UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

High Performance Computing

Lecturer:
Prof. Stefano Cozzini

Authors:

Andrea Spinelli
Christian Faccio

March 17, 2025

github.com/Spina02,christianfaccio

andreaspinelli2002@gmail.com,christianfaccio@outlook.it

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.

This comprehensive approach not only equips you with technical skills but also fosters a critical perspective on technological innovation in computing.

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.

Contents

1 Introduction

1.1 Base Concepts

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

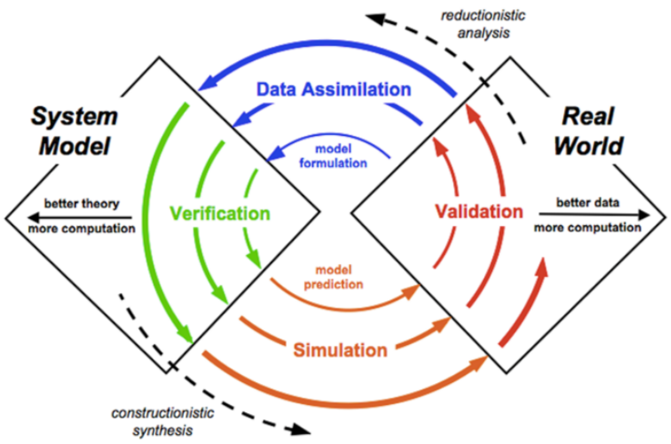


Figure 1.1: Research and Development

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. [**europaHighPerformance**]

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.

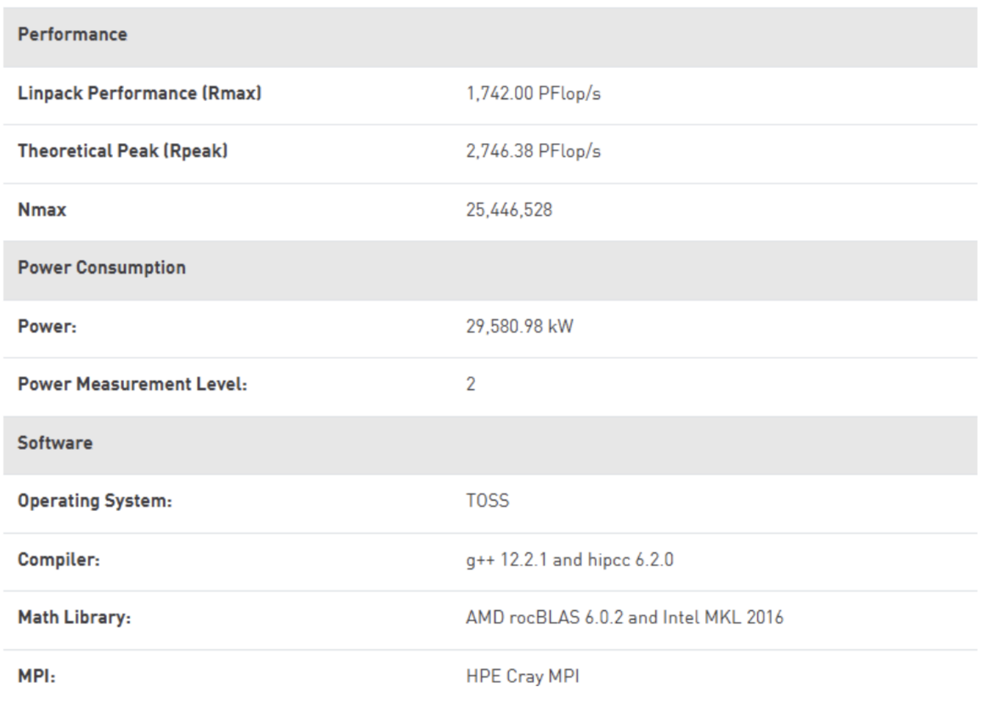
Sustained performance refers to the effective throughput that an HPC system attains when executing actual workloads. Since it is challenging to exactly measure the number of floating point operations performed by every application, standardized benchmarks are commonly used to assess this performance. 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 Supercomputers and TOP500

Supercomputers are the most powerful and advanced computing systems available today. They are designed to handle the most complex and demanding computational tasks, such as weather forecasting, climate modeling, and nuclear simulations. Supercomputers are typically used in scientific research, engineering, and other fields that require massive computational power. The TOP500 list (www.top500.org) is a ranking of the world's most powerful supercomputers. It is published twice a year and provides a snapshot of the current state of high-performance computing. The list ranks supercomputers based on their performance on the Linpack benchmark, which measures the speed at which a system can solve a dense system of linear equations. The TOP500 list is widely regarded as the most authoritative ranking of supercomputers and is used by researchers, industry professionals, and policymakers to track trends in high-performance computing.

The **HPL Linpack benchmark** measures how fast a system can solve a large dense system of linear equations. It estimates floating-point performance by stressing both CPU and memory



Performance	
Linpack Performance (Rmax)	1,742.00 PFlop/s
Theoretical Peak (Rpeak)	2,746.38 PFlop/s
Nmax	25,446,528
Power Consumption	
Power:	29,580.98 kW
Power Measurement Level:	2
Software	
Operating System:	TOSS
Compiler:	g++ 12.2.1 and hipcc 6.2.0
Math Library:	AMD rocBLAS 6.0.2 and Intel MKL 2016
MPI:	HPE Cray MPI

Figure 1.2: Best Supercomputers in the World at the moment (March 2025)

resources, making it a standard method for comparing supercomputers. HPL results are reported in floating-point operations per second (FLOPS), reflecting sustained system throughput under a realistic workload. This test underpins the widely recognized TOP500 list, where higher HPL scores signify more capable machines in real-world, data-intensive scenarios.

For each machine the following numbers are reported using HPL:

- **Rmax:** The maximum performance achieved by the system.
- **Rpeak:** The theoretical peak performance of the system.
- **Efficiency:** The ratio of Rmax to Rpeak, indicating how effectively the system utilizes its computational resources.

What about the AI workload? Well, floating point in TOP500 is double precision, but for AI works we need single or half precision. So, the TOP500 list is not the best for AI workload. Moreover, data movement is not considered in TOP500, but it is very important for AI workload.

2 Hardware

In the classical Von Neumann architecture there is only one processing unit (CPU) that processes instructions, the ALU is responsible for math and logic operations and the register stores data.

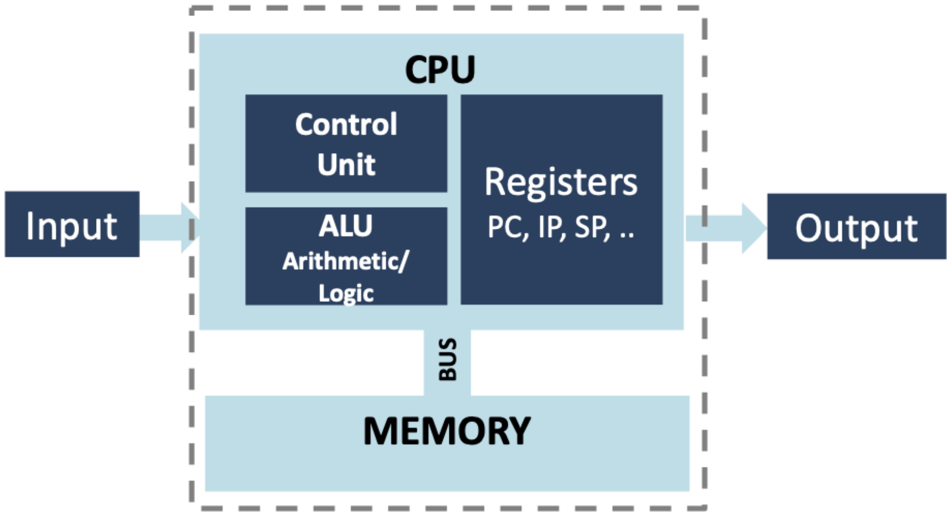


Figure 2.1: Von Neumann architecture

One instruction is executed at a time, the CPU fetches the instruction from the memory, decodes it and executes it. The CPU can access the memory to read or write data. Accessing any location in the mempory has always the same cost, this is called the **uniform memory access** (UMA).

2.1 Moore's Law

Definition: ***Moore's Law***

It states that the number of transistors in a dense integrated circuit doubles about every two years.

How can we go from Moore's Law to processor performance? Through Dennard Scaling:

”Power density stays constant as transistors get smaller”

Intuitively,

- **Smaller transistors** $\rightarrow$ shorter propagation delay $\rightarrow$ faster frequency
- **Smaller transistors** $\rightarrow$ smaller capacitance $\rightarrow$ lower voltage

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

But... even with smaller transistors, we cannot continue reducting power, there are then two options:

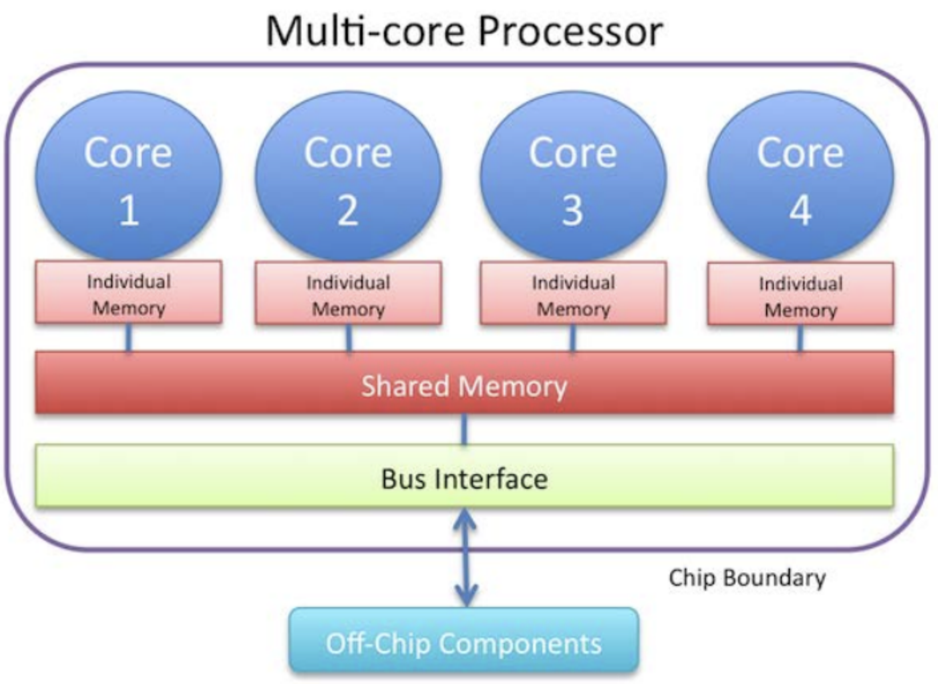


Figure 2.2: Multicore processors

- **Increase power**
- **Stop frequency scaling**

From 2006, single-core performance stopped increasing, so the only way to increase performance is to use more cores. The first solution is to write efficient software to make the efficient use of hardware resources. The second is to use specialized architectural solutions, like GPUs, FPGAs, etc.

Today, **CPUs are multicore processors**, lowering clock frequency because of power and heat dissipation, but packing more computing cores onto a chip. These cores will share some resources (memory, network, disk, . . .) but are still capable of independent calculations.

2.2 Parallel Computers

Flynn Taxonomy (1966) is used to classify parallel computers based on the number of instruction streams and data streams.

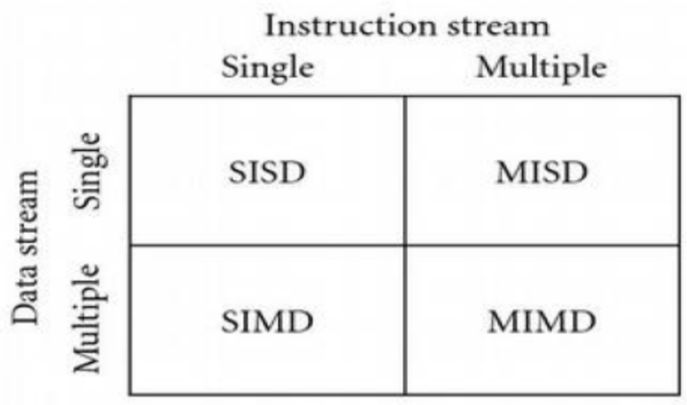
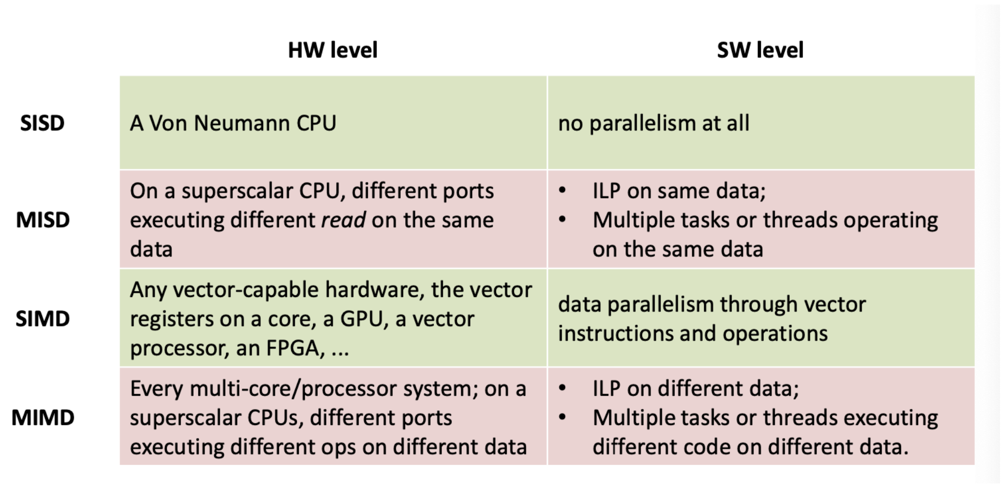


Figure 2.3: Flynn Taxonomy



	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, the vector registers on a core, a GPU, a vector processor, an FPGA, ...	data parallelism through vector instructions and operations
MIMD	Every multi-core/processor system; on a superscalar CPUs, different ports executing different ops on different data	• ILP on different data; • Multiple tasks or threads executing different code on different data.

Figure 2.4: Parallel computers

The essential components of a HPC cluster are:

- **Compute nodes**: the actual computers that perform the calculations
- **Interconnect**: the network that connects the compute nodes
- **Storage**: the disk space where data is stored
- **Software**: the operating system and the software stack that runs on the cluster

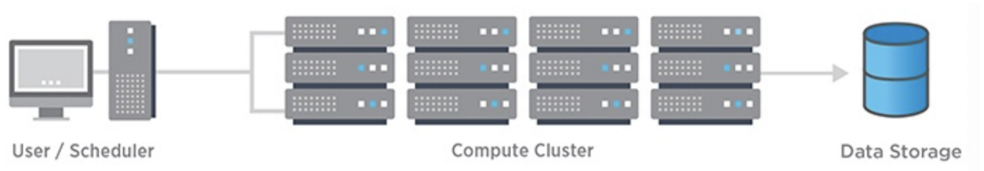


Figure 2.5: HPC cluster

A core is the smallest unit of computing, having one or more threads and is responsible for executing instructions.

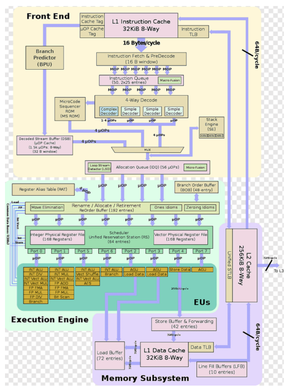


Figure 2.6: Core

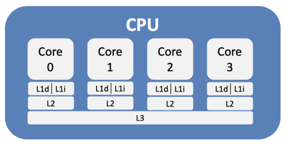


Figure 2.7: Cache hierarchy

Cache hierarchy can have different topologies.

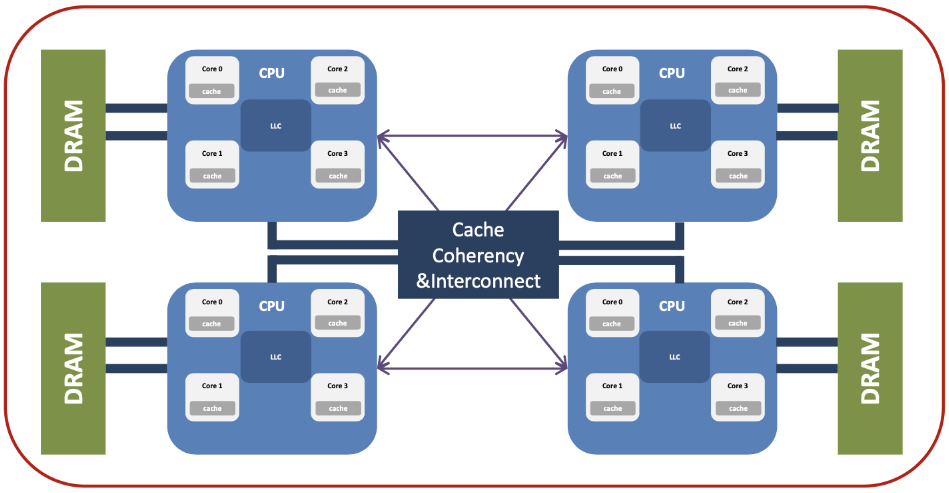


Figure 2.8: Node topology

Tip:

Cache hierarchy has been invented cause accessing memory (moving data) takes almost 100x times computing (doing operations). The cache is a small memory that stores the most frequently accessed data. The cache is faster than the main memory but smaller. The cache is divided into levels, the first level is the fastest but the smallest, the second level is slower but bigger, and so on.

In a cluster there are different types of networks:

- **High-speed network**: used for communication between nodes (parallel computation, low latency/high bandwidth, infiniband or ethernet as examples)
- **I/O NETWORK**: used for communication with storage (I/O requests, latency not fundamental/-good bandwidth, NFS, Lustre, GPFS as examples)
- **In band Management Network**: used for cluster management, monitoring, and control, LRMS (Load Resource Management System) as examples
- **Out of band Management Network**: used for cluster management, remote control of nodes and any other device, IPMI (Intelligent Platform Management Interface) as examples

What about **memory**?

It is fundamental and on a supercomputer there is a hybrid approach as for the memory placement. The memory on a single node can be accessed directly by all the cores on that node (**shared-memory**). When many nodes are used at a time, a process cannot directly access the memory on a different node, it needs to issue a request for that. That is named **distributed-memory**.

Shared Memory

- **Uniform Memory Access (UMA)**:
  Each processor has uniform access to memory. Also known as symmetric multiprocessors (SMP).

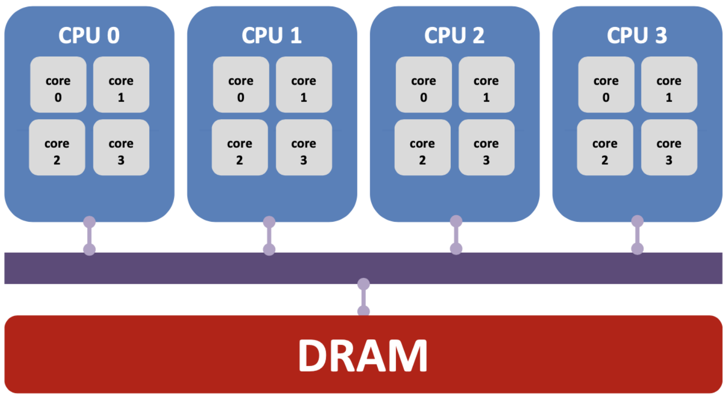


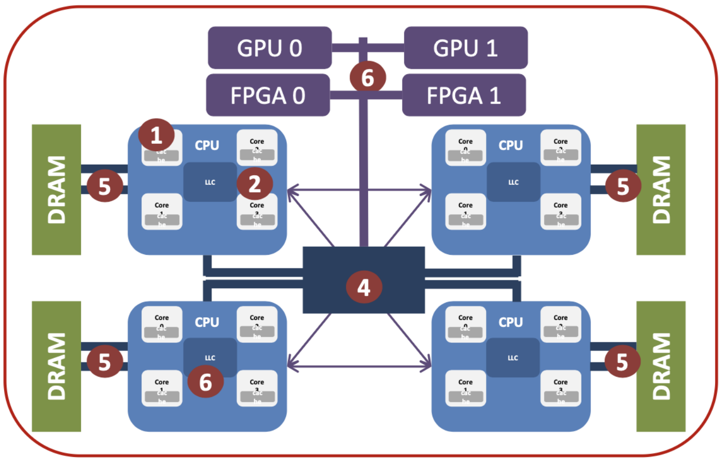
Figure 2.9: Shared memory - UMA

- **Non-Uniform Memory Access (NUMA)**: Time for memory access depends on location of data. Local access is faster on non-local access.

Warning: ***Challenges for multicore***

It aggravates the **Memory Wall problem**, where the memory access time is much slower than the CPU speed.

- **Memory bandwidth**: the memory bandwidth is limited and shared among all the cores
- **Memory latency**: the memory latency is high and can be a bottleneck
- **Memory contention**: the memory contention can be a problem when multiple cores access the same memory location



1. ILP/SIMD units
2. Cores
3.
4. Socket/ccNuma domains
5. Inner cache levels
6. Multiple accelerators

Figure 2.10: Parallelism within a HPC node

3
HPC Software Stack

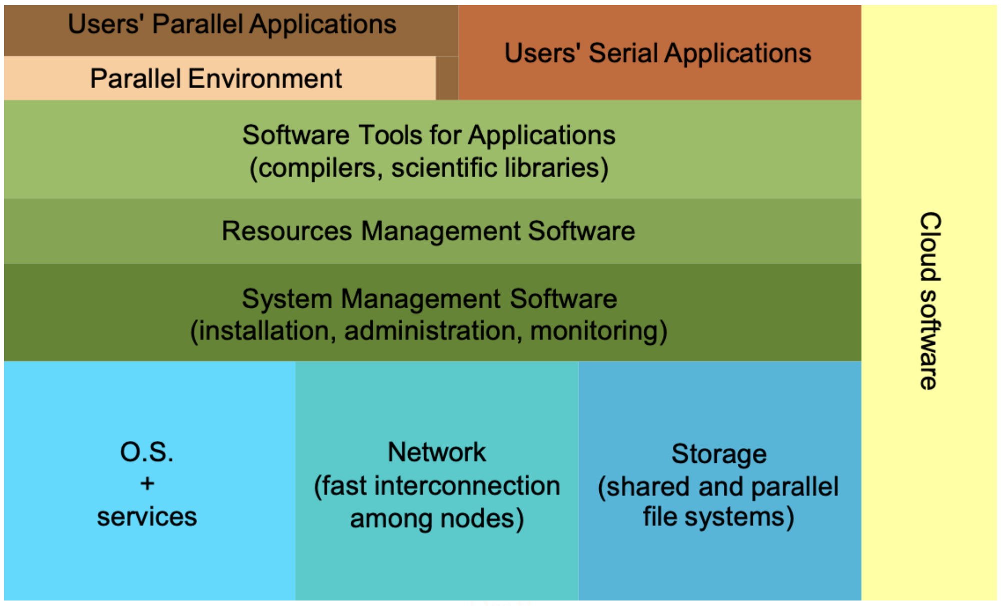

Users' Parallel Applications
Parallel Environment
Users' Serial Applications
Software Tools for Applications
(compilers, scientific libraries)
Resources Management Software
System Management Software
(installation, administration, monitoring)
O.S.
+
services
Network
(fast interconnection
among nodes)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

Figure 3.1: What we need

We refer to **Cluster Middleware** as the software stack that is used to manage the cluster. It is composed of several layers, each one with its own purpose. The first layer is the **Operating System**, which is the base layer of the stack. The second layer is the **Resource Manager**, which is used to manage the resources of the cluster. The third layer is the **Job Scheduler**, which is used to schedule the jobs on the cluster.

- The **Administration Software** is used to manage the cluster. It is used to install, configure, and monitor the cluster. It is also used to manage the users and groups on the cluster.
- The **Resource management and scheduling software (LRMS)** takes care of having many users and distribute process, balances the load among different users and schedules multiple tasks.

3.1 Resource Management Problem

Once we have a pool of users and a pool of resources, then some software is needed to control the available resources, to decide which application to execute based on available resources and to execute applications.

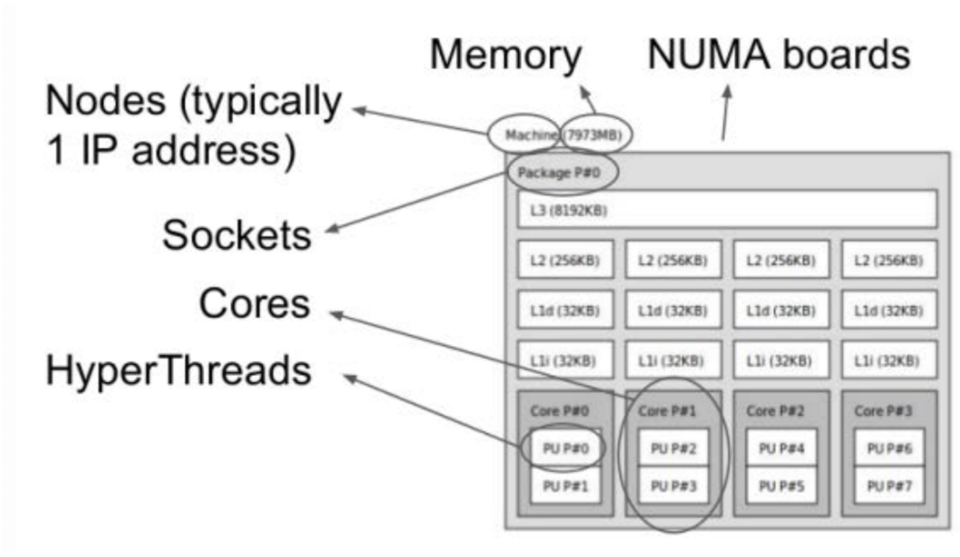


Figure 3.2: HPC resources

plus:

- network resources
- GPU/Accelerator
- Software resources

Let's now define some terms:

- **Batch Scheduler**: a software that manages the execution of jobs in a batch mode. It is used to schedule the jobs on the cluster.

Observation: ***Scheduling***

`Scheduling` is the process of assigning resources to jobs.

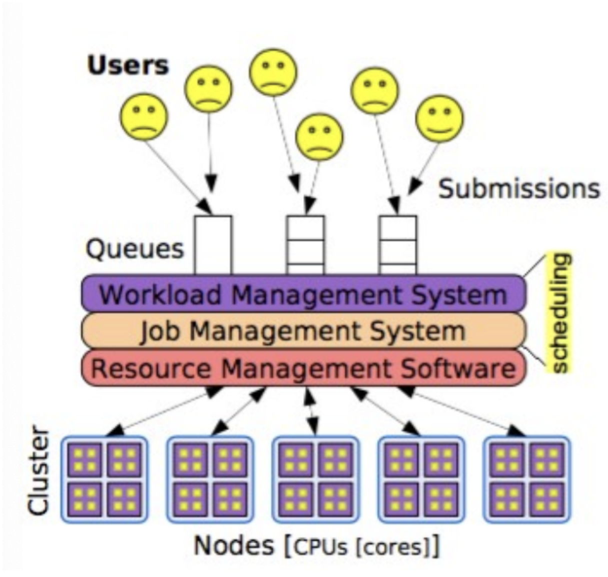


Figure 3.3: Batch scheduler

- **Resources Manager**: software that enable the jobs to connect the nodes and run.

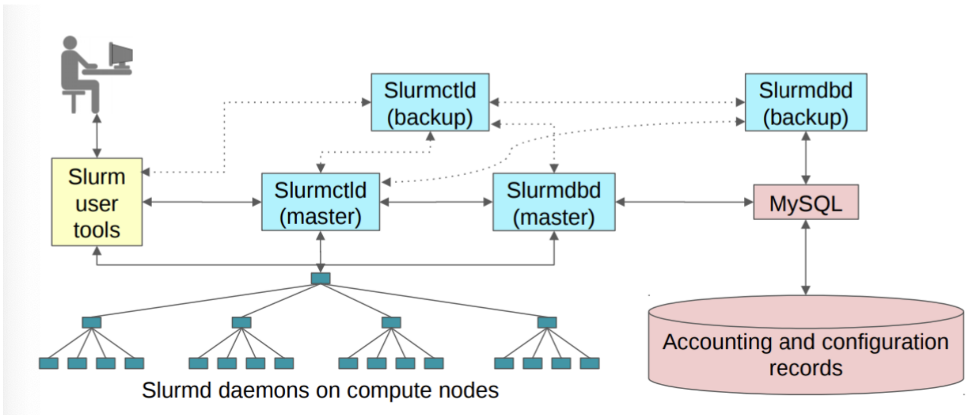


Figure 3.4: SLURM architecture

- **Node (aka Computing Node)**: computer used for its computational power.
- **Login/Master node**: it's through this node that the users will submit/launch/manage jobs.

Definition: ***SLURM***

SLURM (Simple Linux Utility for Resource Management) is an open-source job scheduler for Linux and Unix-like kernels, used by many of the world's supercomputers and computer clusters.

- **Jobs**: Resource allocation requests.
- **Job Steps**: A job can be divided into multiple steps:
  - Typically an MPI and/or multi-threaded application program
  - Allocated resources from the job's allocation
  - A job that contains multiple job steps which can execute sequentially or concurrently
  - Lighter weight than jobs
- **Partitions**: Job queues with limits and access control
- **Qos**: Limits and policies

Table 3.1: SLURM Jobfile Parameters

Directive	**Meaning**
`#!/bin/bash`	Shebang directive indicating the shell to use
`#SBATCH --job-name=<job_name>`	Assigns a name to the job
`#SBATCH --output=<file.out>`	Defines the output file
`#SBATCH --error=<file.err>`	Defines the error file
`#SBATCH --ntasks=<num_tasks>`	Specifies total number of tasks
`#SBATCH --cpus-per-task=<cpus>`	Sets desired CPUs per task
`#SBATCH --mem=<memory>`	Requests memory per node or per CPU option
`#SBATCH --time=<HH:MM:SS>`	Sets maximum runtime
`#SBATCH --partition=<partition>`	Selects partition (queue)
`#SBATCH --qos=<qos_class>`	Specifies quality of service class

3.2 Scientific Software

This refers to the software above the Middleware:

- User's application (parallel and serial)
- Parallel Libraries and Tools
- Mathematical/Scientific Libraries
- I/O libraries
- Compilers

Here there is not so much standardization in HPC: every machine/app has a different software stack. Every code is optimized specifically for the hardware it runs on. This creates the so called **Dependency Hell**.

Definition: ***Dependency Hell***

Dependency hell is a colloquial term for the frustration of some software users who have installed software packages which have dependencies on specific versions of other software packages.

Thus, scientific software is:

- Generally available cluster-wide
- Installed in /opt/cluster/software (or similar) and mounted read-only on the nodes via nfs
- Generally managed by modules packages
- Several versions managed by some agreement

4 Optimization

4.1 Preliminaries

The objective of this chapter is to provide a general overview over how to optimize the code on single-core.
High Performance Computing requires, by the name itself, to squeeze the maximum effectiveness from your code on the machine you run it.
"Optimizing" is, obviously, a key step in this process.

Observation: ***Optimization***

Premature optimization is the root of all evil.

Which means that even if some of the stuff you'll learn may sound cool, you first focus must be in:

- the **correctedness** of your code,
- the **data model**,
- the **algorithm** you choose.

You'd better start thinking in terms of "improved" code.

Do neither add unnecessary code nor duplicate code.

- **Unnecessary code** icreases the amount of needed work to maintain the code (debugging or updating it) or to extend its functionalities
- **Dupliated code** increases you bad technical debt, that alreasy has a large enough number of sources.

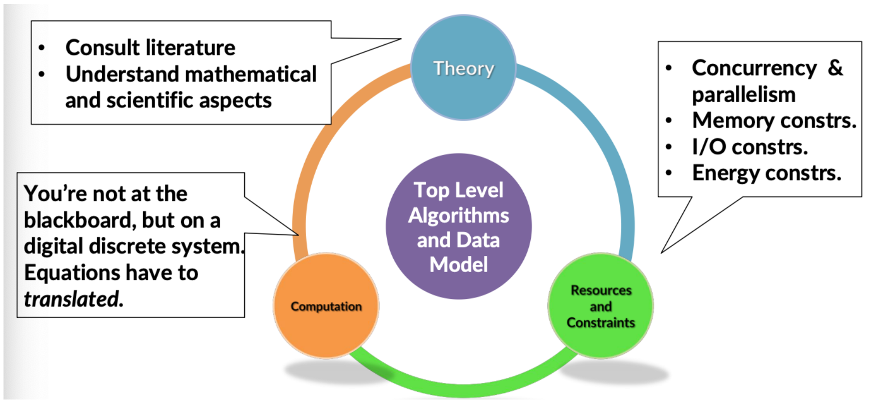

• Consult literature
• Understand mathematical and scientific aspects
Theory
• Concurrency & parallelism
• Memory constrs.
• I/O constrs.
• Energy constrs.
You're not at the blackboard, but on a digital discrete system. Equations have to *translated*.
Top Level Algorithms and Data Model
Computation
Resources and Constraints

Figure 4.1: Optimization

- **Testing** is part of the design, and it is a key step in the optimization process.
- **Validation** ensures that the code does what it was meant to do, and ensures the results are correct

First things first

1. The first goal is to have a program that delivers the **correct answers** and behaves correctly under all conditions.
The code must be as much clear, clean, concise, and documented as possible.
2. The first step towards optimization is to adopt the **best-suited algorithms** and data structures. What ”best-suited” means must be related to the constraints framing your activity (time-to-solution, energy-to-solution, memory, ...).
3. The second step is that the **source code be optimizable by the compiler**.
Then, you must have a firm understanding of the compiler’s capabilities and limitations, as well as those of your target architecture.
Understand the best trade-off between portability and ”performance” (accounting for the human effort into the latter).
4. The third step is to get a **data-driven, task-based workflow**, which possibly, almost certainly, will be parallel in either distributed- or shared-memory paradigms, or both.
5. Profile the code under different conditions (workload/problem size, parallelism, platforms, ...) and **spot bottlenecks and inefficiencies**.
6. Apply optimization techniques by modifying hot-spots in your code to **remove optimization blockers** and/or to better expose intrinsic instruction/data parallelisms.
7. `IF (needed) GOTO point 1.`

That is not a simple and linear process. Optimizing a code may require several trial-and-error steps, and modern architectures evolve so fast that modeling a code’s performance accurately can be challenging. Even promising techniques may sometimes fail.

Observation: ***Compiler job***

Recalling the job of the compiler:

- **Syntax analysis** (parsing)
- **Semantic analysis** (type checking)
- **Intermediate code generation**
- **Optimization**
- **Code generation**

The compiler is a tool that can help you in the optimization process.
It is also able to perform **sophisticated analysis** of the source code so that to produce a target code (usually assembly) which is highly optimized for a given target architecture.

To optimize your code through the compiler, use `-O3` flag, where the number indicates the level of optimization.
`-O3` is the highest level of optimization, and it is the most aggressive one.
It is not granted, though, that the highest level of optimization is the best one for your code.
For instance, sometimes expensive optimization may generate more code that on some architecture (e.g. with smaller caches) run slower, and using `-Os` may be better.
Obviously, the compiler knows the architecture it is compiling on, but it will generate a **portable** code, i.e. a code that can run on any cpu belonging to that class of architecture. Using appropriate switch (in gcc `-march=native -mtune=native`), the compiler will generate code that is optimized for the specific architecture it is running on.

Profile-guided optimization is a technique that uses the information gathered by the compiler when the code is run to optimize the code. Compilers are able to instrument the code so to generate run-time information to be used in a subsequent compilation.
Knowing the typical execution patterns enables the compiler to perform more focused optimizations, especially if several branches are present.

```
gcc -O3 -fprofile-generate -o myprog myprog.c
./myprog
gcc -O3 -fprofile-use -o myprog myprog.c
```

Observation: ***Optimization blockers***

Optimization blockers are those parts of the code that prevent the compiler from applying optimizations.
They can be:

- **Aliasing**: when two pointers point to the same memory location
- **Loop-carried dependencies**: when a loop iteration depends on the result of the previous one
- **Function calls**: when the compiler cannot inline the function
- **Memory access patterns**: when the memory access pattern is not predictable

Memory Aliasing

Memory Aliasing necessitates some attention.
We said that it refers to the situation where two pointers point to the same memory location.
This is a problem because the compiler cannot assume that the memory location is not modified by the other pointer, and it must reload the value from memory each time it is accessed. Help your C compiler in doing the best effort, either writing a clean code or using restrict or using `-fstrict-aliasing -Wstrict-aliasing` options.

Example: ***Memory aliasing***

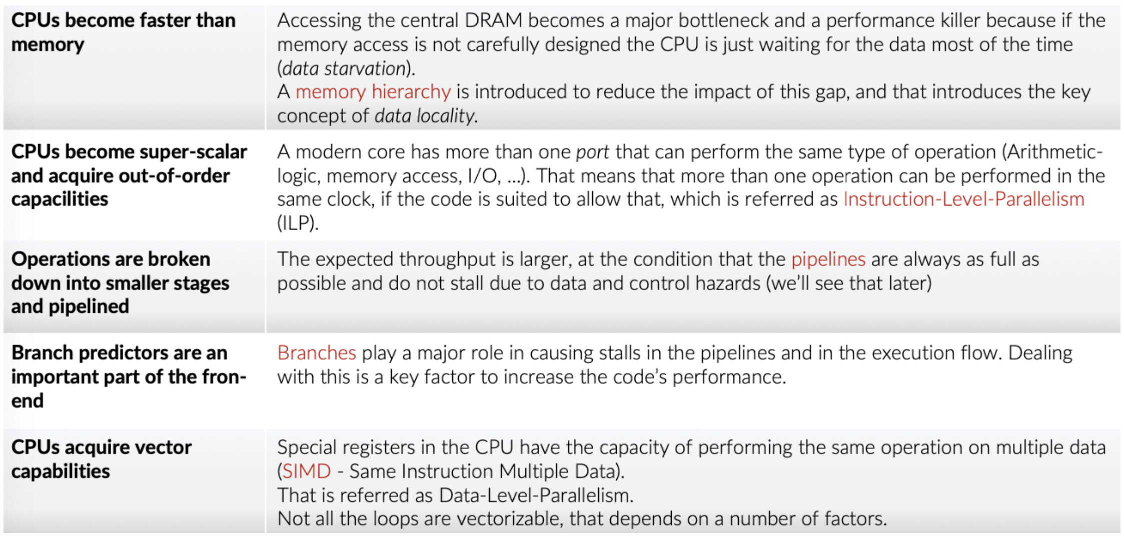
```c
        void my_fun(double *a, double *b, int n) {
            for (int i = 0; i < n; i++) {
                a[i] = b[i] + 1.0;
            }
        }

        // can be optimized to
        void my_fun(double *restrict a, double *restrict b, int n)
            {
            for (int i = 0; i < n; i++) {
                a[i] = b[i] + 1.0;
            }
        }
```

Now you are telling the compiler that the memory regions referenced by a and b will never overlap. So, it will feel confident in optimizing the memory accesses as much as it can (basically avoiding to re-read locations).

4.1.1 Modern Architectures

This section presents the fundamental traits of the **single-core** modern architectures.



CPUs become faster than memory	Accessing the central DRAM becomes a major bottleneck and a performance killer because if the memory access is not carefully designed the CPU is just waiting for the data most of the time (*data starvation*). A memory hierarchy is introduced to reduce the impact of this gap, and that introduces the key concept of *data locality*.
CPUs become super-scalar and acquire out-of-order capacilities	A modern core has more than one *port* that can perform the same type of operation (Arithmetic-logic, memory access, I/O, ...). That means that more than one operation can be performed in the same clock, if the code is suited to allow that, which is referred as Instruction-Level-Parallelism (ILP).
Operations are broken down into smaller stages and pipelined	The expected throughput is larger, at the condition that the pipelines are always as full as possible and do not stall due to data and control hazards (we'll see that later)
Branch predictors are an important part of the fron-end	Branches play a major role in causing stalls in the pipelines and in the execution flow. Dealing with this is a key factor to increase the code's performance.
CPUs acquire vector capabilities	Special registers in the CPU have the capacity of performing the same operation on multiple data (SIMD - Same Instruction Multiple Data). That is referred as Data-Level-Parallelism. Not all the loops are vectorizable, that depends on a number of factors.

Figure 4.2: At a glance

In the **Von Neumann** architecture there is only 1 processing unit, 1 instruction is executed at a time and the memory is ”flat”. It was much simpler than today’s architectures, but it is still the basis of modern computers.

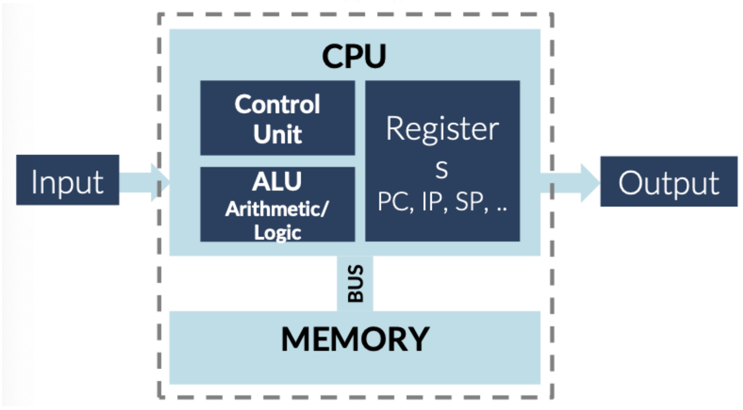


Figure 4.3: Von Neumann architecture

Today, instead:

- there are many processing UniTs
- many instructions can be executed at a time
- many data can be processed at a time
- ”instructions” are internally broken down into many simpler operations that are pipelined
- memory is strongly not ”flat”, there is a strong memory hierarchy, access memory can have very differnt costs depending on the location and accessing RAM is way more costly than performing operations on internal registers.

We have seen that the power required per transistor is

$$C \times V^2 \times f$$

Roughly the capacitance and the voltage of transistor shrinks with the feature size, whereas the scaling is much more complicated for the wires. Overall, a typical CPU got from 2W power to 100W which is at the limit of the air cooling capacity.
To cope with the power wall one can:

- Turn off inactive circuits
- Downscale Voltage and Frequency for both cores and DRAM
- Thermal Power Design, or design for typical case
- Overclocking for a short period of time and possibly for just a fraction of the chip

CPU became faster than memory in the early 90s.
The CPU may spend more time waiting for data coming from RAM than executing operations. That is part of the so called ”memory wall”. The solution is to use a memory hierarchy, where the data is stored in different levels of memory, each one with different access times and sizes. Furthermore, to be faster it ought to be extremelty closer. The new memory that will be called **cache**, will be much smaller than the RAM.
The cache itself has a hierarchy:

- L1 cache: very small, very fast, very close to the core
- L2 cache: bigger, slower, still close to the core
- L3 cache: bigger, slower, shared among cores

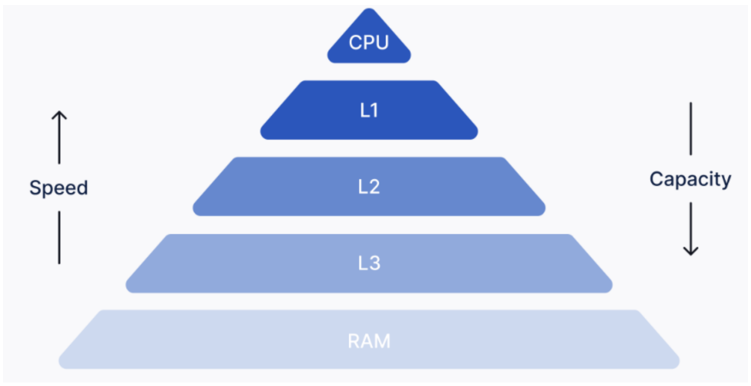


Figure 4.4: Memory hierarchy

Definition: ***Principle of Locality***

Data are defined ”local” when they reside in a small portion of the address space that is accessed in some short period of time.
There are two types of locality:

- **Temporal locality**: if an item is referenced, it will tend to be referenced again soon.
- **Spatial locality**: if an item is referenced, items whose addresses are close by will tend to be referenced soon.

4.2 Cache Hierarchy

The RAM contains 10^9 bytes, while L1 contains 10^4 bytes (32KB for data and 32KB for instructions). So, how do we map the RAM into a given level of cache, for instance L1, in an effective way?

- Where to map an address?
- What if the location in L1 is already occupied?

Let's say that both the RAM and the cache are subdivided in blocks of equal size (64B): you do not load just a byte in your cache, but an entire block (called *line*)

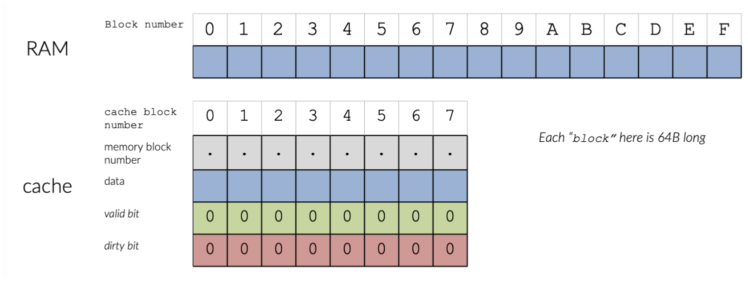


Figure 4.5: Cache hierarchy

There exist three main strategies to map the RAM into the cache:

- **Full mapping**: Data can be placed in any free cache block. This is the simplest strategy, but it is not efficient.

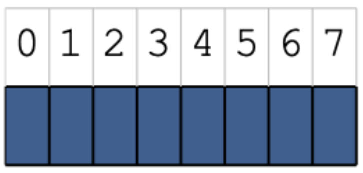


Figure 4.6: Full mapping

- **Direct mapping**: Each block of RAM can be placed in only one block of the cache. This is a simple strategy, but it is not efficient.

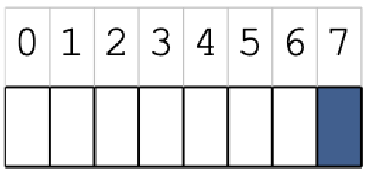


Figure 4.7: Direct mapping

- **n-way associative**: Each block of RAM can be placed in a set of n blocks of the cache. This is a more complex strategy, but it is more efficient.

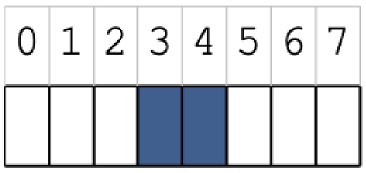


Figure 4.8: n-way associative

Table 4.1: Cache Mapping Strategies

Strategy	Description	Pros	Cons
Full mapping	Any block can store any address	Very flexible, good space use	More complex to search, higher hardware cost
Direct mapping	Each address maps to exactly one cache block	Simple design, fast lookups	Higher chances of conflicts, limited flexibility
n-way associative	Cache is divided into sets of n blocks	Balances speed and flexibility	More complex than direct mapping

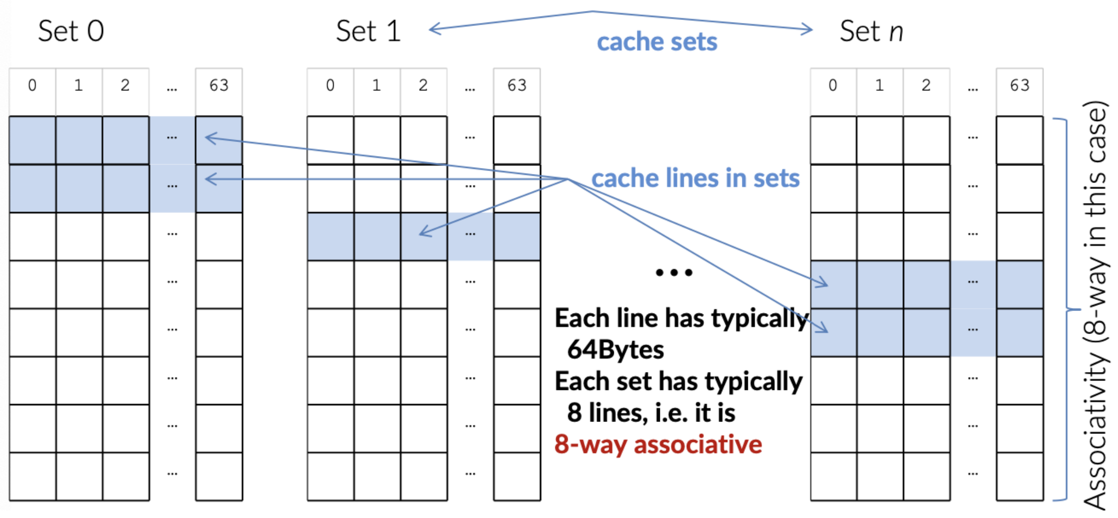

Set 0
Set 1
cache sets
Set *n*
0
1
2
...
63
cache lines in sets
...
Each line has typically 64Bytes
Each set has typically 8 lines, i.e. it is
8-way associative
Associativity (8-way in this case)

Figure 4.9: A typical today cache

Advanced Concept: *Cache mapping*

How a byte is actually mapped into a cache location?

The memory access pattern

Consider a simple direct mapped **16 byte data cache** with **two cache lines**, each of size 8B. Consider the following code sequence, in which the array `X` is cache-aligned (i.e., `x[0]` is always loaded into the beginning of the first cache line) and accessed twice in consecutive order:

```
float X[8];
for(int j=0; j<2; j++)
   for(int i=0; i<8; i++)
     access(X[i]);
```

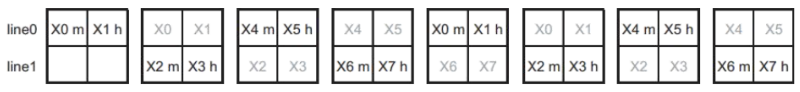


Figure 4.10: The hit-miss pattern is: MH MH MH MH MH MH MH MH, the miss-rate is 50% (the first miss is compulsory miss)

Let's consider another code sequence that access the array twice as before, but with a strided access.

```
float X[8];
for(int j=0; j<2; j++)
   {
      for(int i=0; i<7; i+=2)
         access(X[i]);
      for(int i=1; i<8; i+=2)
         access(X[i]);
   }
```

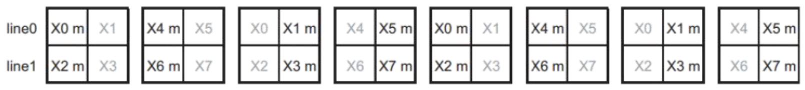


Figure 4.11: The hit-miss pattern now is: MM MM MM MM MM MM MM MM, the miss-rate is 100%

Finally, consider a third code sequence that again access the array twice:

```
for(int k = 0; k < 2; k++)
   for(int i = 0; i < 2; i++)
      for(int j = 4*k; j < (k+1)*4; j ++)
         access(X[ j ]);
```

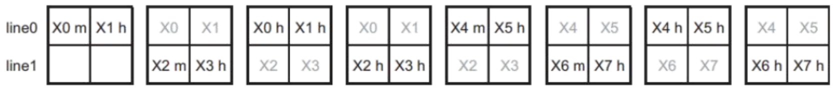


Figure 4.12: The hit-miss pattern now is: MH MH HH HH MH MH HH HH, the miss-rate is 25%