An Introduction to HPC

Hinnerk Stüben (UHH/RRZ), Kai Himstedt (HITeC)

07.05.2025

Funded by the European Union

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

HITeC
HAMBURGER INFORMATIK TECHNOLOGIE-CENTER

Funded by the
IFB HAMBURG | Hamburgische Investitions- und Förderbank

Agenda / Major Topics

- What is HPC?
- HPC Cluster Architecture
- File Systems in HPC Clusters
- Performance of HPC Systems
- Levels of Parallelism
- CPU and GPU Computing
- Benchmarking
- Job Scheduling and Workload Managers
- Editing Files on HPC Systems

What is HPC?

- Tautological definition
  - "You are doing HPC when you are using HPC hardware"
- Traditional definition
  - Run (parallel) programs (computer simulations, machine learning, …) as fast as possible
- Performance metrics
  - FLOPS: Floating Point Operations Per Second (also: Flop/s)
  - OPS: [mathematical] Operations Per Second (AI equivalent to FLOPS)
  - Time-to-solution (e.g. runtimes)
  - Power-to-solution (e.g. FLOPS or OPS per Watt)
  - Search operations per second
  - …
- Common denominator
  - Powerful hardware is used (according to current state of the art)
  - HPC cluster: interconnected nodes ("servers/PCs") that are suitable for complex computations

The HPC (High-Performance Computing) Paradox

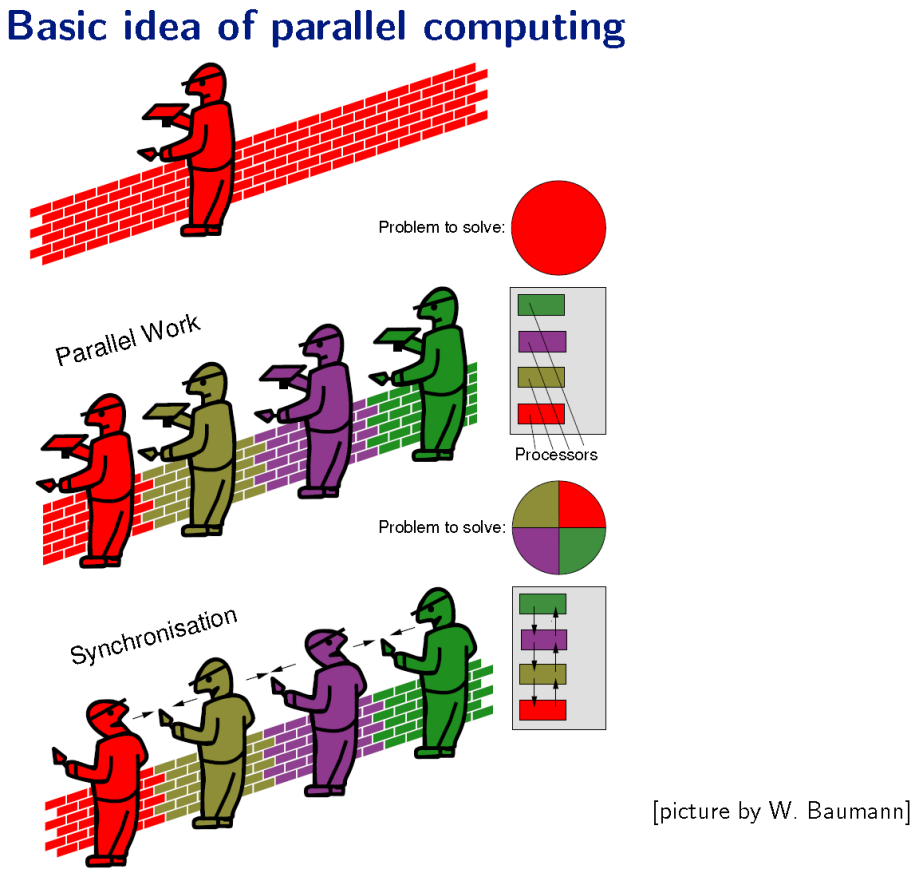
- An HPC cluster is not automatically faster than your PC
- In order to become faster parallel processing techniques must be employed
- I/O (per CPU core) will perform worse than with the SSD/NVMe of your computer
- For I/O intensive tasks usage of the file systems needs planning

HPC Software Environment

- Operating system is typically Linux (e.g. for nearly all TOP500 systems since 2017 (top500.org))
- Batch system based on submitting jobs
  - Workload manager controls where and when which job should run
- Batch job characterization
  - Sequence of Linux operating system commands in batch file to run program
  - User submits batch file for execution via Command Line Interface (CLI)
  - No interaction between user and job
    - Job inputs must be prepared beforehand
  - More jobs are submitted than can be executed immediately
    - Avoids expensive resources from being idle
    - Submitted jobs are queued for later execution
- Interactive use is also supported
  - Nowadays – for example – JupyterHub / JupyterLab / Jupyter Notebooks

Gaining Speedup on HPC Systems

- Applications do not get faster on HPC systems by magic
- Applications have to be parallelized

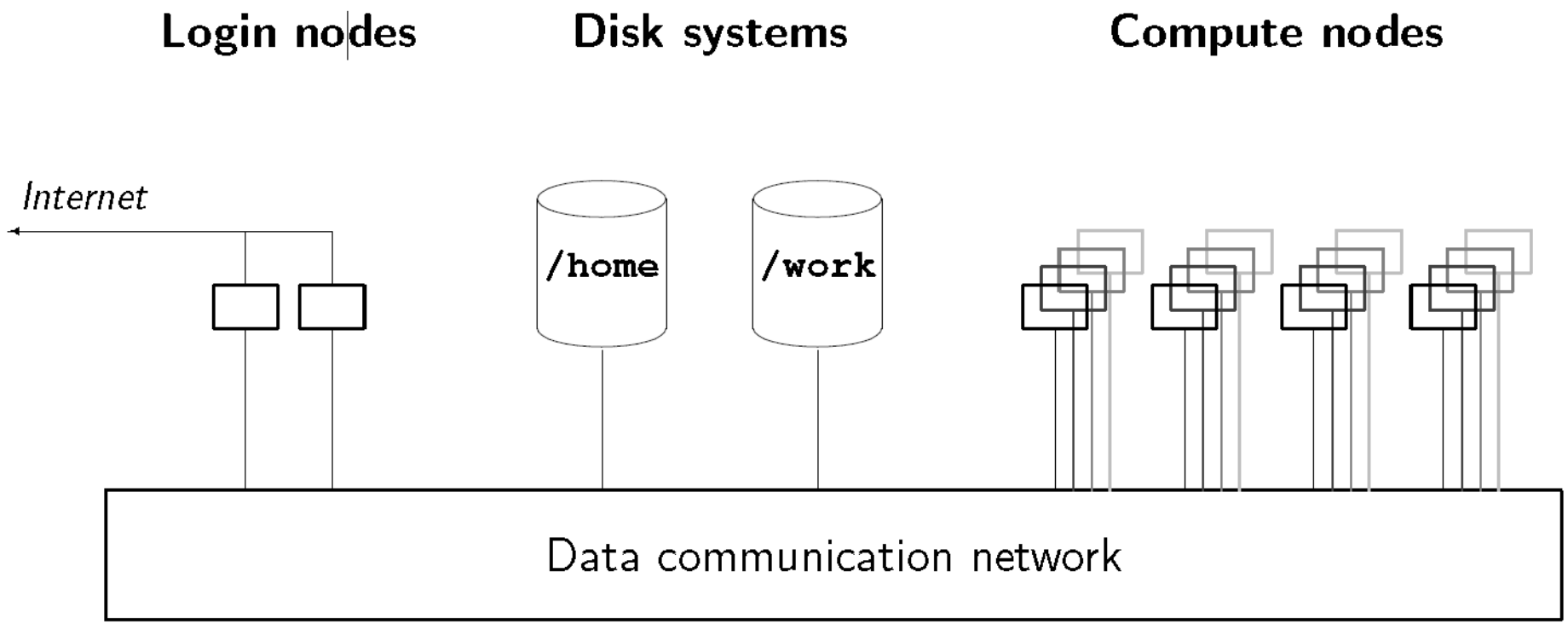


[picture by W. Baumann]

Need for Parallel Processing

- Node level parallelization
  - Multiple CPUs, each with many cores
  - Multiple GPUs, each with many cores
- Cluster level parallelization
  - Multiple nodes that work together, each with potentially multiple CPUs and multiple GPUs
  - Typical for numerical simulation models (e.g. climate models)
  - Nowadays also available for machine learning (e.g. PyTorch, TensorFlow)
- Parallel performance needs to be checked
  - Is runtime (almost) n times shorter using n times as many computing resources?

HPC Cluster Architecture



HPC Cluster Architecture

- What the user sees
  - Login node(s) (also: head node(s))
  - Compute nodes
    - CPU nodes
    - GPU nodes
  - File systems (e.g. /home, /work, /local)
  - Possibly special nodes (e.g. for visualization of results)
- Background functionalities
  - Storage nodes
  - Central user management (e.g. LDAP (Lightweight Directory Access Protocol))
  - …

Parallel Computer Architecture

- Major parallel computer components in HPC clusters
  - Compute units
    - CPUs
    - GPUs
  - Main memory
    - CPU: RAM (Random Access Memory)
    - GPU: VRAM (Video RAM)
  - High speed network
    - E.g. 1 Gbit/s, 10 Gbit/s, …, 400 Gbit/s
- Memory architectures
  - Shared memory at individual node
    - All CPU cores can access shared RAM
    - All GPU cores of a single graphics card can access shared VRAM
  - Distributed memory in the cluster
    - Nodes communicate to coordinate data usage of the local main memories

File Systems in HPC Clusters

- Global file systems, accessible from all nodes
  - Distributed file systems
    - Example: NFS (Network File System), (/home/…)
    - Not allowed: Concurrent writes to same file
    - Primary use: User-specific files and directories, profiles and settings, source code, …
  - Parallel cluster file systems
    - Example: BeeGFS (Bee Grid File System) (/work/…)
    - Allowed: Concurrent writes to same file
    - Provide potentially high I/O bandwidths
    - Primary use: data files, job inputs, job outputs, temporary files, scratch files
- Local file systems, accessible inside node
  - Standard Linux file systems
    - Example: EXT4 (Extended File system 4), (/local/…)
    - Provide potentially very high I/O performance for specific node
    - Primary use: scratch files, temporary files

Performance of HPC Systems

- Floating Point Operations per Second (FLOPS (also: Flop/s)
  - Popular way to measure computational power of HPC systems
  - In the order of several hundred PetaFLOPS (PFLOPS) for top systems 2025
    - $1\ PFLOPS = 1000\ TFLOPS = 10^{15}\ FLOPS$
    - Powerful PC: ca. 1 TFLOPS (1 CPU), ca. 50 TFLOPS (1 GPU)
- TOP 500 list (www.top500.org/lists)
  - Lists the most powerful systems ranked by FLOPS
  - Top system November 2024: El Capitan (USA), ca. 11 million cores (CPU + GPU), 1742 PFLOPS, 29600 kW
  - Based on *Linpack* benchmark
    - Solving dense systems of linear equations is representative for other compute intensive tasks
  - Updated twice a year
  - Shows past and current trends in HPC
    - E.g. many cores architectures, GPU computing

Moore's Law

- Moore's law (1965, revised in 1975) states
  - Complexity of integrated circuits doubles approximately every two years
  - True in the past
  - No more improvements of *sequential* performance since around 2005
    - CPU clock speeds have settled in the range from ca. 3 to 5 GHz
  - Parallel computing became increasingly relevant
    - Many core CPUs
    - Many core GPUs
- Post-Moore era (begun around 2016)
  - Pace of advancements started to slow down
  - Parallel computing still increasingly relevant

Speedup, Efficiency, and Scalability

- Speedup
  - Relation between sequential and parallel runtime of program
  - $S_n = \frac{T_1}{T_n}$
- Where
  - $T_1 = runtime\ using\ a\ single\ processing\ element$ (e. g. 1 core)
  - $T_n = runtime\ using\ n\ processoring\ elements$ (e. g. n cores)
- Ideal case: linear scaling, i.e. linear speedup
  - $S_n = n$
- Linear speedup not achievable in practice
  - Synchronization overheads
    - E.g. for waiting for partial results
  - Communication overheads
    - E.g. for distributing partial tasks and collecting partial results

Speedup, **Efficiency**, and **Scalability**

- Efficiency
  - Relation between speedup and number of processing elements used to achieve the speedup
  - $E_n = \frac{S_n}{n}$
- Scalability
  - Goal: Efficiency remains high when number of processing elements is increased
    - Also called: Good scalability of parallel program
  - Some problems can be parallelized trivially
    - Rendering (independent) computer animation images, …
  - There are problems that are extremely hard to parallelize
    - Some search algorithms, sorting highly dependent data, …
  - Typical problems are somewhere in between these extremes
    - Scientific computing (climate models, …)
    - Machine learning (training of neural networks, …)
    - Standard technique: Domain decomposition (e.g. atmosphere divided into cells, images divided into tiles)

General Challenge in HPC

- When number of processing elements is (steadily) increased the major challenge is to achieve
  - Good speedups
  - Good efficiencies
  - What is “good” will depend on the domain in practice

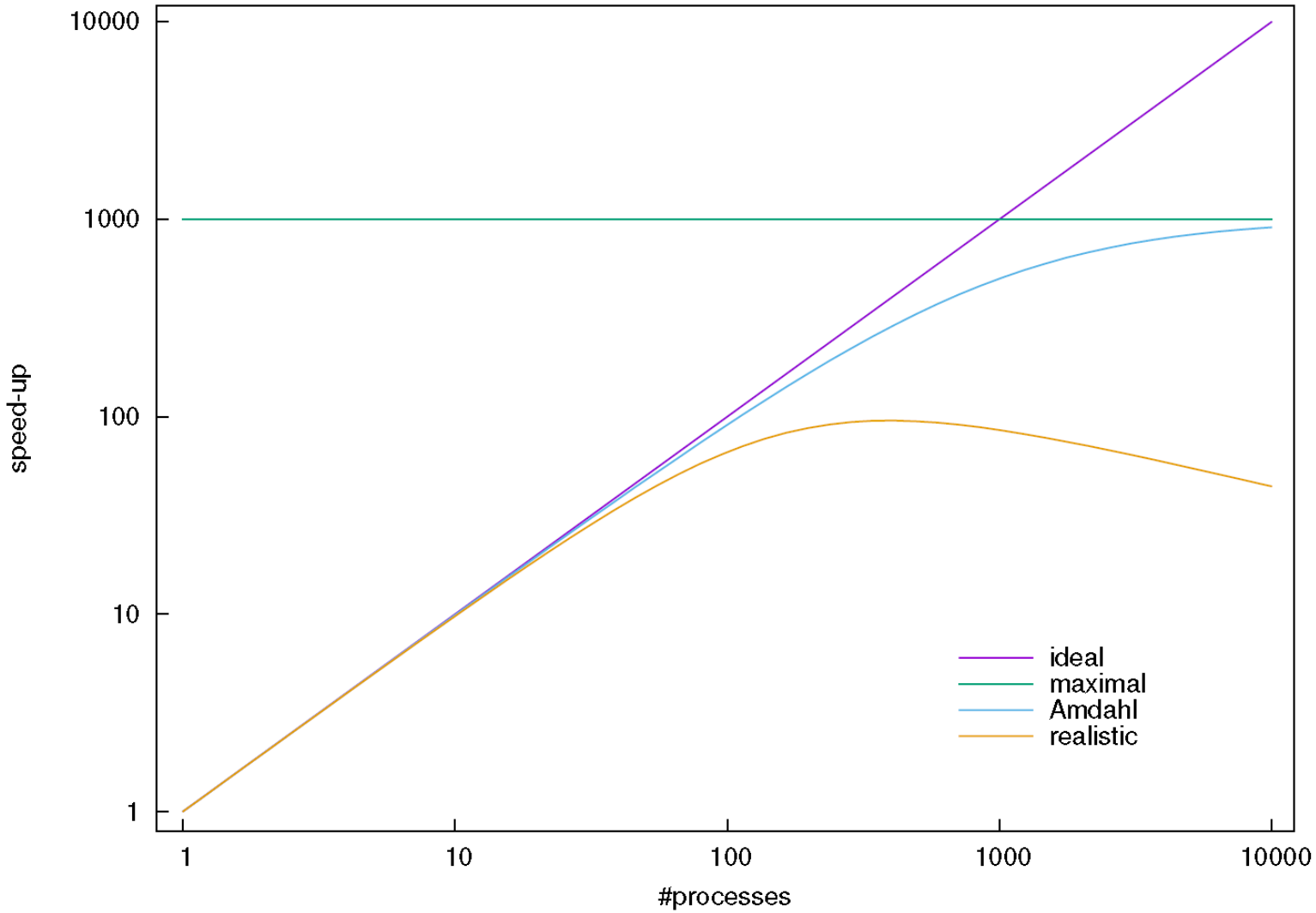
Amdahl's Law

- Amdahl's law (1967) states
- There is an upper limit for the maximum speedup of a parallel program
- Upper limit is determined by sequential, i.e. non parallelizable part
- Sequential parts
  - Initialization and I/O operations
  - More generally: Synchronization and communication overheads

Amdahl's Law: Example

- Example
  - Sequential runtime: $20\ hours$ on a single core
  - Non-parallelizable part: 10% (i.e. $2\ hours$)
  - Total runtime would be at least $2\ hours$
  - Parallelizable part: 90% (i.e. $18\ hours$)
  - Maximum speedup is limited by $\frac{20\ hours}{2\ hours} = 10$

Amdahl's Law: Graphical Representation



Programming Languages for HPC

- Fortran
  - Widely used in scientific and engineering applications for a very long time
  - Performance close to the machine-oriented level with high accuracy in numerical operations
- C
  - Low level of abstraction
  - Performance close to the machine-oriented level
- C++
  - Extends C by object oriented paradigm, i.e. high level of abstraction
  - Performance still close to the machine-oriented level
  - Complex syntax
- Python
  - High level of abstraction (object oriented paradigm) and easy to use for fast prototyping
  - Very good library (package) support, in particular for scientific computing and machine learning
  - Libraries: NumPy, SciPy, CuPy, PyTorch, TensorFlow, …
    - Under the hood: C++, C
    - Some packages also have GPU support
  - In connection with HPC, Python is more of a frontend or wrapper

Levels of Parallelism

- Application level → *trivial* parallelism
  - Independent runs of one or more programs
- Algorithmic level → parallel computing
  - Work is decomposed for multiple processing elements
  - Tasks are divided based on their specific functions which are executed in parallel
- Loop level → compiler
  - SIMD (Single Instruction Multiple Data) vectorization
  - Auto-Parallelization, e.g. using multiple threads
- Machine level → hardware
  - Multiple functional (arithmetic) units
  - Pipelines for loading and executing machine instructions
  - SIMD units
  - …

GPU (Graphics Processing Unit) Computing

- Game programming has driven GPU development
  - Need for more realism trying to make beautiful games (i.e. rendering of 2D and 3D images/scenes)
- Ian Buck (NVIDIA, formerly at Stanford University)
  - Before year 2000: tiny programs running on every pixel
    - Program has only four maybe eight instructions
    - But: A million pixels, sixty times per second
    - Essentially a massively parallel program
  - Since about 2000: Using GPUs to fit a wider class of applications (matrix multiplies, linear algebra)
    - Threading concept similar to CPUs
    - But: Tens of thousands of GPU threads instead of 2, 4, 8 (or even e.g. 128 in 2025) CPU threads
    - CUDA (Compute Unified Device Architecture), released 2007
      - Application Programming Interface (API)
      - If programmer knew C, C++, or Fortran then programming a GPU would not be a great challenge

General Purpose Computing on GPUs

- GPGPU means using GPUs for accelerated general-purpose processing
- Some GPUs no longer even have a connection for a monitor
  - E.g. NVIDIA Tesla V100, Tesla A100
- GPUs are particularly suitable for typical AI/ML workloads, for example
  - Training neural networks
  - Performing inference
- Tasks unsuitable for GPUs
  - Single-threaded tasks
    - Parallel potential can not be exploited
  - Tasks with complex branching logic
    - Parallel execution flow might be disrupted
  - Tasks with an unfavorable data transfer between CPU and GPU
    - Overhead for transferring data to and from GPU can outweigh computational benefits

Overview of Current GPUs From NIVIDIA

	V100	A100	H100	L40 (EDIH Ham.)
Available	June 2017	May 2020	March 2023	October 2022
Architecture	Volta	Ampere	Hopper	Ada Lovelace
VRAM	16 GB	40 GB or 80 GB	80 GB	48 GB
Mem. Bandwidth	900 GB/s	1.6 TB/s	2 TB/s	864 GB/s
FP64	7 TFLOPS	9.7 TFLOPS	26 TFLOPS	1.4 TFLOPS
FP32	14 TFLOPS	19.5 TFLOPS	51 TFLOPS	90 TFLOPS
FP16	112 TFLOPS	312 TFLOPS	1513 TFLOPS	181 TFLOPS
Price (ca.)	7000 €	12000 € (80 GB)	40000 €	9000 €

In comparison: AMD CPU (Zen4) with 64 phys. cores (mem. bandwidth 460 GB/s):

GFLOPS = (Clock in GHz) × (Number of Cores) × (FP32 FLOP per Cycle)

9523 GFLOPS = 3.1 GHz × 64 × 48

High Level GPU Programming

- CUDA vs. OpenACC
  - CUDA: Access to all GPU hardware features
  - OpenACC: Simplifies programming

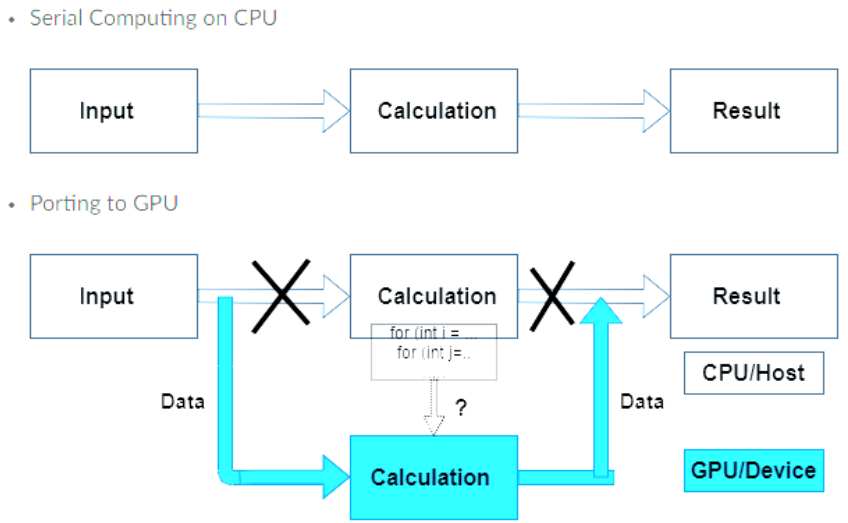
```c
#include <stdio.h>
#include <openacc.h>

#define N 100000

// Function for vector addition
void vector_add(float *a, float *b, float *c, int n) {
    #pragma acc parallel loop
    for(int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Parallelizing a vector addition in C using OpenACC

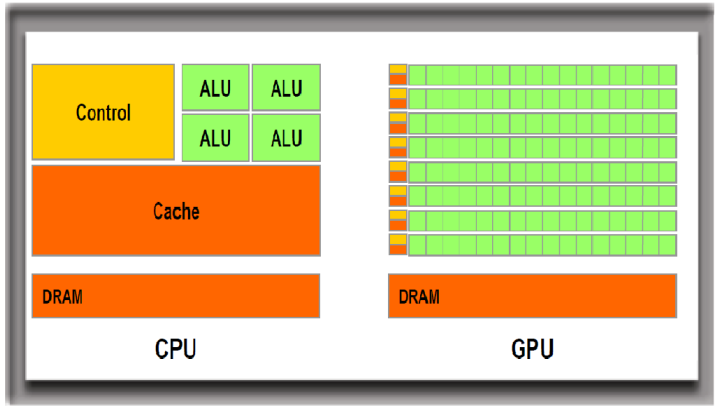
- CPU vs. GPU execution models
  - CPU: Sequential / Serial execution
  - GPU: Parallel execution by offloading of code
    - CPU ("Host") copies data to/from GPU ("Device")



Offloading of code regions from host CPU to GPU
(https://enccs.github.io/OpenACC-CUDA-beginners/1.02_openacc-introduction/)

CPU vs. GPU

	CPU	GPU
Flexibility	General purpose processor	Specialized tasks at massive scale
Arithmetic Accuracy	High precision	Preferably low precision
Cache Memory	Large sets of linear instructions	More shared
Cores	Intel: max. 64 AMD: max 128	NVIDIA H100: 14592 (CUDA FP32)
Parallelism	Few (complex, independent) threads	Thousands of (simple) threads
Data throughput	Low: sequential instruction stream, possibly increased by SIMD / AVX	High: same operation on many data points in parallel
Development	Mature technology reaching limits	Higher potential to improve
Cost	5500 € (AMD 128 cores)	40000 € (NVIDIA H100)



Basic CPU and GPU architectures
(https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/)

Benchmarking

- Definition
  - Determination of hard- or software performance by controlled experiments
  - Benchmark
    - Controlled series of experiments with single program
      - E.g. using 1, 2, and 4 GPUs for training of the same neural network
- Motivation
  - Understanding performance of parallel applications
    - Is there a speedup achieved using more resources?
    - Is the speedup reasonably large in relation to the resources used?
    - What is the scalability of my program?
    - How many resources can be maximally used before efficiency drops to unacceptable values?

Job Scheduling and Workload Managers

- HPC resources can be
  - Shared
    - Login node(s), global file system, network, …
  - Non-shared
    - Compute nodes (at least regarding their cores), GPUs, …
- Job scheduling
  - Process of selecting waiting jobs for execution and allocating resources for them
  - Goals
    - Maximize resource utilization and throughput
    - Minimize waiting time and turnaround time ($waiting\ time\ +\ execution\ time$)
    - Treat users fairly
- Workload managers
  - Implement job scheduling
  - State of the art: SLURM (Simple Linux Utility for Resource Management)

Scheduling Algorithms

- First-Come-First-Served (FCFS)
  - Poor performance but basis for more sophisticated algorithms
- Shortest-Job-First (SJF)
  - Minimizes average waiting time but *starvation* problem
  - Implemented via execution time limits for jobs
- Priority
  - Internal: Job size, time limit, job aging, licenses, …
  - External (per user or group): "deadlines", amount of founds paid for the cluster
- Fair-share
  - Resource utilization proportionate to shares based on job history
- Backfilling
  - Fill nodes with jobs that have lower priority than bigger jobs still waiting for some resources
  - Use already partially allocated resources of bigger job until planned start of bigger job to fill holes

Tasks of a Workload Manager

- Job control
  - Submission
  - Monitoring
  - Cancellation
- Scheduling and resource management
  - Select waiting job for execution
  - Allocate and monitor resources
- Accounting
  - Record resource usage

Popular Workload Managers

- SLURM
  - Simple Linux Utility for Resource Management
  - Includes scheduler
  - Supports a variety of scheduling algorithms
  - Used for Hummel2 Cluster at RRZ and for RCI in EDIH Hamburg
  - Further example: AWS ParallelCluster natively supports Slurm
- TORQUE
  - Terascale Open-source Resource and QUEue Manager
  - Needs additional scheduler component (e.g. Maui or Moab)

Workload Manager SLURM

- Gained much popularity in the recent past
- Open source, GNU General Public License (GPL), Version 2
- Commercial support since 2010
- Command names begin with an ‘s’:
  - `sinfo`
  - `squeue`
  - `sbatch`
  - `salloc`
  - `scancel`
  - …

SLURM Commands `sinfo` and `squeue`

- `sinfo`
  - Shows information on nodes and partitions
- `squeue`
  - Shows information about jobs in the scheduling queue

SLURM Command `sbatch`

- `sbatch [options] [filename]`
  - `options`
    - Resource requirements
      - number of nodes
      - processes per node
      - threads per process
      - number of GPUs
      - queue/partition
      - ...
    - Other job properties
  - `filename`
    - Batch script name
  - As a result
    - *`jobID`* is assigned
    - Job is inserted in the scheduling queue
    - Job will run non-interactively in the background

SLURM Batch Script Example

```
1   #!/bin/bash
2   # script will be executed with the Bash shell
3
4   #SBATCH --nodes=1                # requests 1 node for the job
5   #SBATCH --gpus=L40:2             # requests 2 GPUs of type NVIDIA L40 per node
6   #SBATCH --ntasks-per-node=1      # requests 1 task per node
7   #SBATCH --cpus-per-task=1        # requests 1 CPU (i.e. core) per task
8
9   srun ./gpu_burn 180              # executes program gpu_burn for 180 seconds
```

Interactive Batch Jobs: SLURM Command `salloc`

- `salloc [options]`
  - SLURM grants interactive shell prompt on the allocated resources
    - Otherwise very similar to `sbatch` command
  - `options`
    - Resource requirements
      - number of nodes
      - processes per node
      - threads per process
      - number of GPUs
      - queue/partition
      - ...
    - Other interactive job properties
  - Program to be executed is preceded by `srun`
    - Input and output are directed to terminal session where `salloc` and `srun` were initiated

Three Ways to Use SLURM Command `srun`

- `srun` executed in batch script
  - Launches (parallel) program on allocated resources
- `srun` executed within shell which is granted via `salloc`
  - Launches (parallel) program on allocated resources
- `srun [options]` executed directly within normal login shell
  - Launches (parallel) program on allocated resources
  - Similar to execution within shell which is granted via `salloc`
    - Unlike `salloc`, the allocated resources are immediately released after execution

Editing Files on HPC Systems (Usual Way)

- Remote use of classic Unix/Linux text editors
  - vi, Vim (Vi improved) (available on all Linux systems)
    - Requires a certain learning curve
    - Popular choice among programmers and power users
  - GNU Emacs (Editing MACroS) (actually always available on Linux Systems)
    - Highly customizable text editor
    - More beginner-friendly than vi/Vim
    - X Window support
  - nano (often available on Linux systems)
    - Simple and intuitive
    - More beginner-friendly than Emacs

Editing Files on HPC Systems (Alternatives)

- Local use of text editors
  - Mount and interact with remote file system via SSHFS (Secure SHell File System)
  - Editor of choice can be used to access files as if they were locally stored
  - Drawback: Larger amounts of text may have to be transferred via the underlying ssh connection
- Local use of current IDEs (Integrated Development Environments)
  - VSCode (Visual Studio Code)
    - Powerful code editor, featuring debugging, code completion, Git integration, …
    - Plugins to extend functionality
      - Language support, debuggers, linters, themes, previews (WYSIWYG), …
    - Remote-SSH plugin allows to open remote directories, edit files, run commands as if they were on local machine
      - Server components are automatically installed when connection to remote system is established
  - Further powerful IDEs which support remote editing of files / remote development
    - PyCharm
    - Eclipse
    - IntelliJ IDEA

Kontakt via www.hitec-hamburg.de

End

Backup Slides

Pitfalls of FLOPS

- Other critical performances than the performance of arithmetic units
  - Memory performance
  - Network performance
  - I/O performance
- No clear correlation of FLOPS to real performance, i.e. to "best algorithm"
  - Combinations:
    - Wasteful application with high FLOPS
    - Wasteful application with low FLOPS
    - Highly optimized application with high FLOPS
    - Highly optimized application with low FLOPS
  - FLOPS give no indication how wasteful or optimized a code is

Benchmarking Pitfalls

- Features of current CPU and GPU architectures
  - Varying clock rates and *turbo* modes
    - For benchmarking CPUs and GPUs should be in "thermal equilibrium"
  - Hardware threads vs. hyper-threads (CPUs only)
    - Logical cores for hyper-threads are counted as "CPUs", e.g. by the operating system and SLURM
    - It might not be clear what counts as a core
- Shared resources
  - Activities of other cluster users can influence runtime
    - I/O on global file systems
    - Network load by other programs
    - Program execution on shared nodes

Important Aspect in Benchmark Series

- Use best known sequential algorithm for comparisons
- Focus is on fair speedup results
- Example:
  - Time-to-solution for problem X is 16 minutes using best known sequential algorithm (using 1 processing element)
  - Time-to-solution for problem X is 20 minutes for parallel variant using 1 processing element (pseudo sequential)
  - Time-to-solution for problem X is 4 minutes for parallel variant using 6 processing elements
  - Fair speedup is 4 (not 5); fair efficiency is $4/6 = 66.6\%$ (not $5/6 = 83.3\%$)

SIMD – Data Parallel Processing

- Each operation is executed with multiple data (conceptually) at the same time
- Loop that can be executed in SIMD mode

```
for i := 1 to  N
    a[i] := b[i] + c[i]
```

- Loop that can not be executed in SIMD mode

```
for i := 1 to N
    if (x[i] < eps)
        y[i] := 1.0 - x[i] * x[i] / 6.0
    else
        y[i] := sin(x[i]) / x[i]
```

SIMD (Single Instruction Multiple Data)

- SIMD: Overarching concept for CPUs and GPUs
- CPU
  - AVX (Advanced Vector Extensions) for Intel (since 2008) and AMD CPUs
    - Extensions to the x86 instruction set
    - 16 registers each with 8 single-precision floating point numbers (FP32) (i.e. 8×32 Bit $= 256$ Bit per register)
    - AVX-512: 32 register each with 16 FP32 (i.e. 16×32 Bit $= 512$ Bit per register)
    - Specific implementation of SIMD concept
  - SSE (Streaming SIMD Extensions)
    - Operates on 128 Bit vectors, "predecessor" to AVX
- GPU
  - Multiple SIMD processing blocks organized as Streaming Multiprocessors (SM)
  - Warp: Group of 32 threads that execute the same instruction simultaneously based on CUDA cores
    - SM executes several Warps at the same time and performs Warp-Scheduling
    - SIMT (Single Instruction Multiple Threads): Using threads hide memory access and arithmetic pipeline latencies
      - Analogy: Hyper-Threading for CPUs
  - CUDA or OpenACC implement SIMD concept

Scaling

- Good scalability
  - Efficiency remains high when number of processing elements is increased
- Weak scaling
  - Problem size is increased in proportion with number of processing elements
  - “How big may they problems be that can be solved?”
- Strong scaling
  - Problem size remains constant while number of processing elements is increased
  - “How fast can a problem of a given size be solved?”

SLURM Environment Variables

- Accessibility of resource specifications at runtime of job
  - Nodes allocated in `$SLURM_JOB_NUM_NODES`
  - Type of GPUs allocated in `$SLURM_GPUS`
  - List of GPUs allocated `$SLURM_JOB_GPUS`
    - Also available in `$CUDA_VISIBLE_DEVICES`
      - For NVIDIA GPUs and depending on the SLURM version
  - Processes per node in `$SLURM_TASKS_PER_NODE`
  - Threads (cores) per process in `$SLURM_CPUS_PER_TASK`
  - ...