UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

High Performance Computing

Lecturer:
Prof. Stefano Cozzini

Author:
Andrea Spinelli

March 25, 2025

This document is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA) license. You may share and adapt this material, provided you give appropriate credit, do not use it for commercial purposes, and distribute your contributions under the same license.

github.com/Spina02 ✉ andreaspinelli2002@gmail.com

Preface

As a student of Scientific and Data Intensive Computing, I've created these notes while attending the **High Performance Computing** module of **High Performance and Cloude Computing** course.

The course will introduce the fundamentals of High Performance Computing, exploring both its concepts and practical applications. The notes cover a wide range of topics, including:

- An overview of High Performance Computing and its importance in solving complex, real-world problems.
- The principles behind modern computer architectures and how they influence performance.
- Essential tools and techniques for parallel programming, alongside strategies to optimize code for advanced architectures.
- The evolution of computing facilities and how to effectively leverage them for large-scale computational challenges.
- Developing a proactive mindset, moving beyond the use of pre-packaged tools to a deeper understanding of the underlying systems.

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in High Performance Computing.

Draft

Contents

Draft

1 Introduction

1.1 Basic Concepts

High Performance Computing (HPC), also known as ***supercomputing***, refers to computing systems with extremely high computational power that are able to solve hugely complex and demanding problems. [1]

Often, high precision and accuracy are required in scientific and engineering simulations, which can be achieved by increasing the computational power of the system. This is where HPC comes into play, as it allows for the execution of large-scale **simulations** of complex problems in a reasonable amount of time. Simulations have become the key method for researching and developing innovative solutions in both scientific and engineering fields. They are especially prominent in leading domains such as the aerospace industry and astrophysics, where they enable the investigation and resolution of highly complex problems. However, the increasing reliance on simulation also introduces significant **challenges related to complexity, scalability, and data management**, which in turn impact the supporting IT infrastructure.

As scientific inquiry progresses along what is known as the *Inference Spiral of System Science*, the complexity of models intensifies and the influx of new data enriches these systems with additional insights. Consequently, this dynamic evolution necessitates ever increasing computational power to efficiently handle the enhanced simulations and data management challenges.

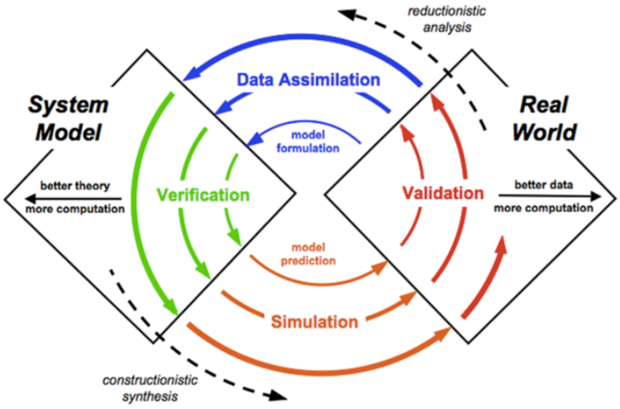

Figure 1.1: Research and Development

Prefix	Symbol	Value
Yotta	Y	10^{24}
Zetta	Z	10^{21}
Exa	E	10^{18}
Peta	P	10^{15}
Tera	T	10^{12}
Giga	G	10^{9}
Mega	M	10^{6}
Kilo	K	10^{3}

Table 1.1: Prefixes in HPC

Observation:

In today's world, larger and larger amounts of data are constantly being generated, from 33 zettabytes globally in 2018 to an expected 181 zettabytes in 2025. This exponential growth is driving a shift towards data-intensive applications, making HPC indispensable for processing and analyzing these vast datasets efficiently. Consequently, HPC is key to unlocking valuable insights that benefit citizens, businesses, researchers, and public administrations. [1]

1.1.1 What is High Performance Computing?

High Performance Computing (HPC) involves using powerful servers, clusters, and supercomputers, along with specialized software, tools, components, storage, and services, to solve computationally intensive scientific, engineering, or analytical tasks.

HPC is used by scientists and engineers both in research and in production across industry, government and academia.

Key elements of the HPC ecosystem include:

- **Hardware:** High-performance servers, clusters, and supercomputers.
- **Software:** Specialized tools and applications designed to optimize complex computations.
- **Applications:** Scientific, engineering, and analytical tasks that leverage high computational power.

People in HPC

Human capital is by far the most important aspect in the HPC landscape. Two crucial roles include HPC providers, who plan, install, and manage the resources, and HPC users, who leverage these resources to their fullest potential. The mixing and interplaying of these roles not only enhances individual competence but also drives overall advancements in high-performance computing.

1.1.2 Performance and metrics

Performance in the realm of high-performance computing is a multifaceted concept that extends far beyond a mere measure of speed. While terms such as "how fast" something operates are often used to describe performance, they tend to be vague. Many factors contribute to the overall performance of a system, and the interpretation of these factors can vary depending on the specific context and objectives of the computational task. Performance, therefore, remains a complex and central issue in the field of HPC, as it involves more than just the raw computational speed.

The discussion often extends to the idea that the ”P” in HPC might stand for more than just performance. A growing sentiment among professionals in the field suggests that high performance should be complemented by high productivity. This broader view recognizes that the true efficiency of a computing system is not only determined by its ability to perform tasks quickly but also by the ease and speed with which applications can be developed and maintained. In other words, while raw performance is critical, the overall productivity of a system—combining the system's speed with the programmer's effort—plays an equally important role.

To further clarify the distinction, consider that performance can be seen as a measure of how effectively a system executes tasks, whereas productivity is the outcome achieved relative to the effort invested in developing the application. For instance, if a code optimization leads to a system that runs twice as fast but requires an extensive period of development—say, six months of work—the benefits of the improvement must be weighed against the increased effort required. This example underlines the importance of balancing performance gains with the associated development costs.

Ultimately, the challenge lies in understanding and optimizing both aspects. A successful HPC system is one that not only achieves high computational throughput but also enhances the productivity of the developers who create and refine the applications. This balance is essential for advancing the capabilities of high-performance computing in both research and production environments.

Number Crunching on CPU

When evaluating the performance of a high-performance computing (HPC) system, one of the most fundamental metrics is the rate at which floating point operations are executed. This rate is typically expressed in millions (Mflops) or billions (Gflops) of operations per second. In essence, it quantifies how many calculations, such as additions and multiplications, the system is capable of performing every second.

To estimate this capability, we rely on the concept of theoretical peak performance. This value is computed by considering the system's clock rate, the number of floating point operations that can be executed in a single clock cycle, and the total number of processing cores available. Under ideal conditions, the theoretical peak performance can be expressed as follows:

$$\text{FLOPS} = \text{clock\_rate} \times \text{Number\_of\_FP\_operations} \times \text{Number\_of\_cores}$$

This formula provides an upper bound on the computational power of the system. However, it is important to note that this is a best-case scenario estimate and does not always reflect the performance achievable in real applications.

Sustained (Peak) Performance

While the theoretical peak performance offers insight into the maximum potential of an HPC system, the actual performance observed during real-world operations is better captured by the sustained (or peak) performance. In practice, several factors such as memory bandwidth limitations, communication latencies, and input/output overhead can prevent a system from reaching its theoretical maximum.

Sustained performance refers to the effective throughput that an HPC system attains when executing actual workloads. Since it is challenging to exactly measure the number of floating point operations performed by every application, standardized benchmarks are commonly used to assess this performance. One widely recognized benchmark is the HPL Linpack test, which forms the basis for the TOP500 list of supercomputers. This benchmark emphasizes the importance of sustained performance, as it reflects the system's efficiency and reliability under realistic operational conditions.

Understanding both the theoretical and sustained performance metrics is crucial. While the former provides an idealized estimate of a system's capabilities, the latter offers a more practical perspective, thereby guiding decisions on system improvements and resource allocation in high-performance computing environments.

…

1.1.3 Moore Law

Tipically, the Moore Law is stated as: ”Performance doubles every 18 months”. However, it is actually closer to ”The number of transistors per unit cost doubles every 18 months”.

The original Moore Law was formulated by Gordon Moore, co-founder of Intel, in 1965. He predicted that:

Definition:

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. [...] Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."
~Gordon Moore, 1965

Dennard Scaling: From Moore's Law to performance

Definition:

"Power density stays constant as transistors get smaller"
~Robert H. Dennard, 1974

The concept of Dennard scaling, named after Robert Dennard, an IBM researcher, is closely related to Moore's Law. Dennard scaling refers to the observation that as transistors shrink in size, their power density remains constant. This phenomenon allowed for the continuous increase in clock speeds and performance of microprocessors over the years.

However, Dennard scaling began to break down around the early 2000s, as power consumption and heat dissipation became significant challenges. Consequently, the industry shifted its focus from increasing clock speeds to improving parallelism and energy efficiency.

Intuitively:

- Smaller transistors → shorter propagation delay → faster frequency
- Smaller transistors → smaller capacitance → lower power consumption

Draft

$$Power \propto Capacitance \times Voltage^2 \times Frequency$$

End of Dennard Scaling: Power wall

The power wall is a fundamental limit on the amount of power that can be dissipated by a chip. This limit is determined by the chip's thermal design power (TDP), which is the maximum amount of heat that the cooling system can dissipate. As the number of transistors on a chip increases, the power consumed by the chip also increases, eventually reaching the TDP limit. When this limit is reached, the chip can no longer dissipate the heat generated by the transistors, leading to overheating and reduced performance.

However, the original Moore's Law is still valid, as the number of transistors per unit cost continues to double every 18 months, but no more on a single core. Instead, the industry has shifted towards multi-core processors and parallel computing to continue improving performance.

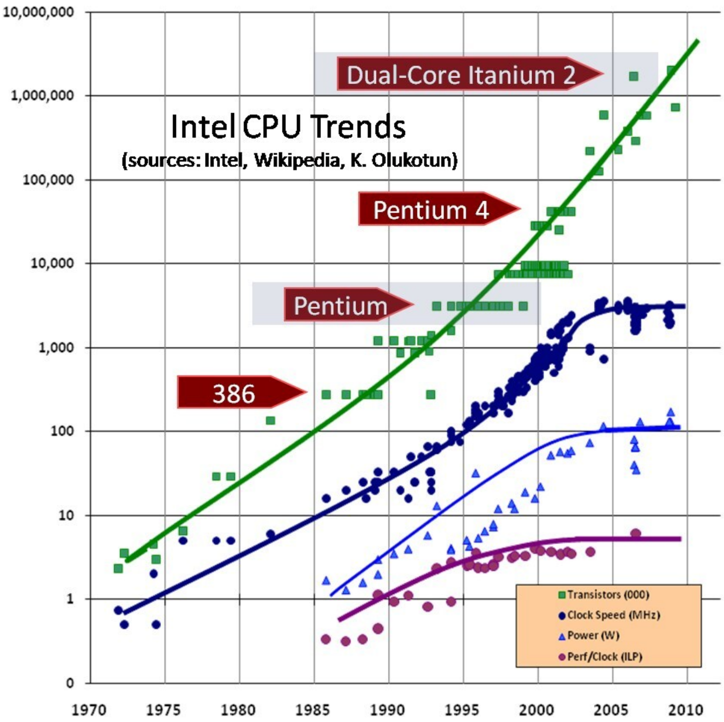

Figure 1.2: Moore's Law

This evolution marks what many computer scientists and engineers refer to as the end of the ”free lunch” era, which began around 2006. Prior to this shift, software developers could rely on hardware improvements to automatically enhance their applications' performance without significant code optimization. Single-core performance scaling, which had been the primary driver of computational advances for decades, effectively plateaued as the industry encountered fundamental physical limitations.

The computing community has responded to this challenge through two complementary approaches:

The Software Solution: This approach emphasizes the critical importance of efficient software design and implementation. As hardware improvements no longer automatically translate to performance gains, developers must engage in deliberate ”performance engineering”—applying sophisticated optimization techniques informed by deep understanding of hardware architecture. This involves careful algorithm selection, memory access pattern optimization, and exploitation of instruction-level parallelism to maximize the utilization of available hardware resources.

The Specialized Architectural Solution: The second approach acknowledges a fundamental shift in design constraints: while chip space has become relatively inexpensive, power consumption has emerged as the primary limiting factor. Rather than continuing to develop increasingly complex general-purpose processing cores, this approach advocates for heterogeneous computing systems. Such systems incorporate specialized accelerators (such as GPUs, TPUs, and FPGAs) that are optimized for specific computational patterns. This architectural diversification allows for significant performance improvements in targeted application domains while maintaining reasonable power consumption profiles.

These complementary strategies represent the computing industry's response to the physical limitations that have constrained traditional performance scaling. By combining software optimization with hardware specialization, the field continues to advance computational capabilities even as the straightforward scaling of single-core performance has reached its practical limits.

1.1.4 The Shift to Multicore Architecture

Modern CPUs have evolved into multicore processors due to physical constraints in power consumption and heat dissipation, with manufacturers reducing clock frequencies while increasing core count to deliver greater computational throughput within manageable thermal profiles. These independent cores can execute separate instruction streams simultaneously but share critical resources including memory hierarchies, controllers, and peripheral subsystems, creating a complex environment where cores must cooperate and compete for resources. This architectural shift effectively circumvents the limitations of traditional single-core scaling but presents new challenges for software developers, who must now explicitly design for parallelism to fully leverage available computational capabilities.

[Hardware accelerators image]

1.1.5 Parallel Compuers

Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously. Flynn Taxonomy is a classification of parallel computer architectures, proposed by Michael J. Flynn in 1966. It categorizes computer systems based on the number of instruction streams and data streams that can be processed concurrently. The four categories are shown in Table 1.2.

	HW level	**SW level**
SISD	A Von Neumann CPU	no parallelism at all
MISD	On a superscalar CPU, different ports executing different *read* on the same data	ILP on same data; multiple tasks or threads operating on the same data
SIMD	Any vector-capable hardware (vector registers on a core, a GPU, a vector processor, an FPGA, ...)	data parallelism through vector instructions and operations
MIMD	Every multi-core processor; on a superscalar CPU, different ports executing different ops on different data	ILP on different data; multiple tasks or threads with different data on each core

Table 1.2: Comparison of SISD, MISD, SIMD, and MIMD at HW and SW levels

While Flynn's taxonomy provided a foundational classification system in 1966, its utility for categorizing modern HPC infrastructure has diminished significantly. The dramatic evolution of CPUs and computing architectures over the past six decades has produced systems with hybrid designs that transcend these simple classifications. Nevertheless, the fundamental concepts of SIMD and MIMD remain relevant principles that continue to influence the design and implementation of contemporary HPC hardware solutions.

1.1.6 Essential Components of a HPC Cluster

- Several computers (nodes)
  Often in special cases (1U) for easy mounting in racks
- One or more networks (interconnects) to hook the nodes together
- Some kind of storage
- A login/access node

[Cluster image]

1.2 Single CPU topology

Modern CPUs are multy- (or many-) core processors.

Definition:

A **core** is the smallest unit of computing, having one or more (hardware/software) threads and is responsible for executing instructions.

A CPU uses a **Cache hierarchy** to store data and instructions. The cache hierarchy consists of several levels of cache, each with different sizes and access times. The cache hierarchy is designed to minimize the time it takes to access data and instructions, which can significantly improve the performance of the CPU.

[CPU layout image]

[Node topology image]

[Overall topology image]

…

1.2.1 memory

on a supercomputer there is a hybrid approach as for the memory placement:

- **Shared memory:** the memory on a single nodes can be accessed directly by all the cores on that node, meaning that memory access is a “read/write” instructions irrespectively of what exact memory bank it refers to.
- **distributed memory:** when you use many nodes at a time, a process can not directly access the memory on a different node. It need to issue a request for that, not a read/write instruction.

These are hardware concepts, i.e. they describe how the memory is physically accessible. However, they do also refer to programming paradigms, as we’ll see in a while.

Tip: ***Notation***

- **Multiprocessor:** server with more than 1 CPU
- **Multicore:** CPU with more than 1 core
- **Processor** = CPU = Socket

Note that sometimes the term “processor” is used to refer to the CPU, sometimes to the core.

Shared Memory: UMA

Uniform memory access (UMA): Each processor has uniform access to memory. Also known as symmetric multiprocessing (SMP).

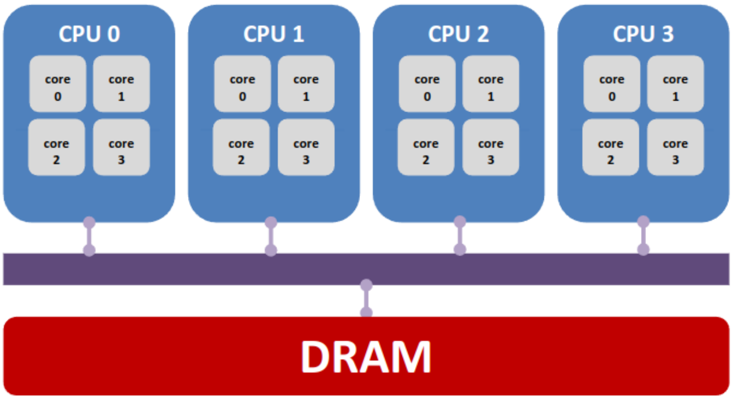

Figure 1.3: Uniform Memory Access (UMA)

Shared memory: NUMA

Non-uniform memory access (NUMA): Time for memory access depends on location of data. Local access is faster then non-local access.

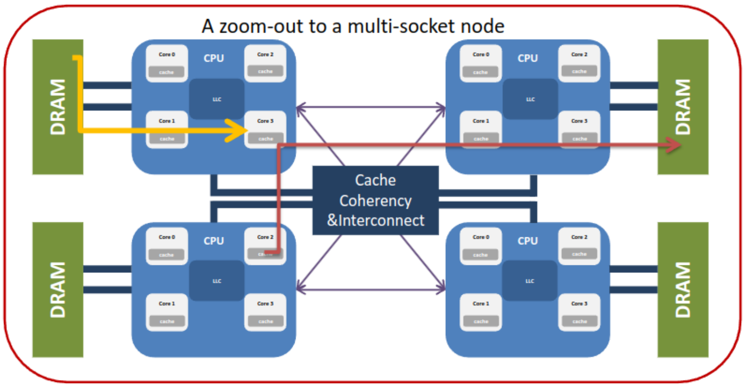
A zoom-out to a multi-socket node
DRAM
CPU
LLC
Cache Coherency &Interconnect

Figure 1.4: Non-Uniform Memory Access (NUMA)

Parallelism within a HPC node

A single node can have multiple cores, each with multiple hardware threads. This introduces several levels of parallelism:

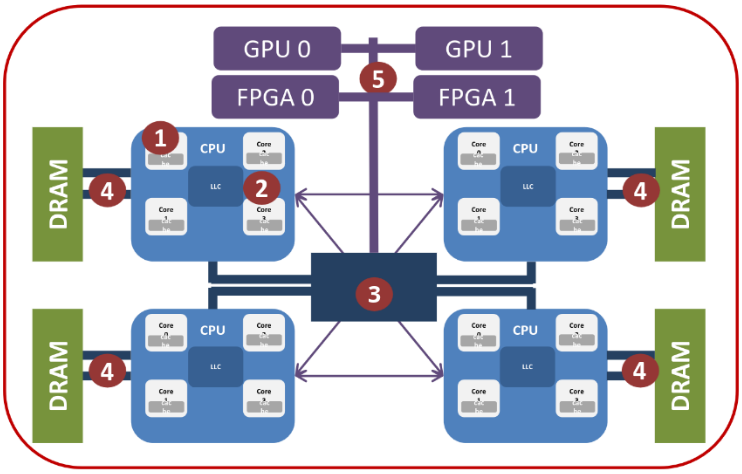

Figure 1.5: Levels of parallelism

1. The first level parallelism is in a single core of a CPU
2. The second level of parallelism is between cores of a single CPU
3. The third level of parallelism is introduced by inner cache levels
4. The fourth level of parallelism is between CPUs of a single node.
5. A node can also have accelerators, like GPUs or FPGAs which introduce another level of parallelism.

Draft

1.3 The Software Stack

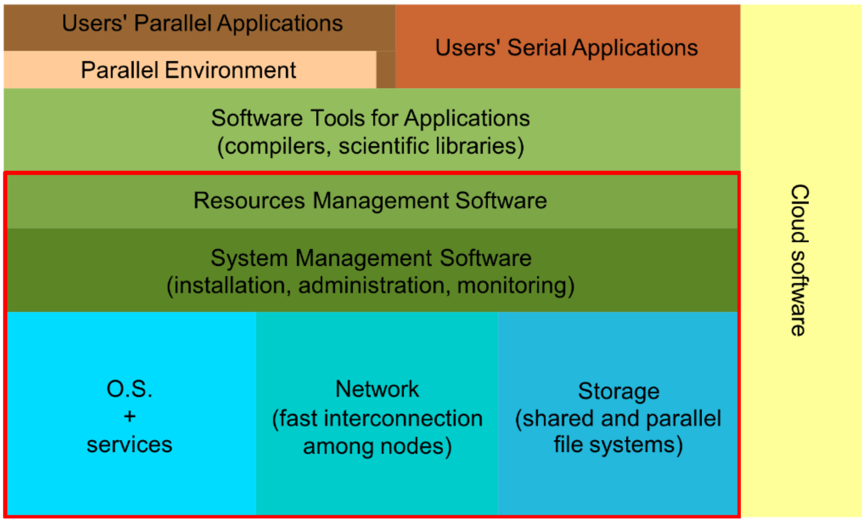
Users' Parallel Applications
Parallel Environment
Users' Serial Applications
Software Tools for Applications
(compilers, scientific libraries)
Resources Management Software
System Management Software
(installation, administration, monitoring)
O.S.
+
services
Network
(fast interconnection
among nodes)
Storage
(shared and parallel
file systems)
Cloud software

Figure 1.6: The Software Stack: in red the *cluster middleware*

The Cluster middleware

The cluster middleware (the one in the red box in Figure 1.6) is the software layer that sits between the hardware and the applications.

The cluster middleware includes administration software responsible for managing user accounts and network services such as NTP and NFS, ensuring system consistency and time synchronization. Additionally, it encompasses resource management and scheduling tools (LRMS) that efficiently distribute processes, balance system load, and schedule jobs for multiple tasks, thereby optimizing overall cluster performance.

Resource Management Problem

The Resource Management Problem in HPC environments centers around the efficient allocation of computing resources among competing users and applications. We have a pool of users and a pool of resources, but this alone is insufficient for effective operation. Three key software components bridge this gap: resource controllers that monitor and manage the available computational assets, scheduling systems that make intelligent decisions about which applications to execute based on resource availability and prioritization policies, and execution engines that handle the actual deployment and running of applications on the allocated hardware. This layered approach ensures optimal utilization of expensive HPC infrastructure while providing fair access to multiple users with diverse computational needs. The complexity of this management increases with system scale, particularly as modern supercomputers accommodate thousands of simultaneous users competing for limited computational resources.

Resources

HPC systems manage a variety of computational resources, including CPUs, memory, storage, network bandwidth, and specialized accelerators like GPUs and FPGAs. In modern supercomputing environments, resources are often virtualized and dynamically allocated based on workload demands. This approach enables flexible resource management but introduces additional complexity in tracking, optimizing, and maintaining the system. Resource management systems must balance competing priorities such as maximizing throughput, ensuring fairness among users, accommodating urgent jobs, and maintaining energy efficiency. As illustrated in Figure 1.7, these resources form an interconnected ecosystem where efficient allocation directly impacts overall system performance and user satisfaction.

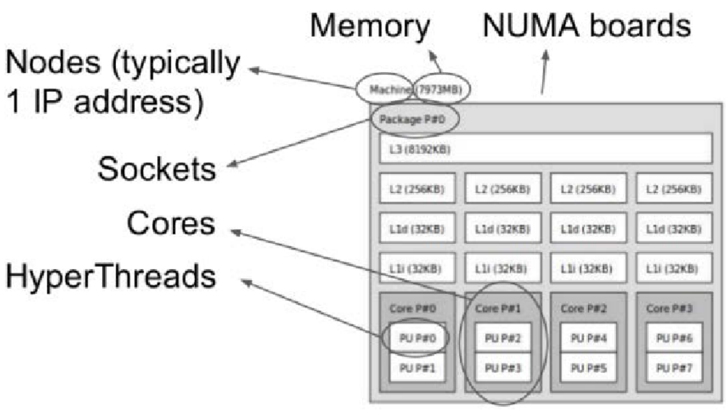

Figure 1.7: Resources in a HPC cluster

Definition: ***Scheduling***

Scheduling is the method by which work specified by some means is assigned to resources that complete the work

Some definitions of scheduling:

- **Batch Scheduler**: software responsible for scheduling the users' jobs on the cluster.
- **Resource Manager**: software that enable the jobs to connect the nodes and run
- **Node**: computer used for its computational power
- **Login/Master node**: it's through this node that the users will submit/launch/manage jobs

Batch Scheduler

The **batch scheduler** is a critical component of the HPC software stack, responsible for managing the allocation of computational resources to user applications. It serves as the primary interface between users and the underlying hardware, ensuring that jobs are executed efficiently and fairly. The batch scheduler receives job submissions from users, evaluates resource availability, and makes intelligent decisions about job placement and execution. By optimizing resource utilization and minimizing job wait times, the batch scheduler plays a central role in maximizing the overall performance of the HPC system.

The batch scheduler faces the challenging task of balancing multiple competing objectives:

- **User Satisfaction:** Allocating resources for applications according to their specific requirements and users' rights, while ensuring minimal response time and high reliability.
- **Administrative Efficiency:** Meeting administrative goals by maintaining high resource utilization, operational efficiency, and effective energy management.

This balancing act requires sophisticated algorithms and policies that can adapt to changing workloads and priorities within the HPC environment.

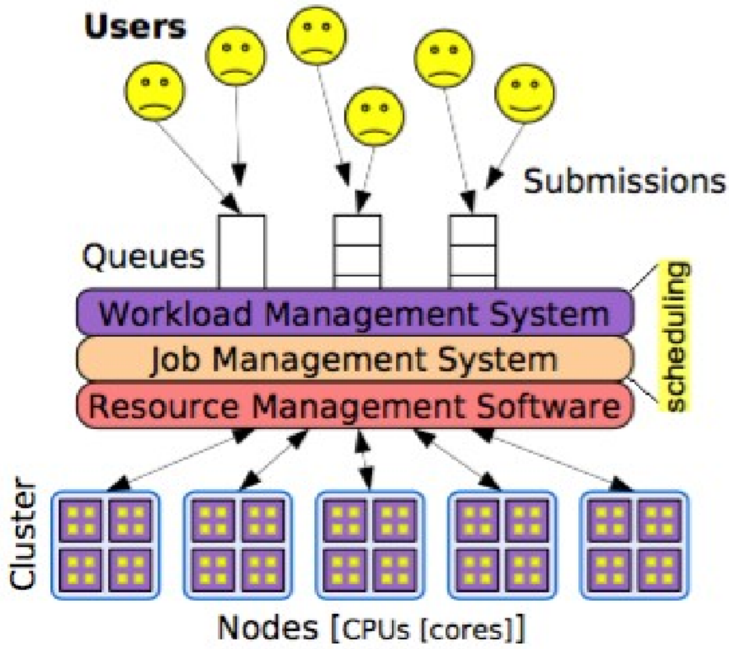

Figure 1.8: Batch Scheduler Architecture

Modern HPC schedulers typically implement a layered architecture:

- **Resource Management Layer:** Handles the low-level aspects of job execution, including launching processes, cleaning up after completion, and continuous monitoring of resource usage.
- **Job Management Layer:** Manages both batch and interactive jobs with features such as:
  - **Backfilling:** Filling idle resources with smaller jobs that won't delay scheduled larger jobs
  - **Advanced scheduling:** Optimizing job placement based on multiple constraints
  - **Job control:** Supporting suspend/resume operations and preemption when needed
  - **Workflow management:** Handling job dependencies and automatic resubmission
  - **Resource reservation:** Enabling advance booking of computational resources
- **Workload Management Layer:** Implements comprehensive scheduling policies including:
  - Fair-sharing mechanisms to allocate resources equitably among users and groups
  - Quality of Service (QoS) provisions for prioritizing critical applications
  - Service Level Agreement (SLA) enforcement to meet contracted performance metrics
  - Energy-saving strategies to optimize power consumption

In many large-scale HPC environments, this workload management functionality may be provided by dedicated software that interfaces with the underlying resource manager, creating a flexible and powerful scheduling ecosystem that can adapt to the specific needs of the organization.

1.3.1 Local Resource Manager

A Local Resource Manager System (LRMS) provides the critical interface between the computing resources and the users' workloads. These systems are responsible for allocating resources, launching jobs, tracking their execution, and managing the overall utilization of the HPC cluster.

Main LRMS packages

Several LRMS solutions have emerged to address the complex scheduling and resource management needs of HPC environments. Each offers different features, advantages, and licensing models:

- **IBM Platform LSF (Load Sharing Facility)**
  - Commercial solution with enterprise-level support
  - Offers advanced workload management capabilities for heterogeneous environments
  - Provides comprehensive policy management, reporting, and analytics
  - Notable for its fault tolerance and high availability features
- **Univa Grid Engine (UGE)**
  - Commercial solution that evolved from Sun Grid Engine (SGE)
  - Specializes in managing complex workloads across distributed computing resources
  - Features advanced job scheduling algorithms and resource allocation policies
  - Supports containerization and cloud integration
- **PBS Professional (PBSPRO)**
  - Originally commercial, now available in both open-source and commercial versions
  - Commercial support provided through Altair Engineering
  - Offers sophisticated scheduling capabilities for heterogeneous computing resources
  - Previously available on ORFEO but has been replaced
- **SLURM (Simple Linux Utility for Resource Management)**
  - Open-source solution with commercial support options
  - Currently deployed on ORFEO for student access
  - Highly scalable and fault-tolerant architecture
  - Used by many of the world's top supercomputers

SLURM in Depth

SLURM's development began in 2002 at Lawrence Livermore National Laboratory, where it was originally designed as a resource manager for Linux clusters. The name initially stood for *Simple Linux Utility for Resource Management*. The system evolved significantly over time, with advanced scheduling plugins being added in 2008 to enhance its capabilities. Today, SLURM consists of approximately 550,000 lines of C code and maintains an active global user community and development ecosystem that continues to improve and extend its functionality.

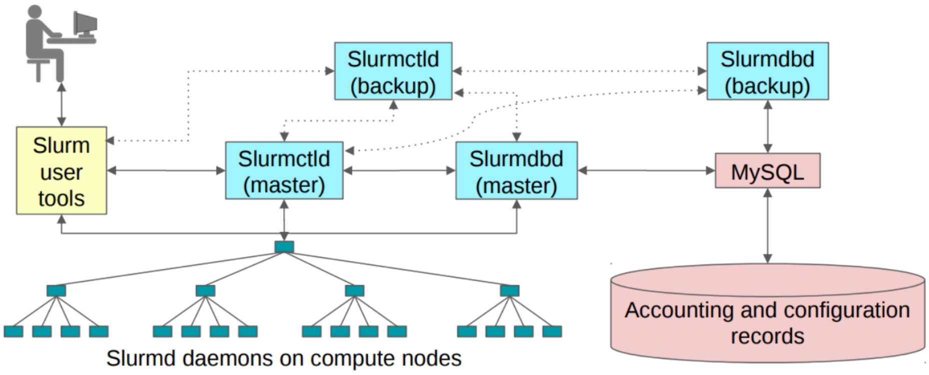

Figure 1.9: Simplified SLURM Architecture

Key Entities in SLURM:

- **Jobs:** Resource allocation requests that define the computational resources required
  - Specified through command-line tools or script directives
  - Include parameters such as required nodes, cores, memory, and time limits
- **Job Steps:** Sets of (typically parallel) tasks within a job allocation
  - Usually correspond to MPI applications or multi-threaded programs
  - Utilize resources from the parent job's allocation
  - Multiple steps can execute sequentially or concurrently within a job
  - Offer lower overhead than full job submissions
- **Partitions:** Job queues with specific limits and access controls
  - Configure access policies and resource limits for different user groups
  - Enable prioritization of workloads based on organizational needs
  - Allow for specialized hardware to be allocated to appropriate jobs
- **QoS (Quality of Service):** Defines limits, policies, and priorities
  - Controls maximum resource allocation per user or group
  - Enforces priorities between competing workloads
  - Implements site-specific policies for resource allocation
  - Provides mechanisms for preemption and resource guarantees

Architecture Components:

- **slurmctld** - Central controller daemon managing the overall state of the cluster
- **slurmd** - Node-level daemon running on each compute node
- **slurmdbd** - Optional database daemon for accounting records
- **User commands** - Tools like `sbatch`, `srun`, `squeue`, and `scancel` for job submission and management

SLURM's modular design allows for customization through plugins, making it adaptable to various hardware configurations and scheduling policies. Its scalability has been demonstrated on systems with over 100,000 compute nodes, making it suitable for the largest supercomputing installations in the world.

1.3.2 Scientific Software

…

1.3.3 Complilers

High level languages need to be compiled to a stream of machine instructions that can be executed by the CPU. The **compiler** is the software that does this job.

In HPC environments, the choice of compiler can significantly impact application performance. Several options are available:

Free Compilers: GNU Suite

- Always available on virtually all Linux/Unix platforms
- Includes GCC (C/C++) and GFortran (Fortran) compilers
- Multiple versions with varying feature support
- Fundamental and reliable, but may lack performance optimizations for specific architectures

- Open source with strong community support

Commercial Compilers: Intel Suite

- Provides a comprehensive software stack including:
  - Highly optimized C, C++, and Fortran compilers
  - Performance libraries (MKL, IPP, TBB)
  - Profiling and benchmarking tools
  - MPI implementations
- Specifically optimized for Intel architectures
- Often delivers superior performance for floating-point computations
- Excellent vectorization capabilities

NVIDIA HPC SDK (formerly PGI)

- Strong compiler suite with good performance characteristics
- Features valuable HPC extensions:
  - OpenACC directives for GPU programming
  - CUDA Fortran for direct GPU programming from Fortran
  - Advanced optimization capabilities
- Community edition available for free
- Particularly well-suited for heterogeneous CPU/GPU computing

The choice of compiler depends on various factors including the target architecture, specific performance requirements, and available budget. Many HPC centers provide multiple compiler options, allowing users to select the most appropriate tool for their particular application requirements.

Observation: ***ORFEO Compiler***

On ORFEO there is an openMPI installation, which includes the GNU compilers.

1.3.4 Libraries

…

...

1.4 Cache Mapping

Modern systems subdivide both RAM and cache into equally sized blocks (often called *lines*). For instance, a 64 B block is commonly used: when a byte is requested, the entire 64 B block containing that byte is fetched into the cache. This approach reduces the overhead of fetching data from memory and exploits spatial locality.

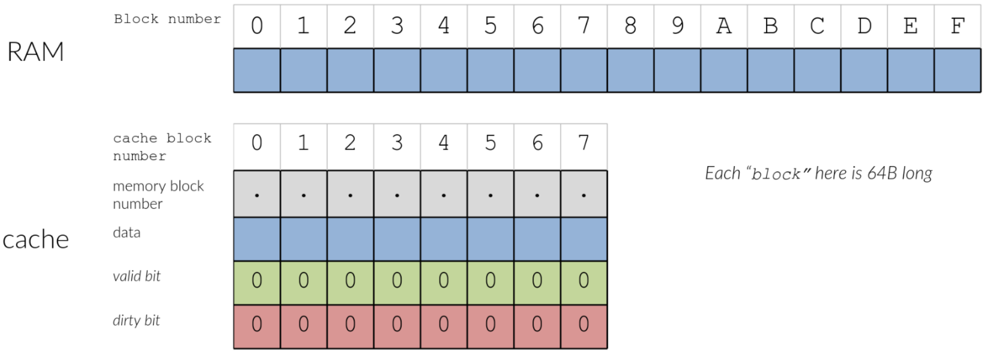

Figure 1.10: Main memory and cache are both split into blocks (lines).

There are three primary strategies for mapping main memory blocks into cache:

- **Fully Associative Mapping**
- **Direct Mapping**
- **N-way (Set) Associative Mapping**

Each strategy differs in how flexible it is in placing a main memory block into cache and how it handles conflicts when multiple blocks compete for the same cache location. A high-level comparison is shown in Figure 1.11.

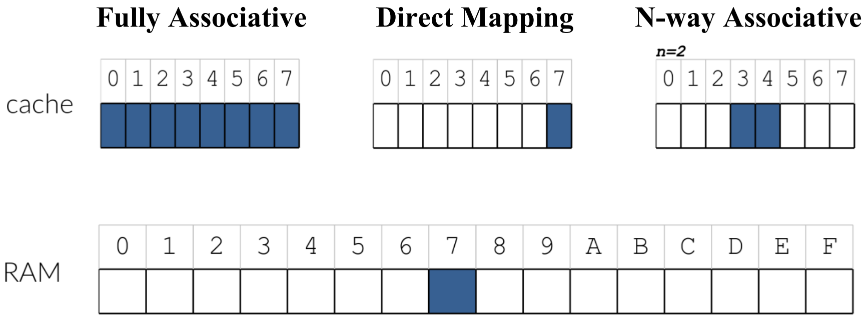

Figure 1.11: Fully Associative, Direct, and N-way Set Associative mappings.

Fully Associative Mapping

In a **fully associative** cache, each block of main memory ***can be placed in any block*** of the cache. This offers maximum flexibility because there is no restriction on which cache line a particular memory block can occupy. However, this scheme also requires more complex hardware for searching (to locate a given address in any cache line), making it more expensive.

Pros

- Minimizes conflicts during *writes*, as any free cache line can be used.
- Offers the greatest flexibility in block placement.

Cons

- Potentially inefficient for *reads* because the requested data could be in any cache line, requiring a more expensive search mechanism (e.g., parallel or associative search).
- Higher hardware complexity and cost (larger tag comparators, fully associative lookups).

Direct Mapping

In a **direct-mapped** cache, each block of main memory ***can be placed in exactly one block*** of the cache. This mapping is determined by some bits of the memory address (e.g., index bits), which directly select the cache line. It is the simplest scheme in terms of hardware.

Pros

- Very efficient in locating or writing data because each memory block maps to a single known location (no associative search needed).
- Simpler and cheaper hardware implementation.

Cons

- Maximizes cache conflicts when multiple addresses map to the same cache line (known as *conflict misses*).
- A single cache line might be heavily contended by multiple memory blocks, reducing overall hit rate.

N-way Set Associative Mapping

In an **N-way set associative** cache, the cache is divided into sets, each containing *N* lines. A block of main memory ***can be placed in any of the*** *N* ***lines*** within the specific set dictated by the address. This approach strikes a balance between fully associative and direct-mapped schemes.

Pros

- Reduces conflict misses compared to direct mapping by allowing multiple possible lines in each set.
- Lower hardware complexity than fully associative (search is limited to *N* lines in the set, not the entire cache).

Cons

- More complex than direct mapping (requires searching up to *N* lines in the set).
- Additional hardware and logic needed to manage the multiple lines per set.

The choice of cache mapping strategy involves a trade-off between hardware complexity, access speed, and conflict rate.

…

2
Lecture 25/03/2025

Coherency problem

We saw that:

Symmetric Multiprocessing (SMP)
In SMP systems, multiple processors share a common physical memory, enabling concurrent access to shared data and resources.

Distributed NUMA Each CPU has its own local memory, but can also access the memory of other CPUs albeit with increased latency due to the non-uniform memory access characteristics.

—

When a CPUs wants to modify a value in memory, they firstly need to copy that value in a register, perform the operation and then write it back in memory.

If we are in a shared memory environment, and two or more CPUs are trying to modify the same value, we have a problem: the value in memory is not updated until the CPU writes it back. This can lead to a situation where the CPUs are working on an outdated value.

—

Draft

The cpu is able to perform multiple operations at the same time. This is possible because the CPU has a pipeline divided in stages. Each stage is responsible for a specific operation. The CPU can start a new operation in a new stage before the previous operation is completed.

If many independent logical units exist to perform each step, they could operate subsequently on different instructioms:

If the stage delays are not uniform, the throughput is limited by the latency:

$$F + (D+E) - (F+D) = E \sim 220ps$$

which means we have a throughput of $\sim 4.5GHz$.just becouse of logic units separation.

Therefore, introducing the instruction pipelining, we can increase the throughput of our system by a large factor:

missing something here

Modern computers have multiple pipelines, each one dedicated to a specific type of instruction (int op, float op, memory, ...).

—

Modern CPUs have ”***vector registers***”, which are registers that can store multiple values at the same time. This allows the CPU to perform the same operation on multiple values at the same time, increasing the throughput of the system.

3 branches

Whenever either

1. the sequence of operations that must be executed
2. the sequence of data to be processed depends on some condition, i.e. on the outcome of a test performed onsome data or result

we have a **conditional execution**.

Modern architecture offer 2 distinct low-level instructions to implement a conditional execution upon a test:

- modifying the control flow → data-dependent execution
- modifying the data flow → data-dependent data-flow

At machine level, the way to alter the execution flow is through a **jump instruction**, that causes the control to be passed to a different code section. The jump instruction can be ***conditional***, when its execution depends on the outcome of some operation (a test), or ***unconditional*** if it is not.

Example: ***Example***

Let's consider a simple snippet of code in C and its corresponding assembly code:

```c
if (a < b) {
    c = a + b;
} else {
    c = a - b;
}
```

```asm
    mov   eax, DWORD PTR [rbp-8]   ; moves a to eax
    cmp   eax, DWORD PTR [rbp-4]   ; compares a and b
    jge   .L2                      ; jumps to L2 if a >= b

    mov   edx, DWORD PTR [rbp-8]   ; moves a to edx
    mov   eax, DWORD PTR [rbp-4]   ; moves b to eax
    add   eax, edx                 ; adds a and b
    mov   DWORD PTR [rbp-12], eax  ; moves res to c
    jmp   .L3                      ; jumps to .L3
.L2:
    mov   eax, DWORD PTR [rbp-8]   ; moves a to eax
    sub   eax, DWORD PTR [rbp-4]   ; sub. b from a
    mov   DWORD PTR [rbp-12], eax  ; moves res to c
.L3:
    ; (rest of the code, if any)
```

Note: The true branch is the closest to the test condition, while the false branch is reached upon a jump.

Tip: ***Branch prediction***

When coding, if possible pay attention to what is most likely to be true, to preserve the code locality: *It is possible to suggest to compiler which branch will most probably be true*

Bibliography

[1] *High Performance Computing — digital-strategy.ec.europa.eu*. https://digital-strategy.ec.europa.eu/en/policies/high-performance-computing. [Accessed 03-03-2025].

Draft