INTRODUCTION TO SUPERCOMPUTING AT JSC

HPC IN A NUTSHELL

20.11.2023 I ILYA ZHUKOV

(with content used with permission from tutorials by Bernd Mohr/JSC)

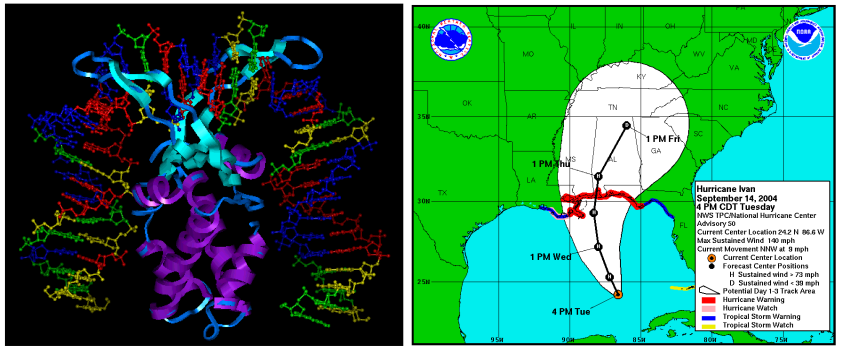
BUILDING BLOCKS OF HPC

WHAT IS HPC?

High-**p**erformance **c**omputing

- **Computer simulation augments theory and experiments**
  - Needed whenever real experiments would be too large/small, complex, expensive, dangerous, or simply impossible
  - Became third pillar of science
- **Computational science**
  - Multidisciplinary field that uses advanced computing capabilities to understand and solve complex problems
- Challenging applications
  - In science
  - In industry

⇨ **Realistic simulations need enormous computer resources (time, memory) !**

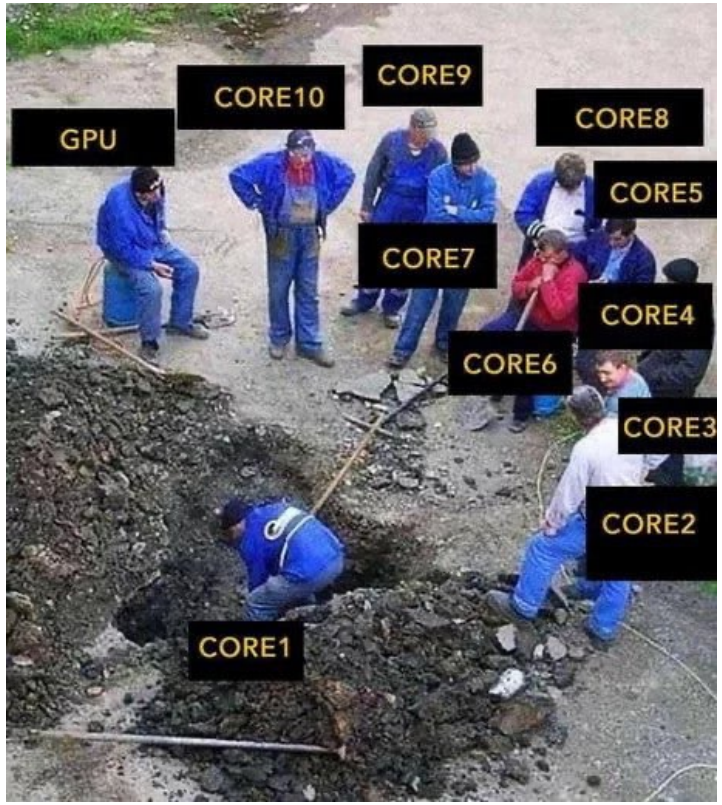


WHY USE PARALLEL COMPUTERS?

- Parallel computers **can** be the only way to achieve specific computational goals in a given time
  - Sequential system is too “slow”
    - Calculation takes days, weeks, months, years, …

    ⇨ **Use more than one processor to get calculation faster**
  - Sequential system is too “small”
    - Data does not fit into the memory

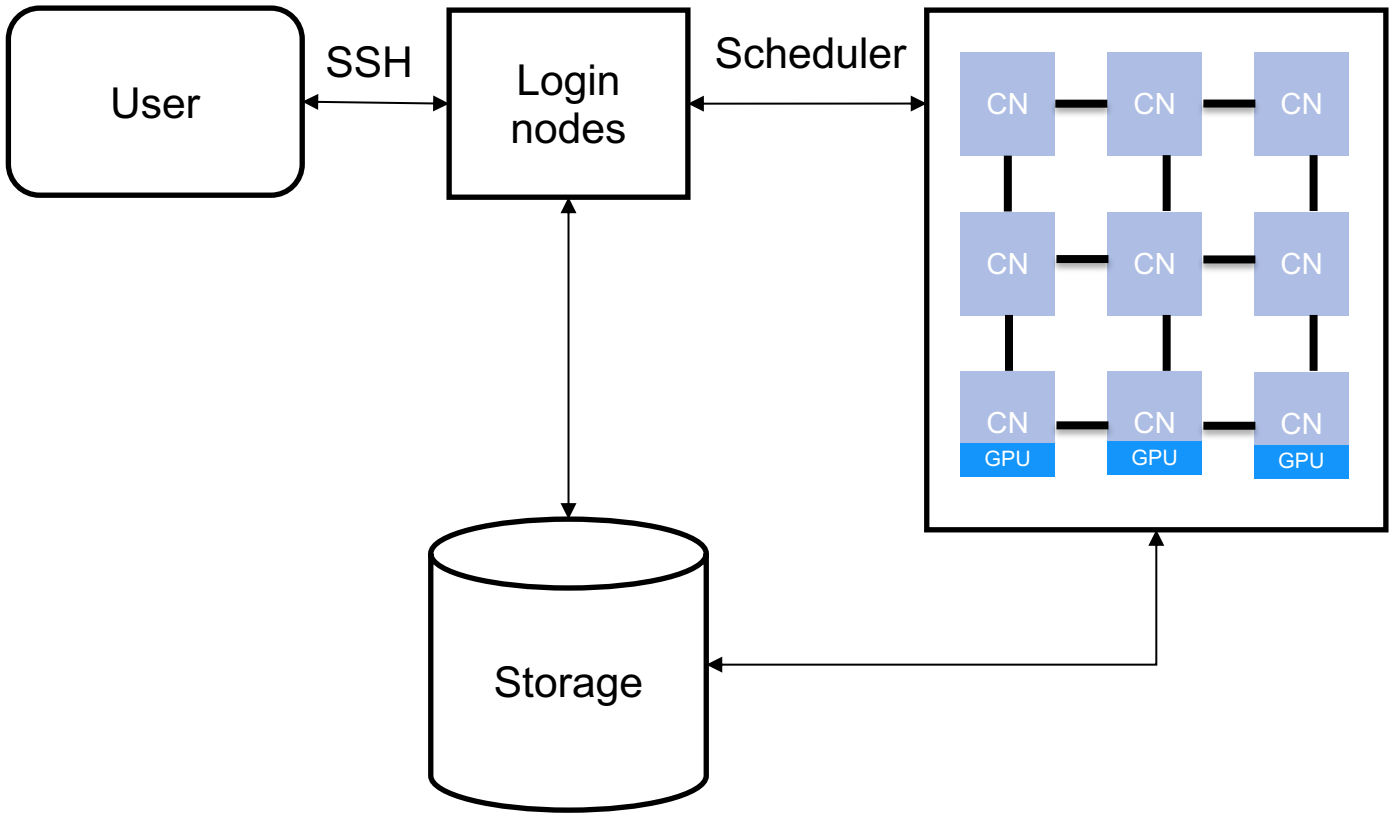
    ⇨ **Use parallel system to get access to more memory**
- You realize you have a parallel system (⇨ **multicore**) and you want to make use of its special features
- Your advisor / boss tells you to do it ;-)



* https://9gag.com/gag/av5vmzd

HIGH-PERFORMANCE COMPUTER

HPC building blocks

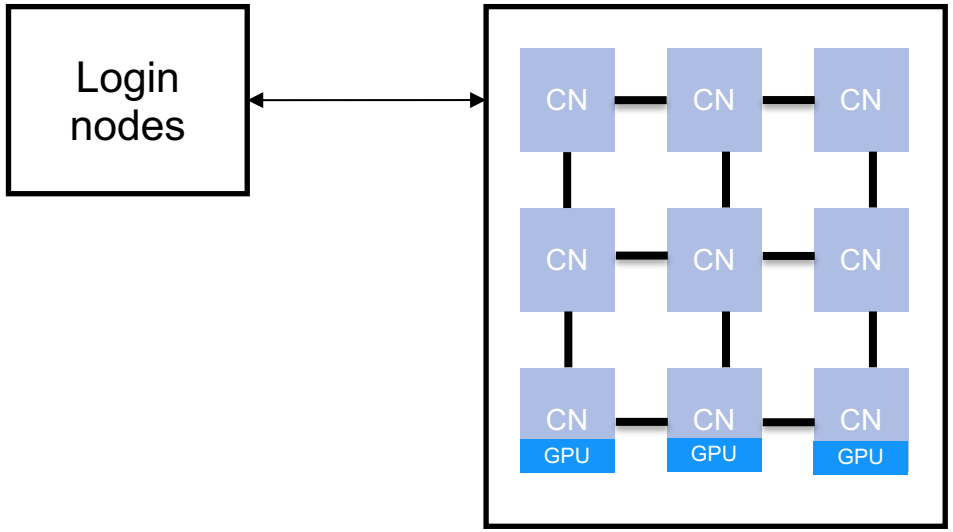


- **Hardware**
  - Login and compute nodes (CN)
  - Network
  - Storage
- **Software**
  - Operating System (OS)
  - Compilers
  - Libraries
  - Scheduler

HIGH-PERFORMANCE COMPUTER

Hardware

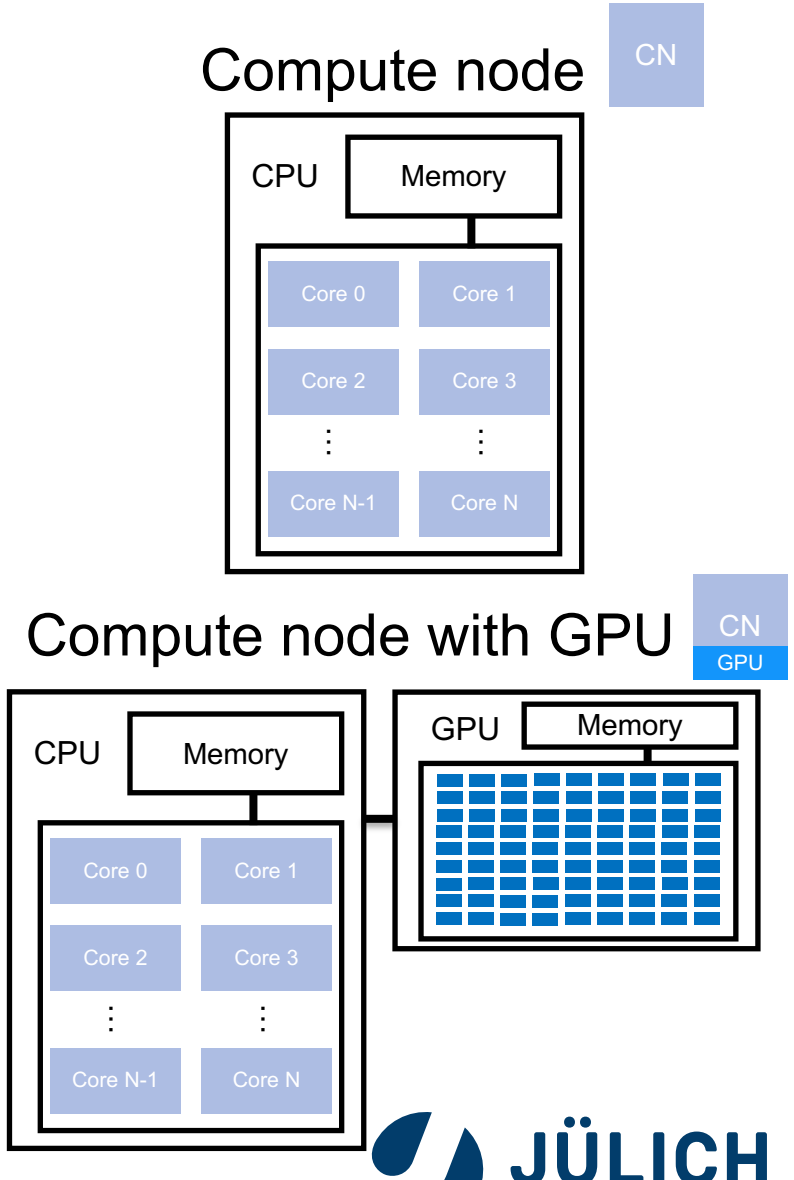
- The **Nodes**
  - Individual computers that compose a cluster are typically called nodes



HIGH-PERFORMANCE COMPUTER

Hardware

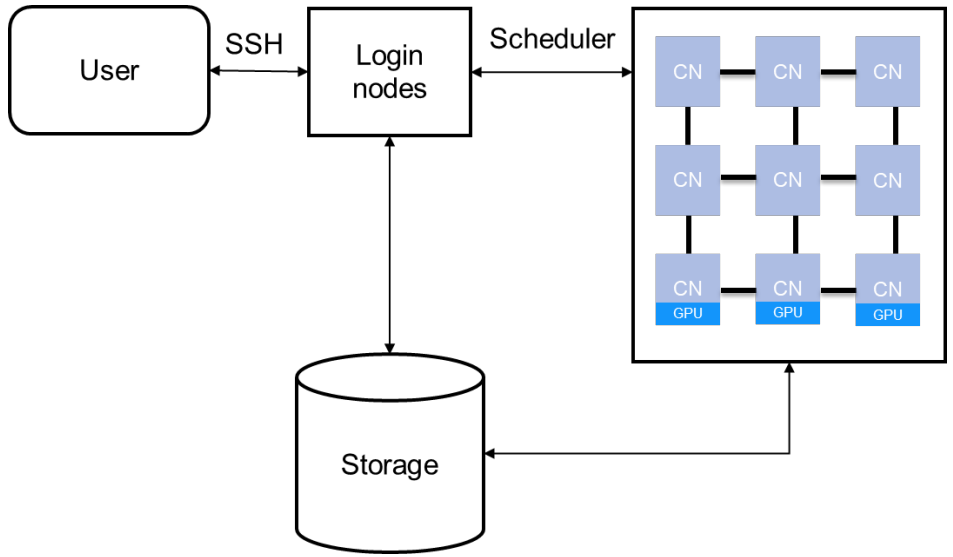
- The **Nodes**
  - Individual computers that compose a cluster are typically called nodes
  - Components of the node
    - Central Processing Unit (**CPU/processor**)
      - CPU can have a single **core** or multiple **cores** (execution unit of a CPU)
    - Memory (RAM, DRAM)
    - Optional: disk space (HDD, SSD, NVMe)
    - Optional: **GPU** (Graphics Processing Unit)
  - Nodes can be grouped into **partitions**: a group of nodes which are characterised by their hardware or purpose, e.g. GPU partition, large memory partition, visualisation partition etc.



HIGH-PERFORMANCE COMPUTER

Hardware

- The **Login (head) nodes**
  - Suited for uploading/downloading files, installing and setting up software, and running quick tests
  - Entry point to the cluster
  - Accessible outside the cluster
  - Only a few nodes are available and they are shared among all users
  - **Please use with respect for other users!**



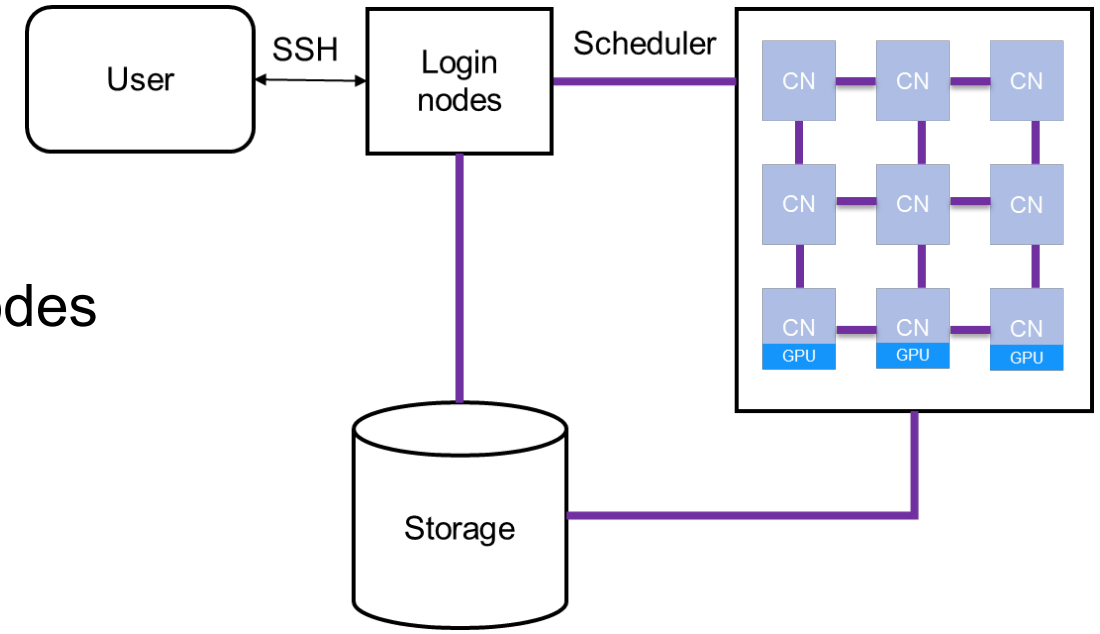
- The **Compute (worker) nodes**
  - Typically dedicated to long or hard tasks that require a lot of computational resources
  - Smallest unit available for allocation (use it wisely!)
  - Accessible only inside the cluster

Note: you'll learn more during **"JSC systems – JUWELS, JURECA & JUSUF"** talk

HIGH-PERFORMANCE COMPUTER

Hardware

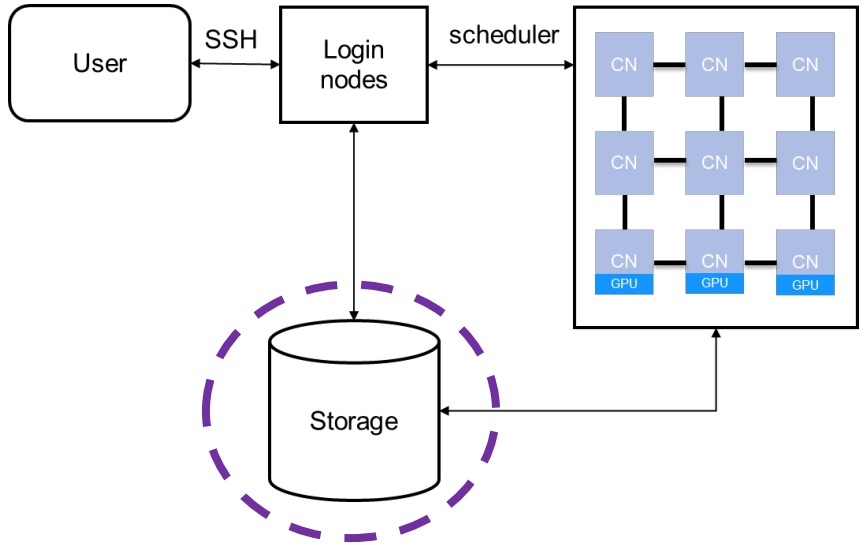
- The **Network** connects nodes in order to share resources and data
  - Characteristics of a Network
    - **Latency** is the response time a node experiences when contacting another nodes (nanoseconds, microseconds)
    - **Bandwidth** is the maximum data rate (Megabytes or Gigabytes per second)
    - **Topology** is the way how nodes are interconnected, e.g. ring, mesh, torus, etc.



HIGH-PERFORMANCE COMPUTER

Hardware

- The **Storage** is a hardware system for storing and manipulating data
  - Login and compute nodes are attached to the storage
  - Storage typically has various file systems which have different properties, e.g.
    - Size
    - Backup policies
    - Access time
    - E.g in JSC: $HOME, $PROJECT, $SCRATCH, etc



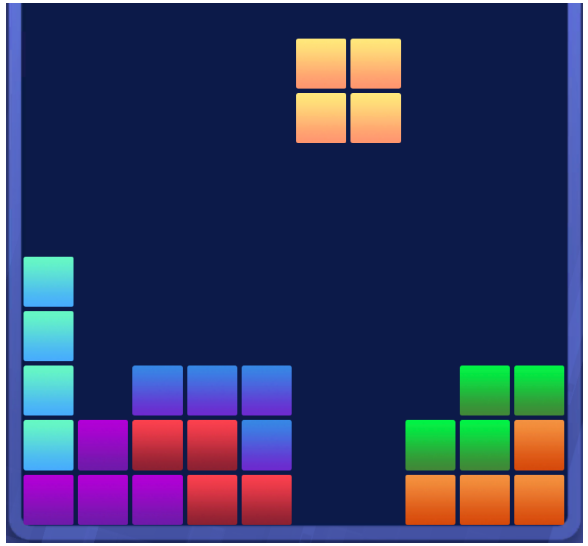
Note: you'll learn more during **"JUST: Juelich Storage Cluster"** talk

HIGH-PERFORMANCE COMPUTER

Software

- **Operating system (OS)** is a system layer that allocates and manages hardware resources, enforces resource protection, provides standardized services, and schedules execution of application
- **Compilers**, e.g. GNU, Intel, NVHPC
- **Libraries**, e.g. MPI, FFTW, etc.
- The **Scheduler** is a special software that manages which **jobs** (set of commands to be run on the cluster) run where and when
  - The most basic use of the scheduler is to run a command non-interactively. This process is called a **batch job submission**
  - An **interactive job** allows a user to interact with applications in real time within an HPC environment

Note: you'll learn more during **"HPC Software – Modules, Libraries & Software"** talk



Note: you'll learn more during **"Work load management with Slurm"** talk

ALL BLOCKS ARE IN PLACE! HOW TO PLAY WITH THEM?

Typical Workflow

1. Write proposal and get compute time on preferred HPC system or join existing project
2. Login to the system
3. Transfer your data to the HPC system
4. Use available software or build your own
5. Make sure your software works and provides correct results! (Hint: start with a small testcase)
6. Optimise it for the available hardware, e.g. set pinning, use high-performance storage, GPUs, etc
7. Analyse and optimise performance with performance analysis tools if necessary
8. Run production jobs to get results and monitor them for correctness
9. Analyse and visualise the results

This is the general cycle. In your individual case some steps may be redundant, some can require several iterations.

Note: some of these topics will be covered during our lectures and practical exercises. Do not miss them!

TIPS AND TRICKS

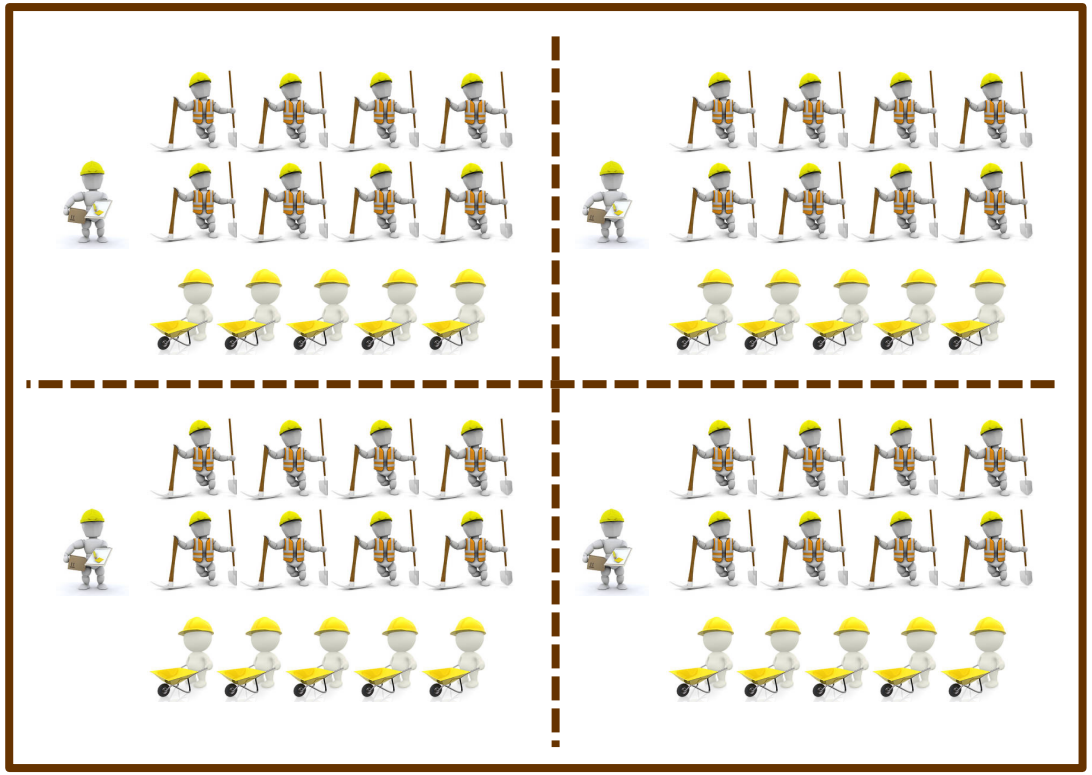
- Always read documentation and manuals!
  - **Status page:** https://status.jsc.fz-juelich.de
  - **JUWELS:** https://apps.fz-juelich.de/jsc/hps/juwels/
  - **JURECA:** https://apps.fz-juelich.de/jsc/hps/jureca/
  - **JUSUF:** https://apps.fz-juelich.de/jsc/hps/jusuf/
- Be gentle with login nodes
  - Never use login nodes for doing actual/production work
  - Do not spawn too many threads, e.g. do not use **“make -j”** use **“make -j 4”** instead
  - Do not use too much memory (can be verified with **“ps ux”** or **“top”** commands)
  - You can use **“kill”** with the PID to terminate any of your intrusive processes
- Have a backup plan
  - Use version control (e.g. git)
  - Use backup file systems for important and frequently used data
  - Archive data that is not used frequently
  - Transfer your data off the system before your access finishes
- Test your setup before running at a big scale or for a long time
- Do you have questions? Just ask! **sc@fz-juelich.de**

INTRODUCTION TO PARALLEL PROGRAMMING

PROGRAMMING PARALLEL COMPUTERS

- Application programmer needs to
  - **Distribute data and work**
    - **Domain decomposition:** different processors do similar (same) work on different pieces
    - **Functional decomposition:** different processors work on different types of tasks
  - **Organize and synchronize work** and dataflow
    - Balance load

- Extra HPC constraint
  - **Do it with least resources most effective way!**



SIMPLE PROGRAMMING EXAMPLE

- Determine maximum value of polynomial 4th grade
  - $y = a \times x^3 + b \times x^2 + c \times x + d$
- Infinitive number of possible values
  - **Discretization**: select huge but finite number of numerical values representing a specific **resolution** determining accuracy

- Program
  1. Read coefficients (a, b, c, d), domain (x_{min}, x_{max}), resolution (numsteps)
  2. maximum = smallest-possible-value
  3. For $x = x_{min}$ to x_{max} in numsteps
     Calculate polynomial y(x)
     If y larger than maximum, then maximum = y
  4. Print maximum

POSSIBLE PARALLEL PROGRAM

- Determine maximum value of polynomial 4^{th} grade

- **On selected master processor**
  1. Read coefficients (a, b, c, d), domain (x_{min}, x_{max}), resolution (numsteps)
  2. Distribute values to all processors
- **Concurrently for all processors P**
  3. **processor**-maximum = smallest-possible-value
  4. For **processor-subset-of** x = x_{min} to x_{max} in numsteps **# work distribution**
     Calculate polynomial y(x)
     If y larger than **processor-**maximum
     then **processor-**maximum = y
  - **On selected master processor**
    5. Collect all maximums from processors
    6. Determine global maximum
    7. Print maximum

PERFORMANCE METRICS I

- For a given problem A, let
  - **T(N,1)** = Time of the best serial algorithm to solve A for input of size N
  - **T(N,P)** = Time of the parallel algorithm + architecture to solve A for input size N, using P processors

Speedup

$$\text{Speedup}(N,P) = \frac{T(N,1)}{T(N,P)}$$

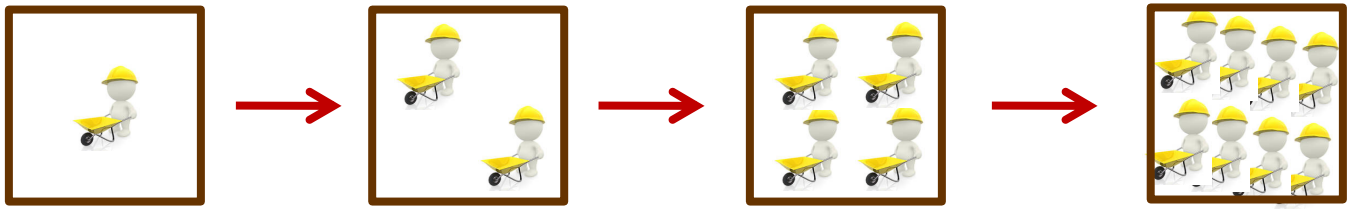
Parallel efficiency

$$\text{Efficiency}(N,P) = \frac{T(N,1)}{P \cdot T(N,P)} = \frac{S(N,P)}{P}$$

PERFORMANCE METRICS II

- In general, expect
  - 0 ≤ Speedup(P) ≤ P
  - 0 ≤ Efficiency ≤ 1
- **Linear speedup**: if there is a constant c > 0 so that speedup is at least c • P.
  - Many use this term to mean c = 1.
- **Perfect** or **ideal speedup**: Speedup(P) = P
- **Superlinear speedup**: Speedup(P) > P (Efficiency > 1)
  - Typical reason: Parallel computer has P times more memory (cache), so higher fraction of program data fits in memory instead of disk (cache instead of memory)

AMDAHL'S LAW



- Assumption
  - total problem size stays the same as the number of processors increases (**strong scaling**)
  - α is a completely serial fraction
  - parallel part is 100% efficient
- Parallel runtime

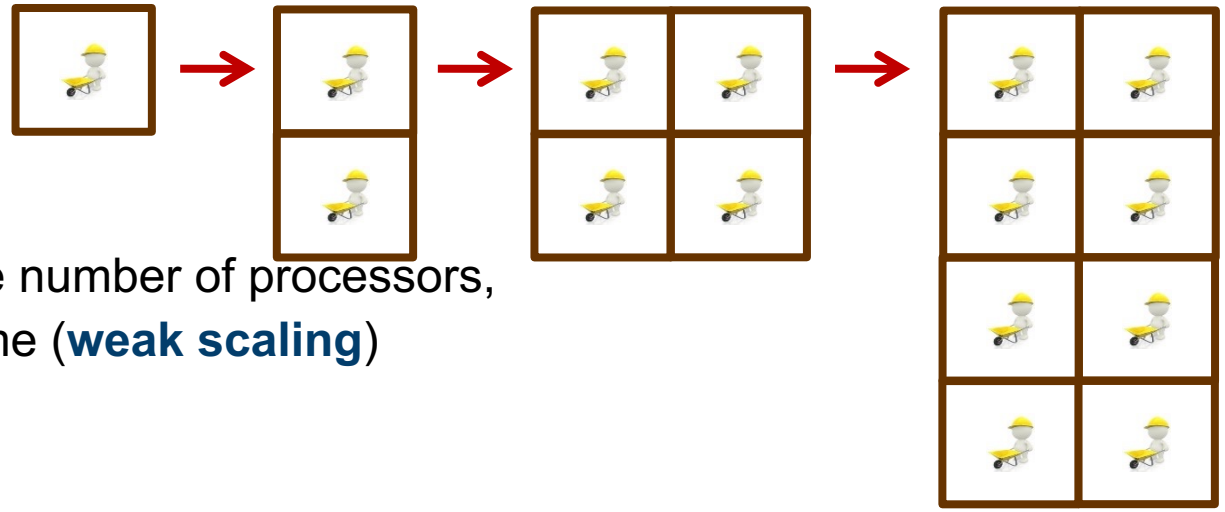
$$T(N,P) = \alpha T(N,1) + \frac{(1 - \alpha)T(N,1)}{P}$$

- Parallel speedup

$$\text{Speedup}(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{1}{\alpha + \frac{(1 - \alpha)}{P}}$$

- Our software is fundamentally limited by the serial fraction
  - $\alpha = 0$, Speedup = P
  - $\alpha = 0.1$, max speedup is 10, e.g. Speedup(N,10) = 5.26, Speedup(N,1000) = 9.91

GUSTAFSON'S LAW



- Assumption
  - the problem size increases at the same rate as the number of processors, keeping the amount of work per processor the same (**weak scaling**)
  - α is a completely serial fraction
  - parallel part is 100% efficient
- Runtime on single process

$$T(N,1) = \alpha T(N,1) + (1 - \alpha)PT(N,1)$$

- Parallel runtime

$$T(N,P) = \alpha T(N,1) + (1 - \alpha)T(N,1)$$

- Parallel speedup

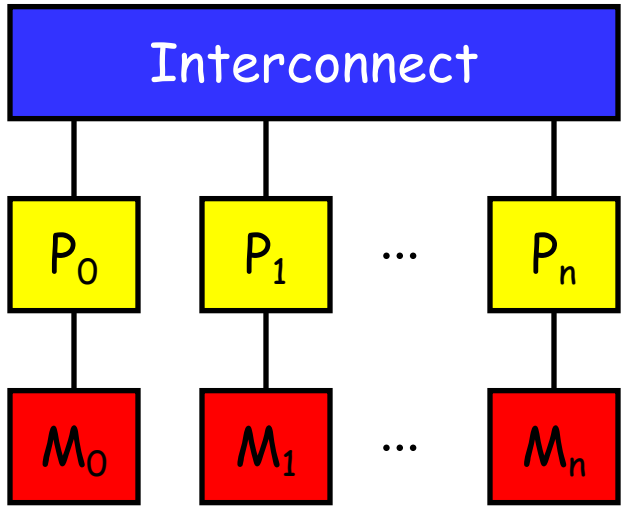
$$Speedup(N,P) = \frac{T(N,1)}{T(N,P)} = \alpha + (1 - \alpha)P$$

- Limitation by the serial fraction becomes less
  - $\alpha = 0$, Speedup = P
  - $\alpha = 0.1$, e.g. Speedup(N,10) = 9.10, Speedup(N,1000) = 900.10

HARDWARE ARCHITECTURE

PARALLEL ARCHITECTURES: DISTRIBUTED MEMORY I

- Interconnected nodes (processor + memory)
- All memory is associated with processors



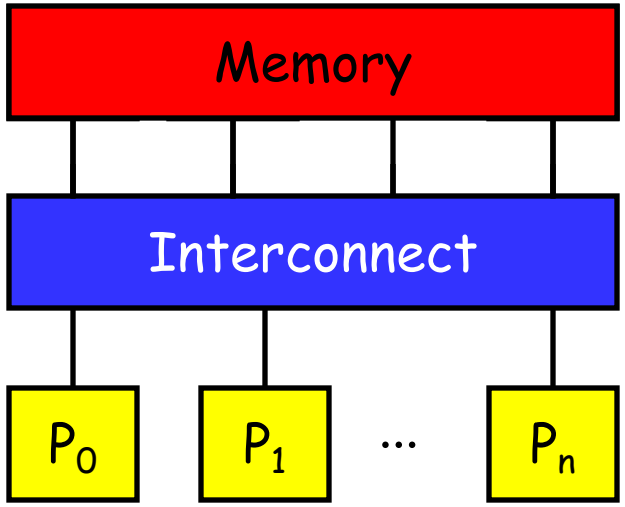
- **Advantages**
  - Memory is scalable with number of processors
    - Can build very large machines (10000's of nodes)
  - Each processor has rapid access to its own memory without interference or cache coherency problems
  - Cost effective and easier to build: can use commodity parts

PARALLEL ARCHITECTURES: DISTRIBUTED MEMORY II

- **Disadvantages**
  - To retrieve information from another processor's memory a **message** must be sent over the network to the home processor
  - Programmer is responsible for many of the details of the communication;
    easy to make mistakes
    - **Explicit** data distribution
    - **Explicit** communication via messages
    - **Explicit** synchronization
  - May be difficult to distribute the data structures, often additional data structures needed (ghost cells, location tables, …)
- Programming Models
  - Message passing: **MPI**, PVM, shmem, ...

PARALLEL ARCHITECTURES: SHARED MEMORY

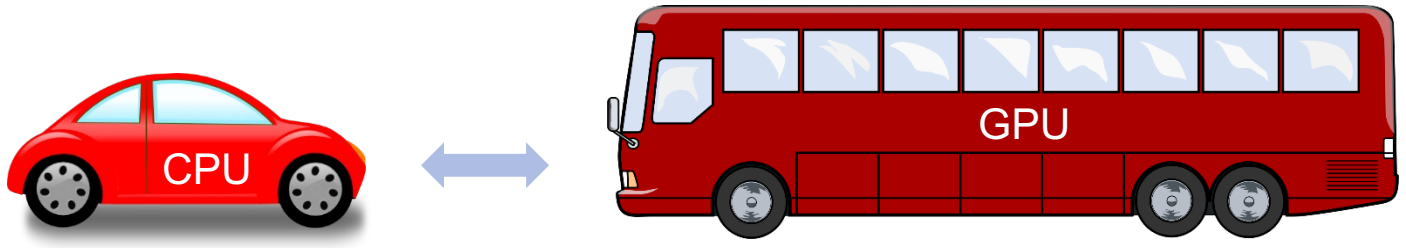
- More exact: **shared address space** accessible by all processors
  - physical memory modules may be distributed
- Processors may have local memory (e.g., **caches**) to hold copies of some global memory. Consistency of these copies is usually maintained by special hardware (**cache coherence**)
- Programming Models
  - Automatic parallelization via compiler
  - Explicit threading (e.g. POSIX threads)
  - **OpenMP**
  - [**MPI**]



ACCELERATORS

- Special hardware for accelerating computations has long tradition in HPC
  - Floating-point units
  - **SIMD/vector units**
    - MMX, SSE (Intel), 3DNow! (AMD), AltiVec (IBM)
  - **FPGA** (**F**ield **P**rogrammable **G**ate **A**rrays)
  - **G**eneral **P**urpose computing on **G**raphics **P**rocessing **U**nits (**GPGPU**)

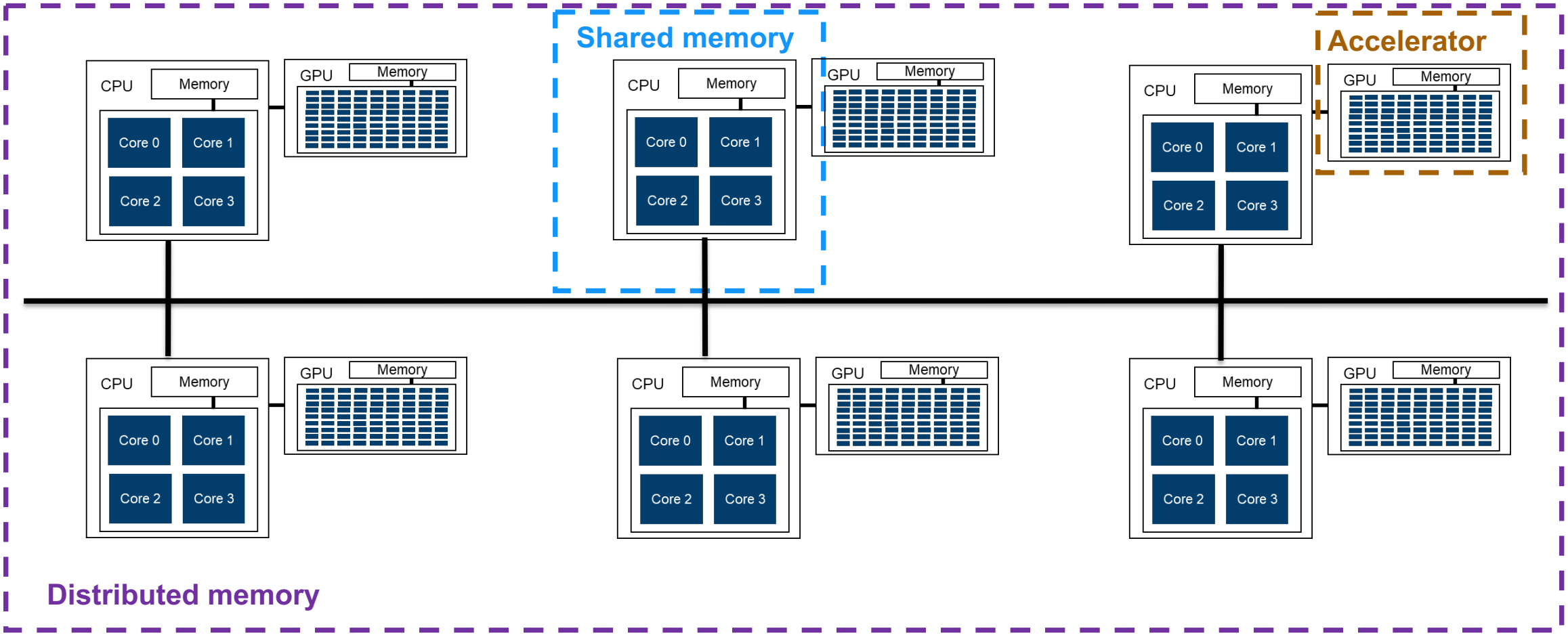
GPGPU



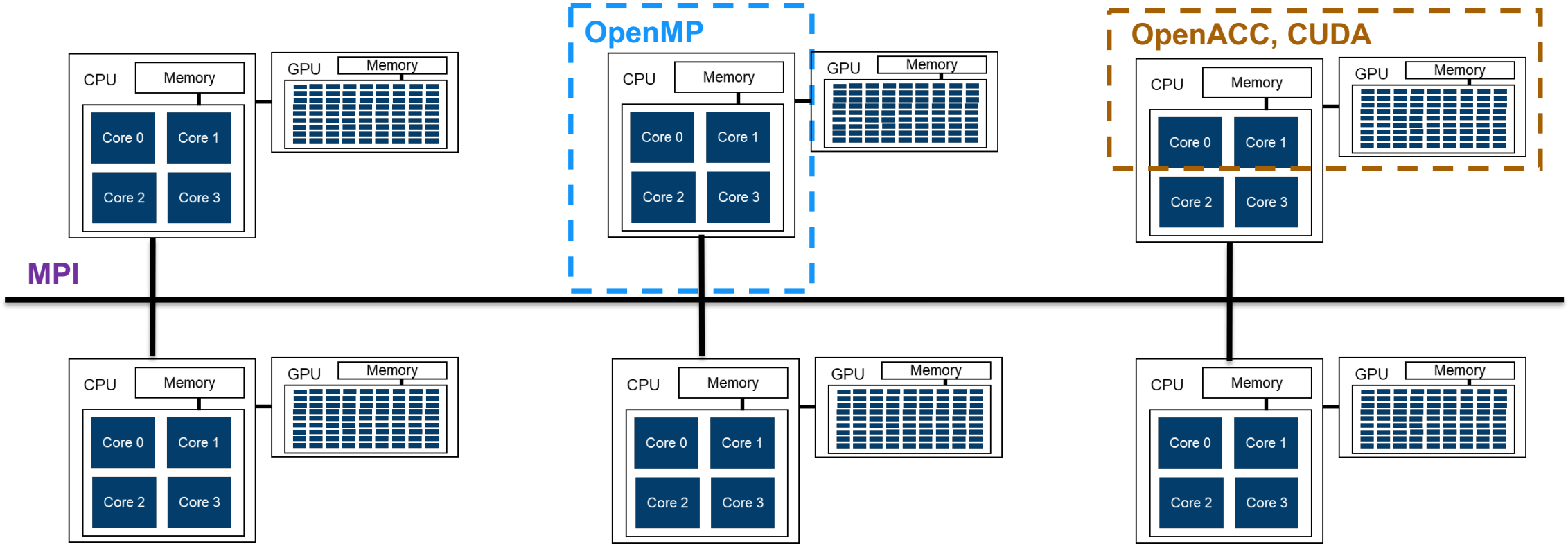
- **Modern GPUs**
  - Have a parallel many-core architecture
    - Each core capable of running 1000s of threads simultaneously
  - **Independent blocks with fine-grain data-parallelism (SIMT)**
  - Highly parallel structure makes them more effective than general-purpose CPUs for **some** (vectorizable) algorithms
- More difficult to use hardware effectively than “standard” CPUs
  - High-level portable programming interfaces still evolving
    - **OpenACC, OpenMP 5.0**
  - Main disadvantage: data must be moved to and from main memory to GPU memory
  - Data locality important, otherwise performance degrades significantly

Note: you’ll learn more during **“Using GPU accelerators of JURECA and JUWELS”** talk

FROM THEORY TO PRACTICE I



FROM THEORY TO PRACTICE II



TYPICAL PARALLELISATION WORKFLOW

1. Identify what you want to parallelise
   - What is your common testcase?
   - Where do you spend most of your time?
2. Identify what hardware do you want to use (CPU, GPU, CPU+GPU, …)
3. How do you want parallelise
   - Library, MPI, OpenMP, OpenACC, CUDA, **MPI+X**, …
4. Implement your choices
5. Validate correctness
6. Evaluate scalability (speedup and efficiency, strong vs. weak scaling)
7. Tune and optimise

Repeat the cycle if necessary!

Note: some of these topics will be covered during our lectures and practical exercises. Do not miss them!