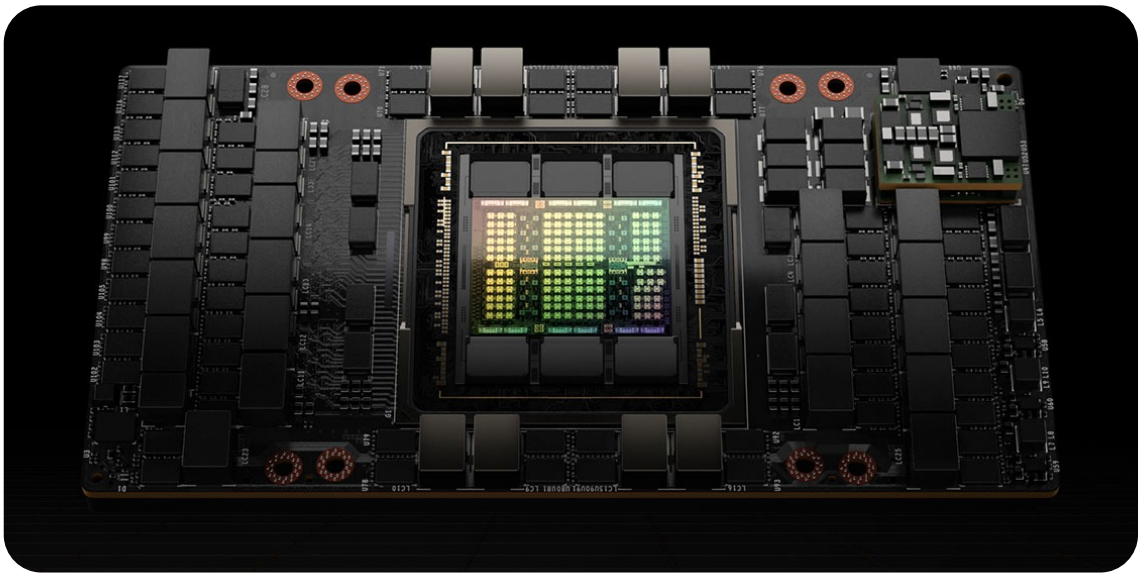
Sapienza, University of Rome
Course of Computer Science
Extra, 1st Semester

High Performance Computing



Notes by Leonardo Biason
Course taught by prof. Salvatore Pontarelli and Daniele De Sensi

Sapienza Università di Roma

About these notes

Those notes were made during my three years of university at Sapienza, and **do not** replace any professor, they can be an help though when having to remember some particular details. If you are considering of using *only* these notes to study, then **don't do it**. Buy a book, borrow one from a library, whatever you prefer: these notes won't be enough.

License

The decision of licensing this work was taken since these notes come from **university classes**, which are protected, in turn, by the **Italian Copyright Law** and the **University's Policy** (thus Sapienza Policy). By licensing these works I'm **not claiming as mine** the materials that are used, but rather the creative input and the work of assembling everything into one file[1]. All the materials used will be listed here below, as well as the names of the professors (and their contact emails) that held the courses[2]. The notes are freely readable and can be shared, but **can't be modified**. If you find an error, then feel free to contact me via the socials listed in my website. If you want to share them, remember to **credit me** and remember to **not** obscure the **footer** of these notes.

The "*High Performance Computing*" course was taught in the Winter semester in 2025 by professors Salvatore Pontarelli (salvatore.pontarelli@uniroma1.it) and Daniele De Sensi (desensi@di.uniroma1.it)

I hope that this introductory chapter was helpful. Please reach out to me if you ever feel like. You can find my contacts on my website. Good luck[3]! Leonardo Biason

→ leonardo@biason.org

Contents

Chapter 1
The rise of HPC

HPC is an old topic in CS: it allows for performing powerful simulations and computations, which would be usually impossible in modern, domestic systems. But why do we need to make simulations? Generally, simulations are considered the Third Pillar of Science: they allow humans to go over the limits of experimental science (over dangerous limits, costs and time limitations), so since the rise of this field of CS, society has been needing coherent machines to overcome even greater limits.

In time, there has been a shift in the kind of architecture used in HPC settings. We stopped focusing about the max frequency of a processor since 2015 (Dennard Law broke), now we focus on novel architectures because they harness different capabilities. Nowadays chips in GPUs, CPUs and so on are made with chiplets, a manufacturing process for which different components are built by stacking layers of silicon together.

The results are impressive: supercomputers from the Top500 (list started in 1993) can reach, jointly, up to 4.8 Eflop/s. A modern computer may reach the power of the most performant supercomputer from 1996!

Back in the days, when the Top500 list was born, Intel was the predominant architecture, using the `i860` architecture. Nowadays, 78% of the supercomputers use Intel with the `x86-64` architecture (97% of the supercomputers use `x86-64`). It costs too much to change architecture, so the scientific community sticked with it. However, nowadays we don't use anymore solely CPUs: GPUs are the dealbreaker.

Curiously, at the time (during the first phase of HPC) it was more convenient to use shared memory architecture, opposed to today, where distributed is the new standard. In 1994 an experiment, the Beowulf cluster, was run: it consisted in multiple Linux machines running together in parallel. This showed how multiple, singular machines could work way better than single, monolithic supercomputers. This marked the beginning of the second phase of HPC. Now we are in the 3rd phase: we moved to heterogeneous architectures, using GPUs, and we are heading to the 4th phase, where experimentation and specialized architectures are investigated and used (consider TPUs, DPUs and special architectures such as Cerebras, Graphcore and Tenstorrent).

Modern supercomputers have some commonalities: NVIDIA dominates in these contexts, interconnects use either Ethernet or InfiniBand (made by NVIDIA; indeed, 426 out of 500 supercomputers from the modern Top500 use this kind of interconnect), and most of these computers (if not all of them!) use Linux as OS.

Programs are not built in Python, but usually in fast, low-level languages (such as C) with the aid of libraries such as MPI, OpenMP and CUDA. Floating point operations also employ different precision standards (from 64 to 8 bits), and are used in different contexts

and scenarios (AI training doesn't need the highest precision, simulating a bridge does).

SECTION 1.1 Components of HPC

HPC systems have different components:

- CPUs, which nowadays are all multicore CPUs;
- accelerators, such as general purpose GPUs;
- memory;
- storage systems;
- interconnects.

Novel architectures employ hardware such as TPUs, NPUs or FPGAs (Field Programmable Gated Arrays).

Nodes is a system can be used for different tasks: for computation, for storage, for data processing, etc... And all nodes in a supercomputer are interconnected through some way. Interconnections can be inter-node, intra-rack (node to node connections) and inter-rack.

Inter-node communications happen within a compute node, between CPU, GPU, memory, etc... Technologies used vary depending on the node: we may use CPU-CPU interconnects, GPU-GPU links (such as the NVLink), or CPU-Accelerator (such as PCIe Gen 5/6, which are used also in domestic machines).

Intra-rack interconnection happens within nodes in the same rack. Technologies used are for instance the InfiniBand from NVIDIA, the Slingshot, Ethernet or UltraEthernet.

Inter-rack communication happens between racks. The main problem with this kind of connections is that we don't have enough bandwidth to connect all the racks together. The technologies used are the same of the intra-rack communication, but with some tweaks for allowing scalability (examples are Dragonfly, Fat-Tree, Torus, or HyperX). There has been some interest lately for optical interconnects. This kind of technologies allow for scalability, fault tolerance and congestion control.

SECTION 1.2 Study case: Leonardo

Let us consider a very familiar supercomputer: Leonardo. Nodes are based on the Atos BullSequana XH2000 design, and nodes either designed for hosting a CPU partition or a booster partition with NVIDIA A100 GPUs. It uses Mellanox interconnects.

There are some nodes used as front-end nodes (16), which are needed for interfacing with the supercomputer. The partitions are as follows:

- **Data Centric** and **General Purpose partitions**, which are nodes with 3-nodes blades, where each node counts 2 CPUs;

- **Booster GPU partition**, also called "Da Vinci" blades, they are diskless and are intended to host 4 NVIDIA Ampere A100 GPUs.

The topology used to connect the nodes is called Dragonfly+, and is based on the NVIDIA Mellanox Infiniband, capable of transmitting up to 200 Gb/s.

Now Leonardo is also being expanded with an other partition, called LISA, made specifically for AI purposes. In this partition, the GPUs (which use last generation accelerators, such as the NVIDIA H100 GPUs) are collected all together through an NVIDIA NVLink, so there is no CPU work involved in the communication.

SECTION 1.3 The performance of an HPC system

Performances on HPC systems are usually measured by a benchmark which solves dense matrix linear systems (HPL benchmark). Another kind of benchmark (HPCG) is based on sparse matrices and conjugate gradient (hence the CG): with these kind of benchmarks, we have no more sequential access to data, rather a discontinuous one (due to the sparse matrices).

The performances vary depending on the benchmarks: indeed, computers achieving peak performance in HPL might not have peak performance in HPCG (or may not keep the same position).

Nowadays we also measure performance based on AI/ML benchmarks. While AI/ML have been around for a while, we've been only able to better tackle them now because of increasing data and computational power. This means that the support from researchers/industries towards the AI/ML scenery has deeply increased. The interesting part of AI/ML tasks is that we have simple operations that are repeated many times on different data.

Also, we are shifting from higher-precision formats to smaller ones (for instance, from 32 to 16). Interestingly, other precision format have been developed, such as the Google Bfloat format (B standing for Brain), which uses a different amount of bits for explaining exponents and mantissa. In this case, Bfloat can represent a greater range at the cost of a lower precision. Moving from IEEE to Bfloat has no cost whatsoever.

SECTION 1.4 High performance programming

When we measure a HPC system, we use the $x$flop/s measurement (where x is kilo, mega, giga, etc...), which denoted how many 64-bits floating-point operations (either addition or multiplication) are done in a second by a system. The theoretical peak performance of a system is given by the theoretical amount of flop/s that a system can compute over a specific time range.

An interesting example: an Intel Skylake processor has 2.1 GHz per core, and can do 32 flops per cycle, per core. But how can it achieve that? This is because there are

some instructions that work on vectorized data. Such instructions are the SIMD (Single Instruction Multiple Data) instructions, which work on multiple data together.

As an example of how fast these operations can be: let us consider a naïve implementation in Python of a DGEMM operation between two $\mathbb{R}^{4K \times 4K}$ matrices as the baseline; if we consider a C program, with the same algorithm, we can have a speedup of 47×. If we use vectorized operations, we can reach a speedup of 23.000×, but if we use instead intrinsic Assembly operations (AVX intrinsic), we can have up to 62.000× speedup.

How can we improve a single-node application though? There are many ways to do so, but in order to identify the best course of action we should:

1) measure the performance of the application;
2) understand the hardware of the machine running the program;
3) identify any possible bottleneck (check if the problem is memory bound, compute bound or related to under-utilization of the machine's resources);
4) solve the bottlenecks, if possible;
5) repeat until you are satisfied with your performance.

Usually, some good pieces of advice are the following:

1) optimize the algorithms used and the data structures (aka use quick sort and hash tables), but don't be greedy: some algorithms might perform better than others only in some specific situations. An example is quick sort vs counting sort: while counting sort is fast, it only holds for limited ranges of values. Choose the best algorithm depending on your problem;
2) perform memory optimizations: most applications usually suffer from memory bottlenecks, mostly because of the moving of the data. It's important to carefully use caching and avoid cache misses. The idea is to have a balance between flops and data transfer rate (hence balancing the flop/*s* and the loaded word/*s*). Reaching the balance of 1/1 is nearly impossible, but we should strive to reach it;
3) exploit hardware parallelism: don't have unutilized resources if you can avoid it. There are two types of parallelism that we can achieve: data parallelism (same task on different data) and thread parallelism (use different threads to execute an algorithm. Might give false sharing issues).

SECTION 1.5 CPU microarchitecture

All CPUs run on a given Instruction Set Architecture (ISA). There are two categories:

- register-based, load-store memory: `ARM` / `RISC-V`;
- register-memory architecture: `x86`.

An ISA provides basic functions (such as load, store, control and arithmetic operations) and extensions for different precision formats (even the ones for AI functions), vectors processing (Intel AVX2, RISC-V "V" extension) and even matrix/tensor instructions (in newer architectures).

In order to improve efficiency in a CPU, we can use pipelining in order to perform multiple operations together, although in different stages. Usually, modern pipelines have around 10-20 pipeline stages.

We consider the **throughput** of a pipeline as the number of instructions that complete and exit the pipeline for a unit of time. Instead, the **latency** of a pipeline is the total time through all the stages of a pipeline.

A pipeline contains hazards:

- structural hazards are caused by resource conflicting, which can be eliminated by replicating hardware resources;
- control hazards are caused by changes in the program flow. Branching is one of the main causes of control hazards;
- data hazards are caused by data dependencies, and are of three types:
  - read after write (RAW): can be solved through data forwarding. The longer the pipeline, the more effective data forwarding is;
  - write after read (WAR): usually happens in superscalar/Out of Order (OoO) processors. Can be solved through register renaming;
  - write after write (WAW): also here, we can solve them through register renaming.

We mentioned Out of Order (OoO) processors: these are CPUs that allow for instructions to enter in the execution stage in an arbitrary stage with respect to the sequential order they have been written; the order depends also on any dependency or resource availability. We say that an instruction is called retired if the results of the CPU is visible in the architectural state (aka, instructions should go out from the pipeline in the same order in which they got inside the pipeline). In order to ensure correctness, a CPU should retire all the instructions in the order specified by the program (this constraint is called in-order retire).

Superscalar processors have the capability of issuing more than an instruction per clock cycle. The maximum number of instructions that can be executed in a clock cycle is called issue width.

In order to contrast static scheduling, CPUs use dynamic scheduling. For this kind of scheduling, CPUs employ:

- the Tomasulo algorithm for register renaming, in order to eliminate false dependencies (which are also known as dependency chains, and are usually found in loops). It does not solve RAW dependencies;

- a Reorder Buffer (ROB), which is a buffer that keeps track of the state of each instruction. Here, instructions are inserted in order, executed out of order, and retired in order. Instructions are also inserted in the RS. The size of the ROB allows for a better future lookup. It's here that register renaming happens;
- a Reservation Station (RS) holds instructions until the resource needed is free. There are different entries in the RS depending on the execution unit type. Moreover, it supports speculative execution: it can allow to execute the instructions of a branch, even if it's known whether that branch will be taken or not. In the worst case, the instruction will be cancelled. Speculative execution is used because, in the best case, it can save a lot of cycles.

Speculative execution is backed by Branch Prediction, which can make speculative execution much more accurate and confident. Branches can be of three types:

1) **unconditional branches** and **direct calls**: always taken;
2) **conditional branches**: can be either taken or not. Forward conditional branches are generated from if-else statements, backward conditional jumps are frequent in loops, and are usually taken;
3) **indirect call**: has many targets, perhaps because of a switch statement.

Prediction mechanisms are based on temporal (if a branch has been taken many times in the past, it's likely that it's going to be taken again) and spatial correlation (if the branch is based on nearby data, it's possible to make inference).

In order to aid branch prediction, we have a Branch Prediction Unit (BPU), which is composed of:

- a Branch Target Buffer (BTB), which caches the target addresses of every taken branch;
- a Pattern History Table (PHT), which remembers the pattern of all taken branches with just 2 bits.

SECTION 1.6 Thread-level parallelism

What we've seen so far is completely transparent for the user function-wise, but performance-wise this is completely opaque. Instead, thread-level parallelism is transparent by all means.

Thread-parallelism can be achieved through simultaneous multi-threading (SMT), which allow for multiple threads to run simultaneously on different cores. The greatest issue with SMT is given by the cache handling and by the fact that it's very hard to measure performances of an SMT application.

SMT scheduling also becomes harder in hybrid architectures (architectures where different types of cores are merged in a single CPU. An example is the Apple M1 chip, which uses high-performance "Firestorm" cores and energy-efficient "Icestorm" cores). In the Intel P-cores and E-cores there is also a difference with respect to support for SMT: P-cores do support it, E-cores don't.

1.6.1 Caches and High Bandwidth Memory

We know that the major bottleneck is moving data from the memory to the CPU. We can take advantage of caches in order to avoid wastes of time in data fetching/storing, which are based on the concept of locality (be it spatial or temporal). In a CPU we have 3 levels of caches: L1, L2 and L3.

A new type of GPU/CPU memory that has been developed recently is the High Bandwidth Memory. This kind if memory sees a series of HBM DRAM dies stacked all together, and that communicate through a very short interconnect (which is also very large, allowing for up to 1024 bits to be read simultaneously from each HBM stack) with both the CPU and the GPU. It's not as fast as the SRAM used within the CPU caches, but it's faster than the usual DRAM that we may find in domestic computers.