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

May 28, 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

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.

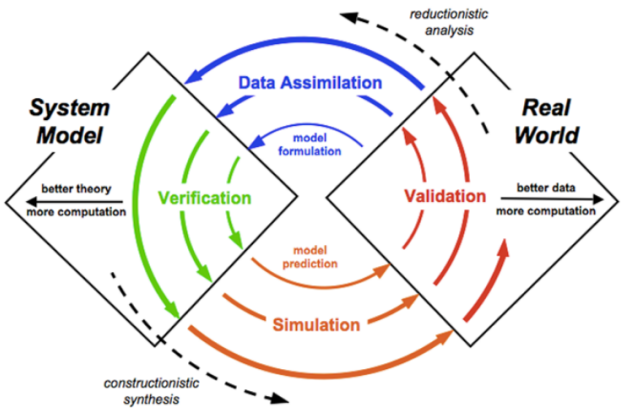


Figure 1.1: Research and Development

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 $\rightarrow$ shorter propagation delay $\rightarrow$ faster frequency
- Smaller transistors $\rightarrow$ smaller capacitance $\rightarrow$ lower power consumption

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

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.

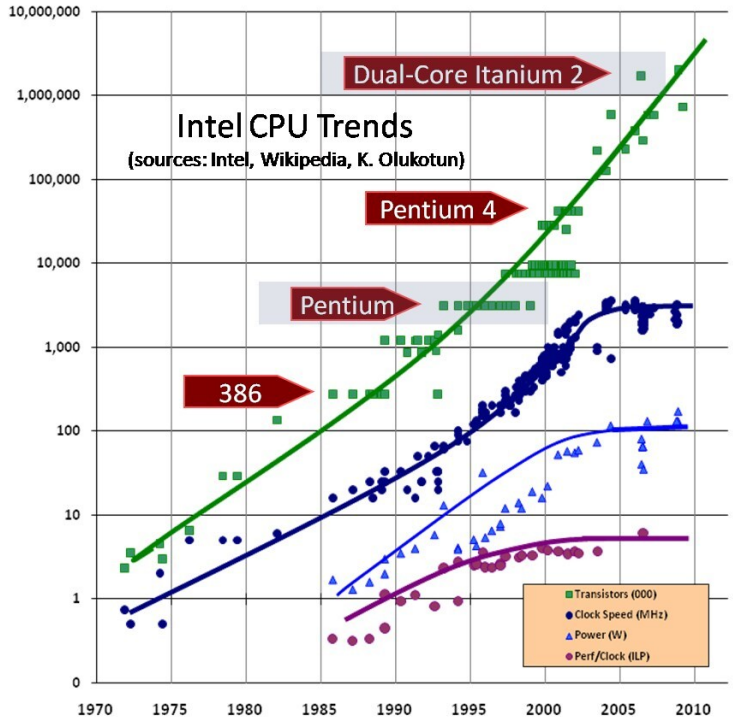


Figure 1.2: Moore's Law

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).

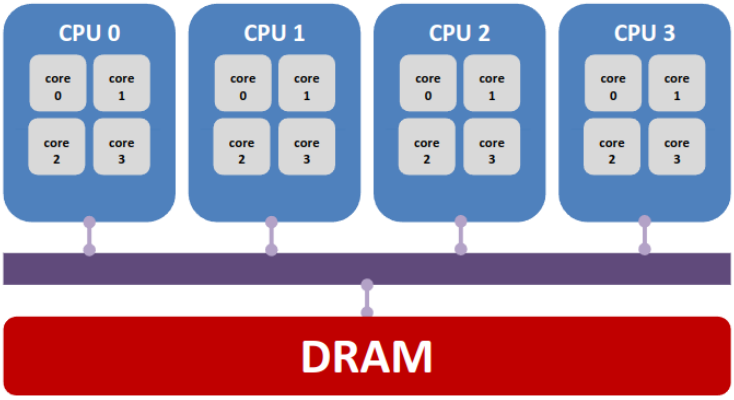


Figure 1.3: Uniform Memory Access (UMA)

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.

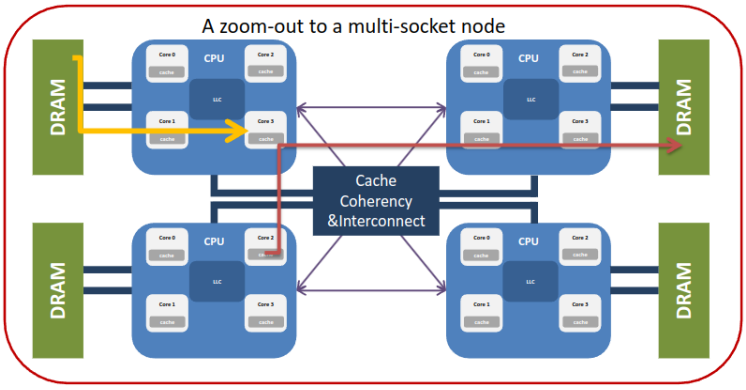


Figure 1.4: Non-Uniform Memory Access (NUMA)

Parallelism within a HPC node

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

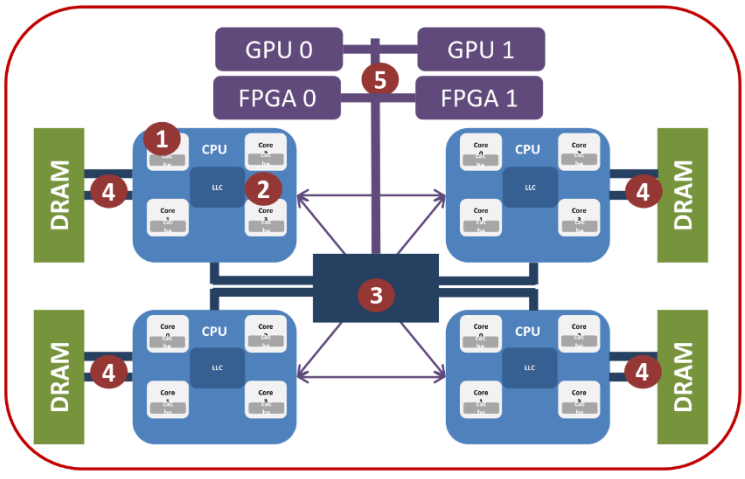


Figure 1.5: Levels of parallelism

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

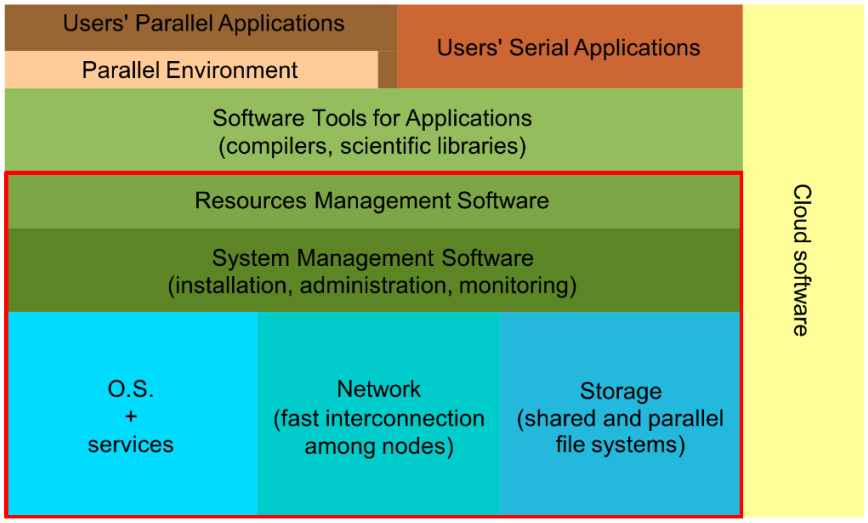


Figure 1.6: The Software Stack: in red the *cluster middleware*

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.

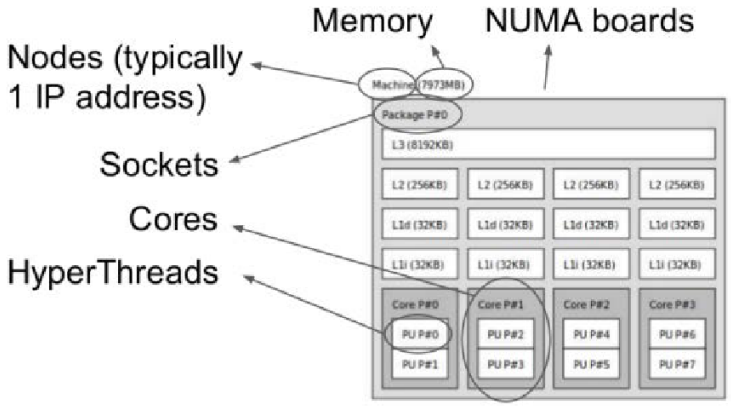


Figure 1.7: Resources in a HPC cluster

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.

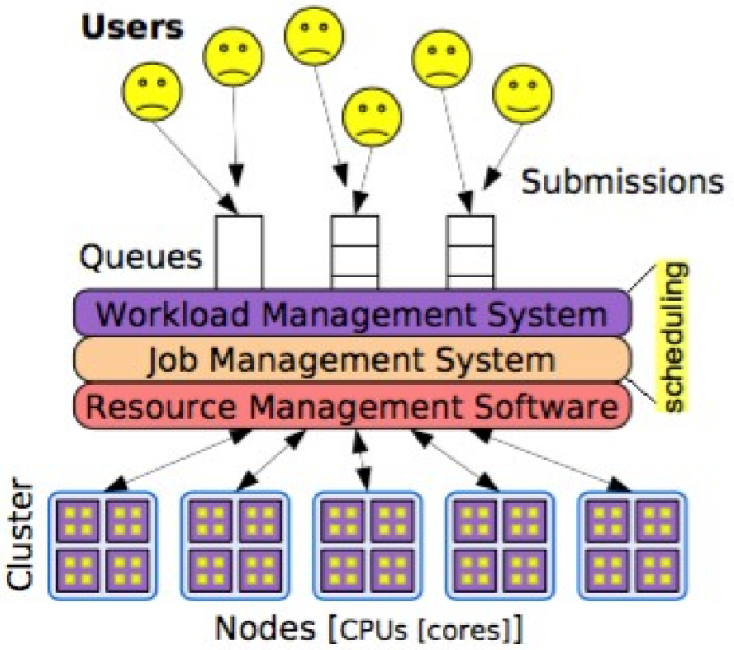


Figure 1.8: Batch Scheduler Architecture

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.

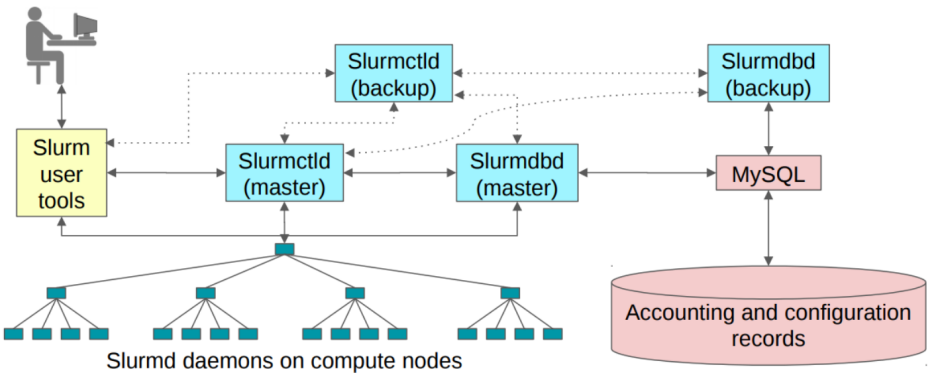


Figure 1.9: Simplified SLURM Architecture

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

…

...

2 Optimization

2.1 Introduction to Optimization

High Performance Computing (HPC) requires extracting maximum effectiveness from code on available hardware. Optimization is thus essential, though both ”optimization” and ”effectiveness” are context-dependent concepts.

There is no universally ”optimal” code, as optimality varies based on specific requirements and constraints. In HPC, optimization encompasses several competing dimensions:

- **Memory constraints:** Minimizing memory footprint for both data structures and executable code, especially in resource-limited environments.
- **I/O constraints:** Efficient input/output operations across various media, critical for data-intensive applications where I/O bottlenecks limit performance.
- **Time constraints:** Reducing time-to-solution, whether optimizing for worst-case scenarios, average performance, or specific use cases.
- **Robustness:** Ensuring reliability and fault tolerance for mission-critical applications, sometimes at the expense of raw speed.
- **Energy constraints:** Minimizing power consumption, particularly important for large-scale HPC installations and power-limited systems.

This multidimensional nature creates inherent trade-offs between performance aspects. The fundamental challenge lies in determining which dimensions are most important for a specific application and finding an appropriate balance among competing objectives.

Tip: ***Premature Optimization***

Premature optimization is the root of all evil.

~Donald Knuth

2.1.1 First steps to consider

- Dryness: Do not add unnecessary code nor duplicate code. This will make the code more difficult to maintain and debug.

- Readibility: Make the code as readable as possible. Write comments and use meaningful variable names.

- Test is part of the design: - unit test: separately stress each unit of the code; - integration test: stress the integrated behavior of all the units together; - system test: stress the system as a whole.

- Validation and verification are part of the design: - Validation ensures that the code does what it was meant to do (all modules are designed accordingly to design specifications)

- Verification ensures that the code does what it does correctly (black-box testing against test-cases, ...)

2.1.2 Compiler optimization

Compilers are those tools that translate the code you write into machine code that the computer can understand.

Compilers are also able to perform **sophisticated analysis** of the source code so that to produce a target code (usually an assembly code) which is **highly optimized for a given target architecture**.

- write non-obfuscated code - design a good data structure layout - design a “good” workflow - take advantage of the modern out-of-order, super-scalar, multi-core architectures

Optimization levels

It is not granted that -O3, although often generating a faster code, is what you really need.

For instance, sometimes expensive optimizations may generate more code that on some architecture (e.g. with smaller caches) run slower, and using -Os may bring surprising results.

Take into accounts that modern compilers allow for local specific optimizations or compilation flags

Compile for specific CPU model

The compiler knows the architecture it is compiling on, of course. However, it will generate a portable code , i.e. a code that can run on any cpu belonging to that class of architecture.

Example: x86_64, x86_32, ARM, POWER9, are all classes of architecture

Besides a general set of instructions that all the cpus of a given class can understand, specific models have specific different ISA that are not compatible with others (normally you have back-compatibility).

Using appropriate switch (in gcc `-march=native -mtune=native`, in icc texttt-xHost), the compiler will optimize for exactly the specific cpu it’s running on, much probably producing a more performant code for it

... automatic profiling ... ... memory allocation ... ... C-specific hints ...

Memory aliasing:

```
1 void func1 (int *a, int *b)  {
2     *a += *b;
3     *a += *b;
4 }
5
6 void func2 (int *a, int *b)  {
7     *a += 2 * *b;
8 }
```

An incautious analysis may conclude that a compiler, or even a programmer, should immediately transform func1() into func2() because, having three less memory references, it should yield to a better assembly code.

However, is it really true that the two functions behave exactly the same way in all possible conditions?

What if a = b, i.e. if a and b points to the same memory location?

The compiler can not optimize the access to a and b because it can not assume that a and b are pointing to the same memory locations or, in general, that the references will never overlap.

That is called aliasing, formally forbidden in FORTRAN: which is the reason why in some cases fortran may compile in faster executables without you paying any attention

Help your C compiler in doing the best effort, either writing a clean code or using restrict or using `-fstrict-aliasing -Wstrict-aliasing` options.

```
1  void my_function (double *restrict a, double *restrict b)  {
2      *a += *b;
3      *a += *b;
4  }
```

Now you're telling the compiler that the memory regions referenced by a and b will never overlap.

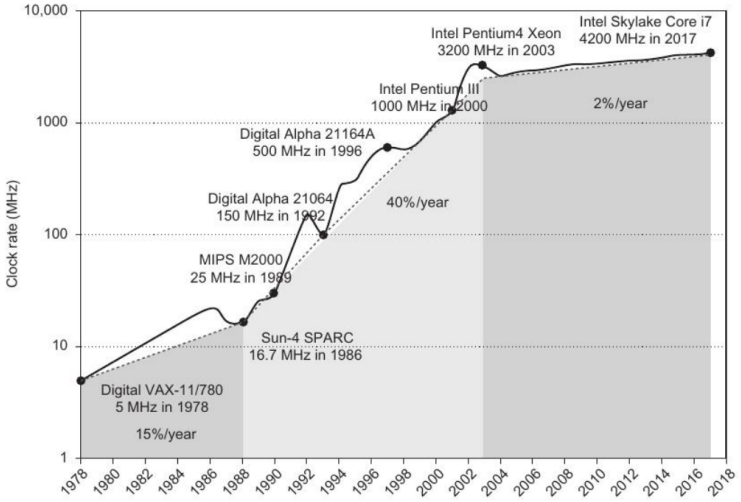
So, it will feel confident in optimizing the memory accesses as much as it can (basically avoiding to re-read locations)

2.2 Single core optimization

Modern CPU architecture has evolved significantly to address performance challenges, fundamentally changing optimization approaches. As CPUs have dramatically outpaced memory speed, memory access has become a critical bottleneck, leading to the development of hierarchical memory systems that leverage data locality principles to reduce latency. Concurrently, CPUs have adopted super-scalar designs with out-of-order execution capabilities, enabling multiple operations to execute simultaneously (Instruction-Level Parallelism or ILP), provided code is appropriately structured. Operations are now broken into smaller pipelined stages to increase throughput, though these pipelines remain vulnerable to data and control hazards that can cause performance-degrading stalls. Branch predictors have become crucial front-end components that minimize pipeline disruptions from control flow changes. Additionally, modern processors incorporate vector processing through SIMD (Same Instruction Multiple Data) registers, enabling Data-Level Parallelism where a single instruction operates on multiple data elements simultaneously. However, not all computational patterns can benefit from vectorization, as this depends on data independence and access patterns.

The energy and power are a major design issue: 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.

Primary strategies to cope with the power wall:

1. Turn off inactive circuits (dark silicon)
2. Downscale Voltage and Frequency for both cores and DRAM
3. Thermal Power Design, or design for typical case
4. Overclocking for a short period of time and possibly for just a fraction of the chip

Static power, $Pstatic \propto Currentstatic \times V$, is also an important factor in the power consumption. It increases with the number of transistor while the current leakage increases as the feature size decreases and could amount to 25% of the total power consumption.

Cache

Another problem is the one colles the memory wall: the memory speed is not increasing as fast as the CPU speed, and the latency is not decreasing as fast as the CPU latency:

The CPU may spend more time waiting for data coming from RAM than executing ops.

The solution is to equip the CPU with a faster memory, that, by the way, is also way more expensive and way more energy consuming.

Furthermore, to be faster it ought to be extremely closer. All in all, the new memory that will be called cahce, will be much smaller then RAM.

We are quite naturally led to the ”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

Local data are more likely to be in the cache.

- **Temporal locality**: if a data is accessed, it is likely to be accessed again soon
- **Spatial locality**: if a data is accessed, it is likely that nearby data will be accessed soon

Cache misses

The cache is organized in lines, each line is a fixed number of bytes (e.g. 64 bytes) and each line is associated with a tag that identifies the memory address of the first byte of the line.

When the CPU needs to access a memory location, it first checks if the line containing that memory location is in the cache. If it is, it is a cache hit, otherwise it is a cache miss.

There are different types of cache misses:

- **Compulsory miss**: Unavoidable misses when the cache is empty and the CPU needs to access a memory location for the first time
- **Capacity miss**: It occurs when there is not enaugh space to hold all data, or if there is too much data accessed in between successive use.
- **Conflict miss**: the cache is full and the CPU needs to evict a line to make space for the new one, but there is no other line that can be evicted

Cache optimization

It is possible to reduce the number of cache misses by:

- **Rearrange (code and data)**: Design layout to improve temoral and spatial locality;
- **Reduce (size)**: Reduce the size of the data structures and the number of instructions;
- **Reuse (cache lines)**: Increase spatial and temporal locality, keep resident data in cache for more operations.

2.3 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.

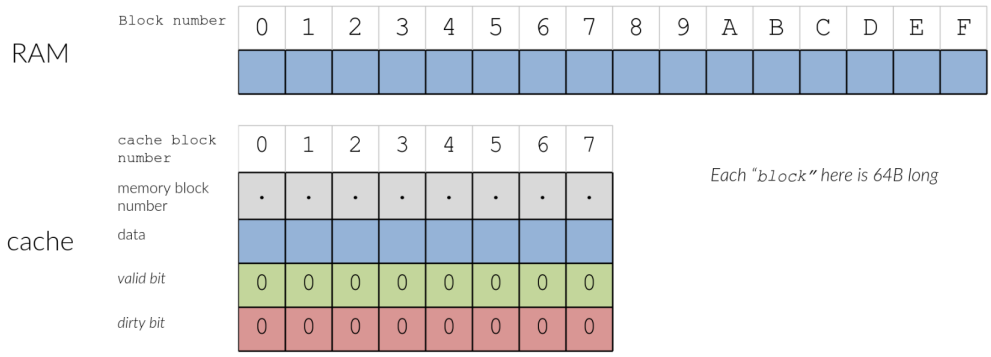


Figure 2.1: Main memory and cache are both split into blocks (lines).

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 2.2.

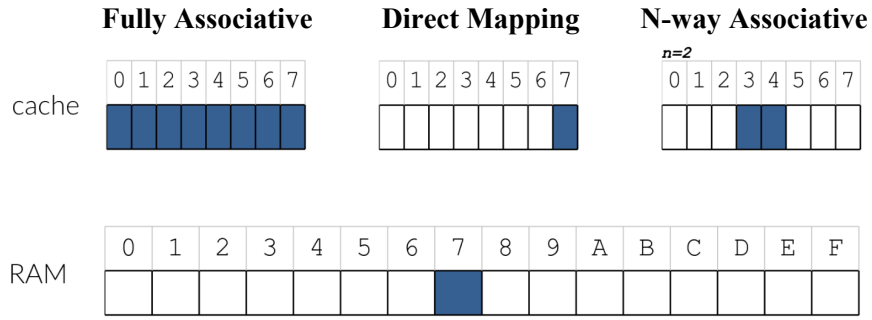


Figure 2.2: Fully Associative, Direct, and N-way Set Associative mappings.

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.

…

3
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.

—

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.

4
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*

5 Loop Optimization and Prefetching

Loop CLassification

Linear Loops $O(N)/O(N)$:

2-level loop $O(N^2)/O(N^2)$:

- **Avoid unecessary loads / stores**
- **Loop unrolling** Unroll outern loop and fuse in the inner loop; there is potential vectorisation

```
1  for (int i = 0; i < N; i++)
2      S += A[i]
```

If we change this into:

```
1  for (int i = 0; i < N; i++)
2      S = S + A[i]
```

then,

```
1  for (int i = 0; i < N; i+=2)
2      S = (S + A[i]) + A[i+1]
```

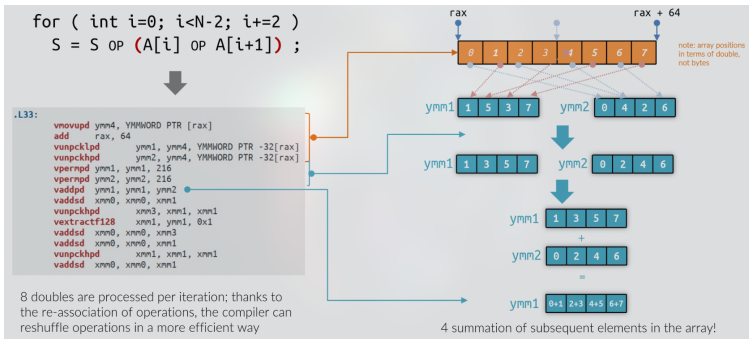
but we can do better, just moving a parenthesis:

```
1  for (int i = 0; i < N; i+=2)
2      S = S + (A[i] + A[i+1])
```

In this case the compiler is able to ***vectorize*** the operations

3rd type: $O(N^3)/O(N^2)$

Figure 5.1: Loop unrolling



6 Parallelism

why parallelism?

We wnt to parallelize our code for two main reasons:

- To speed up a code
- The problem size exceeds the computer capacity

we have two types of problems:

1. **Embarassingly paralell problems**: The problem solution for each data point is completely independent of the solution for any another data point
2. All the rest

Of course, normally you have some blending of computations that are independent of each other and computations that are not, with a significant umbalance towards the second case.

...

6.1 Parallel Computing

Draft

6.1.1 Domain decomposition

Domain decomposition is a technique used to solve large problems by dividing them into smaller, more manageable subproblems. This approach is particularly useful when the goal is to distribute the workload across multiple processors or cores.

...

when the work-load is strongly dependent on the properties of the data, and moreover those propertie change in time during the computation, there is not a general rule.

You can, for instance, sort your data by computational intensity and distribuite them to the different players, so to achieve an even work-load wirth a minimum imbalance; then, repeating the procedure as often as needed while the data evolves during a multi-step computation.

6.1.2 Functional decomposition

Functional decomposition is, as domain decomposition, a technique used to break down a complex problem into smaller subproblems. This time, the decomposition is based on the functions or operations that need to be performed.

Quite often, your computation is made of several different “sections”. Let’s call them tasks.

There will be, in general, a dependency graph among the tasks: some will be completely independent of any other, some will be dependent on 1, 2 or more “previous” tasks and will feed some “subsequent” tasks.

Then, you may decompose not the data but the tasks among your workers. In general, that requires some synchronization to manage the dependencies

…

Definition: ***What is parallel computing?***

1. A ***parallel computer*** s a computational system that offers simultaneous access to many computational units fed by memory units. The computational units are required to be able to co-operate in some way, meaning exchanging data and instructions with the other computational units.
2. ***Parallel processing*** is the ensemble of techniques and algorithms that makes you able to actually use a parallel computer to successfully and efficiently solve a problem.

The parallel processing is expressed by **software entities** that have an increasing level of granularity (processes, threads, routines, loops, instructions..)

The software entities run on underlying **computational hardware entities** (as processors, cores, accelerators)

The data to be processed/created live and travel in **storage hardware entities** (as Memory, caches, NVM, networks, DMA)

The exploitation/access of hardware resources (computational and storage) is **concurrent** among software entities

Shared vs Distribuited Memory

Shared-memory programming leverages a common memory accessible to all computational units, thus facilitating direct data exchange without explicit communication routines. In contrast, distributed-memory programming relies on explicit communication, often via message passing, to exchange data since each unit has its own local memory. This distinction reflects both the physical memory layout and the programming approach.

Distributed Memory A typical programming paradigm is the message-passing approach, where processes communicate via "messages" and each process has its own memory space. Communication can happen either over the network (different protocols are possible at hardware/middleware level) or via shared memory techniques if the communicating processes can directly access the same memory. The user, however, still treats the process as if it were using message passing, and the actual communication is managed by the middleware.

A well-known standard is MPI (Message-Passing Interface). Since version 2.0, MPI has provided interfaces for direct memory access, mimicking shared-memory mechanisms.

Shared Memory A typical programming paradigm is multi-threading, where multiple threads concurrently access the same virtual address space. There are no "messages"; communications and synchronizations must be directly managed in shared memory.

A very widely used high-level standard is OpenMP (openmp.org), or Open Multi-Processing. On all platforms, a very low-level threading library is available. On POSIX systems it is named `pthread`.

On some systems, a software middleware can hide the physical details from the programmer and expose the memory of all nodes as a unified shared memory. In reality, remote memory access may still happen over the network under the hood.

…

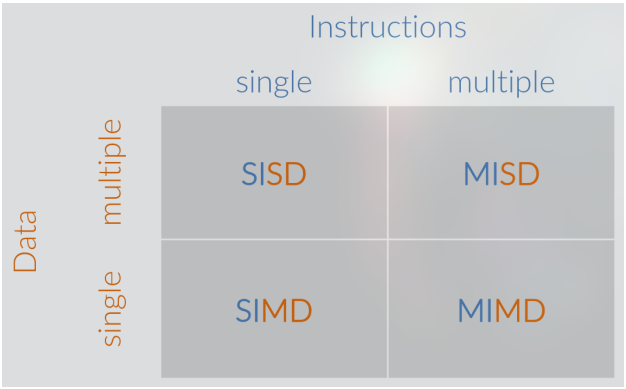
MPI and OpenMP in Practice The two (by far) most used implementations of distributed-memory and shared-memory paradigms are MPI (for distributed memory) and OpenMP (for shared memory), as mentioned above. OpenMP is delivered by the C/C++/Fortran compilers themselves by enabling OpenMP support via a command-line option (for example, `-fopenmp` for `gcc` and `icc`, `-mp` for `pgi` and `nvc`). MPI, on the other hand, is offered by various implementations such as OpenMPI (no relation to OpenMP), MPICH, and MVAPICH, among others.

…

Flynn's Taxonomy

The Flynn's taxonomy helps in understanding the logical subdivision of parallel systems, but it is no more up-to date; it mainly refers to HW capabilities but in 60yrs the HW evolved a lot and today we can imagine it refers to a mix of HW and SW

Figure 6.1: Flynn's Taxonomy



- **SISD** - Single Instruction on Single Data
- **MISD** - Multiple Instructions on Single Data
- **SIMD** - Single Instruction on Multiple Data
- **MIMD** - Multiple Instructions on Multiple Data

[HW level - SW level table]

We can now refine our HPC definition:

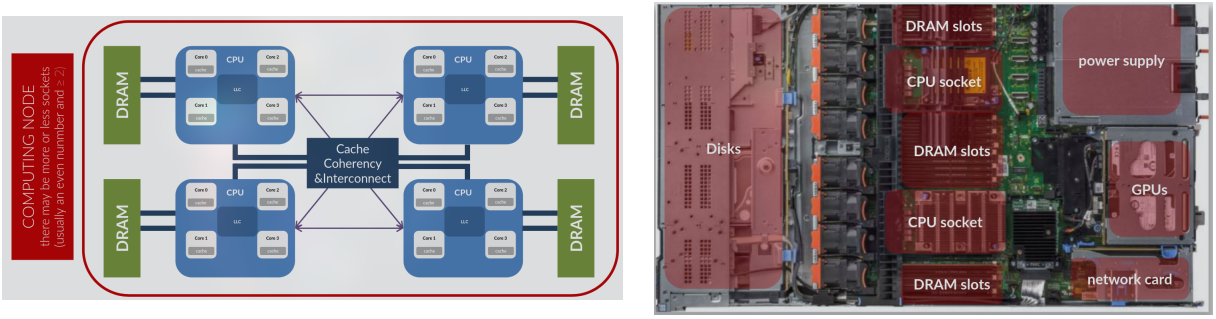
High Performance Computing (HPC) is the use of servers, clusters, and supercomputers - plus associated software, tools, components, storage, and services - for scientific, engineering, or analytical tasks that are particularly intensive in computation, memory usage, or data management.

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

…

In old times all the cores of th e cpu were connected to a Front Side Bus, which was a single bus connected at the same time a North Bridge and a South Bridge. The North Bridge was connected to the CPU and the RAM, while the South Bridge was connected to the I/O devices.

Nowadays, he NorthBridge has been moved inside the cpu and there are many (2-8) lanes that connect the cpu to the DRAM bank(s). The SouthBridge is what is called "the chipset"

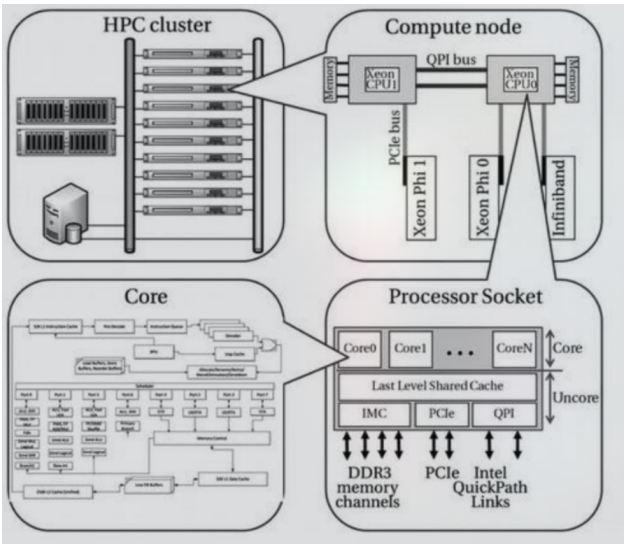


Nowadays, in general, as it's possible to see in the figure above HPCs has more than a socket connected to the same DRAM

The overall topology

Many (on the order of 10 to 10^5) nodes are connected by a switch-based network, whose topology may vary significantly. The details can severely affect overall performance. Typical figures for latency and bandwidth are around 1 µs and 100 Gbit/s, respectively. The most common standards are InfiniBand and OmniPath.

Figure 6.2: Flynn's Taxonomy



Note that on a supercomputer there is a hybrid approach to memory placement:

- The memory on a single node can be accessed directly by all its cores. This is called **shared memory**.
- When using many nodes together, a process cannot directly access the memory on a different node; it must issue a request. This is known as **distributed memory**.

These concepts describe physical memory accessibility but also refer to programming paradigms, as discussed later.

...

[little callback: NUMA vs UMA]

When we are in a NUMA architecture, the memory is not shared among all the cores. Each core has its own memory bank, and the access to the memory is not uniform. This means that some cores can access their own memory bank faster than others.

In this case, if a core wants to access the memory of another core, it has to go through a switch and it will take longer.

	0	1	2	3	4	5	6	7
0	10	12	12	12	30	30	30	30
1	12	10	12	12	30	30	30	30
2	12	12	10	12	30	30	30	30
3	12	12	12	10	30	30	30	30
4	30	30	30	30	10	12	12	12
5	30	30	30	30	12	10	12	12
6	30	30	30	30	12	12	10	12
7	30	30	30	30	12	12	12	10

The one in the table are the relative ”distances” between the nodes. This distance is not the physical distance, but a figure of merit that takes into account the time it takes to access the memory of the other cores.

The first four cores (0, 1, 2, 3) are in the same socket and they can access their own memory bank in 10 cycles and the memory of the other cores in 12 cycles. But when they access the memory of the other socket (4, 5, 6, 7), it takes 30 cycles.

6.2 Cache Coherence

Cache coherence ensures that multiple cores sharing data in separate caches observe a consistent view of memory. This is crucial in multi-core architectures, where performance can degrade significantly if data synchronization is poorly managed. In particular:

- When a memory location is accessed by two or more cores, each core typically holds that location in its private caches. If one core updates the data, it must be propagated to other caches holding a copy.
- When a thread is migrated from one core to another, its original cached data may still reside on the previous core, requiring updates or invalidations to keep the caches consistent.

Such synchronization overhead can be a severe performance bottleneck if data is frequently updated by multiple cores. Concurrent writing and migrating data between caches are two common sources of delays because hardware-level consistency mechanisms kick in often.

To address these issues, most modern systems use the **MESI** (Modified, Exclusive, Shared, Invalid) protocol (successor to the MSI protocol and precursor to MESOI). Its states are:

- **Modified (M):** The core has exclusively modified this cache line, and no other core has a valid copy.
- **Exclusive (E):** The core owns the cache line exclusively, but there have been no modifications yet. No other core holds a copy.
- **Shared (S):** Multiple cores share the cache line. Write attempts require notifying other cores or changing the state.
- **Invalid (I):** The cache line is invalid because another core modified it, or it is simply not in use.

Example: *MESI*

Suppose three threads (on three different cores) share a variable representing the wall-clock time. A “timer” thread (Thread0) updates this variable periodically, while the other two threads (Thread1 and Thread2) read its value. The MESI protocol ensures all cores see a

consistent value even though the data is replicated in each core's local cache.
[image]

…

Has we have seen, «performance» is a tag that can stand for many things.

In this frame, with «performance» we mean the relation between the computational requirements and the computational resources needed to meet those requirements.

$$\textbf{Performance} \approx \frac{1}{resources}, \qquad \textbf{Performance ratio} \approx \frac{resource_1}{resource_2}$$

where the resources are the time, the hardware (CPU, memory, etc.) and, if we want, money.

Key factors

n:	problem size
p:	number of computing units
$T_s(n)$:	Serial run-time for a problem of size n
$T_p(n)$:	Parallel run-time with p processors for a problem of size n
f_n:	Intrinsic sequential fraction of the problem
$k(n,t)$:	Overhead of the parallel algorithm
Speedup:	$S(n,p) = \frac{T_s(n)}{T_p(n)}$
Efficiency:	$E(n,p) = \frac{S(n,p)}{p} = \frac{T_s(n)}{pT_p(n)}$

The sequential execution time for a problem of size n is

$$T_s(n) = T_s(n) \times f_n + T_s(n) \times (1 - f_n)$$

Assuming that the parallel fraction of the computation is perfectly parallel, meaning that its run time scales as 1/p, then we can express the parallel execution time as:

$$T_p(n) = T_s(n,1) \times f_n + \frac{T_s(n)}{p} \times (1 - f_n)$$

and then:

$$\textbf{Speedup}: \quad S(n,p) = \frac{T_s(n)}{T_p(n)} = \frac{1}{f + \frac{1-f}{p}} \qquad \lim_{p \gg 1} S(n,p) = \frac{1}{f}$$

$$\textbf{Efficiency}: \quad E(n,p) = \frac{S(n,p)}{p} = \frac{1}{f(p-1)+1} \qquad \lim_{p \gg 1} E(n,p) = 0$$

…

Amdahl's Law

…

6.3 OpenMP

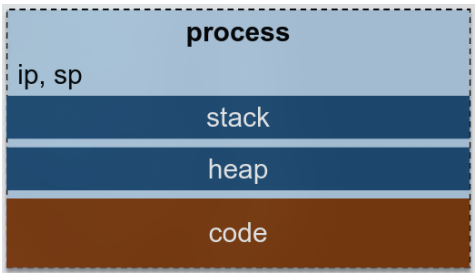
OpenMP is a standard API for shared-memory parallel programming. The name stands for **Open specifications for MultiProcessing**. It enables the development of multi-threaded programs with consistent, portable behavior by using a set of compiler directives embedded in the source code:

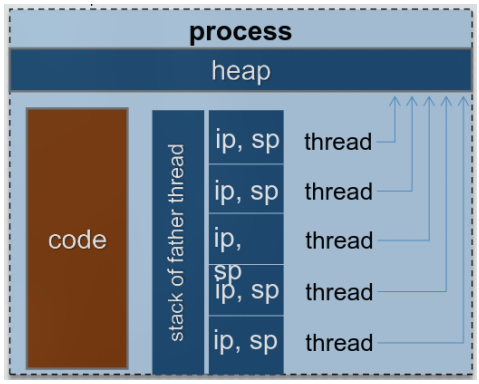
- **Pragmas** (`#pragma`) in C/C++
- **Specially formatted comments** in Fortran

OpenMP supports both fine-grained and coarse-grained parallelism, ranging from loop-level parallelism to explicit assignment of tasks to threads.

6.3.1 Threads and Processes

A **process** is an independent sequence of instructions and the ensemble of resources needed for their execution. A program needs much more than just its binary code (i.e. the list of ops to be executed): it needs to access to a protected memory space and to access system resources (e.g. files and network). A "process" is then a program that has been allocated with the necessary resources by the operating system. There may be different instances of the same program as different, independent processes.





A **thread** is an independent instance of code execution within a process. There may be from one to many threads within the same process. Each thread shares the same code, memory address space and resources than its father process. While each thread has its own private stack for local variables and function calls (allocated within the process's address space), they share access to the process's heap and global data segments. In geneal, spawning threads inside a process is much less costly than creating processes.

A thread can run either on the same computational units of its father process or on a different one.

A computational unit nowadays amounts to a core, either inside the same CPU (socket) on which the father process runs, or inside a sibling socket in the same NUMA region.

OpenMP vs MPI

Shared-Memory (e.g., OpenMP)

- A single process spawns multiple threads.
- All threads share a common address space and can directly access shared variables.
- Communication between threads occurs via shared variables in memory.
- Synchronization is required to avoid race conditions (e.g., critical sections, barriers).
- Typically used for parallelism within a single node (multi-core CPU).

Distributed-Memory (e.g., MPI)

- Multiple independent processes are launched, each with its own memory space.
- Processes cannot directly access each other's memory.
- Communication occurs by explicit message passing (send/receive).
- Synchronization is achieved via communication primitives (e.g., barriers, broadcasts).
- Suitable for parallelism across multiple nodes in a cluster or supercomputer.

Hybrid Programming: Modern HPC applications often combine OpenMP and MPI to exploit both shared-memory and distributed-memory parallelism. For example, OpenMP is used for intra-node (within a node) parallelism, while MPI handles inter-node (across nodes) communication.

Observation: *MPI Shared Memory Extensions*

Since MPI 3.0, the standard introduced features to:

1. Allow shared-memory-like access among MPI processes running on the same node (using `MPI_Win_allocate_shared`).
2. Enable direct remote memory access (RMA) to other processes' memory, known as Remote Memory Access (one-sided communication).

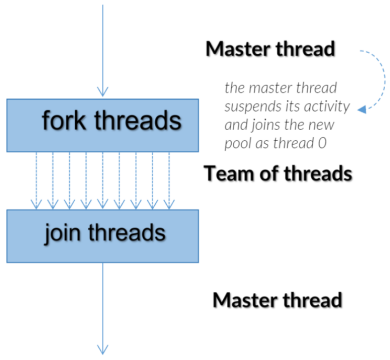
These features blur the strict boundary between shared and distributed memory, enabling more flexible hybrid programming models.

OpenMP programming model

The main advantages of a directive-based approach are the following ones:

- **Abstraction**: Subtleties of pthread and hardware-specific aspects are hidden. You can focus on data and workflow much more easily.
- **Efficiency**: The learning curve to achieve reasonable results is much hallower. The code's design is easier, the result/effort ratio is avourable with respect to pthread.
- **Incremental** approach No need to re-write your whole code. You start oncentrating on some sections only, following the suggestions from rofiling.
- **Portability**: The compiler will take care of this for you. You still ave to develop a design able to adapt to different topologies.
- **One source**: Through conditional compilation, serial and parallel ersions can easily coexist.

The main logic behind OpenMP is to use a single master thread for serial operations and to spawn a team of threads to perform parallel work. This is called ***fork-join* model**: a thread meets, at some point in its existence, a directive that activates the creation of a pool of children threads.



In OpenMP, threads operate by accessing and modifying shared memory regions. To prevent race conditions or situations where multiple threads attempt to read and write shared data simultaneously, synchronization mechanisms are employed, either explicitly (such as critical sections and locks) or implicitly (such as barriers at the end of parallel regions). Unlike distributed-memory paradigms, there is no explicit message-passing between threads; all communication occurs through shared variables.

However, parallelizing code is not always straightforward. For example, if a loop contains dependencies between iterations (loop-carried dependencies), it can severely limit or even prevent parallel speedup. Therefore, careful management of shared and private variable attributes is essential to minimize race conditions and reduce the need for synchronization.

Each thread executes its portion of the parallel workload in its own stack and private memory space, which are isolated from other threads and from the serial regions of the program. OpenMP also supports nested parallelism, allowing parallel regions to be created within other parallel regions if needed. Additionally, the number of threads used in a parallel region can be set dynamically, providing flexibility to adapt to different workloads or system resources.

6.3.2 OpenMP Directives

An OpenMP directive is a specially-formatted pragma for C/C++ and comment for FORTRAN codes. Most of the directives apply to structured code block, i.e. a block with a single input and a single output points and no branch within it.

The directives allows to:

- create team of threads for parallel execution
- manage the sharing of workload among threads
- specify which memory regions (i.e. variables) are shared and which are private to each threads
- drive the update of shared memory regions
- synchronize threads and determine atomic/exclusive operations

```
#pragma omp parallel
{
    // This code block will be executed by multiple threads in parallel
    ...
}
```

The lexical scope of structured blocks defines the static extent of an OpenMP parallel region. Every function call from within a parallel region determines the creation of a dynamic extent to which the same directives apply.

```
// static extent:
#pragma omp parallel
{
    double *array;
    int N;
    ...
    sum = foo(array, N);
    ...
}

// dynamic extent:
double foo(double *A, int N) {
    double mysum = 0;
    #pragma parallel for reduction(+:sum) // "orphan" directive
    for (int ii = 0; ii < N; ii++) {
        mysum += A[ii];
    }
    return mysum;
}
```

The dynamic extent includes the original static extent and all the instructions and further calls along the call tree.

Observation: *Dynamic Extent*

The functions called in the dynamic extent can contain additional OpenMP directives.

As we have seen, the region starts at the opening { brace and ends at the closing } one. An implicit synchronization barrier is present at the end of the region.

The general OpenMP directive in C/C++ is:

```
#pragma omp <directive> [<clause> [, <clause> [ ... ]]]
```

to associate multiple statements use a block:

```
#pragma omp <directive> [<clause> [, <clause> [ ... ]]]
{
statement;
statement;
}
```

with C++ syntax (requires ≥ 5.1)

```
[[ omp :: <directive> [<clause> [, <clause> [ ... ]]] ]]
[[ using omp : <directive> [<clause> [, <clause> [ ... ]]] ]]
[[ omp :: sequence( [omp::<directive>... [, omp::<directive>...]] ) ]]
```

A parallel region can be as short as a single line:

```
#pragma omp parallel _some_clauses_here_
single-line-here
```

There are no limits on the size of the code included within {...}:

```
#pragma omp parallel _some_clauses_here_
{ … }
```

The specific construct about for loops:

```
#pragma omp parallel for _some_clauses_here_
{ … }
```

A more general work-sharing construct

```
#pragma omp sections _some_clauses_here_
{ … }
```

This allows task-based parallelism

```
#pragma omp task _some_clauses_here_
{ … }
```

Observation: ***Threads and Parallel Regions***

For efficiency reasons, it may be, and usually it is, that the threads are not created/killed at the begin/end of each region; instead, they are created at the begin of the run and kept sleeping outside of the parallel regions.

6.3.3 shared/private memory

Let's now start to clarify the meaning of "private" and "shared" memory, and how to specify what variables are either one.

The basic rule is that everything that is defined in a serial region is inherited as shared in a parallel region that originates from the former serial region. Global variables are always shared (we'll see an exception)

```
int i, j, k;
double *array;

#pragma omp parallel
{
    // i,j,k and array
    // are shared here
    ...
}
```

The `private` directive

You can specify that a list of variables that exists in the local stack at the moment of the creation of the parallel region are **private** to each thread inside the parallel region.

```
int i, j, k;
double *array;

#pragma omp parallel private(i,k)
{
// j and array are shared here i and k are unique to each thread, they
    live in each thread's stack.

// They are different than the original j and k, and array does NOT
    point to the region pointed by the original array pointer
     ...
}
```

That means that the needed space will be reserved in the threads' stack to host variables of the same types. As such, those memory regions are different than the original ones, although within the parallel region those variables are referred in the source code with the same names.

Example: *Private variables*

```
int nthreads = 1;
int my_thread_id = 0;

#ifdef _OPENMP
#pragma omp parallel
{
    my_thread_id = omp_get_thread_num();
    #pragma omp master
    nthreads = omp_get_num_threads();
}
#endif
```

- all threads are writing in `my_thread_id`, in undefined order
- only the master thread is writing in `nthreads`

The value of `my_thread_id` unpredictable, because it depends on the run-time order by

which the threads access it and by each a thread accesses it again to write it.

Warning: *Private variables*

A private variable used in the parallel region refers to a memory region that is different than the same variable (outside) in the serial region: as such, this coding style looks only a source of confusion and lacks of clearness.
It's highly recommended to avoid using the "private" directive and to use a different name for the private variable, or to use the `threadprivate` directive (see below).

The `firstprivate` directive

If the clause `firstprivate` is used instead, every variable listed is private in the same sense than before, but its value is not randomic but it is *initialized* at the value that the corresponding variable in the serial region has at the moment of entering in the parallel region.

```
1  int i, j, k;
2  double *array;
3  array = (double*)malloc(...);
4  #pragma omp parallel
5  firstprivate(array)
6  {
7  // now array is unique to each thread, BUT each copy is initialized to
       the value that the original array has at the entry of the parallel
       region.
8
9  // As such, now array can be used to access the previously allocated
       memory.
10 ...
11 }
```

The `lastprivate` directive

The clause `lastprivate` pertains only to the for construct. When used, every variable listed is private in the same sense than before, and its value is not initialized.

```
1  double last_result;
2
3  #pragma omp parallel
4  lastprivate(last_result)
5  {
6  ...
7      #pragma omp for
8      for( int j = 0; j < some_value; j++ )
9      last_result = calculation( j, ...);
10     ...
11 }
12
13 other_calculations( last_result, ... );
14
15 // at this point, last_result has the last value from the last iteration
       in the for loop in the parallel region
```

What is affected is the value that the original variable, the one that is declared in the master thread's stack, has after the parallel region. It will have the value that the private copy of it has in the last iteration of the for loop.

The `threadprivate` directive

The clause `threadprivate` applies to global variables and has a global scope. threadprivate variables are private variables that do exist all along the lifetime of the process.

I.e. they are private variables that do not die in between of two different parallel regions.

```
1  int myN;
2  double *array;
3  #pragma omp threadprivate(myN, array)
4
5  #pragma omp parallel
6  {
7      // array does exist and 'its private
8      // to each thread
9      array = ... allocate memory...;
10 }
11 // .. something serial here..
12 #pragma omp parallel
13 {
14     // array does exist here and
15     // the allocated memory is available
16 }
```

Warning: *Threadprivate and Dynamic Threads*

When using threadprivate, the dynamic thread creation is not allowed, i.e. the number of threads in each parallel region is constant.

The `copyin` directive

The clause `copyin` applies to parallel and worksharing (e.g. for) constructs. This clause basically provides a way to perform a **broadcast of `threadprivate` variables** from the master thread (i.e. the thread 0) to the corresponding threadprivate variables of other threads.

```
1  int golden_values[3];
2  #pragma omp threadprivate(golden_values)
3
4  for( int i = 0; i < N; i++ )
5  {
6      get_golden_values();
7      #pragma omp parallel copyin(golden_values)
8      {
9          // each thread uses golden_values[]
10     }
11 }
```

The copying happens at the creation of the region, before the associated structured block is executed.

The `copyprivate` directive

The clause `copyprivate` applies only to `single` construct.

This clause provides a mechanism to propagate the values of private variables, including threadprivate, from a thread to the others inside a parallel region.

The copying is ultimated before any threads leave the implicit barrier at then end of the construct.

```
#pragma omp parallel
{
    double seed[2];

    #pragma omp single copyprivate(seed)
    {
        // something happens here and the
        // thread that executes this block
        // initializes the seed[2] array
    }
    // at this point the values of the seed[2]
    // array have been propagated to all the
    // other threads
}
```

Other to know about OpenMP

OpenMP is made of 3 components:

1. **Compiler directives** give indication to the compiler about how to manage threads internals
2. **Run-time libraries** linked by the compiler
3. **Environment variables** set by the user, determine the behaviour of the omp library; for instance, the number of threads to be spawned or the requirements about the thread-cores-memory affinity

By default, when the compiler is instructed to activate the processing of OpenMP directives:

```
gcc -fopenmp …
icc -fopenmp …
pgcc -mp …
```

it defines a macro that let you to conditionally compile sections of the code:

```
#ifdef _OPENMP
my_thread_id = omp_some_function();
#endif…
```

Observation: `_OPENMP`

The macro `_OPENMP` is automatically defined by the compiler when OpenMP support is enabled (e.g., with `-fopenmp`). You can use it to conditionally compile code that should only be included when OpenMP is available.
This approach allows your code to remain portable and functional even when compiled without OpenMP support. It is especially useful for writing hybrid codes (e.g., MPI+OpenMP) or for

debugging and performance comparisons between serial and parallel versions.

```
#ifdef _OPENMP
    // OpenMP-specific code here
#else
    // Fallback or serial code here
#endif
```

Draft

7
Lecture 29/04/2025

As we have seen when we discussed the modern architectures, a unique central memory with a fixed bandwidth would be a major bottleneck in a system with a fast growing number of cores/sockets and sockets. The problem is avoided by physically disjointing the memory in separated units (the memory banks) each of which is connected to a socket; All the sockets are inter-connected so that each core can access all the memory and a cache- coherency system “glues” the data. This way, the resulting aggregated bandwidth scales as the number of sockets (although, we know, the cache-coherency becomes the new limiting factor). However, the major drawback is that the access time is no more uniform. This has severe consequences on how you have to write and run your codes.

OpemMP and OS offer the capacity to decide where each thread have to tun, i.e. on which core and/or how the threads have to distribute on the available cores.

We know that each core may have the capability of running more than one thread, which is called (□) Simultaneous MultiThreading (SMT). In the next slides, let’s call strands or hwthreads (hardware threads) the different threads that a physical core could run, as opposed to “swthreads” (software threads) that indicates the OpenMP threads.

The placement of OpenMP threads on cores is called “threads affinity”.

MISSING: part to ”Threads affinity - PLACES”

Places

To pass OpemMP the information about the placement of threads, we have to use the environment variable `OMP_PLACES`:

```
export OMP_PLACES = { sockets | cores | threads }

# examples:
    # 1 socket
    export OMP_PLACES = “{0}:4:12
    # 2 sockets
    export OMP_PLACES = “{0:11,48:11},{24:12,72:12}
```

MISSING: part to ”Threads affinity - BIND”

Binding

Binding defines how the OpenMP software threads (swthreads) are mapped onto hardware *places*.

Binding Types

- **NONE**: Placement is left up to the operating system (OS).
- **CLOSE**: Threads are placed as close as possible to each other (assigned in a round-robin way).
- **SPREAD**: Threads are distributed as evenly as possible, then filled in round-robin.

- **MASTER**: Threads run on the same place as the master thread.

Using `OMP_PROC_BIND`

You can specify binding in OpenMP using the environment variable:

```
export OMP_PROC_BIND = { false | true | master | close | spread }
```

- `false`: No placement policy; OS decides and may migrate threads.
- `true`: No placement policy; OS decides but cannot migrate threads.
- `master`, `close`, `spread`: Specify exact placement; OS cannot migrate threads.

8
Message Passing Interface

The Message Passing Interface (MPI) is a standardized library that implements the distributed memory programming model. Its primary purpose is to facilitate data movement between distinct address spaces in a parallel computing environment.

In MPI, each process operates as an independent entity with its own private memory space. From a programming perspective, memory management within each process follows the same principles as sequential programming, making it conceptually straightforward for developers.

Processes interact through explicit message passing for both synchronization and data exchange. This communication model requires explicit collaboration between processes - data is only shared when processes explicitly send and receive messages.

It's important to note that MPI is implemented as a library rather than a programming language. This means that all parallel operations are performed through library function calls, which provides flexibility in terms of language integration while maintaining a consistent interface across different platforms.

So here must exist a way to send a message from one process to another; this method should accept the following parameters:

- the message to send
- the length of the message
- the receiver process
- the framework this message belongs to

In the same way, there must be a way to receive a message from another process; this method should accept the following parameters:

- where to store the message
- the length of the message
- the sender process
- the framework this message belongs to

So, we expect that there must be point-to-point

- a way to send messages
- a corresponding way to receive messages
- a way to know "the names" in "the framework"

Since sometimes we need to broadcast messages, it would be nice if:

- there was a broadcasting (one-to-many) mechanism
- there was a collection mechanism (many-to-one) mechanism
- the answer/collection was possible many-to-many

Communicators

Communicators and groups are a very central concepts in MPI: tasks can form groups (a task can belong to more than one group) and the same group can be in different situations.

The ***communicator*** is the combination of a group and its "context".

You can build as many groups as you want, and they may or not have a communicator. However, if you want to communicate among tasks in a group, you need a communicator.

This functionality offers the capability of isolating communication between application modules with an effective "sandbox" for different contexts.

For instance, a parallel library and your application will use internally their own communicator, separating contexts.

- By creating groups of MPI processes, that may or not overlap with each other, it is possible to
  - separate contexts within different modules of the same application (useful or even advisable)
  - express multiple levels of parallelism

MPI provides several predefined communicators that are available after initialization:

- `MPI_COMM_WORLD` is the default communicator available right after the call to `MPI_Init`. Its group contains all the tasks started by your job.
- `MPI_COMM_NULL` signals an invalid or non-existent communicator. This is often used as a return value to indicate errors in communicator operations.
- `MPI_COMM_SELF` contains only the process itself. This is useful when a process needs to communicate with itself.
- `MPI_GROUP_NULL` signals an invalid or non-existent group. Similar to `MPI_COMM_NULL`, it's used to indicate errors in group operations.

These predefined communicators serve as fundamental building blocks for MPI communication patterns and are essential for both basic and advanced MPI programming.

Send and Receive

MPI provides two basic functions for sending and receiving messages between processes:

```
int MPI_Send(
    const void *buf,         /* starting address of send buffer */
    int count,               /* number of elements in send buffer */
    MPI_Datatype datatype,   /* datatype of each buffer element */
    int dest,                /* rank of destination */
    int tag,                 /* message tag */
    MPI_Comm comm            /* communicator */
)
```

```
int MPI_Recv(
    void *buf,               /* starting address of receive buffer */
    int count,               /* number of elements in receive buffer */
    MPI_Datatype datatype,   /* datatype of each buffer element */
    int source,              /* rank of source */
    int tag,                 /* message tag */
    MPI_Comm comm,           /* communicator */
    MPI_Status *status       /* status object */
)
```

The data type is a fundamental concept in MPI. It defines the type of data being sent or received. MPI provides a variety of predefined data types, including:

MPI DataType	C DataType
MPI_CHAR	char
MPI_BYTE	unsigned char
MPI_SHORT , MPI_UNSIGNED_SHORT	(unsigned) short int
MPI_INT , MPI_UNSIGNED_INT	(unsigned) int
MPI_LONG , MPI_UNSIGNED_LONG	(unsigned) long int
MPI_LONG_LONG , MPI_UNSIGNED_LONG_LONG	(unsigned) long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	--

Table 8.1: Correspondence between MPI predefined datatypes and C datatypes.

Example: *Send and Receive Example*

```
1  int N;
2  if ( Myrank == 0 )
3      MPI_Send( &N, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
4  else if ( Myrank == 1 ) {
5      MPI_Status status;
6      MPI_Recv( &N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
7  }
8  else if ( Myrank == 1 )
9      MPI_Recv( &N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE );
```

NOTE: `MPI_STATUS_IGNORE` is always valid instead of putting an `MPI_Status` varible's address as last argument of `MPI_Recv`

Example: *Sending things of different types*

```c
typedef struct {
    int i, j;
    double d, f;
    char s[4];
} my_data;

unsigned int length = N * sizeof(my_data);

if (Myrank == 0) {
    MPI_Send(data, length, MPI_BYTE, 1, 0, MPI_COMM_WORLD);
} else if (Myrank == 1) {
    MPI_Recv(data, length, MPI_BYTE, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```

…

MISSING: probe

…

When we do send a message, we specify the memory region that contains the data, at what point in the future it is safe to modify the memory region involved in the send? Or better, how can we be sure that the communication has ended and the data have all been received?

The MPI standard prescribes that `MPI_Send` returns when it is safe to modify the send buffer. So, whenever `MPI_Send` returns, it is safe to act on the memory region that has been sent.

…

MISSING: protocols

…

Code that rely on the system's bufferization to run correctly are called ***unsafe***.

Solution to cure, or better, to avoid the unsafe situations are:

- designing more carefully the communication pattern
- checking the runnability by substituting `MPI_Send` with `MPI_Ssend`
- using `MPI_Sendrecv`
- supplying explicitly buffer with `MPI_Bsend`
- non-blovking operations

deadlock

```
if (Rank == 0) {
  Recv ( from 1 );
  Send ( to 1 ); }
if (Rank == 1) {
  Recv ( from 0 );
  Send ( to 0 ); }
```

unsafe

```
if (Rank == 0) {
  Send ( from 1 );
  Recv ( from 1 ); }
if (Rank == 1) {
  Send ( to 0 );
  Recv ( from 0 ); }
```

safe

```
if (Rank == 0) {
  Recv ( from 1 );
  Send ( to 1 ); }
if (Rank == 1) {
  Send ( to 0 );
  Recv ( from 0 ); }
```

Figure 8.1: Deadlock, Unsafe and Safe code

MISSING: boh

There are different routines to send and receive messages:

mode	routine	notes
standard	`MPI_Send`	Safe to modify data once returns. Equiv. to synchronous or asynchronous mode (uses sys buffers) depending on msg size and implementation choices.
synchronous	`MPI_Ssend`	Completes when the receive has started (it does not wait for the receive completion). Unsafe communication patterns will deadlock → a way to check safety.
asynchronous or ”buffered”	`MPI_Bsend`	Completes after the buffer has been copied. Needs an explicit buffer.
ready	`MPI_Rsend`	Mandatory that the matching receive has already been posted. May be the fastest solution, but it is quite problematic.
all	`MPI_Recv`	One size serves ’em all

Table 8.2: Different MPI send and receive routines

MISSING: examples

Non-Blocking communications

Until now we have considered point-to-point communication functions that do not return until some conditions are met (either the copy of the data into a buffer or the actual delivery of the data to the recipient, in case of the sender, or the actual arrival of the data into their local destination in case of the receiver).

As such, the caller is blocked into the call and cannot perform any other operation. Those functions are consequently identified as ***blocking functions***.

If what follows the Send/Recv depends on the fact that the operations mentioned above actually completed, then the usage of those functions reflects an actual dependency and there is little to be done.

However, if there are other instructions that could be executed while waiting for the data to arrive at destination, by using blocking functions we are losing parallelism.

To obviate to this issue, MPI offers the non-blocking functions, i.e. a set of functions that return immediately.

However, their return does not mean that the communication has completed but only that it has been posted on an internal queue system that will execute it at some point in the future.

To assess, at any moment, whether the communication has been executed, MPI provides dedicated routines:

```
1 MPI_Test ( MPI_Request *, int *flag, MPI_Status *)
2 MPI_Wait ( MPI_Request *, MPI_Status *)
3 MPI_Waitall (int count, MPI_Request array_of_req[], MPI_Status
      array_of_st[] )
```

The non-blocking counterparts of the send and receive functions are:

```
1 int MPI_Isend( void *buf, int count, MPI_Datatype dtype,
2                int dest, int tag, MPI_Comm comm,
3                MPI_Request *request );
```

```
4
5 int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
6                int source, int tag, MPI_Comm comm,
7                MPI_Request *request );
```

The request variable is used to handle the status of the `MPI_Isend` or `MPI_Irecv` operation posted. At any point after the call, the status can be determined by the immediately-returning call

Tip: ***Non-blocking communication pattern***

When using non-blocking communication, always follow this three-step pattern:

1. **Post the operation**: Call `MPI_Isend` or `MPI_Irecv` to initiate the communication. This returns immediately with a request handle.
2. **Overlap computation**: Perform other useful work while the communication proceeds asynchronously in the background. This is where the performance benefit comes from.
3. **Ensure completion**: Use `MPI_Wait` (blocking) or `MPI_Test` (non-blocking check) to verify the operation has finished before accessing the communicated data.

Warning: ***Critical safety rule***

Never access the send buffer (for sends) or receive buffer (for receives) until the operation is confirmed complete via `MPI_Wait` or a successful `MPI_Test`. Violating this rule can lead to race conditions, data corruption, and undefined behavior.

…

All the sending routines have a correspondent non-blocking version:

mode	Blocking routine	Non-blocking routine
standard	`MPI_Send`	`MPI_Isend`
synchronous	`MPI_Ssend`	`MPI_Issend`
asynchronous or ”buffered”	`MPI_Bsend`	`MPI_Ibsend`
ready	`MPI_Rsend`	`MPI_Irsend`
all	`MPI_Recv`	`MPI_Irecv`

Table 8.3: Comparison of blocking and non-blocking MPI communication routines

…

MISSING: boh

8.0.1 Collective operations

Collective communication operations in MPI are fundamental for coordinating tasks and exchanging data among multiple processes simultaneously. These operations always occur within a specific group of processes defined by a communicator.

Key characteristics of collective communications include:

- **Group-wide participation**: All processes within the communicator’s group must participate in the collective operation. If even one process in the group does not call the collective function, the program will likely hang or behave unpredictably.

- **Implicit synchronization**: Collective operations often imply a synchronization point. Processes may block until all participating processes have reached the collective call. This synchronization can lead to some degree of serialization, potentially impacting overall parallelism.
- **Optimized algorithms**: MPI implementations typically use sophisticated and highly optimized algorithms for collective operations to ensure efficiency across various network topologies and system architectures.
- **Blocking and non-blocking variants**: Similar to point-to-point communication, many collective operations have both blocking and non-blocking versions (e.g., `MPI_Bcast` and `MPI_Ibcast`). Non-blocking collectives allow for potential overlap of computation and communication.

Collective operations can be broadly categorized into three main classes:

1. **Synchronization**: These operations are used to synchronize processes within a group. The most common example is `MPI_Barrier`, which blocks each process until all processes in the communicator have called it.
2. **Data Movement**: These operations involve distributing, gathering, or re-arranging data among processes.
3. **Collective Computation (Reductions)**: These operations perform a computation (e.g., sum, max, min, logical AND) across data provided by all processes in the communicator, with the result being available at one (e.g., `MPI_Reduce`) or all (e.g., `MPI_Allreduce`) processes.

We will now discuss the most common collective operations.

Synchronization

The `MPI_Barrier` function is a collective synchronization operation. Its primary purpose is to ensure that all processes within a specified communicator reach a certain point in their execution before any of them proceed further.

The function call `int MPI_Barrier(MPI_Comm comm)` takes a communicator `comm` as an argument. A process calling `MPI_Barrier` will block until all other processes in the group associated with `comm` have also called this function. Once all processes have reached the barrier, they are all unblocked and can continue execution. While `MPI_Barrier` is useful for synchronizing tasks, it should be used judiciously.

Like any synchronization mechanism, it can introduce serialization into the parallel execution, potentially limiting performance gains by forcing faster processes to wait for slower ones.

Collective data movement

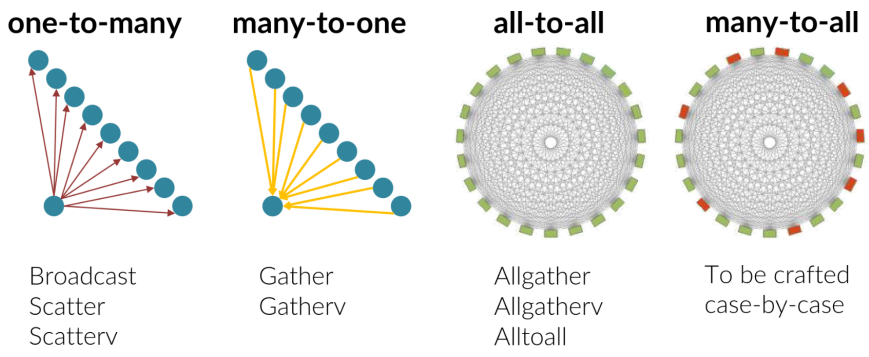
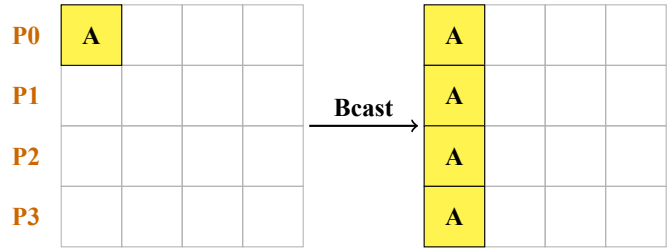
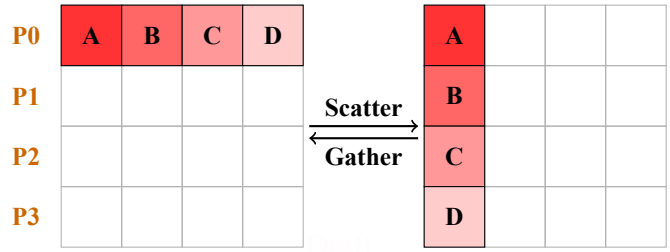


Figure 8.2: Data movement operations

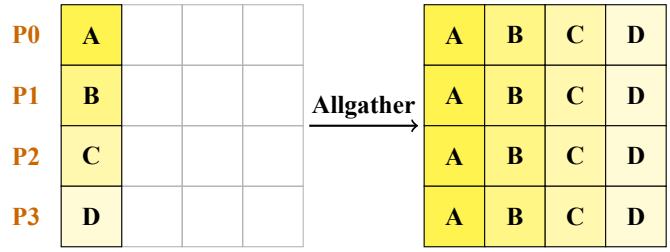
- **Broadcast:** The `MPI_Bcast` function is one of the most fundamental collective operations in MPI. It allows a single process (the root) to send the same data to all other processes in the communicator. The root process (P0) holds the data (A) and broadcasts it to all other processes. After the broadcast, all processes have a copy of the data.



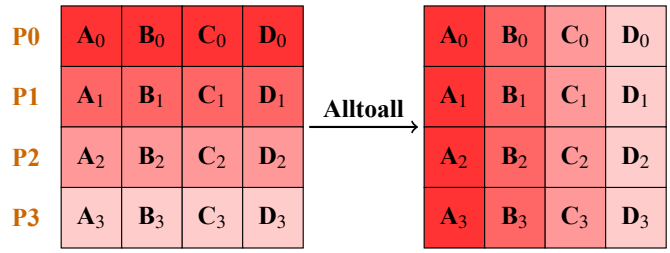
- **Scatter and Gather:** The `MPI_Scatter` and `MPI_Gather` functions are complementary collective operations that distribute and collect data among processes. `MPI_Scatter` takes data from a single process (the root) and distributes it among all processes, while `MPI_Gather` does the opposite - it collects data from all processes and combines it at the root.



- **Allgather:** The `MPI_Allgather` function is a collective operation that allows all processes in a communicator to exchange data with all other processes. It collects data from all processes and combines it at the root.



- **Alltoall:** The `MPI_Alltoall` function is a collective operation that allows all processes in a communicator to exchange data with every other process. Each process sends data to every other process, and each process receives data from every other process.



Bibliography

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

Draft