D 1 Parallel Computer Architectures [1]

D. Pleiter
Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH

Contents

[1] Lecture Notes of the 45^{th} IFF Spring School "Computing Solids - Models, ab initio methods and supercomputing" (Forschungszentrum Jülich, 2014).

1 Introduction

Since John von Neumann [Neumann(1945)] and others established modern computer processor architecture concepts in the 1940s, the speed at which arithmetic computations could be performed has increased by many orders of magnitude. One ingredient for this increase in performance has been improvements in computer architectures. The other important driver for this development has been the technology for building integrated circuits. Already in the 1960s it had been observed by Gordon Moore that the number of components per integrated circuit increases roughly by a factor 2 per year [Moore(2006)]. A more long-term analysis of the transistor count for Intel microprocessors revealed a doubling every 2 years [Bohr(2007)]. It has become popular to refer to a so-called "Moore's Law", which today is commonly understood as a doubling of the performance every 18 months.

Over some period of time this increase in performance could be achieved by means of increasing the frequency f at which computing devices are clocked. Today all these devices are based on CMOS technology for which dissipated (active) power scales proportional to $V^2 \cdot f$, where V is the voltage. As voltage has to be increased for larger frequencies, power dissipation grows more than linear when f is increased. Since a couple of years frequency scaling has reached its limit. Today's high-performance processors are typically operated in a frequency range $2.5 \lesssim f \lesssim 4\,\text{GHz}$. Further increase of performance can thus largely only be achieved by scaling-out to a larger number of computational devices running in parallel.

High-performance computing today therefore means massively-parallel computing. In this way still further performance increases can be achieved. This is, e.g., documented by the Top500 project,[2] which publishes twice a year an international list of computers which are fastest in terms of floating-point operation throughput while executing the High-Performance LINPACK (HPL) benchmark. In Fig. 1 we show how the (aggregate) performance of the top 1, 10 and 100 systems increased over time. Let us consider the aggregate performance of the top 100 systems in more detail. From June 1993 to November 2013 their performance doubled almost every 14 months on average, now reaching 180 PFlop/s. This corresponds to an performance increase by a factor $2.5 \cdot 10^5$. During that time the number of processor cores increased by roughly a factor 750. This number reflects only a fraction of the increase in parallelism as we observe an increase of parallelism at many levels of the computer architectures, as we will see later.

This lecture is written for computational scientists for which availability of these massively-parallel computing resources are critical for their research. The purpose is to improve the efficient use of such resources by means of better understanding of computer architectures. We will discussed some of the most relevant conceptional aspects which are realised in today's architectures. But we will also have a detailed quantitative look into crucial performance parameters of parallel computer architectures and the underlying technologies.

This lecture starts with a general overview on system hardware architectures and its components in section 2, followed by a discussion of the term performance and an introduction into performance models in section 3. In the following sections we will have a closer look into specific components like the processor (section 4), the memory (section 5) as well as the network (section 7). In section 6 we will introduce a special type of computing devices, so-called accelerators, which play an increasingly important role in high-performance computing. As we will see in section 8, many of the current trends in the development of massively-parallel computers are due to the need for improved power and energy efficiency. In section 9 we will

[2] `http://www.top500.org`

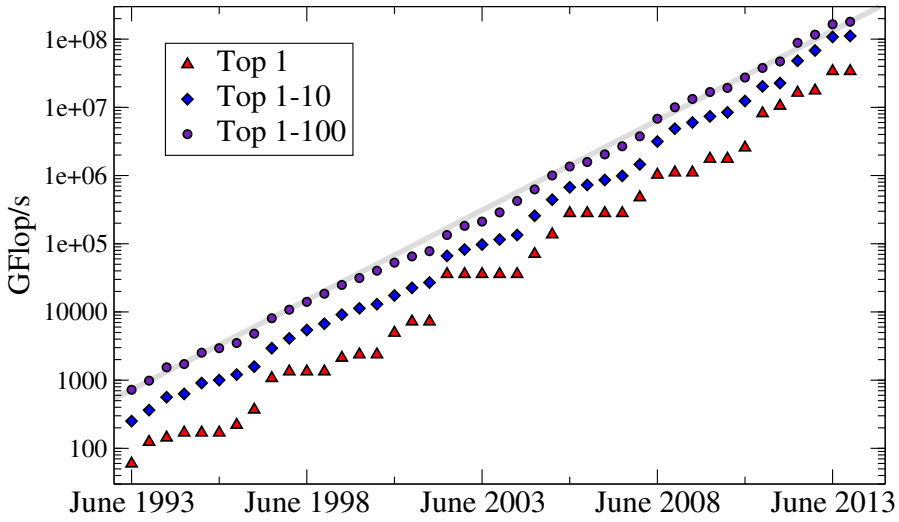

Fig. 1: *The aggregate double-precision floating-point performance achieved by the top 1, 10 and 100 systems on the Top500 list as a function of time. The grey line shows the performance trend for the top 100 systems indicating a doubling of performance every 13.7 months.*

review the trend to increased parallelism, followed in section 10 by a comparison of several high-performance computer architectures, which as of today are intensively used for scientific computations. We will close with an outlook in section 11.

2 Abstracted View on System Hardware Architectures

Today's processor architectures are surprisingly similar to the architecture proposed by von Neumann [Neumann(1945)]. We thus use the latter, significantly simpler architecture as a model to introduce concepts which are helpful to understand modern architectures. In Fig. 2 we show a variant of the Von Neumann architecture. The memory unit *MEM* holds both data and instructions which are processed by the central processing unit *C*. The latter consists of a central control unit *CC* which directs operations, i.e. fetching of instructions and operands, dispatch of instructions and storing results. The central arithmetic unit *CA* executes the arithmetic instructions. Communication with the processor requires the availability of input and output units. All these units are interconnected by a central bus.

Let us assume that data and program have been loaded into the processor's memory and execution of instructions can start. This means that the first instruction can be fetched followed by a fetch of the operands. To hold instruction and operands in the central processing unit small pieces of (fast) memory, called *registers*, are required. Once instruction and input operations are available for execution, the instruction can be issued and results written back to memory. A program counter, which points to the current instruction, then needs to be incremented before the next instruction is loaded. In a very simple program this cycle continues until all instructions have been processed.

This architecture has a crucial limitation, called the Von Neumann bottleneck. Both instructions and data are kept in the same memory and have to be moved along the same data path to the central processing unit resulting in potential congestion on the internal bus. This problem can

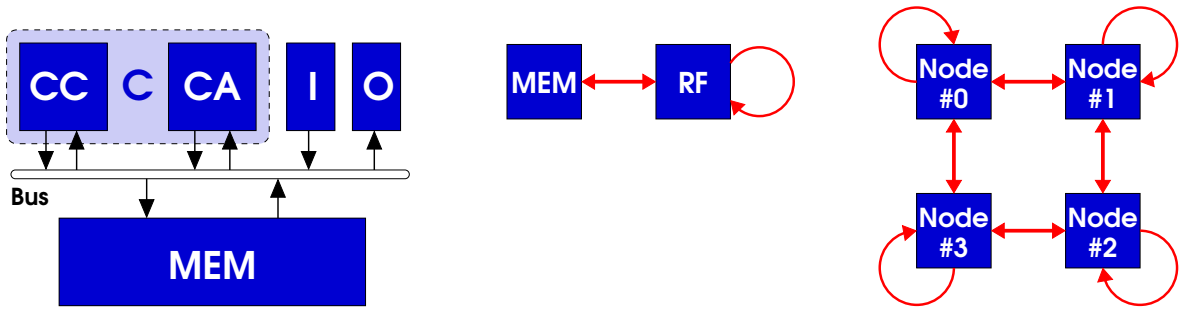

Fig. 2: *On the left a version of the Von Neumann architecture is shown. The central processing unit* C *consists of the central control and arithmetic units* CC *and* CA*, respectively. A common bus connects* C *to the memory* MEM *as well as to the input and output units (*I *and* O*). The middle figure shows an abstract processor architecture model comprising 2 storage devices: The memory* MEM *and a register file* RF*. The graph on the right represents a parallel architecture comprising 4 nodes connected by network links in a ring topology.*

be mitigated by modifying the architecture such that instructions are kept in a separate memory unit directly connected to the central processing unit.[3] Now instructions and operands can in principle be loaded in parallel.
This short digression on modified Von Neumann architectures highlights the importance of data transport within a system hardware architecture. In the following we introduce an abstract system hardware architecture model which focusses on this aspect. The model comprises two types of devices:

Storage devices: Devices which can store data and are characterized by their capacity.

Transport or processing devices: Devices which read data from one storage device, process the data and write the result to another or the same storage device. The data does not have to be processed but may just be copied from one device to another. The performance of the device is in particular characterized by throughput or bandwidth, i.e. the speed of data processing or transport, respectively.

The model can naturally be represented by a graph where the vertices and edges refer to storage and transport devices, respectively. Fig. 2 (middle) shows such a graph for a very simple processor architecture comprising of 2 storage devices: the main memory and a set of registers, called *register file*. These are connected by a memory bus through which data can be read from memory to the register file or written back to memory. The register file is connected to a processing pipeline which takes its input operands from the register file and writes results back to the same register file. The processing pipeline may, e.g., be capable to perform fused multiply-add operations.
It should be noted that the abstraction level of this model can be adjusted. The previous example describes a processor architecture and is relatively close to the actual hardware architecture of very simple processors. Let us consider a parallel architecture with multiple nodes where each node has potentially a complex internal structure, e.g., comprising multiple processors with separate memory units attached to each of them. Each of the nodes could be modeled as a single storage device with one or more processing devices attached, which read data from the

[3] This modified architecture is often called Harvard architecture.

storage device, processes it and write it back to the storage device. The different storage devices, i.e. nodes, are connected by transport devices representing network links. Fig. 2 (right) shows a graphical representation of such a machine with 4 nodes interconnected in a ring topology.

3 Performance and Performance Models

Before we continue on computer architectures with the goal of obtaining a better understanding of the performance a computational scientist might expect for a given application we first have a closer look on different aspects of the term performance.
Here we assume that the user of a high-performance computing system aims on minimizing *time-to-solution* Δt for a given *problem size*, also called *workload*. Time-to-solution refers to the time needed to execute an application, ignoring the efforts required to implement, port or optimize an application, which depending on the complexity of the application may become relevant, too.[4] In future also other performance metrics like energy-to-solution for a given workload might become relevant.
Start and end of application execution can be seen as events, i.e. points of time where something happens. Let us refer to these events by t_{start} and t_{end} and write time-to-solution as $\Delta t = t_{\mathrm{end}} - t_{\mathrm{start}}$. The problem size refers to all parameters which affect the efforts required to solve a computational problem. For Density Function Theory (DFT) calculations this would, e.g., include both the number of nuclei as well as their position.
Any application can be decomposed into a set of computational tasks that need to be solved. An example for such a task, which can be found in many computational science problems, is solving a linear set of equations, e.g. $A\,x = b$. Within a single application such a task may have to be executed repeatedly for different matrices A and different input vectors b. Depending on the properties of this linear equation different numerical algorithms may be applied. Let us assume iterative methods like the Conjugate Gradient method to be a good choice. In this case the numerical task of solving this equation can be further decomposed in smaller subtasks like matrix-vector multiplications, scalar products and linear combinations. While application developers may organise the implementation of tasks (or subtasks) in separate functions or subroutines, this is not relevant for the theoretical concept formulated here.
In practice, it often can be observed that a large fraction of the total time-to-solution is spent in just a few tasks or subtasks. A set of such tasks, which are executed in sequence, form a *performance critical region*, also called computational kernel.
It is important to distinguish the following types of performance:

Machine performance: This metric refers to the amount of work that our computer architecture performs within a given time unit on a given set of resources (e.g. number of nodes). The work here is defined in terms of hardware related units. An example is the number floating-point operations which the machine is able to execute per second while running a given computational task.

Algorithmic performance: The algorithmic performance quantifies the number of steps per computational task in algorithmic terms.In case of the Conjugate Gradient algorithm or other Krylov-space based iterative methods the amount work can for instance be measured in terms of number of iterations required to obtain a solution within a given precision.

[4] See [Wienke et al.(2013)] for an attempt to take costs of application development into account.

The performance of an application for a given problem size depends on both, the machine as well as the algorithmic performance. The application developer is faced with the challenge to decide on the most promising strategy to increase the performance of an application, i.e. reduce time-to-solution, as machine performance may deteriorate when using a different algorithm with better algorithmic performance and vice versa. The better choice may also be problem size dependent.

While we focus here on measuring machine performance in terms of time-to-solution, it should be noted that for more detailed performance analysis not only the number of clock cycles spent in a certain task or subtask but also other counters, which, e.g., count the number of floating-point operations or load/store operations, can provide important insight. Modern processors comprise performance monitoring units that enable various hardware events to be counted (see, e.g., [Terpstra et al.(2010)] for more details).

As implementing algorithms just to measure time-to-solution is often not affordable, performance models can become important as they allow to make predictions of machine performance. Performance models also allow to understand how the observed performance at system level relates to performance at device level and how, e.g., the performance of a transport device like the network impacts overall performance of an application.

In the previous section we introduced a simple model where we represented a computer architecture by a graph where the nodes and vertices are storage devices and transport or processing devices, respectively. Each task k which is executed by this computer architecture implies that information is transferred from one storage device x to another storage device y. The amount of information is given by the *information exchange* $I^k_{x,y}(W)$, which is a function of the problem size W [Bilardi et al.(2005)].

Let us consider a practical example on how to determine the information exchange where an array a is copied on an array b, i.e. $b_i \leftarrow a_i$. Both arrays comprise N double-precision floating-point numbers of size 64 bit. When executing this task, N elements of a have to be loaded from memory to register file and subsequently N elements of b have to be transferred back to memory. The problem size can thus be parametrized by the input parameter N. In this example we find

$$I^{\mathrm{cp}}_{\mathrm{mem,rf}}(N) = I^{\mathrm{cp}}_{\mathrm{rf,mem}}(N) = 8N\,\mathrm{Byte}. \tag{1}$$

When loading N array elements from memory a certain amount of time will elapse between the event "load instruction issued" until the first data arrives. This time is called *start-up latency*. Once the first element of array a arrives at the register file, the next data will follow. This behaviour can in first approximation be parametrized by the following linear ansatz:

$$\Delta t_{\mathrm{cp}}(N) = 2\left(\lambda_{\mathrm{mem}} + \frac{I^{\mathrm{cp}}(N)}{\beta_{\mathrm{mem}}}\right), \tag{2}$$

where λ_{mem} and β_{mem} are the start-up latency and *asymptotic bandwidth*, respectively. The factor 2 accounts for data being read and written.[5] The asymptotic bandwidth should be distinguished from the effective bandwidth which in our case is

$$b_{\mathrm{mem}}(N) = \frac{2\,I^{\mathrm{cp}}(N)}{\Delta t_{\mathrm{cp}}} = \left(\frac{\lambda_{\mathrm{mem}}}{I^{\mathrm{cp}}(N)} + \frac{1}{\beta_{\mathrm{mem}}}\right)^{-1}, \tag{3}$$

[5]Here we assume start-up latency and bandwidth for memory read and write operations to be the same. This is not alway a valid assumption.

i.e. $b_{\mathrm{mem}}(N) \leq \beta_{\mathrm{mem}}$. The effective bandwidth depends, unlike the asymptotic bandwidth, on the problem size (here N). For small problem sizes the effective bandwidth strongly depends on the start-up latency, while for large N the effective bandwidth approaches the asymptotic bandwidth:

$$\lim_{N\to\infty} b_{\mathrm{mem}}(N) = \beta_{\mathrm{mem}}. \tag{4}$$

The asymptotic bandwidth may be put in relation with the *nominal bandwidth* of a given transport device, here the nominal memory bandwidth B_{mem}. In the simple processor architecture considered here (with, e.g., caches being absent) we have $\beta_{\mathrm{mem}} \leq B_{\mathrm{mem}}$.

Despite the fact that the underlying hardware mechanisms in modern processors are very complex (as we will see later), this simple *latency-bandwidth model* describes the measured timings often well (or at least sufficiently well).[6] Once start-up latency and asymptotic bandwidth have been determined for a given hardware architecture, then the knowledge of the information exchange function, which depends on the computational task of interest and the problem size, allows for a prediction of the execution time based on these simple hardware and performance models.

As a slightly more complex example let us consider the scalar product $c \leftarrow \sum_i a_i\, b_i$. Here N elements of a and b have to be loaded from memory, multiplied with each other and added to the scalar variable c. The latter variable is assumed to be kept in a register, i.e. it does not have to be transferred before writing the final result, an operation which we will ignore as we assume $N \gg 1$. We thus have two information exchange functions describing the memory-to-register file transfer and the processing of the data in the pipeline for fused multiply-add operations:

$$I^{\mathrm{sp}}_{\mathrm{mem,rf}}(N) = 16N\,\mathrm{Byte}, \tag{5}$$

$$I^{\mathrm{sp}}_{\mathrm{rf,rf}}(N) = 2N\,\mathrm{Flop}. \tag{6}$$

Like for the copy operation, the information exchange functions may be used to predict the time needed for loading and processing data, Δt_{mem} and Δt_{proc}, based on the latency-bandwidth model. Making the (realistic) assumption that loading and processing of data can be overlapped, the total execution time for this task can be estimated as $\Delta t_{\mathrm{sp}} = \max(\Delta t_{\mathrm{mem}}, \Delta t_{\mathrm{proc}})$.

For different system hardware architectures Δt_{mem} and Δt_{proc} may differ significantly, depending on how the floating-point processing pipeline's throughput B_{fp} relates to the available memory bandwidth B_{mem}. The ratio of required number of arithmetic operations versus the amount of data, which needs to be transferred, is called *arithmetic intensity* [Harris(2005)]. For the scalar product we can express the arithmetic intensity in terms of the already determined information exchange functions:

$$\mathrm{AI}_{\mathrm{sp}} = \frac{I^{\mathrm{sp}}_{\mathrm{rf,rf}}(N)}{I^{\mathrm{sp}}_{\mathrm{mem,rf}}(N)} = \frac{1}{8}\frac{\mathrm{Flop}}{\mathrm{Byte}}. \tag{7}$$

For the system hardware architecture being balanced, i.e. both memory interface and processing pipeline utilization being approximately equal, we would like to have[7]

$$\frac{B_{\mathrm{fp}}}{B_{\mathrm{mem}}} \simeq \frac{1}{8}\frac{\mathrm{Flop/s}}{\mathrm{Byte/s}}. \tag{8}$$

[6] In case of more complex network protocols, where depending on the amount of data to be communicated different protocols are used, more sophisticated models may be required, like the LogP model [Culler et al.(1993)].

[7] Using the asymptotic bandwidth or throughput (here: β_{mem} and β_{fp}) may be a better choice than the nominal bandwidth or throughput (here: B_{mem} and B_{fp}). While nominal bandwidth figures can be extracted from datasheets, estimates of asymptotic bandwidth are typically more difficult to obtain and may require, e.g., execution of micro-benchmarks.

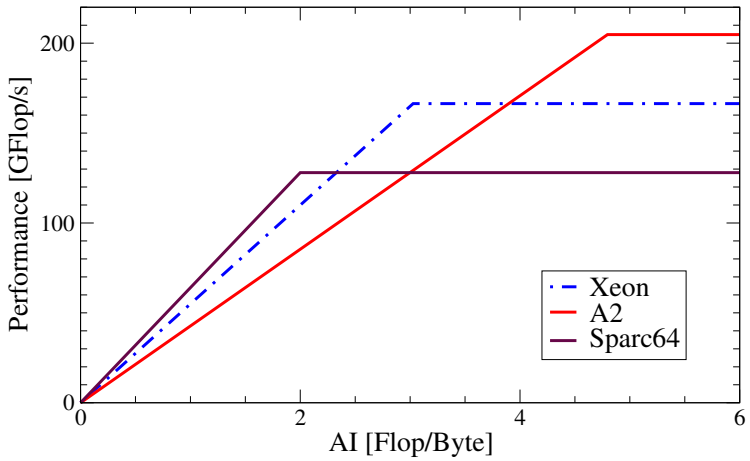

Fig. 3: *Roofline model for the processors listed in Table 1. For small AI (here: AI$\lesssim$ 2) the maximum attainable performance is limited by memory bandwidth, while for large AI (here: AI$\gtrsim$ 4) the performance is limited by floating-point operation throughput. It should be noted that for this (worst case) analysis the existence of caches has been ignored.*

A different look on arithmetic intensity is provided by the roofline model [Williams et al.(2009)]. For small arithmetic intensity AI the floating-point performance is limited by the memory bandwidth. The performance limit increases with larger AI until the point is reached where the throughput of the processing pipeline(s) becomes the limiting factor. In Fig. 3 we show the roofline for different modern server processors. As can easily be seen, a scalar product of two arrays of double-precision floating-point numbers with $\mathrm{AI} = 1/8$ would be strongly bandwidth limited on these processor architectures. Relative to the peak floating-point performance processing capabilities we can not expect to see a sustained performance of more than 6.3 %, even on the for this task presumably most suitable processor, the Sparc64 VIIIfx.

4 Processor Core Architectures

When introducing the Von Neumann architecture we encountered a single central processing unit. Modern processors comprise multiple such units, which we in the following call *processor cores*. Here the instructions of an application are executed. Over the last decades significant advances have been made optimizing instruction throughput, i.e. the number of instructions per cycle (IPC). While logically instructions are executed sequentially, in the order as they appear in the program, at hardware level they may be executed in parallel and/or out-of-order.
All processors, whose hardware parameters are listed in Table 1, comprise cores with multiple execution pipelines and are capable of processing multiple instructions in parallel. Processor cores which implement such instruction-level parallelism (ILP) , and thus allow for $\mathrm{IPC} > 1$, are called superscalar. The number of execution units can vary significantly between different processor core architectures as can be seen from Table 1. The Sparc64 VIIIfx core architecture [Maruyama et al.(2010)] comprises a relatively large number of 8 processing pipelines: 2 for integer instructions, 2 for load/store instructions and 4 for floating-point instructions. The hardware, which decides when which instruction is executed, has to ensure the correctness of instruction execution.
With multiple pipelines being available, it is possible that the order in which instructions are

	Xeon E5-2650	Blue Gene/Q A2	Sparc64 VIIIfx
Core clock speed [GHz]	2.6	1.6	2
Floating-point peak B_{fp} [GFlop/s]	166.4	204.8	128
ISA	x86-64	Power	Sparc-V9
SIMD ISA/width (bits)	AVX/256	QPX/256	HPC-ACE/128
Number of exec. pipelines	5	2	8
Instruction processing	out-of-order	in-order	out-of-order
SMT	2-way	4-way	–
N_{cores}	8	16+1	8
L1 data cache per core [kiByte]	32	16	32
L1 instr. cache per core [kiByte]	32	16	32
L2 cache [MiByte]	2	32	5
L3 cache [MiByte]	20	–	–
Memory bandwidth B_{mem} [GByte/s]	51.2	42.7	64
Memory capacity C_{mem} [GiByte]	O(10-100)	16	16

Table 1: *Hardware parameters of processors used in today's HPC systems: Xeon E5-2650 (Sandybridge) [Intel(2013)] from Intel, Blue Gene/Q A2 [Aho et al.(2013)] from IBM, Sparc64 VIIIfx [Maruyama et al.(2010)] from Fujitsu. The core clock speed refers to the default clock speed. Throughout this lecture we adopt the convention that* $1\,\text{kByte} = 10^3\,\text{Byte}$ *and* $1\,\text{kiByte} = 2^{10}\,\text{Byte} = 1024\,\text{Byte}$. *All 3 processors are programmed using different instruction set architectures (ISA).*

executed differs with respect to program order. This can happen if the execution of instruction i has to be stalled, e.g. because of missing input data, while a later instruction j is ready for execution in a different pipeline. In this way instruction throughput, i.e. average IPC, increases. However, if instructions i and j are to be executed by the same execution unit then this change of order is not possible, at least for processor core architectures where instructions are strictly processed in order. Many processor core architectures meanwhile support out-of-order execution of instructions with different maximum distance at which two instructions can be re-ordered.
Another technique for improving the utilization of processing pipelines is Simultaneous Multi-Threading (SMT). Instead of executing only the instructions of a single software thread, processor core architectures may support concurrent execution of multiple software threads. As the probability increases for at least one of the threads providing instructions ready for execution, the average IPC is likely to increase. Table 1 compares SMT support for different processors. The A2 core used in Blue Gene/Q supports a particular large number of hardware threads to compensate for the lacking support of out-of-order execution of instructions. While SMT may help to increase IPC, it may also lead to increased pressure on resources shared by several threads, e.g. register file.[8] Higher utilization of such potentially scarce resources can have negative impact on performance.
In all of these processors an increasingly often used strategy for enhancing operation throughput has been applied: SIMD (Single Instruction Multiple Data) instructions apply the same operations to multiple data. Let us, e.g., consider a double-precision floating-point fused multiply-add

[8] In the A2 core each thread does have a separate register file.

instruction which implements the operation $d \leftarrow a + b \cdot c$, where a, b, c and d are 64-bit floating-point numbers. A 4-way SIMD version of this instruction would take 3 256-bit input operands and generate a single 256-bit output:

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} \leftarrow \begin{pmatrix} a_0 + b_0 \cdot c_0 \\ a_1 + b_1 \cdot c_1 \\ a_2 + b_2 \cdot c_2 \\ a_3 + b_3 \cdot c_3 \end{pmatrix} \tag{9}$$

While these SIMD instructions enable a significant increase of the operation throughput at moderate hardware costs, the utilization of such hardware capabilities may be challenging. Although the ability of compilers to perform automatic vectorization is improving, often the user has to take care of explicit SIMD programming[9] or at least provide a suitable data layout. SIMD load operations can improve memory access as more data is loaded using a single instruction. On some architectures, however, address alignment requirements, the possible need of gathering data from different memory locations into a single vector and other complicacies may limit the usability of SIMD instructions.

5 Memory Architectures

Within a system hardware architecture the main performance parameters of integrated memory devices are the following: capacity C_{mem}, bandwidth B_{mem} and access latency λ_{mem}. The performance of the currently most widely used memory technology, DDR-SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory), and similar technologies has mainly increased in terms of capacity. Improvements in terms of bandwidth and latency have been very moderate. If external SDRAM based memory devices would be the only memory available to the processor then the performance of almost any scientific application would be severely limited by memory bandwidth and access latencies.
An important empirical observation helps improving processor hardware architectures: Applications tend to reuse data (and instructions) they have used recently. Therefore data localities exist which can be exploited. These localities can be classified in the following way:

Temporal locality: Recently accessed items are likely to be accessed again in the near future.

Spatial locality: Items stored at nearby addresses in memory tend to be referenced close together in time.

As an example let us consider a very simple, 1-dimensional stencil computation:

$$b_i \leftarrow c_- \cdot a_{i-1} + c_+ \cdot a_{i+1}. \tag{10}$$

To compute b_i we have to read 2 elements of array a which are stored at memory addresses which are very close in memory (in this example the distance is 2 times the size of an element of a). When computing b_{i+2} then a_{i+1} is re-used, i.e. the example also exposes temporal locality. Data locality can be exploited to enhance performance by inserting a storage device between register file and main memory which provides high bandwidth and low latency towards the

[9]To support SIMD instruction set architectures, like SSE, AVX, QPX, HPC-ACE, many compilers provide suitable extensions.

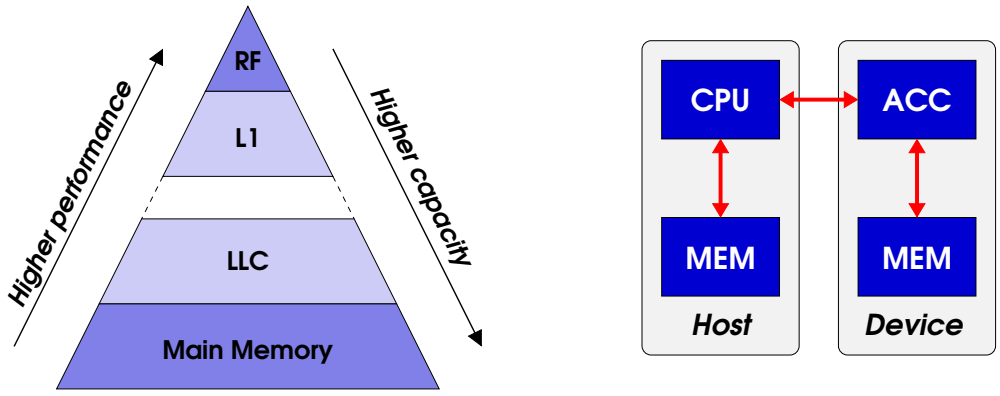

Fig. 4: *A schematic view on a typical processor memory hierarchy is shown in the left figure. On the right a schematic view on an accelerator architecture is shown comprising a host with a single processor as well as a device with a single compute accelerator.*

processor core. High temporal locality means that it becomes beneficial holding a copy of the data in this faster memory device as it is likely that this data is going to be re-used. Transport of data in larger chunks over the memory bus helps to improve bus utilization. High spatial locality increases the probability of all data loaded into the processor to be used, i.e. not only the initially requested data.

This fast memory may either be managed by software or by hardware. The latter is much more common. All processors used in today's HPC architectures comprise even multiple hardware managed caches, i.e. data is moved from memory through 2-4 caches before it arrives at the processing pipelines. The different caches (or more generally speaking: storage devices) are typically arranged in a hierarchical way and thus the term *memory hierarchy* has been established.[10] Such a hierarchy is shown in Fig. 4 (left).

When a load instruction is executed then first the highest cache level, by convention called L1 cache, is checked on whether the requested data is available, i.e. either a cache hit or miss occurs. In the latter case an attempt is made to fetch the data from the next cache level. Only if the *last-level cache* (LLC) does not contain a valid copy of the requested data then data is read from external memory.

The highest cache level provides highest performance, i.e. highest bandwidth and smallest latency, but also the smallest capacity, while the main memory at the bottom of this hierarchy is relatively slow but provides the largest capacity. To make this statement more quantitative: For typical systems used for high-performance computing systems as of today the memory capacity increases from $O(10)$ kiByte at the L1 cache level to $O(10-100)$ GiByte at the main memory level. While it takes $O(1)$ ns to fetch data from the L1 cache, the latency to the main memory directly attached to the processor is $O(100)$ ns. With processor cores running at a core clock speed $2.5 \lesssim f \lesssim 4$ GHz this means that it in the worst case it may take $O(1000)$ clock cycles from the point of time when a load instruction is executed until the data arrives at the processing pipelines.

At the highest, L1 cache level there are typically separate caches for data and instructions to mitigate the effects of the Von Neumann bottleneck, while at lower levels caches are used both

[10] It is, however, important to note that more complex memory architectures play a role in high-performance computing. See, e.g., architectures comprising of accelerator devices (see Fig. 4, right).

for data and instructions. Caches, which are used by a single processor core only, are called private. In processors comprising multiple cores higher cache levels, in particular the LLC are shared. Selected parameters describing the memory hierarchy for server processors used in current HPC systems are shown in Table 1.
Cache misses can be categorized depending on the circumstances under which the cache miss occurs. *Compulsory cache misses* occur when data has to be loaded for the very first time to the cache. The data stays there until it is evicted. This may happen either because the cache became full or because data from another memory address is loaded, which is mapped to the same location in cache. In these cases consecutive *capacity cache misses* or *conflict cache misses* may occur, respectively.
How efficiently the cache hierarchy is used depends on the numerical task, its implementation as well as the problem size. For small problem sizes data reuse might be easier to realise as for larger problem sizes when temporal distance until data is reused becomes too large, i.e. the probability of a data item being evicted from cache before it could be reused becomes large. For a given numerical task and problem size the number of compulsory cache misses cannot be reduced. Changes in implementation and data layout can, however, have significant impact on capacity and conflict cache misses. Strategies on how to optimize for cache utilization will be a topic of later lectures (also see [Kowarschik and Weiß(2003), Hager and Wellein(2010)]).

6 Compute Accelerators

Processor core architectures as presented in an earlier section have become not only very powerful but also complex and costly, e.g. in terms of integrated circuit resources or power consumption. This limits the number of processing units which can be integrated within a single die. For certain compute intensive tasks it may, however, be sufficient to provide very simple compute units. By integrating a very large number of such compute units within a single integrated circuit very large compute performance is achievable. Such devices are often called compute accelerators as they are tightly coupled with a processor and cannot be operated stand-alone. This architectural feature may change in the future.
One may view an architecture comprising processors and compute accelerators as heterogeneous architectures comprising a smaller set of complex, heavy-weight compute cores and a large number of simple, light-weight compute units. While this concept is likely to play an even more important role in high-performance computing in the future, the way how these different compute units are integrated may change quite significantly.
In such heterogeneous architectures the memory hierarchy is (and will be) typically more complex than discussed so far. In Fig. 4 (right) we show a typical node architecture comprising of a host part which corresponds to the architectures discussed earlier: a processor with attached external memory and typically several levels of caches, which have not been shown here. Additionally there is a similar looking device part comprising an accelerator device with attached external memory. Accelerator devices do also comprise multiple levels of cache. Data which needs to be moved from host to device memory (or vice versa) has to be transferred over 2 memory buses as well as a link connecting processor and accelerator. Today this link is typically based on a technology called PCI Express (PCIe) with a link bandwidth $B_{\mathrm{link}} \simeq 8-16$ GByte/s. If we compare this number to the memory bandwidth numbers listed in Tables 1 and 2 then we observe B_{link} to be about 10-50 times smaller than B_{mem}, i.e. special care needs to be taken in order to minimize data transfer over this link.

	K20	K40	Xeon Phi 7120
Vendor	NVIDIA	NVIDIA	Intel
Architecture codename	Kepler	Kepler	Knights Corner
Core clock speed [GHz]	0.71	0.74	1.24
Number of cores/SMs	13	15	61
DP floating-point peak $B_{\mathrm{fp,DP}}$ [TFlop/s]	1.2	1.4	1.2
SP floating-point peak $B_{\mathrm{fp,SP}}$ [TFlop/s]	3.5	4.3	2.4
Memory bandwidth B_{mem} [GByte/s]	208	288	352
PCIe link bandwidth B_{link} [GByte/s]	8	15.7	8

Table 2: *Hardware parameters of selected high-end compute accelerators.*

Today the most widely used type of compute accelerators are GPUs. More details on GPU architectures and on how to program them will be presented in a later lecture. The architecture of these accelerators, which had been originally designed and are still mainly used for graphics processing, differs significantly from processor architectures discussed so far. The entities which may be considered to be the pendants of processor cores are called *streaming multiprocessors (SM)* which are capable of performing significantly more operations in parallel compared to processor cores. Each SM of the GPUs listed in Table 2 can perform 384 single-precision floating-point operations per clock cycle. By means of this huge amount of parallelism these devices can provide an order of magnitude more compute performance compared to processors, although they operate at a significantly lower clock speed. Whether this high compute capability can be efficiently exploited depends, however, strongly on the kind of computational task. For instance, it should be implementable in such a way that a very large number of threads can execute a significant number of instructions independently of each other.
A different strategy is pursued in Intel's Many Core Architecture (MIC) which has been implemented in Intel's Xeon Phi devices. Here a large number of about 60 standard, less powerful processor cores is used. Each of the cores is able to execute very wide SIMD instructions such that each core of the Xeon Phi device shown in Table 2 is able to perform 32 single-precision or 16 double-precision floating-point operations per clock cycle.
Table 2 shows a set of hardware performance numbers for some selected accelerator devices. While the GPU and MIC architectures differ significantly, they do have some common features as well as features which distinguish them from standard processors, e.g. those listed in Table 1. The floating-point performance as well as the memory bandwidth is about 5-10 times larger. All devices run at relatively low clock speed, but are able to execute thousands of floating-point operations per clock cycle. The bandwidth of the link connecting the accelerator device with the host processor is similar and small compared to the memory bandwidth.

7 Network Architectures

As we have seen in the previous section, using compute accelerators a single compute node can provide floating-point processing capabilities at the TFlop/s scale. For high-end computing systems with compute capabilities in the PFlop/s range thousands of nodes have to be integrated in a fast network.

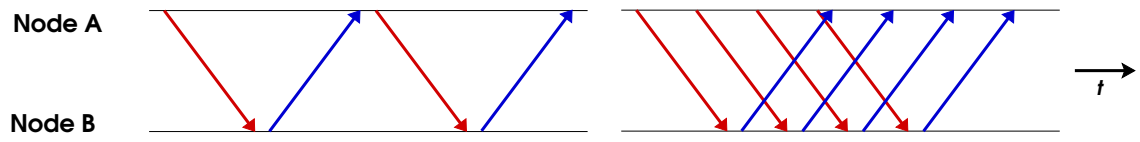

Fig. 5: *The left figure shows a schematic time diagram for a single series of ping-pong communications. In the right figure multiple such series are overlapped to hide communication latency and improve link utilization.*

All state-of-the-art networks are based on point-to-point links which connect compute nodes and/or network devices like switches. Messages which are sent from one node thus usually have to traverse multiple links to reach the receiving node. The most important parameters which describe the performance of such links are bandwidth and latency. While bandwidth continues to improve, further reduction of latency is much more difficult to achieve. Today typical bandwidth performance is $B_{\rm link} = O(1)$ GByte/s, while latency stagnates at $\lambda_{\rm link} = O(100)$ ns.The end-to-end latency of communication at user level is usually $> 1\,\mu$s (frequently even $\gtrsim 10\,\mu$s) because of time needed to communicate between processor and network device plus software overheads.
In terms of core clock cycles the time required to complete a single data transport from one node to the other thus becomes huge. For computational tasks involving significant amounts of small data transfers, i.e. transfers for which execution time is dominated by start-up latency, application developers have to take care of this network architecture limitation. A common strategy, which can be applied for many cases involving fine-grained data transfer, is latency hiding through pipelining. Let us consider a simple pattern of communication between 2 nodes A and B called ping-pong. Node A starts sending a message to B, once this message has been received B sends a message to A, and so on. A schematic version of the time diagram is shown in Fig. 5 (left). Following a pipelining principle the network link connecting the nodes can be used in a much more efficient way if multiple ping-pong communication patterns can be overlapped. As indicated in Fig. 5 (right) a larger number of communications can be performed within the same amount of time in this case. Instead of overlapping one communication operation with other communication operations one can also try to overlap the time between sending and receiving data by performing other work, e.g. computations. The described strategy is commonly called *latency hiding*.
Another key parameter is the network topology. For different communication patterns a given network topology may be more or less suitable. In this lecture we limit us to the following aspects:

Network diameter: The diameter is the maximum distance between 2 nodes. The larger the diameter the more hops may be required to perform a point-to-point communication between nodes and the start-up latency for each communication will thus increase. This also affects the performance of collective operations as a small amount of data has to be collected from all nodes (e.g., in case of a global sum) and/or distributed to all nodes (e.g., in case of a broadcast operation).

Bi-section bandwidth: The bi-section bandwidth refers to the aggregate bandwidth of all links connecting two equal halves of the machine. This metric is important for all-to-all com-

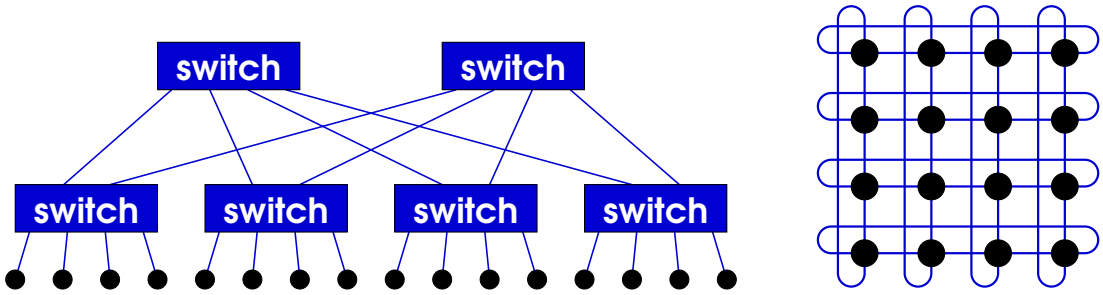

Fig. 6: *Commonly used network topologies: fat-tree (left) and torus (right). The filled circles refer to nodes. The networks shown here are for illustration. In existing systems fat-tree networks are realised using switches with a larger number of ports and torus networks have higher-dimensions.*

munication patters where large amounts of data is exchanged. In this case all nodes communicate with all other nodes and therefore all nodes of one half of the system have to communicate with all nodes of the other half of the system.

A fat-tree topology (see Fig. 6, left) is frequently used for Infiniband-based networks in small and medium size HPC systems. The leave nodes of the tree are the compute nodes, the other nodes are switches. While there is typically only a single path connecting a compute node to the nearest switch, there are usually multiple data paths connecting switches in order to increase the bandwidth at the higher levels, i.e. to fatten the tree. The network diameter as a function of the number of nodes N_{node} scales $\propto \log(N_{\mathrm{node}})$. The network topology is suitable for those collective communication operations which are typically implemented using binary trees, for instance global sums. For fat-tree networks it is difficult to achieve a high bi-section bandwidth when the number of nodes becomes large. For a fixed number of upstream and downstream ports per switch the bi-section bandwidth is independent of the number of nodes.
On high-end systems often d-dimensional torus networks are used with $3 \leq d \leq 5$ (see Fig. 6, right). Compared to fat-tree networks the network diameter may become large for large number of nodes as it scales $\propto N_{\mathrm{node}}^{1/d}$. Note, however, that this scaling argument only holds for very large N_{node}, while for realistic system sizes and large d, latencies are often sufficiently small. The same argument also holds for collective communication patterns, where time needed to perform, e.g., a global sum in the worst case scales $\propto d \cdot N_{\mathrm{node}}^{1/d}$. For bi-section bandwidth a good scaling $\propto N_{\mathrm{node}}^{(d-1)/d}$ is found, i.e. a scaling close to linear for high-dimensional torus networks.
However, for large d the number of network ports per node and total number of network cables becomes large. Furthermore, the distance between nodes, which within the network are nearest-neighbours, may become physically very large for big systems. As this has significant impact on the costs alternatives are considered. For instance, the TOFU network [Ajima et al.(2011)] with its 3-dimensional multi-rail torus structure features a smaller diameter and higher bi-section bandwidth than a standard 3-dimensional network with the same number of nodes. A completely different approach is taken in case of dragonfly topologies (see, e.g., [Faanes et al.(2012a)]). Dragonfly networks are multi-level networks providing an all-to-all connectivity at each level. This network topology features very small network diameters even for very large number of nodes at the expense of bi-section bandwidth scalability.
For any topology and very large number of nodes the time needed for collective communication

operations like broadcast, reduction operations as well as scatter and gather operations may have significant impact on the overall performance of the application. For this reason network designers started to provide hardware support for such communication patterns. For instance, integration of floating-point pipelines in network devices allows to avoid moving data to the processor cores just for performing a small number of arithmetic operations.

8 Power and Energy Efficiency Challenge

While compute performance continues to increase also power consumption is increasing. The power consumed by today's high-end HPC systems already exceeds 10 MWatt (see, e.g., machines listed in Table 3). As a consequence operational costs start to approach the limit of what is considered to be affordable, i.e. 20-30 MWatt. For this reason power and, more precisely, energy efficiency has become a very important topic in high-performance computing. This means that not only time-to-solution Δt but also energy-to-solution ΔE becomes relevant.
To keep operational costs in terms of expenses for electricity low, we would like to minimize energy-to-solution for a given workload W, i.e. the energy efficiency

$$\epsilon_{\mathrm{E}} = \frac{W}{\Delta E} \tag{11}$$

should be maximized. It is important to distinguish energy and power efficiency

$$\epsilon_{\mathrm{P}} = \frac{dW/dt}{P}. \tag{12}$$

For technical reasons and to keep costs of a system low there is an upper limit for the maximum power $P_{\max}$ consumed by a system (or individual system components). Energy-to-solution ΔE, time-to-solution $\Delta t = t_{\mathrm{end}} - t_{\mathrm{start}}$ and power consumption P are related by the equation

$$\Delta E = \int_{t_{\mathrm{start}}}^{t_{\mathrm{end}}} P(t)\, dt \leq P_{\max} \Delta t. \tag{13}$$

High power efficiency means that high compute performance can be provided within a small power envelope. High compute performance may result in small time-to-solution. Therefore, energy-to-solution may be small even if power consumption is high.
If the application is dominated by floating-point operations it becomes natural to quantify the workload in terms of number of floating-point operations N_{fp}. If power consumption is roughly constant we find

$$\epsilon_{\mathrm{E}} = \frac{N_{\mathrm{fp}}}{\Delta E} \simeq \frac{b_{\mathrm{fp}}\, \Delta t}{P\, \Delta t} = \epsilon_{\mathrm{P}}, \tag{14}$$

where $b_{\mathrm{fp}} = N_{\mathrm{fp}}/\Delta t$ is the average throughput of floating-point operations.
One way of comparing power efficiency has been established by the Green500 project.[11] This project ranks the most powerful HPC systems, i.e. systems which are also listed by the Top500 project, in terms of power efficiency $\epsilon_{\mathrm{P}} = b_{\mathrm{fp}}/P$ while executing the HPL benchmark. As of November 2013 an efficiency $\epsilon_{\mathrm{P}} = 4.5$ GFlop/s per Watt could be reached.[12]

[11] `http://www.green500.org`

[12] Note that the HPL benchmark does not foresee W to be kept fixed. System providers are rather allowed to tune W to optimize the resulting performance and power efficiency.

A focus on power efficiency in terms of floating-point operation throughput per Watt instead of energy efficiency can, however, lead to wrong conclusions. In [Bekas and Curioni(2010)] different algorithms for solving a linear system using Iterative Refinement methods have been investigated on a Blue Gene/P system. For a Cholesky-based version a high power efficiency was observed while for a version based on the Conjugate Gradient algorithm time- and energy-to-solution were found to be 5 and 150 times smaller, respectively.
The main reason for the huge difference in terms of energy-to-solution is the difference in energy costs for data processing on the one hand and data transport on the other hand. With today's technology the execution of a double-precision fused multiply-add operations costs about 100 pJ [Shalf et al.(2011)]. For today's commonly used SDRAM technology the average costs for reading and writing from and to external main memory are about 10 pJ/bit [Vogelsang(2010)]. Therefore reading all 3 input operands and writing 1 result costs about 2500 pJ. This means that in this example data transport is 25 times more expensive in terms of energy than data processing. This observation is not expected to change in the future. The use of algorithms with higher data locality, as was shown in [Bekas and Curioni(2010)] for a specific example, will become critical as energy efficiency becomes more important.

9 Continuous Increase of Parallelism

In the previous section we have seen that energy efficiency has become a major concern for high-performance computing. Increasing core clock speed tends to lead to reduced energy efficiency. It furthermore makes power distribution and cooling more challenging and thus more expensive. Further increase in performance thus requires further increase in parallelism. In this lecture we encountered the following levels of parallelism:

Instruction-level parallelism (ILP): Compute devices comprise multiple processing pipelines where instructions can be processed in parallel even if the architecture appears to be scalar. It is mainly task of the compiler to exploit ILP.

Data-level parallelism (DLP): DLP is exploited by SIMD instructions where the same instructions are applied to a parallel stream of data. Compilers may be able to detect this parallelism and generate suitable instructions, alternatively application programmers have to explicitly leverage extensions to standard programming languages to enforce the use of SIMD instructions.

Thread-level parallelism (TLP): To use multiple processor cores or, in case of SMT, to better utilize multiple processing pipelines, computing devices may support concurrent execution of multiple threads of instructions. On processors POSIX threads or OpenMP are established programming models for multi-threaded programming.

Process-level parallelism: To further increase parallelism it becomes necessary (or more efficient) to start multiple processes which process data within a private address space. The underlying memory may either be still shared by multiple processes or physically disjunct, e.g. reside on different nodes. Data has in both cases to be exchanged explicitly by sending and receiving messages. The most widely used communication interface is the Message Passing Interface (MPI) (see [MPI Forum(2012)] for the latest version of this standard). It is based on a model where data is moved from the address space of one process to that of another process through cooperative operations on each process.

MPI provides a full abstraction of the underlying hardware architecture which makes applications using MPI very portable. Details of the underlying network architecture do, however, effect the performance. The use of MPI will be topic of a later lecture.

When parallelizing applications over N computing devices (which here may refer to multiple processing pipelines, cores or nodes), one would like to keep the parallel speedup $S(N)$ close to the N. The *parallel speedup* is defined as

$$S(N) = \frac{\Delta t(1)}{\Delta t(N)}, \tag{15}$$

where $\Delta t(N)$ is time-to-solution using N computational devices. It is convenient to consider the *parallel efficiency* which is the parallel speedup weighted by a factor $1/N$ to account for the N-fold increase of computational resources:

$$\epsilon_{\rm p} = \frac{S(N)}{N}. \tag{16}$$

As parallel efficiency $\epsilon_{\rm p}$ is crucial for scaling to a very large number of nodes it is important to understand the limiting factors. Even if parallelization overheads could be completely removed, $\epsilon_{\rm p}$ might be small due to the fraction of a task which cannot be parallelized. Let us assume that a fraction $f > 0$ of a given task can be parallelized. The serial execution time can then be written as $\Delta t(1) = \Delta t_s + \Delta t_p$, where $\Delta t_s = (1-f)\Delta t$ and $\Delta t_p = f\Delta t$ refer to the execution time of the part of the task which cannot and can be executed in parallel, respectively. Neglecting any overheads due to parallelization, Δt_p reduces by a factor N when executed on N processing devices in parallel. With this assumption we find $\Delta t(N) = \Delta t_s + \Delta t_p/N$ and therefore parallel speedup becomes

$$S(N) = \frac{(1-f)\Delta t + f\Delta t}{(1-f)\Delta t + f\Delta t/N} = \frac{1}{(1-f) + f/N}. \tag{17}$$

This model analysis is called *Amdahl's law* [Amdahl(1967)]. There is always a fraction of a computational task which cannot be parallelized, i.e. $f < 1$. From Amdahl's law it follows that parallel speedup is limited by $\lim_{N\to\infty} S(N) = (1-f)^{-1}$. Even if a large fraction of a task, which, say, accounts for 90 % of the execution time when executed in serial mode, can be perfectly parallelized, the maximum speedup is 10.

Amdahl's law has been criticized by Gustafson [Gustafson(1988)] as being unrealistic because it assumes a fixed workload. From Amdahl's law it follows how parallel speedup scales with fixed workload, i.e. the so-called *strong scaling* case is considered. Alternatively one may consider *weak scaling* where the workload changes linearly with the number of computational devices. In case of perfect weak scaling, i.e. in the absence of any parallelization overheads, the time Δt_p spent in the parallel section of a computational task would stay constant. Concerning the serial part Gustafson argues, based on empirical observations, that this part is typically independent of the workload.[13] The following scaled parallel speedup is known as *Gustafson's law*:

$$\tilde{S}(N) = \frac{(1-f)\Delta t + Nf\Delta t}{(1-f)\Delta t + f\Delta t} = N - (N-1)(1-f). \tag{18}$$

From Gustafson's law a scaled parallel efficiency $\tilde{S}(N)/N > f$ can be derived. Unlike in the strong scaling case described by Amdahl's law, scaled parallel efficiency does not approach zero for large N.

[13] This empirical basis for this assertion was not very strong and in many numerical tasks this does not hold.

	Sequoia	K Computer	Piz Daint
System architecture	Blue Gene/Q	K Computer	XC30
Vendor	IBM	Fujitsu	CRAY
Top500 (11/2013)	#3	#4	#6
Processor type	Blue Gene/Q A2	Sparc64 VIIIfx	Xeon E5-2650
N_{core}	1 572 864	663 552	42 176
Accelerator type	–	–	K20 GPU
N_{acc}	–	–	5 272
Network topology	5d torus	3d toroidal	dragonfly
Link bandwidth B_{link} [GByte/s]	2	5	4.7-5.25
Floating-point peak B_{fp} [PFlop/s]	20.1	10.6	7.8
Memory capacity C_{mem} [PiByte]	1.5	1.3	0.2
Power [MWatt]	7.9	12.7	2.3

Table 3: *Hardware parameters of selected high-end HPC systems: Blue Gene/Q [Aho et al.(2013)] installation at LLNL (USA), K Computer [Yokokawa(2012)] at RIKEN (Japan), Piz Daint at CSCS (Switzerland). Power refers to the amount of electricity consumed during execution of the HPL benchmark as reported in the Top500 list of November 2013.*

10 Today's Supercomputer Architectures

In this section we discuss a selected set of parallel architectures and existing incarnations which as of today are used for large-scale scientific applications.

IBM Blue Gene/Q [Aho et al.(2013)] is the third generation of a series of architectures which first was introduced in 2004. A unique feature of this architecture is the integration of the network on the processor. In this way very high bandwidth to the network and extremely small latencies can be achieved, which has a strong impact on the scalability properties of the architecture. As can be seen from Table 1 the number of cores per processor is large. To efficiently utilize the processing pipelines up to 4 hardware threads may have to be started resulting in up to 64 threads per processor. A separate, 17th core is used just for running a slim operating system to minimize interference of computational task execution and asynchronous management of system services. The nodes are interconnected in a 5-dimensional torus using 10 out of 11 available network ports per processor. Since the Blue Gene/Q architecture has become generally available in 2012 O(10) systems have been installed, including an installation at Jülich Supercomputing Centre, called JUQUEEN, comprising 458 752 cores. In Table 3 we show selected hardware parameters of the largest existing Blue Gene/Q installation at Lawrence Livermore National Laboratory (LLNL, USA).

Also the K-Computer [Yokokawa(2012)] developed by Fujitsu together with the Japanese research lab RIKEN is based on a processor which has been specifically designed for high-end HPC systems. Details of the processor and the only existing installation of the architecture are listed in Table 1 and 3, respectively. The processors are connected to a custom network chip which is part of the TOFU network [Ajima et al.(2011)] (see section 7 for more details). Compared to Blue Gene/Q the architecture provides relatively complex processor cores with more features, higher memory bandwidth and larger memory capacity. This makes the system easier

to use for a larger range of applications. The price is a significantly lower power efficiency.
The Cray XC30 system [Faanes et al.(2012b)] is unlike the other systems based on a commodity processor. These are interconnected by a special high-performance network using a dragonfly topology. The parameters of a Cray XC30 system at the Swiss supercomputing centre CSCS can be found in Table 3. The number of nodes is relatively small as each of the nodes provides significantly more compute performance thanks to GPU-based compute accelerators. Because of these accelerators the power consumption when executing the HPL benchmark is significantly smaller compared to the two other systems.

11 Outlook

As of today parallel computer architectures are capable of providing $O(10)$ PFlop/s compute performance. Around 2017 and after 2020 it is expected that with next generation of parallel computer architectures systems will become available with a peak compute performance of $O(100)$ PFlop/s and $O(1)$ EFlop/s, respectively. This will, however, only be possible by yet another significant increase of parallelism. With $O(10^7)$ threads on today's machines the amount of concurrency which has to be exposed by the application is already very high. It will require significant efforts from developers of algorithms and applications to leverage the increased performance of future parallel computer architectures. While challenging in use, high-end HPC systems open significant opportunities for scientific research and the creation of new scientific knowledge and results.
To learn more about computer architectures in general and the use of high-performance computing systems for scientific applications we recommend [Hennessy and Patterson(2011)] and [Hager and Wellein(2010)] for further reading.

Acknowledgements

The author would like to thank Hubert Simma (DESY, Germany) for many valuable discussions and ideas which found their way into these lecture notes but have not been published elsewhere.

References

[Neumann(1945)] J. v. Neumann, Tech. Rep. (1945).

[Moore(2006)] G. E. Moore, Solid-State Circuits Society Newsletter, IEEE **11**, 33 (2006), ISSN 1098-4232.

[Bohr(2007)] M. Bohr, Solid-State Circuits Society Newsletter, IEEE **12**, 11 (2007), ISSN 1098-4232.

[Wienke et al.(2013)] S. Wienke et al., in *Supercomputing*, edited by J. Kunkel, T. Ludwig, and H. Meuer (Springer Berlin Heidelberg, 2013), vol. 7905 of *Lecture Notes in Computer Science*, pp. 330–342, ISBN 978-3-642-38749-4, URL http://dx.doi.org/10.1007/978-3-642-38750-0_25.

[Terpstra et al.(2010)] D. Terpstra et al., in *Tools for High Performance Computing 2009*, edited by M. S. Mller, M. M. Resch, A. Schulz, and W. E. Nagel (Springer Berlin Heidelberg, 2010), pp. 157–173, ISBN 978-3-642-11260-7, URL http://dx.doi.org/10.1007/978-3-642-11261-4_11.

[Bilardi et al.(2005)] G. Bilardi et al., in *Proceedings of the 12th International Conference on High Performance Computing* (Springer-Verlag, Berlin, Heidelberg, 2005), HiPC'05, pp. 386–397, ISBN 3-540-30936-5, 978-3-540-30936-9, URL http://dx.doi.org/10.1007/11602569_41.

[Culler et al.(1993)] D. Culler et al., in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (ACM, New York, NY, USA, 1993), PPOPP '93, pp. 1–12, ISBN 0-89791-589-5, URL http://doi.acm.org/10.1145/155332.155333.

[Harris(2005)] M. Harris, in *ACM SIGGRAPH 2005 Courses* (ACM, New York, NY, USA, 2005), SIGGRAPH '05, URL http://doi.acm.org/10.1145/1198555.1198768.

[Williams et al.(2009)] S. Williams et al., Commun. ACM **52**, 65 (2009), ISSN 0001-0782, URL http://doi.acm.org/10.1145/1498765.1498785.

[Intel(2013)] Intel, *ARK product specifications* (2013), URL http://ark.intel.com.

[Aho et al.(2013)] M. Aho et al., IBM Journal of Research and Development **57**, 0:1 (2013), ISSN 0018-8646.

[Maruyama et al.(2010)] T. Maruyama et al., IEEE Micro **30**, 30 (2010), ISSN 0272-1732, URL http://dx.doi.org/10.1109/MM.2010.40.

[Kowarschik and Weiß(2003)] M. Kowarschik and C. Weiß, in *Algorithms for Memory Hierarchies*, edited by U. Meyer, P. Sanders, and J. Sibeyn (Springer Berlin Heidelberg, 2003), vol. 2625 of *Lecture Notes in Computer Science*, pp. 213–232, ISBN 978-3-540-00883-5, URL http://dx.doi.org/10.1007/3-540-36574-5_10.

[Hager and Wellein(2010)] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers* (CRC Press, Inc., Boca Raton, FL, USA, 2010), 1st ed., ISBN 143981192X, 9781439811924.

[Drepper(2007)] U. Drepper, *What every programmer should know about memory* (2007).

[Ajima et al.(2011)] Y. Ajima et al., in *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on* (2011), pp. 87–94.

[Faanes et al.(2012a)] G. Faanes et al., in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (2012a), pp. 1–9, ISSN 2167-4329.

[Bekas and Curioni(2010)] C. Bekas and A. Curioni, Computer Science - Research and Development **25**, 187 (2010), ISSN 1865-2034, URL http://dx.doi.org/10.1007/s00450-010-0119-z.

[Shalf et al.(2011)] J. Shalf et al., in *High Performance Computing for Computational Science – VECPAR 2010*, edited by J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes (Springer Berlin Heidelberg, 2011), vol. 6449 of *Lecture Notes in Computer Science*, pp. 1–25, ISBN 978-3-642-19327-9, URL http://dx.doi.org/10.1007/978-3-642-19328-6_1.

[Vogelsang(2010)] T. Vogelsang, in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* (2010), pp. 363–374, ISSN 1072-4451.

[MPI Forum(2012)] MPI Forum, *MPI: A message-passing interface standard version 3.0* (2012), URL http://www.mpi-forum.org/docs/mpi-3.0.

[Amdahl(1967)] G. M. Amdahl, in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (ACM, New York, NY, USA, 1967), AFIPS '67 (Spring), pp. 483–485, URL http://doi.acm.org/10.1145/1465482.1465560.

[Gustafson(1988)] J. L. Gustafson, Commun. ACM **31**, 532 (1988), ISSN 0001-0782, URL http://doi.acm.org/10.1145/42411.42415.

[Yokokawa(2012)] M. Yokokawa, in *Networking and Computing (ICNC), 2012 Third International Conference on* (2012), pp. 21–22.

[Faanes et al.(2012b)] G. Faanes et al., in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (IEEE Computer Society Press, Los Alamitos, CA, USA, 2012b), SC '12, pp. 103:1–103:9, ISBN 978-1-4673-0804-5, URL http://dl.acm.org/citation.cfm?id=2388996.2389136.

[Hennessy and Patterson(2011)] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011), 5th ed., ISBN 012383872X, 9780123838728.