is installed. Good practice is to use different key pairs for different HPC cluster systems.

SSH key pairs

The `ssh-keygen` command is used to generate, manage and convert authentication keys for `ssh`. Below, an example is given on how to generate an RSA type key pair (the newer type ed25519 is also a good choice) with a length of 4096 bits (minimum length should be 2048 bits). Windows users can use the `PuTTYgen` helper program in connection with third-party software like `putty`[36] or `MobaXterm`[37] to generate key pairs. `ssh-keygen` and `PuTTYgen` use different private key file formats. `PuTTYgen` can be used to convert between these formats via its import and export functionality (e.g. to reuse a private key when a user migrates the local PC from Windows to Linux or vice versa).

```
user@your-pc:~$ ssh-keygen -t rsa -b 4096

Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase): ****************
Enter same passphrase again: ****************
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
b8:df:d1:14:48:03:00:68:5e:46:9c:1a:b2:b2:d4:f4 user@your-pc
The key's randomart image is:
+--[ RSA 4096]----+
|   +oo....o      |
|. +.=     . o    |
| =o=.      . .   |
|o.o. E .      .  |
|o.    . S    .   |
|.       .    o   |
|       .   . .   |
|        . . .    |
|         . .     |
+-----------------+
```

SSH-agent and Windows equivalents

[35] https://en.wikipedia.org/wiki/Challenge%E2%80%93response_authentication
[36] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
[37] https://mobaxterm.mobatek.net/

A further advantage of public-key cryptography in connection with the `ssh` command is the possibility to use the `ssh-agent` helper program, that keeps private keys after they have been unlocked with their passphrase. After the user grants access via the `ssh-agent` and the `ssh-add` command to a private key once, the agent can then use the key to log into other servers without having the user to type in the passphrase again:

```
$ ssh-add $HOME/.ssh/id_rsa
Enter passphrase for key '/home/user/.ssh/id_rsa': ****************
Identity added: /home/user/.ssh/id_rsa (/home/user/.ssh/id_rsa)

$ ssh yourHummelUsername@hummel1.rrz.uni-hamburg.de
```

This is similar to the idea of Single Sign-On (SSO[38]). Windows users can use corresponding helper programs for third-party software like `putty`[39] or `MobaXterm`[40] (e.g. the `Pageant` tool, which provides the same functionality).

Once the user is logged in to one of the gateway nodes (`hummel1` or `hummel2`) `$HOME` and `$WORK` directories can be accessed, e.g. to transfer data using the `scp` (secure copy) command.

Multi-hop login

A special security feature of the *Hummel* cluster is that an additional login from the gateway node to one of two front end nodes (`front1` or `front2`) is required to gain full access – e.g. to build programs and submit jobs – to the cluster environment. This is called a *multi-hop* login.

The corresponding `ssh` command is shown below.

```
[yourHummelUsername@login1 14:48:23]~$ ssh front1

[yourHummelUsername@node001 14:48:33]~$
```

The front end nodes are two (in principle arbitrary) nodes of the cluster (usually `node001` and `node002`). The usage of alias names (`front1` and `front2`) ensures to connect to a node providing the front end functionality. In contrast to the other compute nodes of the cluster the front end nodes are meant e.g. for interactive program development and job submitting.

Both `ssh` commands can also be grouped together in a single `ssh` command line using ssh connection chaining:

```
user@your-pc:~$ ssh -t yourHummelUsername@hummel1.rrz.uni-hamburg.de ssh front1

   _                                                              _
__/ \   __/ RRZ HPC Login | Zugang nur mit Berechtigung      \__   / \_
     \_/  \ RRZ HPC login | Access for authorized users only /  \_/

```

[38] https://archive.vn/20140315095827/http://www.authenticationworld.com/Single-Sign-On-Authentication/

[39] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

[40] https://mobaxterm.mobatek.net/

```
7
8 [yourHummelUsername@node001 14:48:33]~$
```

The `-t` option is used to force an interactive terminal connection. For more information about the `ssh` usage and its options also see the `man` page for the `ssh` command (i.e. `man ssh`). `man` is an interface to the on-line reference manuals. For more information about the `man` page usage also see `man man`.

Further possibilities (for more advanced users) to connect transparently to a front end node are the use of proxy commands[41] and ssh tunnels[42]. Windows users can configure third-party software like `putty`[43] or `MobaXterm`[44] to achieve the same convenience.

Agent forwarding

Agent forwarding is used to connect to a third HPC system from an HPC system that you logged into with `ssh` from your computer. This is needed, for example, for copying files between two remote systems. The advantage of agent forwarding is that no secret information needs to be stored on a remote system (a private key), or needs to be passed to it (the password for the third machine).

Agent forwarding is switched off by default. It is switched on by `-A` and is used as shown in this example:

- log into `hpc_system1`

  ```
  user@your-pc$ ssh -A username1@hpc_system1.example.com
  ```

- from there, log into `hpc_system2`

  ```
  hpc_system1$ ssh username2@hpc_system2.example.com
  ```

- or copy a file from `hpc_system1` to `hpc_system2`

  ```
  hpc_system1$ scp example.c username2@hpc_system2.example.com:
  ```

Text editors[45]

On an HPC-cluster one typically works in a *terminal mode* (or *text mode* in contrast to a *graphical mode*), i.e. one can use the keyboard but there is neither mouse support nor graphical output. Accordingly, text editors[46] have to be used in text mode as well. For newcomers this is an obstacle. As a workaround, or for editing large portions, one can use the personal computer, where a graphical mode is available, and copy files to the cluster when editing is completed. However, copying files back and forth can be cumbersome if edit cycles are short: for example, if one is testing code on a cluster, where only small changes are necessary, using an editor directly on the cluster is very helpful.

---

[41] https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Proxies_and_Jump_Hosts#ProxyCommand_with_Netcat

[42] https://www.ssh.com/ssh/tunneling/example

[43] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

[44] https://mobaxterm.mobatek.net/

[45] https://en.wikipedia.org/wiki/Text_editor

[46] https://en.wikipedia.org/wiki/Text_editor

Text user interface

Classic Unix/Linux text editors are vi[47]/vim[48] and GNU Emacs[49]. Both are very powerful but their handling is not intuitive. The least things to know are the key strokes to quit them:

- vi: `<esc>:q!` (quit without saving)
- vi: `<esc>ZZ` (save and quit)
- emacs: `<cntl-x><cntl-c>`

GNU nano[50] is a small text editor that is more intuitive, in particular, because the main control keys are displayed with explanations in two bars at the bottom of the screen. The exit key is:

- nano: `<cntl-x>`.

Graphical user interface

In addition to text user interfaces vim[51] and GNU Emacs[52] have graphical interfaces. On an HPC-cluster it is possible to use graphical interfaces if *X11 forwarding* is enabled (see `-X` option of the ssh[53] command). Two other well know text editors, that you might find on your cluster, are gedit[54] and kate[55].

However, X11 forwarding can be annoyingly slow if the internet connection is not good enough.

A trick for advanced users is to mount file systems from a cluster with SSHFS[56]. Then one can transparently use one's favourite text editor on the local computer for editing files on the remote cluster.

USE1.2-B Using Shell Scripts

Relevant for: Tester, Builder, and Developer
Description:

- You will learn to handle scripting tasks that are often needed in batch scripts: manipulating filenames, temporary files, tracing command execution, error handling, trivial parallelization (basic level)

Level: basic

---

[47] https://en.wikipedia.org/wiki/Vi
[48] https://en.wikipedia.org/wiki/Vim_(text_editor)
[49] https://en.wikipedia.org/wiki/GNU_Emacs
[50] https://en.wikipedia.org/wiki/GNU_nano
[51] https://en.wikipedia.org/wiki/Vim_(text_editor)
[52] https://en.wikipedia.org/wiki/GNU_Emacs
[53] https://en.wikipedia.org/wiki/Secure_Shell
[54] https://en.wikipedia.org/wiki/Gedit
[55] https://en.wikipedia.org/wiki/Kate_(text_editor)
[56] https://en.wikipedia.org/wiki/SSHFS

Using shell scripts

Shell scripts are used to store complicated commands or to automate tasks. A simple script consists of one or a few commands that appear exactly like on the interactive command line. Since the shell is also a programming language, scripts can execute more complicated processes. On HPC systems scripting is needed for creating batch jobs. Here, some scripting tasks are explained that are useful for writing batch scripts.

Manipulating filenames

Handling filenames translates to character string processing. The following table shows some typical examples:

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`

Recommendation: Never use white space in filenames! This is error prone, because quoting becomes necessary, like in: `dir=$(dirname "$c")`.

Temporary files

There are three issues with temporary files: choice of the directory to write them, unique names and automatic deletion.

Assume that a batch job shall work in a temporary directory. Possibly, the computing center provides such a directory for every batch job and deletes that directory when the jobs ends. Then one can just use that directory. If this is not the case one can proceed like this.

- Choose a file system (or top directory) to work in. The classic directory for this purpose is `/tmp`. However, this is not a good choice on (almost all) HPC clusters, because `/tmp` is probably too small on diskless nodes. You should find out for your system, which file system is well suited. There can be local file systems (on nodes that are equipped with local disks) or global file systems. Let us call that filesystem `/scratch` and set:

  ```
  top_tmpdir=/scratch
  ```

- A sub-directory with a unique name can be generated with the `mktemp` command. `mktemp` generates a unique name from a template by replacing a sequence of Xs by

a unique value. It prints the unique name such that it can be stored in a variable. For easy identification of your temporary directories you can use your username (which is contained in the variable `$USER`) in the template, and set:

```
my_tmpdir=$(mktemp -d "$top_tmpdir/$USER.XXXXXXXXXXXX")
```

- The next line in our example handles automatic deletion. Wherever the script exits our temporary directory will be deleted:

```
trap "rm -rf $my_tmpdir" EXIT
```

- Now we can work in our temporary directory:

```
cd $my_tmpdir
...
```

Tracing command execution

There are two shell settings for tracing command execution. After `set -v` all commands are printed as they appear literally in the script. After `set -x` commands are printed as they are being executed (i.e. with variables expanded). Both settings are also useful for debugging.

Error handling

There are two shell settings that can help to handle errors.

The first setting is `set -e` which makes the script exit immediately if a command exits with an error (non-zero) status. The second setting is `set -u` which makes the script exit if an undefined variable is accessed (this is also useful for debugging).

Exceptions can be handled in the following ways. If `-e` is set an error status can be ignored by using the *or* operator `||` and calling the `true` command which is a *no-op* command that always succeeds:

```
command_that_could_go_wrong || true
```

If `-u` is set a null value can be used if variable that is unset is accessed (the braces and the dash after `_set` do the job):

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

Trivial parallelization

In this context *trivial parallelization* means to start more than one executable. For example, two graphics cards can be used in the following way:

```
# start 2 CUDA binaries (in the background)
CUDA_VISIBLE_DEVICES=0 cudaBinary1 input1 &
```

```
CUDA_VISIBLE_DEVICES=1 cudaBinary2 input2 &

# wait for completion of both (all) background jobs
# (do not exit right away)
wait
```

A more powerful way for starting many tasks or processing a task queue is *GNU Parallel*[57].

USE1.3-B Selecting the Software Environment

Relevant for: Tester, Builder, and Developer
Short Background:

- In general, multiple versions of software tools and software environments are installed on an HPC system.
- A widely used system for handling different software environments are "Environment Modules". (basic level)

Description:

- You will learn to how to use Environment Modules (basic level)

Level: basic

Environment Modules

Introduction

This text gives a short overview of *Environment Modules*. More information can be found on the homepage of the project[58].

Environment Modules are a tool for managing environment variables of the shell. The `module load` command extends variables containing search paths (e.g. `PATH` or `MANPATH`). The `module unload` command is the corresponding inverse operation, it removes entries from search paths. By extending search paths software is made callable. Effectively software can be provided through Modules. An advantage over defining environment variables directly in the shell is that Modules allow to undo changes of environment variables. The idea of introducing Modules is to be able to define software environments in a modular way. In the context of HPC, Modules make it easy to switch compilers or libraries, or to choose between different versions of an application software package.

Initialization

The `module` command is a *shell function*. As a consequence this function needs to be defined in every instance of the shell including batch job scripts. Typically, this happens

[57]https://www.gnu.org/software/parallel/
[58]http://modules.sourceforge.net/

by sourcing an initialization script. In interactive environments the module function is probably available without further initialization. But in batch jobs it might be necessary to source an initialization script. In that latter case the documentation of your computing center will provide information on how this is performed.

Naming

Names of Modules have the format `program/version` or just `program`. Modules can be loaded (and always be unloaded) without specifying a version. If the `version` is not specified the default `version` will be loaded. The default `version` is either explicitly defined (and will be marked in the output of module avail) or module will load the `version` that appears to be the latest one. Because defaults can change **`versions` should always be given if reproducibility is required**.

Dependences and conflicts

Modules can have dependences, i.e. a Module can enforce that other Modules that it depends on must be loaded before the Module itself can be loaded.

Module can be conflicting, i.e. these modules must not be loaded at the same time (e.g. two version of a compiler). A conflicting Module must be unloaded before the Module it conflicts with can be loaded.

Caveats

The name Modules suggest that Modules can be picked and combined in a modular fashion. For Modules providing application packages this is true (up to possible dependences and conflicts described above), i.e. it is possible to chose any combination of application software.

However, today, environments for building software are not modular anymore. In particular, it is no longer guaranteed that a library that was built with one compiler can be used with code generated by a different compiler. Hence, the corresponding Modules cannot be modular either. A popular way to handle this situation is to append compiler information to the version information of library Modules. Firstly, this leads to long names and secondly, to very many Modules that are hard to overlook. A more modern way is to build up toolchains with Modules. For example, in such a toolchain only compiler Modules are available at the beginning. Once a compiler Module is loaded, MPI libraries (the next level of tools) become available and after that all other Modules (that were built with that chain).

Important commands

Important Module commands are:

list Modules currently loaded	`module list`
list available Modules	`module avail`

load a Module	`module load program[/version]`
unload a Module	`module unload program`
switch a Module (e.g. compiler version)	`module switch program program/version`
add or remove a directory/path to the Module search path (e.g. by an own Module directory)	`module [un]use [--append] path`

Self-documentation

Modules are self-documented:

show the actions of a Module	`module display program/version`
short description of [one or] all Modules	`module whatis [program/version]`
longer help text on a Module	`module help program/version`
help on `module` itself	`module help`

USE2.1-B Use of a Workload Manager

Relevant for: Tester, Builder, and Developer
Description:

- You will learn to use a workload manager like SLURM or TORQUE to allocate HPC resources (e.g. CPUs) and to submit a batch job (basic level)
- You will learn to use shell scripts (basic level) (see also USE1.2-B Using Shell Scripts, p. 36)
- You will learn to use the command line interface (basic level) (see also USE1.1-B Use of the Command Line Interface, p. 30)

Level: basic

Workload managers

Batch jobs submitted to a job queue define the workloads in batch systems. A workload manager of a cluster system typically deals with:

- *Job Control* to provide a user interface for submitting jobs to job queues, monitoring their state during processing (e.g. to check their estimated starting time), and intervening in their execution (e.g. to abort them manually)
- *Scheduling* and *Resource Management* to select a waiting job for execution and to allocate nodes to the job meeting all its other demands for computing resources (memory, special processing elements like GPUs, etc.)
- *Accounting* to record historical data about how many computing resources (e.g. computing time) have been consumed by a job

SLURM[59] (Simple Linux Utility for Resource Management) and TORQUE[60] (Terascale Open-source Resource and QUEue Manager) are widely used open source workload managers for large and small Linux clusters. Both workload managers are controlled via a CLI (Command Line Interface) and offer similar features.

In recent years there appears to be a general trend that scientific institutions substitute TORQUE (or OpenPBS[61] (Open Portable Batch System), from which TORQUE was derived/forked) by SLURM. One reason for this is that SLURM includes all functionality out of the box, while TORQUE typically has to be integrated with additional software like the Maui[62] scheduler or the commercially available Moab Cluster Suite[63], which was based on Maui, to efficiently make use of the cluster resources.

---

[59] https://www.schedmd.com/
[60] https://en.wikipedia.org/wiki/TORQUE
[61] https://en.wikipedia.org/wiki/Portable_Batch_System
[62] https://en.wikipedia.org/wiki/Maui_Cluster_Scheduler
[63] https://en.wikipedia.org/wiki/%20Moab_Cluster_Suite

SLURM

There are three key functions of SLURM described on the SLURM website[64]:

"... First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work. ... "

SLURM's default scheduling is based on a FIFO-queue, which is typically enhanced with the Multifactor Priority Plugin[65] to achieve a very versatile facility for ordering the queue of jobs waiting to be scheduled. In contrast to other workload managers SLURM does not use several job queues. Cluster nodes in a SLURM configuration can be assigned to multiple partitions by the cluster administrators instead. This enables the same functionality.

A compute center will seek to configure SLURM in a way that resource utilization and throughput are maximized, waiting times and turnaround times are minimized, and all users are treated fairly.

The basic functionality of SLURM can be divided into three areas:

- Job submission and cancellation
- Monitoring job and system information
- Retrieving accounting information

Job submission and cancellation

There are three commands for handling job submissions:

- `sbatch`[66]
  - submits a batch job script to SLURM's job queue for (later) execution. The batch script may be given to `sbatch` by a file name on the command line or can be read from *stdin*. Resources needed by the job may be specified via command line options and/or directly in the job script. A job script may contain several job steps to perform several parallel tasks within the same script. Job steps themselves may be run sequentially or in parallel. SLURM regards the script as the first job step.
- `salloc`[67]
  - allocates a set of nodes, typically for interactive use. Resources needed may be specified via command line options.
- `srun`[68]

[64] https://slurm.schedmd.com/overview.html
[65] https://slurm.schedmd.com/priority_multifactor.html
[66] https://slurm.schedmd.com/sbatch.html
[67] https://slurm.schedmd.com/salloc.html
[68] https://slurm.schedmd.com/srun.html

- usually runs a command on nodes previously allocated via `sbatch` or `salloc`. Each invocation of `srun` within a job script corresponds to a job step and launches parallel tasks across the allocated resources. A task is represented e.g. by a program, command, or script. If `srun` is not invoked within an allocation it will via command line options first create a resource allocation in which to run the parallel job.

SLURM assigns a unique *jobid* (integer number) to each job when it is submitted. This *jobid* is returned at submission time or can be obtained from the `squeue` command.

The `scancel`[69] command is used to abort a job or job step that is running or waiting for execution.

The `scontrol`[70] command is mainly used by cluster administrators to view or modify the configuration of the SLURM system but it also offers the users the possibility to control their jobs (e.g. to hold and release a pending job).

The Table below lists basic user activities for job submission and cancellation and the corresponding SLURM commands.

Table 7: User activities for job submission and cancellation
(user supplied information is given in *italics*)

User activity	SLURM command
Submit a job script for (later) execution	`sbatch` *job-script*
Allocate a set of nodes for interactive use	`salloc` --nodes=*N*
Launch a parallel task (e.g. program, command, or script) within allocated resources by `sbatch` (i.e. within a job script) or `salloc`	`srun` *task*
Allocate a set of nodes and launch a parallel task directly	`srun` --nodes=*N task*
Abort a job that is running or waiting for execution	`scancel` *jobid*
Abort all jobs of a user	`scancel` --user=*username* or generally `scancel` --user=$USER
Put a job on hold (i.e. pause waiting) and	`scontrol` hold *jobid*
Release a job from hold (These related commands are rarely used in standard operation.)	`scontrol` release *jobid*

The major command line options that are used for `sbatch` and `salloc` are listed in the Table below. These options can also be specified for `srun`, if `srun` is not used in the context of nodes previously allocated via `sbatch` or `salloc`.

[69] https://slurm.schedmd.com/scancel.html
[70] https://slurm.schedmd.com/scontrol.html

Table 8: Major sbatch and salloc options

Specification	Option	Comments
Number of nodes requested	--nodes=*N*	
Number of tasks to invoke on each node	--tasks-per-node=*n*	Can be used to specify the number of cores to use per node, e.g. to avoid hyper-threading[71]. (If option is omitted, all cores and hyperthreads are used; Hint: using hyperthreads is not always advantageous.)
Partition	--partition= *partitionname*	SLURM supports multiple partitions instead of several queues
Job time limit	--time=*time-limit*	*time-limit* may be given as minutes or in hh:mm:ss or d-hh:mm:ss format (d means number of days)
Output file	--output=*out*	Location of *stdout* redirection

For the sbatch command these options may also be specified directly in the job script using a pseudo comment directive starting with #SBATCH as a prefix. The directives must precede any executable command in the batch script:

```
#!/bin/bash
#SBATCH --partition=std
#SBATCH --nodes=2
#SBATCH --tasks-per-node=16
#SBATCH --time=00:10:00
...
srun ./helloParallelWorld
```

A complete list of parameters can be retrieved from the man pages for sbatch, salloc, or srun, e.g. via

```
man sbatch
```

Monitoring job and system information

There are four commands for monitoring job and system information:

- sinfo[72]

[71] https://en.wikipedia.org/wiki/Hyper-threading
[72] https://slurm.schedmd.com/sinfo.html

- shows current information about nodes and partitions for a system managed by SLURM. Command line options can be used to filter, sort, and format the output in a variety of ways. By default it essentially shows for each partition if it is available and how many nodes and which nodes in the partition are *allocated* or *idle* (or are possibly in another state like *down* or *drain*, i.e. not available for some time). This is useful for the user e.g. to decide in which partition to run a job. The number of allocated and idle nodes indicates the actual utilization of the cluster.

- `squeue`[73]
  - shows current information about jobs in the SLURM scheduling queue. Command line options can be used to filter, sort, and format the output in a variety of ways. By default it lists all pending jobs, sorted descending by their priority, followed by all running jobs, sorted descending by their priority. The major job states are:
    * R for Running
    * PD for Pending
    * CD for Completed
    * F for Failed
    * CA for Cancelled

    The `TIME` column shows for running jobs their execution time so far (or 0:00 for pending jobs).
    The `NODELIST (REASON)` column shows either on which nodes a job is running or why the job is pending. A job is pending for two main reasons:
    * it is still waiting for resources to become scheduled, shown as `(Resources)`,
    * its priority is still not sufficient for it to become executed, shown as `(Priority)`, i.e. there are other jobs with a higher priority pending in the queue.

    The position of a pending job in the queue indicates how many jobs are executed before and after it. The `squeue` command is the main way to monitor a job and can e.g. also be used to get the information about the expected starting time of a job (see Table below).

- `sstat`[74]
  - is mainly used to display various status information of a running job taken as a snapshot. The information relates to CPU, task, node, Resident Set Size (RSS), and virtual memory (VM), etc.

- `scontrol`[75]
  - is mainly used by cluster administrators to view or modify the configuration of the SLURM system, but it also offers users the possibility to get some information about the cluster configuration (e.g. about partitions, nodes, and jobs).

The Table below lists basic user activities for job and system monitoring and the corresponding SLURM commands.

[73] https://slurm.schedmd.com/squeue.html
[74] https://slurm.schedmd.com/sstat.html
[75] https://slurm.schedmd.com/scontrol.html

Table 9: User Activities for Job and System Monitoring
(user supplied information is given in italics;
brackets indicate optional specifications)

User activity	SLURM command
View information about currently available nodes and partitions. The state of a partition may be `UP`, `DOWN`, or `INACTIVE`. If the state is `INACTIVE`, no new submissions are allowed to the partition.	`sinfo` *[*--partition=*partitionname]*
View summary about currently available nodes and partitions. The `NODES(A/I/O/T)` column contains corresponding number of nodes being *allocated*, *idle*, in some *other* state and the *total* of the three numbers.	`sinfo` -s
Check the state of all jobs.	`squeue`
Check the state of all own jobs.	`squeue` --user=$USER
Check the state of a single job.	`squeue` -j *jobid*
Check the expected starting time of a pending job.	`squeue` --start -j *jobid*
Display status information of a running job (e.g. average CPU time, average Virtual Memory (VM) usage – see `sstat` --helpformat and `man sstat` for information on more options).	`sstat` --format=AveCPU, AveVMSize -j *jobid*
View SLURM configuration information for a partition cluster node (e.g. associated nodes).	`scontrol` show partition *partitionname*
View SLURM configuration information for a cluster node.	`scontrol` show node *nodename*
View detailed job information.	`scontrol` show job *jobid*

Retrieving accounting information

There are two commands for retrieving accounting information:

- `sacct`[76]
  - shows accounting information for jobs and job steps in the SLURM job accounting log or SLURM database. For active jobs the accounting information is accessed via the job accounting log file. For completed jobs it is accessed via the log data saved in the SLURM database. Command line options can be used to filter, sort, and format the output in a variety of ways. Columns for jobid, jobname, partition, account, allocated CPUs, state, and exit code are shown by default for each of the user's jobs eligible after midnight of the current day.
- `sacctmgr`[77]
  - is mainly used by cluster administrators to view or modify the SLURM account information, but it also offers users the possibility to get some information about their account. The account information is maintained within the SLURM

[76] https://slurm.schedmd.com/sacct.html
[77] https://slurm.schedmd.com/sacctmgr.html

database. Command line options can be used to filter, sort, and format the output in a variety of ways.

The Table below lists basic user activities for retrieving accounting information and the corresponding SLURM commands.

Table 10: User Activities for Retrieving Accounting Information
(user supplied information is given in italics)

User Activity	SLURM Command
View job account information for a specific job.	`sacct` -j *jobid*
View all job information from a specific start date (given as `yyyy-mm-dd`).	`sacct` -S *startdate* -u $USER
View execution time for (completed) job (formatted as `days-hh:mm:ss`, cumulated over job steps, and without any header).	`sacct` -n -X -P -o Elapsed -j *jobid*

Examples

Submitting a batch job

Below an example script for a SLURM batch job – in the sense of a hello world program – is given. The job is suited to be run in the *Hummel* HPC cluster[78] at Regional Computing Center / Regionales Rechenzentrum der Universität Hamburg (RRZ[79]). For other cluster systems some appropriate adjustments will probably be necessary.

```
1  #!/bin/bash
2  # Do not forget to select a proper partition if the default
3  # one is no fit for the job! You can do that either in the sbatch
4  # command line or here with the other settings.
5  #SBATCH --job-name=hello
6  #SBATCH --nodes=2
7  #SBATCH --tasks-per-node=16
8  #SBATCH --time=00:10:00
9  # Never forget that! Strange happenings ensue otherwise.
10 #SBATCH --export=NONE
11
12 set -e # Good Idea to stop operation on first error.
13
14 source /sw/batch/init.sh
15
16 # Load environment modules for your application here.
17
18 # Actual work starting here. You might need to call
19 # srun or mpirun depending on your type of application
20 # for proper parallel work.
```

[78] https://www.rrz.uni-hamburg.de/services/hpc/hummel-2015.html
[79] https://www.rrz.uni-hamburg.de/

```
# Example for a simple command (that might itself handle
# parallelisation).
echo "Hello World! I am $(hostname -s) greeting you!"
echo "Also, my current TMPDIR: $TMPDIR"

# Let's pretend our started processes are working on a
# predetermined parameter set, looking up their specific
# parameters using the set number and the process number
# inside the batch job.
export PARAMETER_SET=42
# Simplest way to run an identical command on all allocated
# cores on all allocated nodes. Use environment variables to
# tell apart the instances.
srun bash -c 'echo "process $SLURM_PROCID \
(out of $SLURM_NPROCS total) on $(hostname -s) \
parameter set $PARAMETER_SET"'
```

The job script file above can be stored e.g. in `$HOME/hello_world.sh` (`$HOME` is mapped to the user's home directory). One peculiarity of the *Hummel* cluster is that the user's home directory is write protected for a batch job to avoid storing batch job results unintentionally there.

A corresponding working (sub-) directory can be created before the `hello_world.sh` job is submitted:

```
[exampleusername@node001 14:48:33]~$ mkdir $WORK/hello_workdir
[exampleusername@node001 14:48:33]~$ cd $WORK/hello_workdir
```

The working directory `$WORK` of a user is a location that is accessible to all nodes of a user's job.

The job is submitted to SLURM's batch queue using the default value for partition (*scontrol show partitions* (also see above) can be used to show that information):

```
[exampleusername@node001 14:48:33]~$ sbatch $HOME/hello_world.sh
Submitted batch job 123456
```

The output of `sbatch` will contain the *jobid*, like 123456 in this example.

During execution the output of the job is written to a file, named `slurm-123456.out` in this example:

```
[exampleusername@node001 14:48:33]~$ cat slurm-123456.out
module: loaded site/slurm
module: loaded site/tmpdir
module: loaded site/hummel
module: loaded env/system-gcc
Hello World! I am node223 greeting you!
Also, my current TMPDIR: /scratch/exampleusername.123456
process 8 (out of 32 total) on node223 parameter set 42
process 15 (out of 32 total) on node223 parameter set 42
process 4 (out of 32 total) on node223 parameter set 42
process 5 (out of 32 total) on node223 parameter set 42
```

```
12 process 9 (out of 32 total) on node223 parameter set 42
13 process 7 (out of 32 total) on node223 parameter set 42
14 process 3 (out of 32 total) on node223 parameter set 42
15 process 6 (out of 32 total) on node223 parameter set 42
16 process 11 (out of 32 total) on node223 parameter set 42
17 process 2 (out of 32 total) on node223 parameter set 42
18 process 13 (out of 32 total) on node223 parameter set 42
19 process 12 (out of 32 total) on node223 parameter set 42
20 process 1 (out of 32 total) on node223 parameter set 42
21 process 10 (out of 32 total) on node223 parameter set 42
22 process 0 (out of 32 total) on node223 parameter set 42
23 process 14 (out of 32 total) on node223 parameter set 42
24 process 28 (out of 32 total) on node224 parameter set 42
25 process 23 (out of 32 total) on node224 parameter set 42
26 process 26 (out of 32 total) on node224 parameter set 42
27 process 27 (out of 32 total) on node224 parameter set 42
28 process 30 (out of 32 total) on node224 parameter set 42
29 process 19 (out of 32 total) on node224 parameter set 42
30 process 18 (out of 32 total) on node224 parameter set 42
31 process 22 (out of 32 total) on node224 parameter set 42
32 process 25 (out of 32 total) on node224 parameter set 42
33 process 17 (out of 32 total) on node224 parameter set 42
34 process 29 (out of 32 total) on node224 parameter set 42
35 process 21 (out of 32 total) on node224 parameter set 42
36 process 24 (out of 32 total) on node224 parameter set 42
37 process 16 (out of 32 total) on node224 parameter set 42
38 process 31 (out of 32 total) on node224 parameter set 42
39 process 20 (out of 32 total) on node224 parameter set 42
```

If there had been errors (i.e. any output to the *stderr* stream) a corresponding file named slurm-123456.err would have been created.

Interactive usage under control of the batch system

Interactive sessions under control of the batch system can be created via `salloc`. `salloc` differs from `sbatch` by the fact that resources are initially only reserved (i.e. allocated) without executing a job script. Also, the session is running on the node on which `salloc` was invoked (but not on a compute node in contrast to submission with `sbatch`). This is often useful during the interactive development of a parallel program.

A single node is reserved for interactive usage as follows:

```
1 [exampleusername@node001 14:48:33]~$ salloc
```

When the resources are granted by SLURM, `salloc` will start a new shell on the (login or head) node where `salloc` was executed. This interactive session is terminated by exiting the shell or by reaching the time limit.

An OpenMP program using N threads, for example, can be started on the allocated node as follows:

```
[exampleusername@node001 14:48:33]~$ export OMP_NUM_THREADS=N
[exampleusername@node001 14:48:33]~$ srun my-openmp-binary
```

To start an interactive parallel MPI program N nodes can be allocated as follows:

```
[exampleusername@node001 14:48:33]~$ salloc --nodes=N
```

The MPI Program using $n = 32$ processes, for example, can be started on the allocated nodes as follows:

```
[exampleusername@node001 14:48:33]~$ mpirun -np 32 my-mpi-binary
```

Another way to use the allocated nodes is to use ssh to establish connections to them.

TORQUE

Since PBS and TORQUE offer similar features as SLURM it is appropriate to show the corresponding commands and options in Tables. A good overview to this topic is also given e.g. at the Division of Information Technology at the University of Maryland[80].

Table 11: Command comparison between PBS/TORQUE and SLURM
(user supplied information is given in *italics*)

User Activity	PBS/TORQUE	SLURM
Submitting a job	`qsub` *jobscript*	`sbatch` *jobscript*
Using nodes interactively	`qsub` -I *[options]*	`salloc` *[options]*
Deleting or canceling a job	`qdel` *jobid*	`scancel` *jobid*
Viewing job status	`qstat` *jobid*	`squeue` -j *jobid*
Viewing all of one's own job status	`qstat`-u $USER	`squeue` -u $USER
Viewing all jobs in the queue	`qstat` [-a]	`squeue`
Checking expected starting time of a job		`squeue` --start -j *jobid*

Additional job parameters are listed below. These options are typically used in job scripts using a pseudo comment directive starting with a special prefix. The directives must precede any executable command in the batch script:

Table 12: Option comparison between PBS/TORQUE and SLURM
(user supplied information is given in italics)

Job Parameter	PBS/TORQUE	SLURM
Directive	#PBS	#SBATCH
Job name	-N *name*	--name=*name*
Queue/Partition	-Q *queuename*	--partition= *partitionname*
Nodes	-l nodes=*n*	--nodes=*n*
Processes per node	-l ppn=*n*	--tasks-per-node=*n*
CPUs per task	-l ompthreads=*n*	--cpus-per-task=*n*

[80] https://www.glue.umd.edu/hpcc/help/slurm-vs-moab.html

Job Parameter	PBS/TORQUE	SLURM
Job time limit	-l walltime=*seconds*	--time=*minutes*
	-l walltime=*hh:mm:ss*	--time=*hh:mm:ss*
stdout redirection to a file	-o *filename*	--output=*filename*
stderr redirection to a file	-e *filename*	--error=*filename*
stdout and *stderr* redirection to one file	-oe *filename*	--output=*filename* and *not* specifying --error=*filename*
Email address	-m *address*	--mail-user=*address*
Notify on state change of job	-m b	--mail-type=BEGIN
	-m e	--mail-type=END
	-m a	--mail-type=FAIL
	-m abe	--mail-type=ALL
Do not permit the job to be requeued after a node failure	-r n	--no-requeue

During job runtime several environment variables are automatically set:

Table 13: Environment variable comparison between PBS/TORQUE and SLURM

Content of Environment Variable	PBS/TORQUE	SLURM
JobId	$PBS_JOBID	$SLURM_JOBID
Working directory from which the job was submitted	$PBS_O_WORKDIR	$SLURM_SUBMIT_DIR
List of allocated nodes	$PBS_NODEFILE (a filename)	$SLURM_JOB_NODELIST (node list itself)

PE3-B Benchmarking

Relevant for: Tester, Builder, and Developer
Description:

- You will learn to assess speedups and efficiencies as the key measures for benchmarks of a parallel program (basic level) (see also K2.1-B Performance Frontiers, p. 13)
- You will learn to differentiate between strong and weak scaling (basic level)
- You will learn to benchmark the runtime behavior of parallel programs, performing controlled experiments by providing varying HPC resources (e.g. 1, 2, 4, 8, ... cores on shared memory systems or 1, 2, 4, 8, ... nodes on distributed systems for the benchmarks) (basic level)
- You will learn about the performance impact of certain features of current CPU architectures (temperature and dynamic CPU frequencies) (basic level)

Level: basic

Motivation

Benchmarking means the comparative analysis of measurement results of certain program properties for the performance determination of hard- or software. The measurement results, primarily those of the program runtimes, are determined by controlled experiments.

Such a controlled experiment is named a *benchmark*, but the term is also used – apparent from the context – for the program that is, or set of programs that are used for benchmarking.

For HPC users measuring the performance behavior of the parallel program(s) they use is of primary importance in order to make optimal use of HPC hardware. Hence this is the focus of this learning unit.

Before benchmarking is explained in more detail, standard benchmarks, like Linpack[81], shall briefly be mentioned here for the sake of completeness.

Benchmarking hardware

The Linpack benchmark is used, for example, to build the TOP 500 list[82] of the currently fastest supercomputers, which is updated twice a year. The computer's floating-point rate of execution is measured by running a program that solves a dense system of linear equations. Beside the floating point operations per second (FLOPS), the Top 500 list contains a number of additional details of the different supercomputers, like CPU architectures, power consumption, number of cores, etc. Such information shows current

[81] https://www.top500.org/project/linpack/
[82] https://www.top500.org/lists/

trends in HPC and is of great interest for computing centers with view to the procurement of next generation cluster system.

It is also common practice to execute a mixture of different parallel programs to get impressions of the total performance of a cluster system under conditions simulating practical use.

Benchmarking software

For HPC users, however, synthetic tests to benchmark HPC cluster hardware (like the Linpack benchmark) are of less importance. For them, the emphasis lies on the determination of speedups and efficiencies of the parallel program they want to use with regard to varying the number of cores (or cluster nodes) for the measurements.

This kind of benchmarking is very essential in the HPC environment and can be applied to a variety of issues:

- 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 is also a basis for dealing with questions emerging from tuning, e.g.:

- What is the appropriate task size (big vs. small) that may have a positive performance impact on my program?
- Is the use of hyper-threading[83] technology advantageous?
- What is the best mapping of processes to nodes, pinning[84] of processes/threads to CPUs or cores, and setting memory affinities to NUMA[85] nodes in order to speed up a parallel program?
- What is the best compiler selection for my program (GCC, Intel, PGI, . . .), in combination with the most suitable MPI environment (Open MPI, Intel MPI, . . .)?
- What is the best compiler generation/version for my program?
- What are the best compiler options regarding for example the optimization level `-O2`, `-O3`, . . . , for building the executable program?
- Is the use of PGO[86] (Profile Guided Optimization) or other high level optimization, e.g. using IPA/IPO[87] (Inter-Procedural Analyzer/Inter-Procedural Optimizer), helpful?
- What is the performance behavior after a (parallel) algorithm has been improved, i.e. to what extend are speedup, efficiency, and scalability improved?

[83] https://en.wikipedia.org/wiki/Hyper-threading
[84] https://en.wikipedia.org/wiki/Processor_affinity
[85] https://de.wikipedia.org/wiki/Non-Uniform_Memory_Access
[86] https://en.wikipedia.org/wiki/Profile-guided_optimization
[87] https://en.wikipedia.org/wiki/Interprocedural_optimization

If the focus is on performance improvements, the user should also feel motivated to benchmark the program after any change of the environment, the configuration, or the program itself.

Benchmarking the runtime behavior of parallel programs

For benchmarking a parallel program at the basic level it is essential to measure its runtimes in dependence of the HPC resources provided (e.g. 1, 2, 4, 8, 16, 32, 64, . . . cluster nodes) in a series of experiments.

For a parallel program that uses a single cluster node only, as a typical OpenMP program, controlled experiments to measure the execution time are performed in a similar way providing an increasing number of cores (e.g. 1, 2, 4, 8, 16, . . .) in this case.

The parallel efficiency E is the ratio of the speedup S achieved in an experiment to the number of cluster $nodes$ or $cores$, respectively, used in the experiment:

$$E_{nodes} = \frac{S_{nodes}}{nodes}$$

or

$$E_{cores} = \frac{S_{cores}}{cores}.$$

Below, two Tables with hypothetical benchmark results are given as examples for measuring the runtimes of a parallel program to calculate π with the same vast number of decimal places in each experiment:

Table 14: Benchmark results of π calculation for a varying number of cluster nodes

Program	Runtime [s]	Cluster Nodes	Total Cores	Speedup	Efficiency
calc-pi-MPI	180.5	1	16	1.00	100%
calc-pi-MPI	92.1	2	32	1.96	98%
calc-pi-MPI	47.5	4	64	3.80	95%
calc-pi-MPI	25.1	8	128	7.19	90%

Table 15: Benchmark results of π calculation for a varying number of cores on a single cluster node

Program	Runtime [s]	Total Cores	Speedup	Efficiency
calc-pi-OpenMP	2800.0	1	1.00	100%
calc-pi-OpenMP	1414.1	2	1.98	99%
calc-pi-OpenMP	707.1	4	3.96	99%
calc-pi-OpenMP	360.8	8	7.76	97%

How can runtimes be measured?

For the total runtime of programs started interactively, the answer is simple: the `time` command will do the trick.

Note that many shells (including `bash`) have a built-in `time` command, which may offer less features than the globally installed stand-alone program. For MPI programs the shell built-in gives enough relevant information:

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

For a sequential or an OpenMP program `/usr/bin/time` gives more interesting information than time in addition:

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

Scaling

The scalability[88] of a program is considered to be good when the efficiency remains high when the number of processors is steadily increased.

As work is distributed among more and more processes the overheads for synchronization (e.g. waiting for partial results of other processes) and communication (e.g. distributing partial data or tasks to other processes and collecting partial results from them) are also increasingly growing. Eventually, this will lead to a reduced performance of the program and the efficiency will drop.

Typically, there exists a dependency on the problem size, e.g. on the size of a matrix, to be processed with a parallel algorithm.

Two types of scalability can be distinguished that deal with the question of how many nodes (or cores) can be reasonably used to further reduce the time to solution to solve a problem. It is required for:

- *Weak scaling*, that the problem size increases proportional to the number of parallel processes.
  In other words, weak scaling answers the question: How big may the problems be that I can solve?
- *Strong scaling*, that the problem size remains the same for an increasing number of processes.
  In other words, strong scaling answers the question: How fast can I solve a problem of a given size?

A typical use case for weak scaling is to predict the performance for using very many processes.

For illustration scaling plots are shown below. These were obtained from benchmarking *conjugate gradient solvers* in simulations of *lattice gauge theories*. The main properties of these kinds of calculations are:

[88] https://en.wikipedia.org/wiki/Scalability

- The problems are 4-dimensional. The data access pattern is given by a 9-point stencil.
- Communication patterns are halo boundary exchange and global sums.

Weak scaling

For weak scaling the number of processors n is increased and the domain size $data$ (i.e. the size of the problem) per process is kept constant, i.e. total domain size $data_n \propto n$.

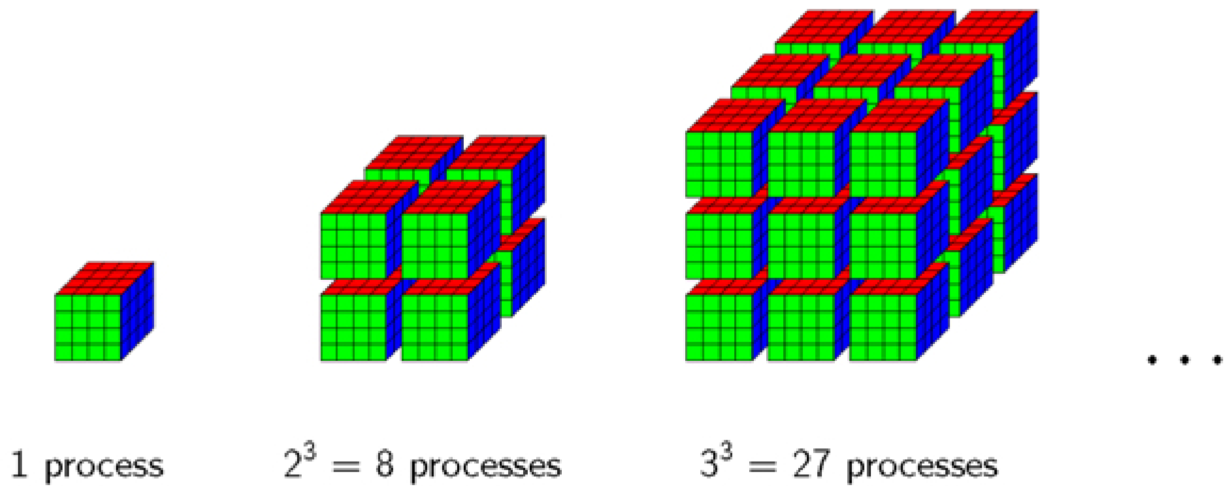

Figure 3: Weak scaling: geometry example

This typically leads to the following observations:

- 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

In the plot below performance per core is plotted over the number of cores used. Typical weak scaling behaviour can be seen: performance per process decreases when more cores are used and then flattens out at some point.

If performance figures are not available execution time can be plotted. Execution time is expected to increase and then flatten out.

A logarithmic x-scale is better suited if the number of processes varies over orders of magnitudes

Strong Scaling

For strong scaling the number of processors n is increased and the total domain size $data$ (i.e. the total size of the problem) is kept constant.

This typically leads to the following observation:

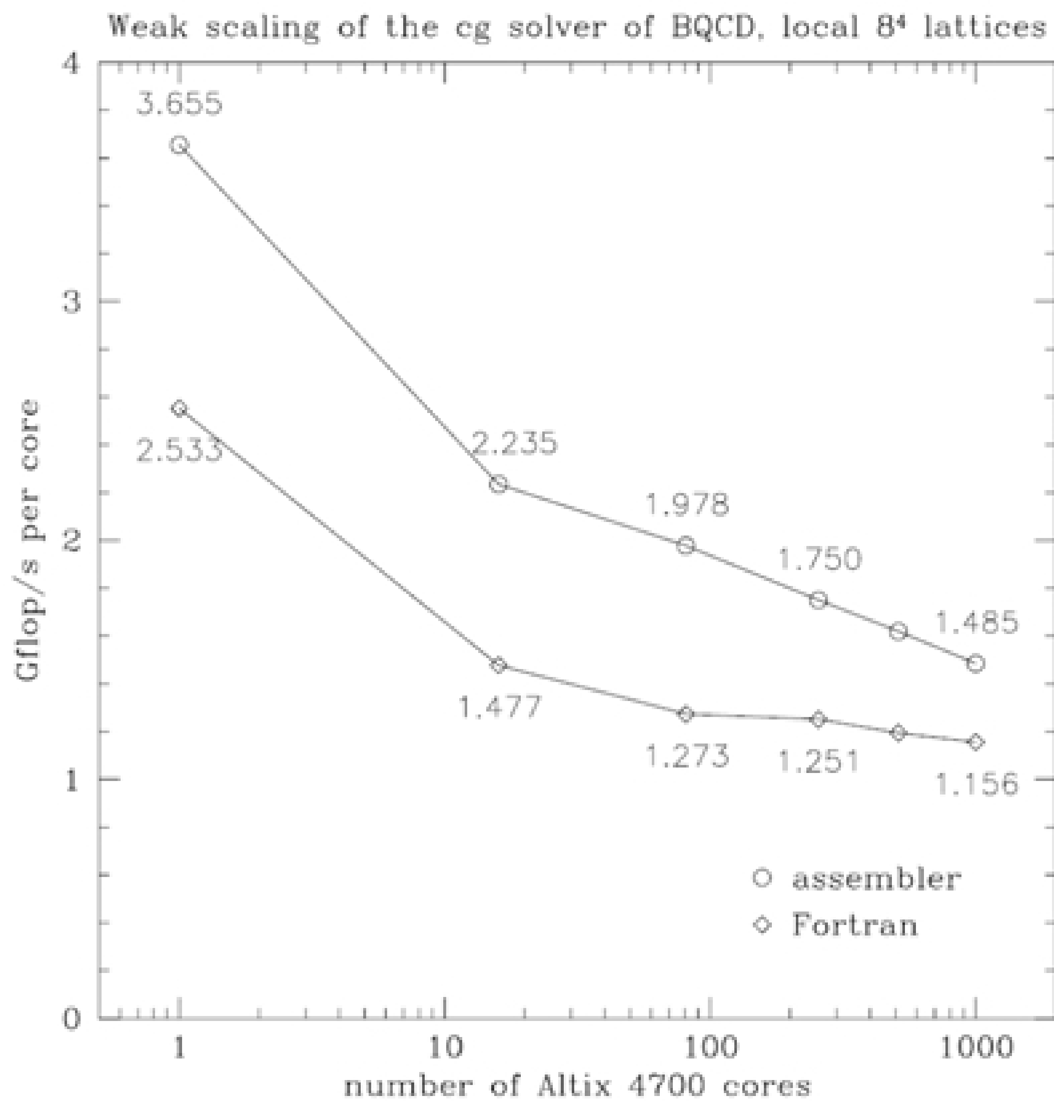

Figure 4: Weak scaling (BQCD on SGI Altix)

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

The motivation for studying strong scaling is to determine an optimal number of processes to use.

The question whether an efficiency may be considered to be acceptable depends on context, e.g. the algorithm a program is based on, and on boundary organizational or “political” conditions: the more important it is to reduce the time to solution, the lower a still acceptable efficiency will be.

Normally, it will obviously be harder to achieve acceptable efficiencies for strong scaling conditions than for weak scaling conditions when the number of processors is steadily increased.

Strong scaling plot examples

A classic strong scaling plot shows speedup over the number of processes. A reference point needs to be defined where the speedup is 1. Originally, a single process was chosen where the (parallel) program is executed sequentially. Nowadays, a single node is a natural reference point. Other points can be chosen, e.g. a rack.

Typically, linear scaling is indicated in a strong scaling plot for comparison. Double logarithmic plots are advantageous because orders of magnitudes can be better resolved (linear scaling is still represented by a straight line in double logarithmic plots).

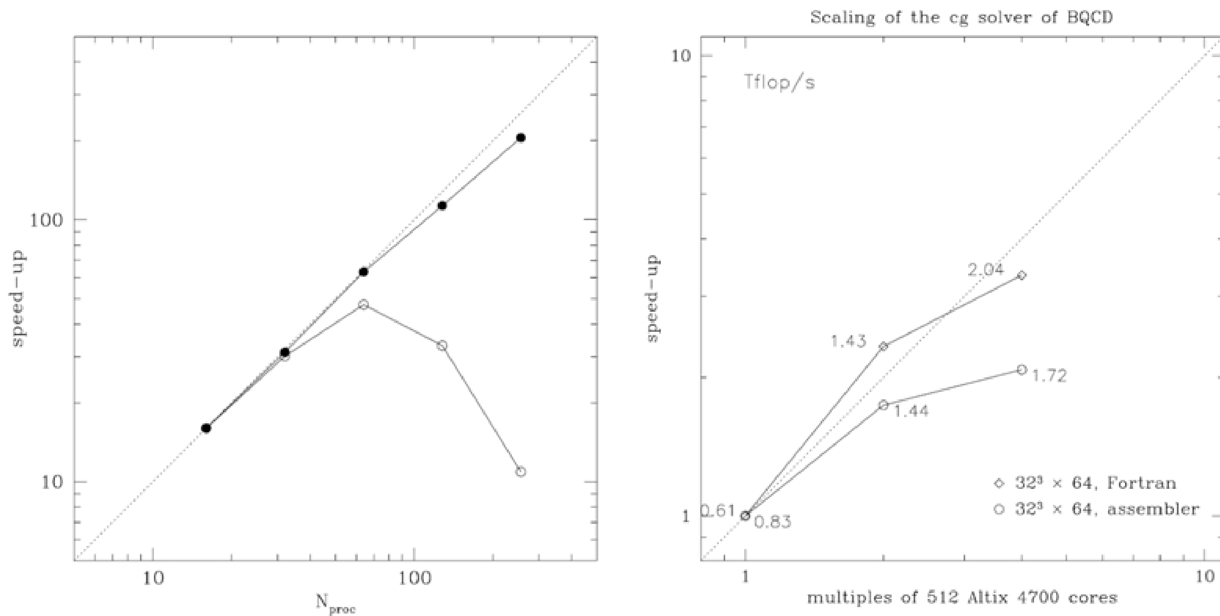

Figure 5: Strong scaling (speedup of QED on Cray T3D and BQCD on SGI Altix)

In the figure above two scaling plots are shown. The plot on the left hand side shows bad scaling (open circles) of an early parallel program in which the global sum was implemented in a naive way (the computational effort was proportional to the number of processes used) and good scaling (filled circles) that was achieved after implementing an improved version of the global sum (using a binary tree algorithm, which has only logarithmic complexity). In this case the reason for bad scaling is a naive algorithm. Two points should be remembered:

- Even the bad implementation could by used with up to 64 processes.
- The speedup curve of *any* parallel program will bend down if too many processes are used.

The plot on the right hand side shows speedup curves for two different implementations of the solver of the BQCD program. The Fortran implementation (diamonds) displays a *superlinear* speedup from 512 to 1024 cores. The assembler implementation (circles) is faster on 512 cores. On 1024 cores both implementations run at about the same speed. On 2048 cores the speedup is unacceptably low in both cases (the efficiency compared with running on 1024 cores is 71% for the Fortran and 60% for assembler implementation). The plot was chosen in order to explain how to decide on the number of processes that can reasonably be used.

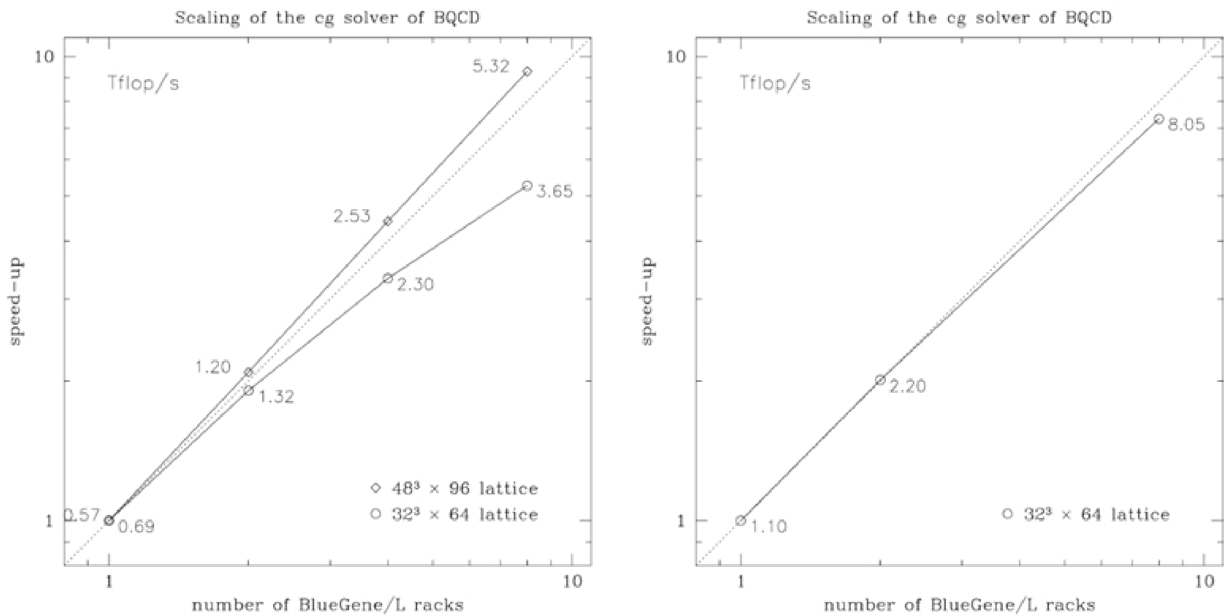

Figure 6: Strong scaling (speedup of BQCD on IBM BlueGene/L)

The next plot, above, shows scaling results for two different problem sizes obtained with a Fortran implementation (left hand side) and an assembler implementation (right hand side). Remarkable is that

- the larger problem (diamonds) displays superlinear speedup from 1 rack (2048 cores) up to 8 racks (16384 cores),
- the assembler implementation can improve performance on the smaller problem (circles) considerably and scales almost linearly (this occurs because communication and computation overlap in the assembler implementation).

While these observations are specific to the problem, important general observations can be made as well:

- Plotting speedup can be misleading when only scaling curves are compared, i.e. scaling can look better, while performance is worse. One has to look at performance, too.
- Generally, slow programs scale better.

In instead of speedup one can directly plot performance. For demonstration this was done with the data from the figure above on the left hand side of the plot shown below. In such a plot one can directly compare scaling and performance. Because there is no reference point there is no single reference line for linear performance (through any point such a line could be drawn). The plot contains three lines that indicate linear scaling to guide the eye.

If performance figures are not available one can make the same kind of plot from execution times. This is shown in the plot on the right hand side in the figure above. (In this case just the inverse of performance data was plotted, which is inverse proportional to performance.) In general, one has to normalize times, e.g. plot the time per iteration and mesh point.

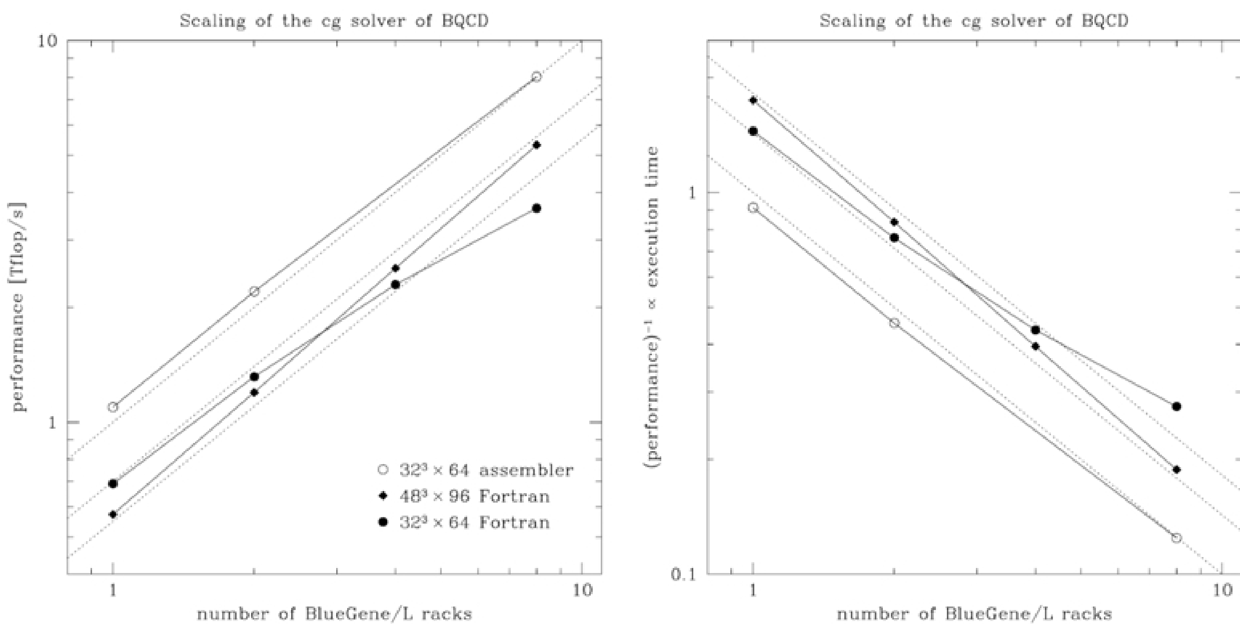

Figure 7: Strong scaling (performance and execution time of BQCD on IBM BlueGene/L)

Further benchmark opportunities

Some possible application areas of benchmarking have already been mentioned in connection with tuning ideas in the motivation section. Benchmarking can be used for example to determine the most suitable compiler (GCC, Intel, PGI, . . .) and MPI environment (Open MPI, Intel MPI, . . .) for building a program that is as fast as possible. It can also be observed now and then, that for some reason a predecessor version of a compiler might create code that is faster than the code created with the current version. This might indicate that the program has to be adopted to the current compiler version so that fast (or even faster) executable code is created again.

To take another example already mentioned in the motivation section, benchmarking can also be used to answer the question if there is a positive trade-off to be expected when performing the typical two steps of the Profile Guided Optimization (PGO) cycle consisting of

- Step 1: running initially the instrumented (and therefore relatively slow) version of the program providing representative input data to collect information about which branches are typically taken and other typical program behavior,
- Step 2: recompiling with this information to build a faster program.

For some algorithms and a given constant problem size there might exist an inverse proportional relationship between the main memory provided for the parallel program and the execution time, e.g. when large hash tables[89] can be used for caching partial results that would otherwise have to be recalculated (multiple times) on demand, to give just one example. In this case it may be beneficial for the user to choose a partition (or batch queue, respectively) of a given cluster that offers him a smaller number of bigger nodes (i.e. nodes with a larger amount of main memory) when compared with

[89] https://en.wikipedia.org/wiki/Hash_table

other partitions that could be chosen. Appropriate benchmarks can be used to examine if it is advantageous to use the nodes with the larger amount of main memory.

Besides performing benchmarks depending essentially on the computing power of a cluster system it might also be interesting to benchmark the network bandwidth (e.g. InfiniBand vs. Ethernet) or the I/O performance of the file system of cluster systems. This is interesting in particular if the user can choose between such possibilities, e.g. if different cluster systems are available to submit batch jobs to. Within the same cluster it may furthermore be interesting to see which positive impact choosing the adequate file system might have on reducing the runtime, e.g. using a fast SSD (Solid State Disk) instead of the parallel file system for locally storing temporary data.

Pitfalls

Several topics are listed below which should be considered carefully in order to avoid typical pitfalls.

Break-even considerations regarding the benchmark effort

It follows from the foregoing that benchmarking, especially with regard to determining the scalability of a parallel program, is important.

However, benchmarking also represents a certain effort, namely

- for providing the HPC resources explicitly used for that purpose,
- and human time for (manually) performing the experiments, collecting and interpreting the results.

Test input data representing a comparatively small problem size can often be helpful to assess reasonably accurate the performance behavior for larger problem sizes, without incurring inappropriately high CPU time usages (also see explanations of weak and strong scaling above).

Depending on the purpose, varying the provided HPC resources for benchmarking can be performed in larger steps, so that for example 1, 4, 16, 64, ... cluster nodes are used for the series of experiments. This way the HPC resource usage as well as the (manual) effort to manage the experiments can be greatly reduced.

Break-even considerations should be kept in mind, but anyway it can be assumed that there will usually be a positive trade-off performing benchmarks in an appropriate manner. This is particularly true in connection with tuning, for which benchmarking is the basis.

Presenting fair speedups

For the determination of speedups the parallel program executed on a single core is often considered as sequential version of an algorithm. The runtime measured on a single core is then used as numerator T_1 for calculations like:

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

A parallel program might scale pretty well under these conditions. Therefore it is important to be aware of the *best known* sequential algorithm for such comparisons in order to get fair speedup results (especially when it is planned to publish the results).

Special features of current CPU architectures

The CPU clock rates of a typical multi-core cluster node supporting features like turbo boost[90] may vary depending on the CPU usage. At low CPU load, when for example only a single core is used, the clock rate of this core is typically considerably higher than the clock rate at times when several cores are fully utilized. If many cores are fully utilized over a period of time, the clock rates of the cores will be usually reduced over time to avoid a rise in CPU temperature, which is heavily affected by the clock rates.

The runtime of benchmarks should therefore not be chosen too short, in order to achieve stable CPU clock rates at first. This also has the advantage that possible overheads in the initialization phase, for example for reading extensive input data using just a single core, or in the de-initialization phase, for example for writing final results to a file, again using just a single core, only have a relatively small impact on the runtime.

A meaningful minimum runtime of course depends on the specific parallel program that is to be benchmarked. As a rule of thumb the runtime should be at least a few minutes.

The hyper-threading[91] technology is a further peculiarity of current multi-core CPU architectures. With hyper-threading enabled, each physical core of the CPU can also be used as two logical hyper-threaded cores, each – simply put – just over half as fast as a physical core. Using two hyper-threaded cores mainly improves exploiting functional subunits of the CPU, so both hyper-threaded cores have slightly more computing power than their corresponding physical core (however, depending on the scalability of a program this does in no way imply a positive trade-off for using hyper-threading). If one core of a hyper-threaded core pair is idle, the other core can run at the full speed of a physical core. This already hints that hyper-threading may add some complexity to benchmarking.

- Provided that all hyper-threads of all cluster nodes are permanently fully utilized by the parallel program, the runtimes measured in a series of benchmark experiments are comparable with each other very well.
- The same applies if hyper-threading is not really used, i.e. if on a node with h hyper-threaded cores no more than $h/2$ cores are utilized by the parallel program. In this case all cores can run at the full speed of a physical core.
- It will be difficult to determine the scalability of a program, if a number of cores is used for the parallel program for which some hyper-threads are running at the full speed of a physical core and some hyper-threads are running only at about the half speed, i.e. if the number n of used cores is in the range $h/2 < n < h$. In such a mixture of core usage it will be hard to assess the speedups achieved in relation to the cores used by calculating efficiencies via $E_{cores} = \frac{S_{cores}}{cores}$.

Shared resources

[90] https://en.wikipedia.org/wiki/Intel_Turbo_Boost
[91] https://en.wikipedia.org/wiki/Hyper-threading

It lies in the nature of benchmarking that cluster nodes of the same type should be made *exclusively* available to the user for benchmark experiments. This avoids the influence of parallel programs from other users – running otherwise possibly on the same nodes at the same time – on the result, i.e. on measuring the execution time as precisely as possible.

In an ideal case, benchmarks are performed (after program changes, for example) even on the same set of nodes they were performed initially, in order to avoid any side effects that might be introduced by using (slightly) different cluster hardware. Such side effects can be caused for instance by manufacturing tolerances of cluster nodes, or differences regarding the interconnection network between nodes.

But even under ideal conditions using the same set of nodes to perform all benchmarks, the program has generally to share some HPC resources with other programs running on the cluster at the same time. These resources include in particular network bandwidth for interprocess communication and I/O bandwidth for accessing a global parallel file system.

In the context of benchmarking, users should be aware of such effects. If they seem actually significant, one idea to alleviate the problem would be to repeat each benchmark several times for calculating averages, or to use the median to avoid the influence of outliers, for example.

Shared nodes

Compute centers often also make shared nodes explicitly available for a cluster system via an appropriate batch queue, or partition, respectively, in which several users may share the cores of the nodes. When the user is only interested in the result of a program, that is usually not much of a problem and it especially enables the compute center to make better use of the expensive HPC resources that would typically be more idling otherwise.

As long as only different cores on the same node are used at any point in time by two or more programs of different users, benchmark results may still be meaningful. If the same cores are potentially shared at times on a node by different programs, the value of the benchmark results may be significantly reduced or even made useless.

Reproducibility

Nowadays, it is increasingly required that scientific results from numerical programs are reproducible, so that they can be verified by external parties.

For parallel programs reproducibility[92] is also important for benchmarking. If the execution of a program depends for example on a randomized component (as used for Monte Carlo algorithms[93]), the random number generator needs to be made deterministic in order to ensure the reproducibility of repeated runs of a program. (Random number generators or, strictly speaking, pseudo random number generators[94], are typically initialized in a fashion (called seeding) that the sequence of the random numbers generated remains the same every time the program is started.)

[92] https://plato.stanford.edu/entries/scientific-reproducibility/
[93] https://en.wikipedia.org/wiki/Monte_Carlo_algorithm
[94] https://en.wikipedia.org/wiki/Pseudorandom_number_generator

However, there are parallel algorithms which may produce non deterministic results, due to inherent effects of concurrency[95]. A variety of parallel tree-search algorithms could serve as a good example. Without going into detail, many tree-search algorithms typically cut-off parts of the search tree during the search. Depending on when branches of the tree are cut off by a search process and how this information is exchanged between all parallel searching processes, usually different parts of the tree are redundantly searched each time the program is restarted. This in turn may lead to different (but generally equivalent) search results and also to strongly differing runtimes of repeated runs.

In the context of benchmarking users should be aware of the existence of parallel algorithms with a non-deterministic behavior (event-driven simulations, for example). One way to alleviate the problem would be to repeat each benchmark several times for calculating averages, another would be to use the median to avoid the influence of outliers.

[95] https://en.wikipedia.org/wiki/Concurrency_(computer_science)