UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence

Department of mathematics informatics and geosciences

High Performance Computing

Lecturer:
Prof. Stefano Cozzini, Prof. Luca Tornatore

Author:
Christian Faccio

September 22, 2025

github.com/christianfaccio christianfaccio@outlook.it

Preface

As a student of Data Science and Artificial Intelligence, 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.

Contents

1 Introduction

1.1 Priors

Expressing Performance

Here we introduce a metrics to estimate the performance of a code in exploiting some of the CPU's resources. Specifically, we will focus on (i) how **fast** a code is and (ii) how **well** a code exploits the instruction-level parallelism capability of a CPU. More in detail, these metrics are more suited for loops, the main bottleneck of most scientific codes.

A CPU's activity is regulated by an internal clock whose pace is of the order of billions of ticks per seconds. Consider that typically a CPU frequency is between 2 and 4 GHz, and if someone (like me) is not updated with physics, it means that a single cycle runs in $\frac{1}{\text{frequency}}$ seconds, meaning around $0.5 - 0.35$ ns.

Usually, one should consider retrieving different metrics and not a single one, and the wall clock time (wct) per se isn't the best metric. Instead, one should focus on quantifying the number of clock cycles spent on a single section since it is more informative than knowing how many seconds it required to execute. Thus, we are interested in knowing how much efficient our code is **per-element** and not per-iteration, since our implementation may be able to process more elements per iteration. Basically, we have access to 3 different types of time:

1. **Wall clock time (wct):** is the time you can find in your nearest physical clock and follows the *POSIX* system, meaning it shows the number of seconds elapsed since the start of the Unix epoch at 1 January 1970 00:00:00 UT.
2. **System time (st):** is the amount of time the system as a *whole* spent executing the code (it includes I/O, system calls, etc.)
3. **Process user-time (ut):** is the amount of time the CPU spent executing the code's instructions.

How can one measure all these times? There are two main ways:

- **Outside** the code, using *perf* profiler of the *time* command
- **Inside** the code, using different functions:
  - `gettimeofday()`: returns the time elapsed since the epoch in seconds and microseconds
  - `clock_t clock()`: returns the number of clock ticks since the program
  - `int clock_gettime(clockid_t clk_id, struct timespec *tp)`: returns the time of the specified clock clk_id
  - `int getrusage(int who, struct rusage *usage)`: returns the resource usage for the calling process, its children, or the whole system

Consider as an example:

Example:

```
#define CPU_TIME (clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &ts ), \
                  (double)ts.tv_sec + (double)ts.tv_nsec * 1e-9)
...
Tstart = CPU_TIME;
// your code segment Here
Time = CPU_TIME - Tstart;
```

Tip: ***Accumulation***

Note that running a program without **binding** it to a specific code and a specific memory bank may result in a non-optimal behaviour. Thus, it is a good practice to ask the OS not to migrate the code using as an example `taskset` or `numactl`:

- `numactl -H` exposes the topology of the node
- `numactl --cpunodebin=n` bind the execution to code n
- `numactl --membind=n` bind the memory to memory bank associated with core n

Pointers in C

A **pointer** is essentially a variable (a memory location of fixed size like 8B in 64 bits systems) that points to a memory address. It can point to an integer (2B), a float (4B) or even an array of 10Gb of items. **De-referencing** a pointer means to get to the pointer variable, meaning the memory location that the variable occupies and getting the value stored in that memory location.

The **memory** is essentially a long 1D string of bytes (8 bits) and every one of them can be uniquely identified by its distance from the byte 0.

Remember that:

- **char** → 1B
- **short int** → 2B
- **(long) int** → 4B
- **long long int** → 8B
- **floating-point single precision** → 4B
- **floating-point double precision** → 8B

(Please refer to this PDF for more details about floating point representation).

Example: *Pointers practice*

```c
#include <stdio.h>

char *c; // points to a char
double *d; // points to a floating-point double precision number
struct who_knows *w; // points to a struct who_knows

c = 0x123456; // assign a value to a pointer variable
          // \* or c = &my_preferred_letter \* where &var is the
             address operator

** ---------------------------------------------------------- **

var = 123456        // &var = 12

ptr = (void*)&var   // (void*) is a type cast that
            // converts the pointer to a
            // generic type
            //
            // ptr = 12
            // *ptr = 123456
```

Example: *Pointer arithmetic*

```c
#include <stdio.h>

int main() {
    int array[8];   // each element has size sizeof(int)
            //
                // &array[i] = &array[0] + i*sizeof(int)
            // &array[i] - &array[j] = (i-j)*sizeof(int)

    for (int i=0; i<8; i++){
        array[i] = i;   // assign a value
    }

    int *ptr1 = &array[0];  // should point to first address
    int *ptr2 = &array[7];

    printf("%d\n", *ptr1);  // prints 0
    printf("%d\n", *ptr2);  // prints 7
    printf("%p\n", &array[0]);  // prints 0x16f2a6d68

    return 0;
}
```

Example: *Allocating memory*

```c
#include <stdio.h>
#include <stdlib.h> // for malloc

int main() {
    int array[8]    // stack allocation
    int *array = malloc( sizeof(int) * 8 )  // heap allocation
    free(array) // must free manually

    // to allocate multidimensional arrays:
    // define array as a pointer to type**
    int **array;

    // allocate n pointers to int*
    array = (int**)malloc( 8 * sizeof( int* ))

    // for each pointer, allocate enough memory to retain m int
    for (int i=0; i<8; i++)
        array[i] = (int*)malloc( 8 * sizeof( int ));
}
```

Observation: *Pointers to functions*

How are **functions** represented in memory? A function is essentially a block of instructions, that are translated in machine code (a sequence of bytes) and stored in the memory. Variables are not called unless the function is called. Thus, a function can be pointed to by a pointer.

```c
#include <stdio.h>
#include <stdlib.h>

int (*func)(); // pointer to a function that returns an int
int (*func)(int, double); // pointer to a function that takes an
   int and a double as input and returns an int
```

Example: *Pointers zoo*

What is `char **monster` ? Well, simply a pointer to a pointer that points to a char. Recalling that a pointer is just a variable, and hence it has a memory address, a pointer to a pointer is just a variable that points to the memory address of another variable.

```c
#include <stdio.h>
#include <stdlib.h> // for malloc

int main() {
    int x=42;                   // variable 42
    int *ptr = &x;              // ptr=address, *ptr=42
    int **ptr_to_ptr = &ptr;    // ptr_to_ptr=address of address,
                                // *ptr_to_ptr=address,
                                // **ptr_to_ptr=42
    printf("%p\n", ptr_to_ptr);     // 0x16d5f6d80
    printf("%p\n", *ptr_to_ptr);    // 0x16d5f6d8c
    printf("%d\n", **ptr_to_ptr);   // 42
}
```

This is the how we access command-line arguments in C:

```c
int main(int argc, char **argv) {
    // argc is the number of arguments
    // argv is a pointer to a pointer to arrays of char
    // **argv is the first argument (the first string)
}
```

The command line arguments are strings, and in C strings are represented by arrays of characters. When a string is passed as argument to a function, it decays to a pointer to its first element. Command line arguments are an array of strings, which are arrays of characters: this is why we use `char **argv` .
Then we can define the Pointers zoo:

- `monster` : phrase (array of strings), ex. `"hello world"`
- `*monster` : pointer to monster -> 1st string (`"hello"`)
- `**monster` : pointer to `*monster` -> 1st char of 1st string (`'h'`)

so that `*monster = "hello"` , `*(monster+1) = "world"` , `**monster = 'h'` , `*(*monster+1) = 'e'` and so on.

INT representation

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int a = atoi(*(argv+1));    // atoi converts string command line
       args to int
    int b = atoi(*(argv+2));
    printf("%d x %d = %d\n", a, b, a*b);
}
```

```
./int_repr 100000 21000 → 2100000000
```

`./int_repr 100000 22000 → -1594967296`

Why is that? Essentially there is an **overflow** in the representation of integers. In C, an integer is typically represented using 4 bytes (32 bits) in a format called **two's complement**. This format allows for the representation of both positive and negative integers, but it has a limited range and if the result of an operation exceeds this range, it wraps around, leading to unexpected results.

1.2 HPC 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. [2]

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.

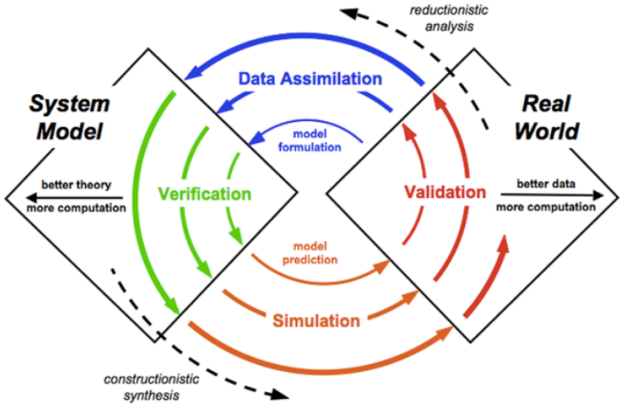


Figure 1.1: Research and Development

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. [2]

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.

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.

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.

2 Hardware and Software

2.1 HPC Hardware

Serial Computers

A serial computer is a machine that processes one instructinon at a time and is defined by the **Von Neumann architecture** you can see in Figure 2.1:

- **Control Unit:** fetches instructions from memory, decodes them, and executes them by coordinating the activities of the other components.
- **Arithmetic Logic Unit (ALU):** performs arithmetic and logical operations on data.
- **Registers:** small, high-speed storage locations within the CPU that hold data and instructions temporarily during processing.
- **Memory:** stores data and instructions that the CPU needs.

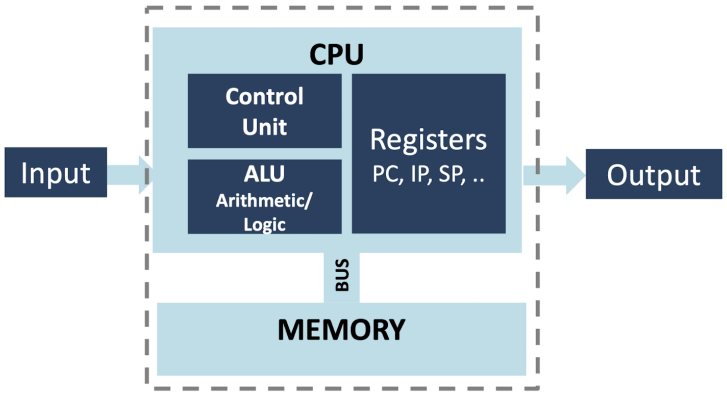


Figure 2.1: Von Neumann Architecture

Here, the CPU fetches instructions from memory **one at a time**, decodes them, and executes them sequentially. This architecture is simple and easy to understand, but it has some limitations, such as the **Von Neumann bottleneck**, which refers to the limited bandwidth between the CPU and memory that can slow down performance. Moreover, memory is **flat**, meaning that there is no hierarchy or differentiation between different types of memory. This can lead to inefficiencies in data access and processing, as the CPU may have to wait for data to be fetched from slower memory locations.

Moore Law

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

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

Definition:

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

~Gordon Moore, 1965

Dennard Scaling → from Moore's Law to performance

Definition:

"Power density stays constant as transistors get smaller"

~Robert H. Dennard, 1974

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

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

Intuitively:

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

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

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.

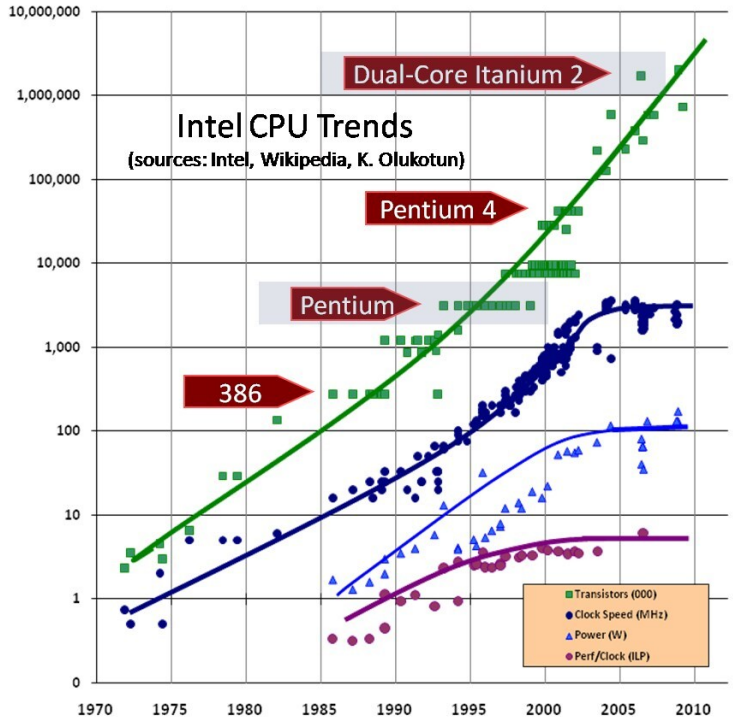


Figure 2.2: Moore's Law

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.

2.2 Parallel computers

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.

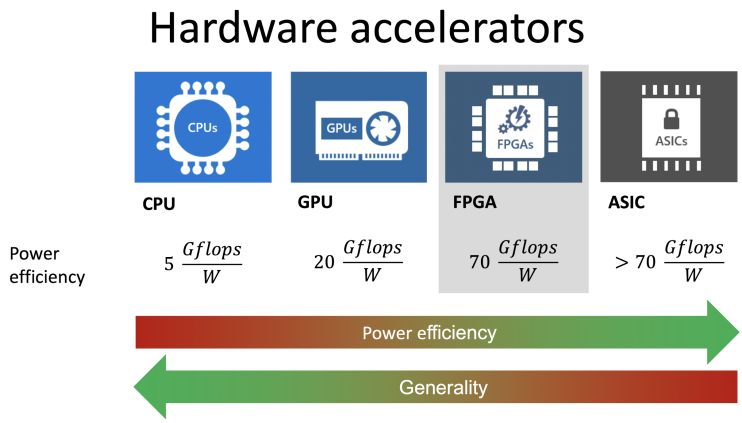


Figure 2.3: Hardware accelerators

Definition:

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

Observation: ***Nomenclature***

- **CPU** = Central Processing Unit = Processor = Socket
- **Core** = execution unit within a CPU
- **Thread** = hardware thread = virtual core

Thus, a **multiprocessor** is a server with more than 1 CPU, while a **multicore** is a CPU with more than 1 core.

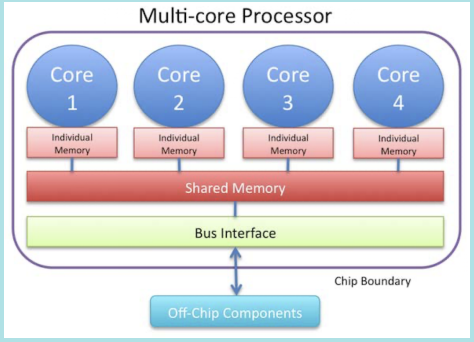


Figure 2.4: Multicore processor

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 2.1.

	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 2.1: 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.

Essential Components of a HPC Cluster

- Several computers (nodes) with multiple sockets, each with multiple cores
- One or more networks (interconnects) to hook the nodes together
- One or more accelerators (GPUs, FPGAs, TPUs, ...)
- One or more levels of memory (cache, RAM, ...)
- One or more levels of storage (SSD, HDD, ...)
- A login/access node

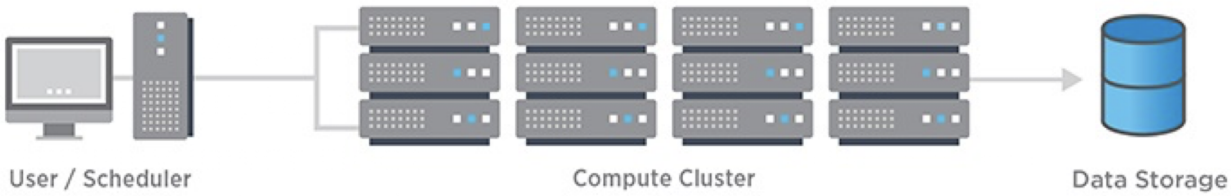


Figure 2.5: HPC Cluster Architecture

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.

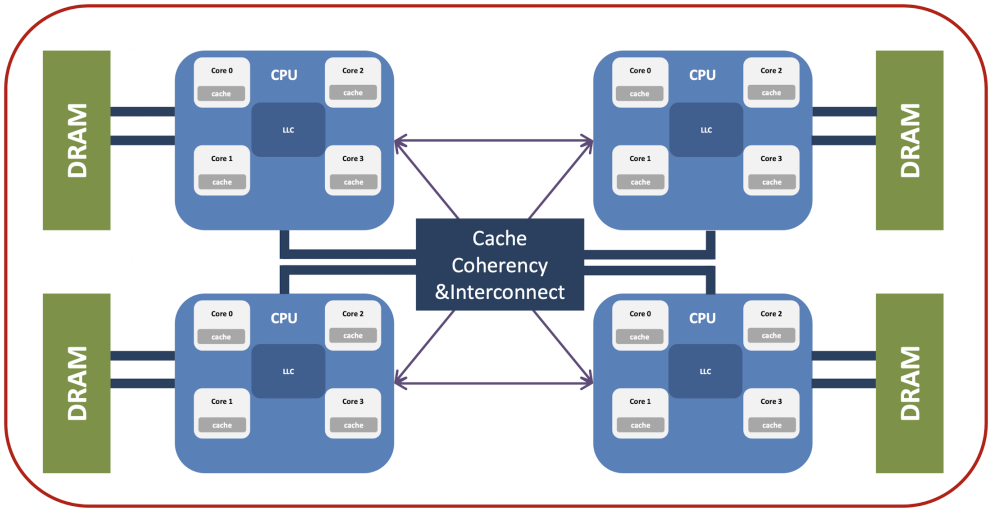


Figure 2.6: Node topology

There are different networks for different purposes:

- **High-speed interconnect:** used for communication between nodes, typically using InfiniBand or high-speed Ethernet. Low latency and high bandwidth are critical for performance.
- **Management network:** used for administrative tasks, such as monitoring and managing the cluster. This network is typically separate from the high-speed interconnect to ensure that management tasks do not interfere with application performance.
- **I/O network:** used for data transfer between the compute nodes and storage systems. This network is optimized for I/O requests (NFS and/or parallel FS), latency is not as critical as bandwidth. Typical choices are Ethernet or Fibre Channel.

Memory

What about 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.

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

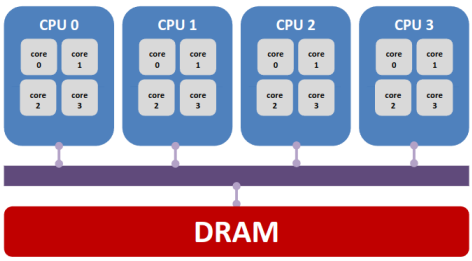
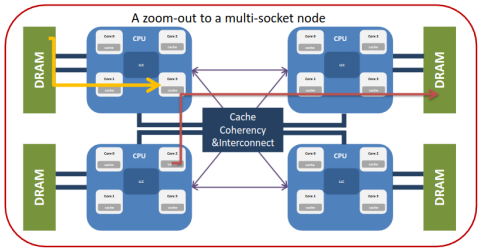


Figure 2.7: UMA



Non-uniform memory access (NUMA): Each processor has its own local memory, but can also access memory local to other processors. Accessing local memory is faster than accessing non-local memory.

Figure 2.8: NUMA

Warning: *Challenges for multicore processors*

- Need parallel algorithms to exploit multiple cores
- Aggravates memory wall problem (cores compete for memory bandwidth)
- Cache sharing and coherency issues

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

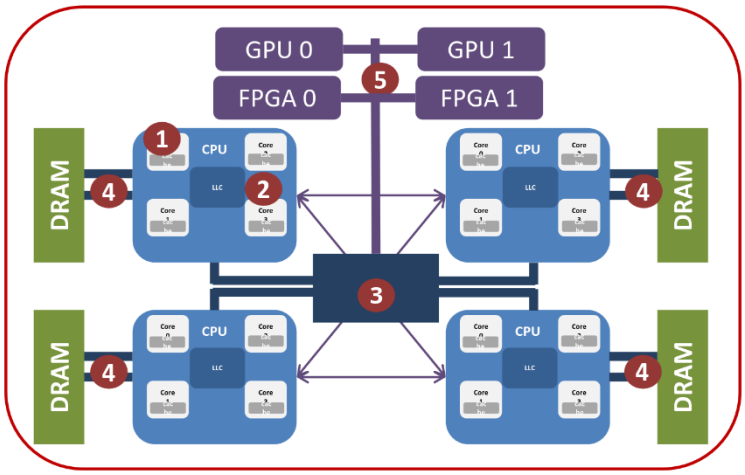


Figure 2.9: Levels of parallelism

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.

2.3 HPC Software

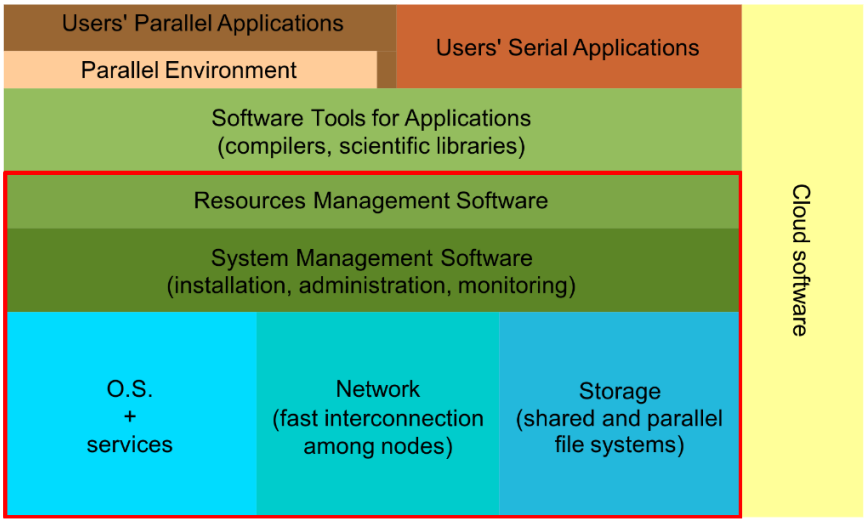


Figure 2.10: The Software Stack: in red the *cluster middleware*

The Cluster middleware

The cluster middleware (the one in the red box in Figure 2.10) 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 2.11, these resources form an interconnected ecosystem where efficient allocation directly impacts overall system performance and user satisfaction.

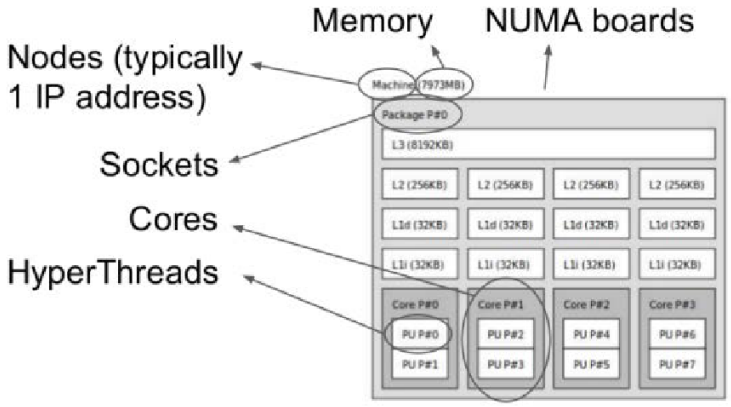


Figure 2.11: Resources in a HPC cluster

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.

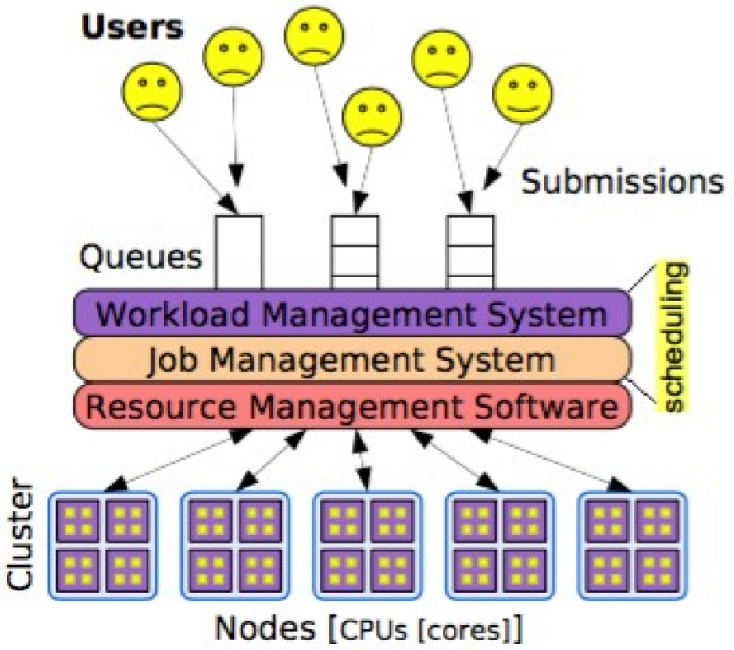


Figure 2.12: Batch Scheduler Architecture

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.

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.

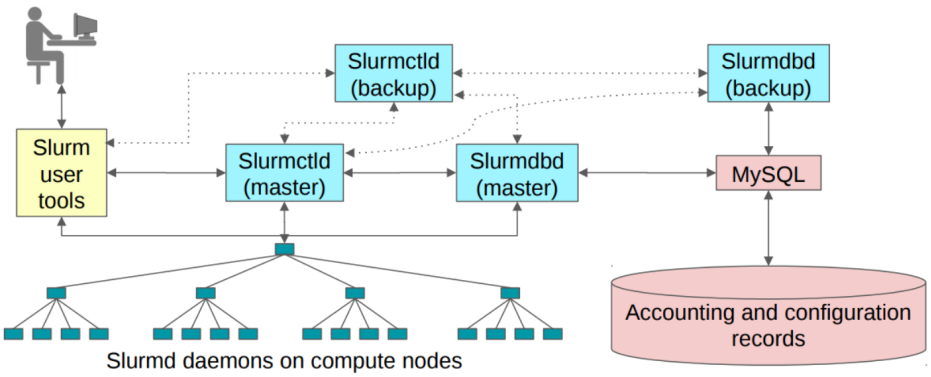


Figure 2.13: Simplified SLURM Architecture

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.

Scientific Software

Scientific software encompasses a wide range of applications and libraries designed to facilitate complex computations, simulations, and data analysis in various scientific domains. It is composed by different layers:

- **User's applicaations (both parallel and serial):** these are the applications that the users run on the cluster. They can be either commercial or open-source.
- **Parallel libraries:** these are libraries that provide parallel functionality to the user's applications. They can be either commercial or open-source.
- **Mathematical/Scientific libraries:** these are libraries that provide mathematical and scientific functionality to the user's applications. They can be either commercial or open-source.
- **I/O libraries:** these are libraries that provide I/O functionality to the user's applications. They can be either commercial or open-source.
- **Compilers:** these are the compilers that the users use to compile their applications. They can be either commercial or open-source.

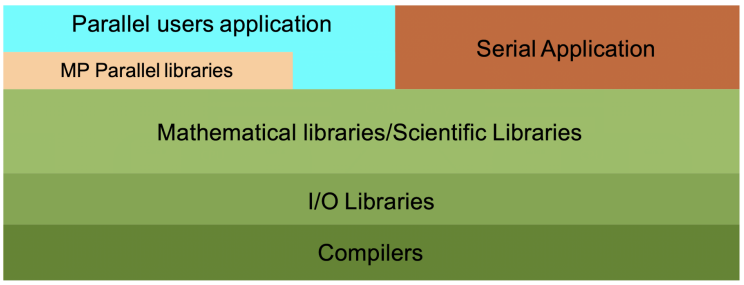


Figure 2.14: Scientific Software Stack

There is not much **standardization** for scientific software in HPC, as every app has a different software stack. This is done to get the best possible performance for the specific application. Usually usability and reuse are traded off for performance. This creates a lot of problems for the users, as they have to learn how to use different software stacks for different applications. Moreover, the software stack is usually not portable, as it is optimized for a specific architecture. Here rises the **Dependency Nightmare**, where different applications require different versions of the same library, which can lead to conflicts and incompatibilities.

Usually the scientific software is installed in `/opt/cluster/software` (or similar) and mounted read-only on the nodes using NFS. It is generally managed using environment modules, which allow users to load and unload different software stacks as needed. Modules allow to dynamically modify the user's environment (`PATH`, `LD_LIBRARY_PATH`, etc.) to use different versions of the same software. Some commands are:

- `module avail`: list all available modules
- `module load <module>`: load a module
- `module unload <module>`: unload a module
- `module list`: list all loaded modules

```
cozzini@login02 ~] > module avail

------------------------------------------------- /orfeo/opt/modules/tools -------------------------------------------------
  IGV/2.18.0              bwa-mem2/2.2.1       gromacs/2022.6      (D)    ont-guppy-cpu/6.5.7 (D)    singularity/3.10.4
  R/4.3.3                 cmake/3.28.1         hwloc/2.10.0               ont-guppy-gpu/6.2.1        singularity/3.11.5 (D)
  R/4.4.1           (D)   cutadapt/4.2         java/8-8u402b06            ont-guppy-gpu/6.5.7 (D)    trim_galore/0.6.10
  STAR/2.7.11b            fastp/0.23.4         java/11.0.22               picard/3.2.0
  bcftools/1.17           fastqc/0.12.1        java/17.0.10               plink/1.90
  bcl2fastq2/2.20.0       foldseek/8-ef4e960   java/21.0.2         (D)    sambamba/1.0
  bedtools2/2.31.1        gromacs/2018.4       ont-guppy-cpu/6.2.1        samtools/1.17

-------------------------------------------------- /orfeo/opt/modules/mpi --------------------------------------------------
  openMPI/4.1.6/gnu/14.2.1

------------------------------------------- /orfeo/opt/modules/libraries -------------------------------------------
  cuda/11.7    cuda/12.0        cuda/12.6.3         openBLAS/0.3.26-omp
  cuda/11.8    cuda/12.1 (D)    cutensor/2.0.2.5    openBLAS/0.3.26     (D)
```

Figure 2.15: Orfeo Environment Modules

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.

3 Optimization

3.1 Preliminaries

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

Observation: ***Premature optimization***

Premature optimization is the root of all evil.

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

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

You’d better start thinking in terms of ”improved” code.

Do neither add unnecessary code nor duplicate code.

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

Writing **comments** is a good practice, but do not overdo it. Comments should explain **why** something is done, not **what** is done (the code itself should be clear enough to explain what is done).

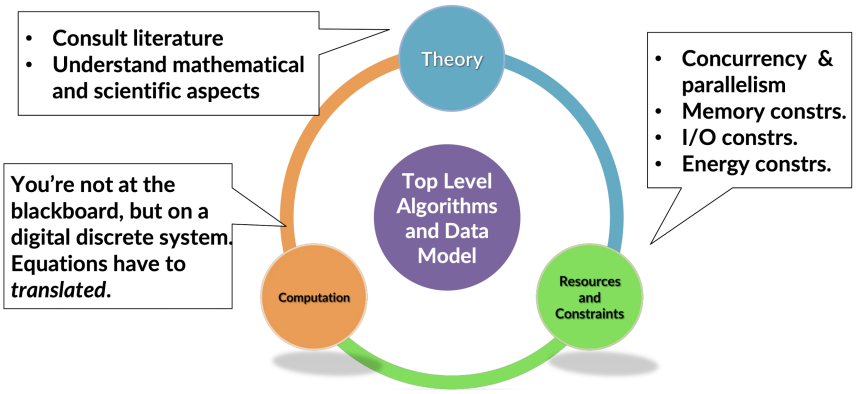


Figure 3.1: Design

Testing is part of the design, and it is a key step in the optimization process (unit test, integration test, system test). **Validation** ensures that the code does what it was meant to do, and ensures the results are correct. **Verification** ensures that the codes does what it does correctly.

First things first

1. The first goal is to have a program that delivers the **correct answers** and behaves correctly under all conditions.
The code must be as much clear, clean, concise, and documented as possible.

2. The first step towards optimization is to adopt the **best-suited algorithms** and data structures. What "best-suited" means must be related to the constraints framing your activity (time-to-solution, energy-to-solution, memory, ...).

3. The second step is that the **source code be optimizable by the compiler**.
Then, you must have a firm understanding of the compiler's capabilities and limitations, as well as those of your target architecture.
Understand the best trade-off between portability and "performance" (accounting for the human effort into the latter).

4. The third step is to get a **data-driven, task-based workflow**, which possibly, almost certainly, will be parallel in either distributed- or shared-memory paradigms, or both.

5. Profile the code under different conditions (workload/problem size, parallelism, platforms, ...) and **spot bottlenecks and inefficiencies**.

6. Apply optimization techniques by modifying hot-spots in your code to **remove optimization blockers** and/or to better expose intrinsic instruction/data parallelisms.

7. IF (needed) GOTO point 1.

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

Observation: ***Compiler job***

Recalling the job of the compiler:
- **Syntax analysis** (parsing)
- **Semantic analysis** (type checking)
- **Intermediate code generation**
- **Optimization**
- **Code generation**

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

To optimize your code through the compiler, use `-Os` flag, where `s` typically is 1,2,3 and refers to the level of optimization.
`-O3` is the highest level of optimization, and it is the most aggressive one.
It is not granted, though, that the highest level of optimization is the best one for your code.
For instance, sometimes expensive optimization may generate more code that on some architecture (e.g. with smaller caches) run slower, and using `-Os` may be better.

Obviously, the compiler knows the architecture it is compiling on, but it will generate a **portable** code, i.e. a code that can run on any cpu belonging to that class of architecture. Using appropriate switch (in gcc `-march=native -mtune=native`), the compiler will generate code that is optimized for the specific architecture it is running on.

Profile-guided optimization is a technique that uses the information gathered by the compiler when the code is run to optimize the code. Compilers are able to instrument the code so to generate run-time information to be used in a subsequent compilation.

Knowing the typical execution patterns enables the compiler to perform more focused optimizations, especially if several branches are present.

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

Tip: ***Some C-specific hints***

- `extern`: Global variables, exist forever.
- `auto`: Local variables, allocated on the stack for a limited scope and then destroyed.
- `register`: Suggests that the compiler puts this variable directly in a CPU register.
- `const`: Indicates that this variable won't be changed in the current variable's scope.
- `volatile`: Indicates that this variable can be accessed and modified from outside the program.
- `restrict`: A memory address is accessed only via the specified pointer.

Observation: ***Optimization blockers***

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

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

Memory Aliasing

Memory Aliasing necessitates some attention.

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

Consider the following functions:

```c
void func1 (int *a, int *b) {
    *a += *b;
    *a += *b;
}
void func2 (int *a, int *b) {
    *a += 2 * *b;
}
```

An incautious analysis may conclude that a compiler should prefer `func2()`, since it contains less pointers. However, is it really true that the two functions behave the same way in all possible conditions? What if **a** = **b** (if a and b point to the same memory location)?

```c
// a and b point to the same memory location, and let's say that *a = 1
void func1 (int *a, int *b) {
    *a += *b; // -> *a and *b now contains 2
    *a += *b; // -> *a and *b now contains 4
}
void func2 (int *a, int *b) {
    *a += 2 * *b; // -> *a and *b now contains 3
}
```

This condition, i.e., when 2 pointer variables reference the same memory address is called **memory aliasing** and is a major performance blocker in those languages that allows pointer arithmetic like C and C++.

Example: *Memory aliasing*

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

// 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.

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

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

3.2 Modern Architectures

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

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

Figure 3.2: At a glance

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

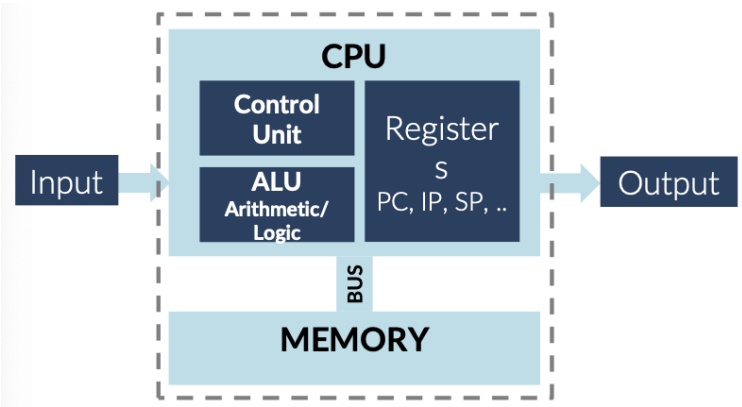


Figure 3.3: Von Neumann architecture

Today, instead:

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

Table 3.1: Cache levels

Level	Size	Latency (cycles)	Bandwidth (GB/s)	Location
L1	32-64KB	3-4	1000+	On-chip
L2	256KB-1MB	10-12	100-300	On-chip
L3	2-32MB	30-40	20-100	On-chip
RAM	4-512GB	100-200	10-20	Off-chip

We have seen that the power required per transistor is

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

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

To cope with the power wall one can:

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

Observation:

CPU became faster than memory in the early 90s.

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

The cache itself has a hierarchy:

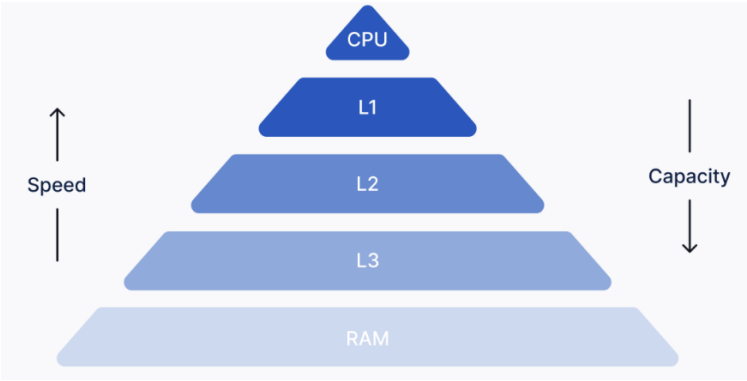


Figure 3.4: Memory hierarchy

Definition: ***Principle of Locality***

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

There are two types of locality:

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

Cache Coherency

There is a difference between **SMP (Symmetric Multi-Processing)** and **Distributed NUMA (Non-Uniform Memory Access)**. In SMP, all processors share the same physical memory and have equal access to it, while in NUMA, each processor has its own local memory and can access remote memory with higher latency. Consider an SMP node, with tens of sockets interconnected by a bus (a colletive interconnect in which messages are broadcasted and everyone is listening for a message dedicated to itself).

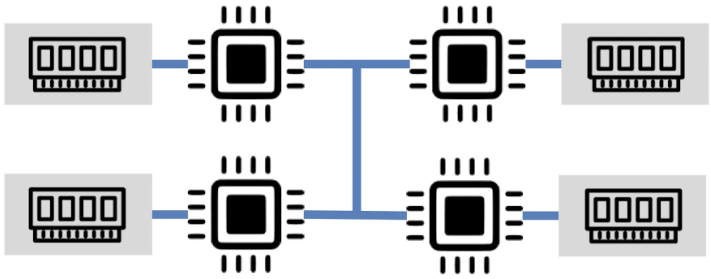


Figure 3.5: SMP architecture

The memory is **shared**, and everybody sees the whole amount of RAM.

Let's now say that the CPUs have caches and some data are loaded in more than one cache. What happens to all the caches and actual data in memory when one CPU modifies the data? This is called the **cache coherency problem**, and the overwhelming difficulty and cost to manage it on too large SMP nodes is the main limit to their size. So, after having introduced the hierarchy of cache memories, you have to deal with a strong memory hierarchy and the fact that your CPU is much faster than the central memory.

Even if you are good in designing the data model and the workflow of your code, that complexity may result in a real performance disaster if you do not understand how the memory hierarchy works and how to exploit it. For instance, the CPU may stall for hundreds of cycles waiting for data coming from RAM, while it could be doing something else if the data were in cache. This leads to the so called "memory wall" and thus to some improvements in the architecture, like the cache hierarchy.

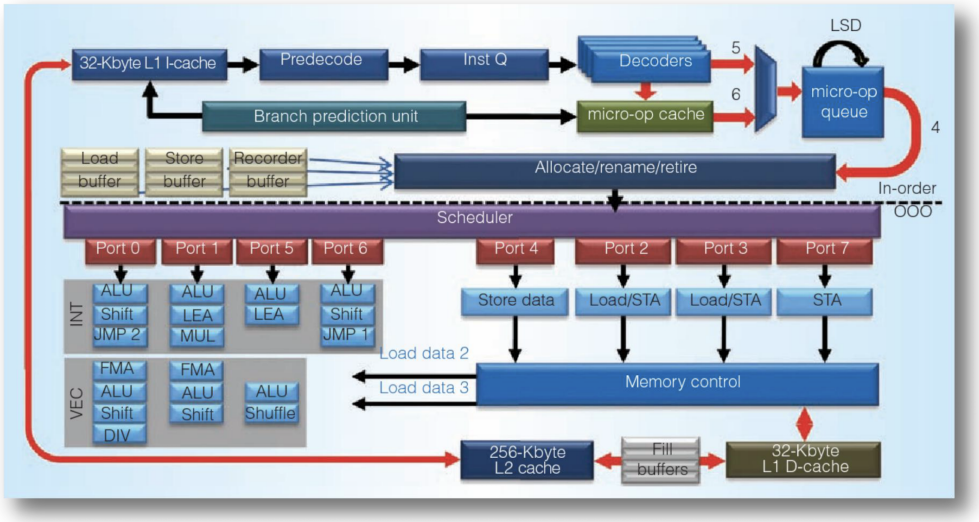
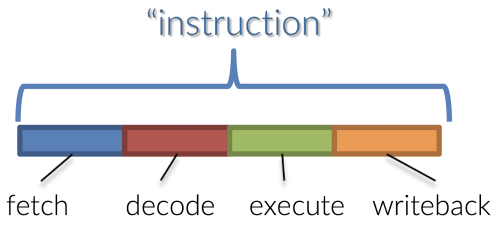


Figure 3.6: 6^{th} generation SkyLake micro-arch.

- **Multiple ports:** more than 1 port is available to execute CPU instructions, although different units have different specializations (ALU, LEA, SHIFT, FMA, ...). This is **superscalar capacity**, meaning the capacity of executing more than 1 instruction per cycle.
- **Front-end:** it basically fetches instructions and the data they operate on from instruction and data caches, decodes instructions, predicts branches and dispatches the instructions to different ports.
- **Back-end:**: it is responsible for the actual instructions execution and for the back-writing of results in memory locations. It is responsible for orchestrating out-of-order operations' execution depending on their instructions/data dependencies.

Pipelining

Nowadays, CPUs are **pipelined**, meaning that the execution of an instruction is broken down into several stages, each one executed in a different clock cycle.



1. **Fetch**: the instruction is fetched from memory.
2. **Decode**: the instruction is decoded and the operands are fetched from
3. **Execute**: the instruction is executed.
4. **Write-back**: the result is written back to memory.

If all the four stages take $\sim 400ps$, then we would obtain a throughput of 2.5GIPS (Giga Instructions Per Second). It is the time required to get a result from an instruction and thus the latency of it. However, if we were able to detach the four stages, we could increase the throughput, since we could have 4 instructions in the pipeline at the same time, each one in a different stage.

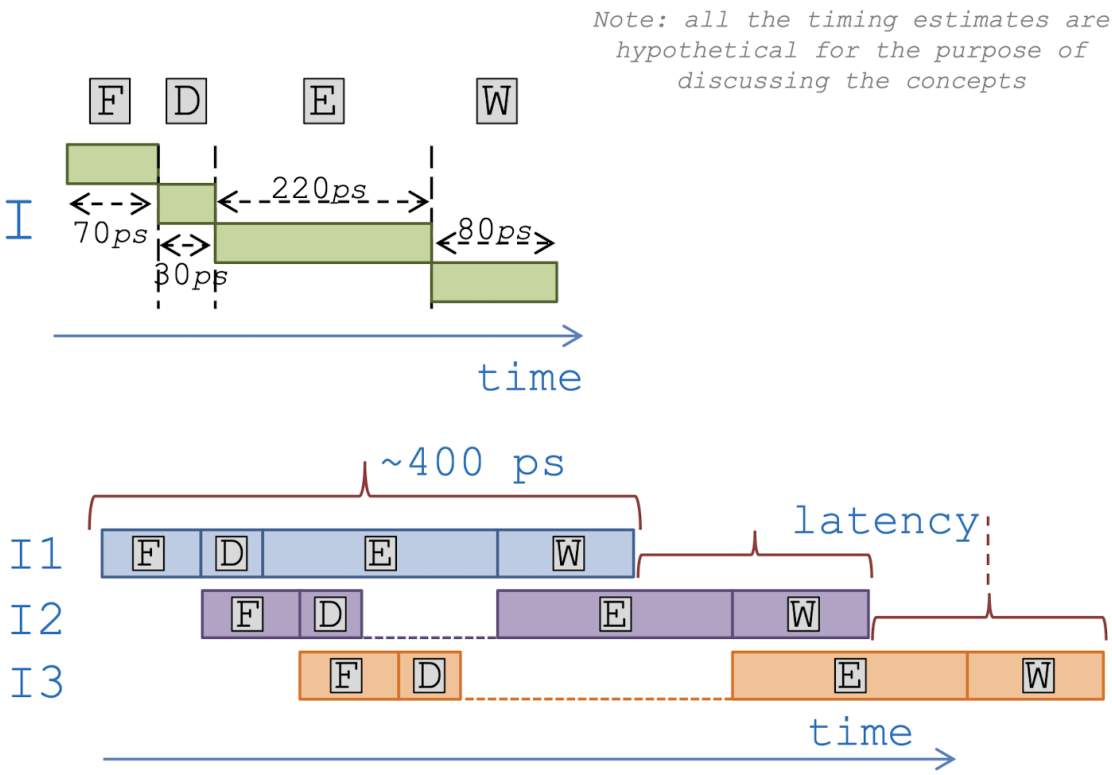


Figure 3.7: Pipelining

If many independent logical units exist to perform each step, they could operate subsequently on **different instructions**. This is called **instruction-level parallelism** (ILP), and it is a key feature of modern CPUs.

However, pipelining has some limitations:

- **Data hazards**: when an instruction depends on the result of a previous instruction
- **Control hazards**: when the pipeline is flushed due to a branch instruction
- **Structural hazards**: when two instructions require the same hardware resource

Vector registers are special registers that can hold multiple data elements, allowing the CPU to perform the same operation on multiple data elements simultaneously. This is called **data-level parallelism** (DLP) or **single instruction, multiple data** (SIMD).

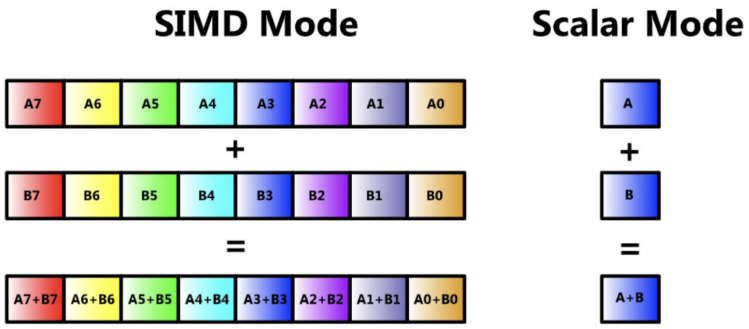


Figure 3.8: Vector registers

3.3 Avoid the avoidable

Let's consider the following example:

- We have a distribution of random data points on a 3D plane which we subdivide in sub-regions using a grid.
- For each point p, we want to collect all the grid cells whose center is closer to p than a given radius r, and to perform some operations accordingly.

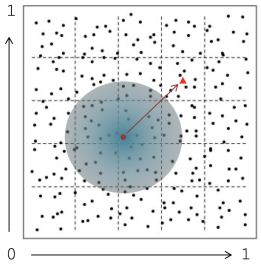


Figure 3.9: Points and grid cells

```
for (int p=0; p<Npoints; p++){ //loop over points
  for (int i=0; i<Ncells; i++){ //loop over cells in x dim
    for (int j=0; j<Ncells; j++){ //loop over cells in y dim
      for (int k=0; k<Ncells; k++){ //loop over cells in z dim
        // assuming 3D grid spans from -half_size to +half_size in each
           dim
        double dist = sqrt(
          pow(x[p] - (double)i/Ncells - half_size, 2) +
          pow(y[p] - (double)j/Ncells - half_size, 2) +
          pow(z[p] - (double)k/Ncells - half_size, 2)
        );
        if (dist < r){
          // do something with cell(i,j,k)
        }
      }
    }
  }
}
```

This code is very inefficient, even if correct. Let's try to improve it:

1. **Avoid expensive function calls:** functions like `sqrt()`, `pow()` and `floating point division` are very expensive, it is better to avoid (or reduce) them as much as possible. Moreover, the computation `(double)i/Ncells - half_size` was performed $N^3 + N^2 + N$ times, always returning the same values. We can precompute them outside the innermost loop.

```
// precompute cell centers
double i[Ncells], j[Ncells], k[Ncells];
for (int idx=0; idx<Ncells; idx++){
  i[idx] = (double)idx/Ncells - half_size;
  j[idx] = (double)idx/Ncells - half_size;
  k[idx] = (double)idx/Ncells - half_size;
}

for (int p=0; p<Npoints; p++){ //loop over points
  for (int i=0; i<Ncells; i++){ //loop over cells in x dim
    for (int j=0; j<Ncells; j++){ //loop over cells in y dim
      for (int k=0; k<Ncells; k++){ //loop over cells in z dim
        dx = x[p] - i[i];
```

```
14          dy = y[p] - j[j];
15          dz = z[p] - k[k];
16
17          dist2 = dx*dx + dy*dy + dz*dz; // avoid pow()
18          if (dist2 < r*r){ // avoid sqrt()
19            // do something with cell(i,j,k)
20          }
21        }
22      }
23    }
24  }
```

2. **Clarify the variable's scope:** variables should be defined in the smallest possible scope. This helps the compiler to optimize the code better, and it also helps the programmer to understand the code better. For example, we can move the computation of `dx` and `dy` in their smallest possible scope.

```
1  for (int p=0; p<Npoints; p++){ //loop over points
2    for (int i=0; i<Ncells; i++){ //loop over cells in x dim
3      dx = x[p] - i[i];
4      for (int j=0; j<Ncells; j++){ //loop over cells in y dim
5        dy = y[p] - j[j];
6        for (int k=0; k<Ncells; k++){ //loop over cells in z dim
7          dz = z[p] - k[k];
8
9          dist2 = dx*dx + dy*dy + dz*dz; // avoid pow()
10         if (dist2 < r*r){ // avoid sqrt()
11           // do something with cell(i,j,k)
12         }
13       }
14     }
15   }
16 }
```

3. **Suggest what is important:** If there are some variables that are often calculated and reused subsequently, keeping a register dedicated to them may be useful. Note that this is a suggestion to the compiler, after analyzing the code it may decide to ignore it.

```
1  double register Ng_inv = 1.0/Ncells; // suggest to keep in register
2  for (int idx=0; idx<Ncells; idx++){
3    i[idx] = (double)idx*Ng_inv - half_size;
4    j[idx] = (double)idx*Ng_inv - half_size;
5    k[idx] = (double)idx*Ng_inv - half_size;
6  }
7
8  for (int p=0; p<Npoints; p++){ //loop over points
9    for (int i=0; i<Ncells; i++){ //loop over cells in x dim
10     dx = x[p] - i[i];
11     for (int j=0; j<Ncells; j++){ //loop over cells in y dim
12       dy = y[p] - j[j];
13       double register dist2_xy = dx*dx + dy*dy; // suggest to keep
             in register
14       for (int k=0; k<Ncells; k++){ //loop over cells in z dim
15         double register dz = z[p] - k[k]; // suggest to keep in
```

```
                register
16          double register dist2 = dist2_xy + dz*dz; // suggest to
                keep in register
17
18          if (dist2 < r*r){ // avoid sqrt()
19            // do something with cell(i,j,k)
20          }
21        }
22      }
23    }
24  }
```

Observation: *Importance of being earnest*

Paying attention to the scope of the variables and keeping local what is local has a twofold advantage:

- All the local variables will reside in the stack, however also the stack is better to be clearly organized, so that the CPU can manage it better. For instance in the example above the variables `dx`, `dy`, `dz` are defined in the smallest possible scope, so that they can be allocated and deallocated in the stack as soon as they are needed, and they are all used packed together in the innermost loop.
- Keep your mind clear and sharp. Think carefully to what you need and where, and make it clear.

Moreover, **don't suppose the compiler is always able to re-arrange calculations**, it may be able to do that for integers but not for floating point numbers, since the order of operations may change the result due to rounding errors. Floating point math is commutative but not associative. For better understanding look at [1] or this link.

4. **Don't repeat unnecessary checks:** if you have to check a condition that is always true or always false, do it once outside the loop and then use a flag to avoid checking it again and again. Moreover, if you are checking a condition that does not change, as an example if you are checking if `dist < r*r` you know that `r*r` does not change, so you can compute it once outside the loop.
5. **Avoid unnecessary memory accesses:** if you are accessing a memory location that is not changing, as an example if you are accessing `x[p]`, you can store it in a local variable and use it instead of accessing the memory location again and again.

Let's now check how much time we have saved with these optimizations:

Table 3.2: Time comparison

Version	Time (s)
Original	0.216138
Optimized	0.020303
Optimized + `-O3`	0.000006

- **Original code:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define CPU_TIME ({ struct timespec ts; clock_gettime(
    CLOCK_PROCESS_CPUTIME_ID, &ts); (double)ts.tv_sec + (double)ts.
    tv_nsec * 1e-9; })

int main(int argc, char **argv){
    int Npoints = 1000;
    int Ncells = 20;
    double half_size = 1.0;
    double r = 0.1; // example radius

    // Example point arrays (you'd initialize these with actual data)
    double *x = malloc(Npoints * sizeof(double));
    double *y = malloc(Npoints * sizeof(double));
    double *z = malloc(Npoints * sizeof(double));

    double Tstart = CPU_TIME;

    for (int p=0; p<Npoints; p++){ //loop over points
        for (int i=0; i<Ncells; i++){ //loop over cells in x dim
            for (int j=0; j<Ncells; j++){ //loop over cells in y dim
                for (int k=0; k<Ncells; k++){ //loop over cells in z
                    dim
                    // Grid cell center coordinates
                    double cell_x = -half_size + (i + 0.5) * (2.0 *
                        half_size) / Ncells;
                    double cell_y = -half_size + (j + 0.5) * (2.0 *
                        half_size) / Ncells;
                    double cell_z = -half_size + (k + 0.5) * (2.0 *
                        half_size) / Ncells;

                    double dist = sqrt(
                        pow(x[p] - cell_x, 2) +
                        pow(y[p] - cell_y, 2) +
                        pow(z[p] - cell_z, 2)
                    );
                    if (dist < r){
                        // do something with cell(i,j,k)
                    }
                }
            }
        }
    }

    double dT = CPU_TIME - Tstart;
    printf("Time taken: %fs\n", dT);

    free(x);
    free(y);
    free(z);

```

```
    return 0;
}
```

- **Optimized code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define CPU_TIME ({ struct timespec ts; clock_gettime(
    CLOCK_PROCESS_CPUTIME_ID, &ts); (double)ts.tv_sec + (double)ts.
    tv_nsec * 1e-9; })

int main(int argc, char **argv){
    int Npoints = 1000;
    int Ncells = 20;
    double half_size = 1.0;
    double r = 0.1;
    double dx, dy, dz;

    double *x = malloc(Npoints * sizeof(double));
    double *y = malloc(Npoints * sizeof(double));
    double *z = malloc(Npoints * sizeof(double));

    double Tstart = CPU_TIME;

    // Precompute all cell centers
    double *cell_centers = malloc(Ncells * sizeof(double));
    double cell_size = (2.0 * half_size) / Ncells;
    for (int idx = 0; idx < Ncells; idx++){
        cell_centers[idx] = -half_size + (idx + 0.5) * cell_size;
    }

    double r2 = r * r;

    for (int p = 0; p < Npoints; p++){
        for (int i = 0; i < Ncells; i++){
            dx = x[p] - cell_centers[i];
            double dx2 = dx * dx;
            for (int j = 0; j < Ncells; j++){
                dy = y[p] - cell_centers[j];
                double dist2_xy = dx2 + dy * dy;
                for (int k = 0; k < Ncells; k++){
                    dz = z[p] - cell_centers[k];
                    double dist2 = dist2_xy + dz * dz;
                    if (dist2 < r2){
                        // do something with cell(i,j,k)
                    }
                }
            }
        }
    }

    double dT = CPU_TIME - Tstart;
    printf("Time taken: %fs\n", dT);

```

```
50      free(x);
51      free(y);
52      free(z);
53      free(cell_centers);
54      return 0;
55  }
```

3.4 Heap and Stack Memory

What happens when we execute a program on a machine? How does the code interact with the OS and how can it find the way to memory and cpu? It needs at least the memory where the code itself and the data must be uploaded, along with access to other resources, the cpu but also I/O.

Observation: ***nix systems***

Here we only consider *nix systems, meaning any OS that is Unix-like or Unix-based. The asterisk is used as a wildcard to represent the various prefixes or variations of Unix-style systems.

We will also clarify the difference between the **stack** and the **heap**, the two fundamental memory regions we will deal with.

Memory Model

When we execute a program, the OS provides a sort of ”memory sandbox” in which our code will run and that offers a **virtual address space** that appear to be homogeneous to our program (this is to enhance portability and security).

The amount of memory that can be addressed in this virtual box depends on the machine we are running on and on the operating system:

- 32 bit systems: 4GB of addressable memory
- 64 bit systems: 16EB of addressable memory (but usually limited to a few TB by the OS)

In the very moment it is created, not the whole memory is obviously at hand for the program, but it is allocated on demand.

The virtual memory inside the sandbox must ne related to the physical memory where the data is actually stored: when the cpu executes a read or write instruction, it translates the virtual address to a physical address that is then given to the memory controller which deals with the physical memory.

The translation from virtual addresses to physical ones is done with the **paging** mechanism. Basically, the physical memory is seen as split in pages, whose size is specific to each architecture. Each physical page can be translated in more than one virtual page. The mappings is described in hierarchical table that the kernel keeps in memory, containing a **Page Table Entry (PTE)** for each page addressed. Since the translation of virtual to physical addresses is done very often, the CPU keeps a cache of the most recent translations in a special memory called **Translation Lookaside Buffer (TLB)**.

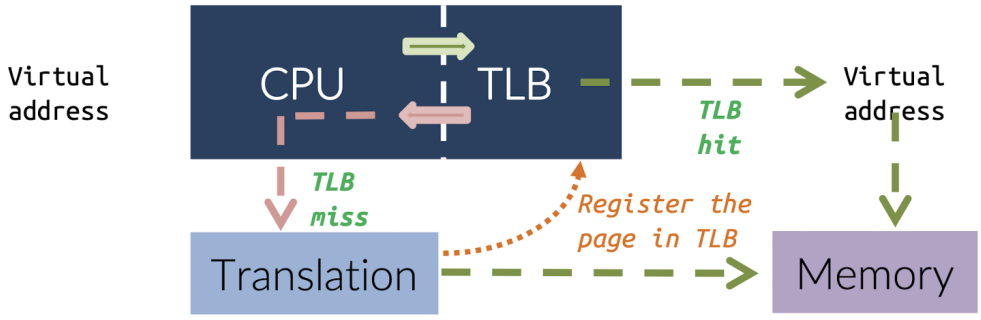


Figure 3.10: TLB

Typically a TLB can have 12 bits of addressing, meaning that it can store 4096 entries. If the TLB is full and a new entry must be added, one of the existing entries is evicted (typically the least recently used one). Moreover, a hit time of 1 cycle is typical, while a miss can cost up to 100 cycles. This is why TLB miss have a high cost in terms of performance, even if it usually stays around $\sim 1-2\%$.

Stack vs Heap

- **Stack:** is a bunch of local memory that is meant to contain local variables of each function. They are all the variables used to serve the workflow of the function, and they are allocated when the function is called and deallocated when the function returns. The stack is managed by the CPU, which keeps track of the stack pointer (SP) that points to the top of the stack and of the base pointer (BP) that points to the bottom of the stack. The stack grows downwards, meaning that when a new variable is pushed onto the stack, the SP is decremented. The stack is usually limited in size (typically a few MB), and if it overflows, a **stack overflow** occurs.
- **Heap:** is meant to host the mare magnum of data that our program needs to work. **Global** variables and data are those that must be accessible from all our functions in all the code units. They are allocated when the program starts and deallocated when the program ends. The heap is managed by the OS, which keeps track of the free and used memory blocks. The heap grows upwards, meaning that when a new variable is allocated on the heap, the heap pointer (HP) is incremented. The heap is usually much larger than the stack (typically a few GB), but it is also more fragmented and slower to access. Its most natural use is then for **dynamic allocation**.

Below in Figure 3.11 you can see a typical memory layout of a process in a *nix system. The **text segment** contains the executable code of the program, the **data segment** contains the global variables and the **bss segment** contains the uninitialized global variables. The **heap** is located above the bss segment and grows upwards, while the **stack** is located below the text segment and grows downwards. The space between the heap and the stack is called **unused memory**, and it is usually not used by the program. Above the stack there is the **kernel space**, which is reserved for the OS and is not accessible by the program.

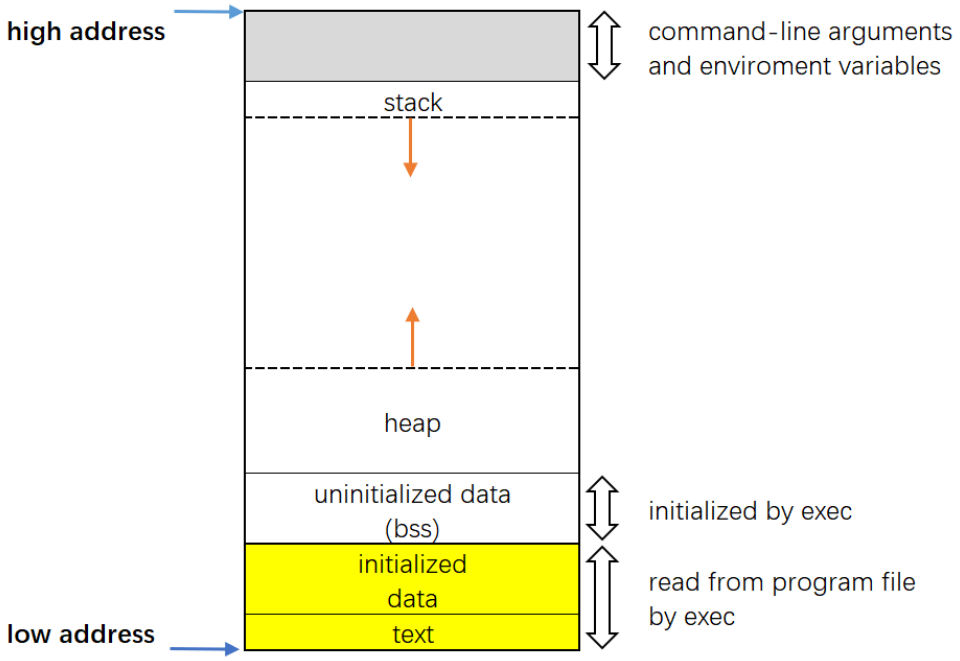


Figure 3.11: Stack vs Heap

We will use the work of Yuriy Georgiev to better understand what happens in the stack when we call a function.

As introduced before, we have three pointers to keep track of the stack:

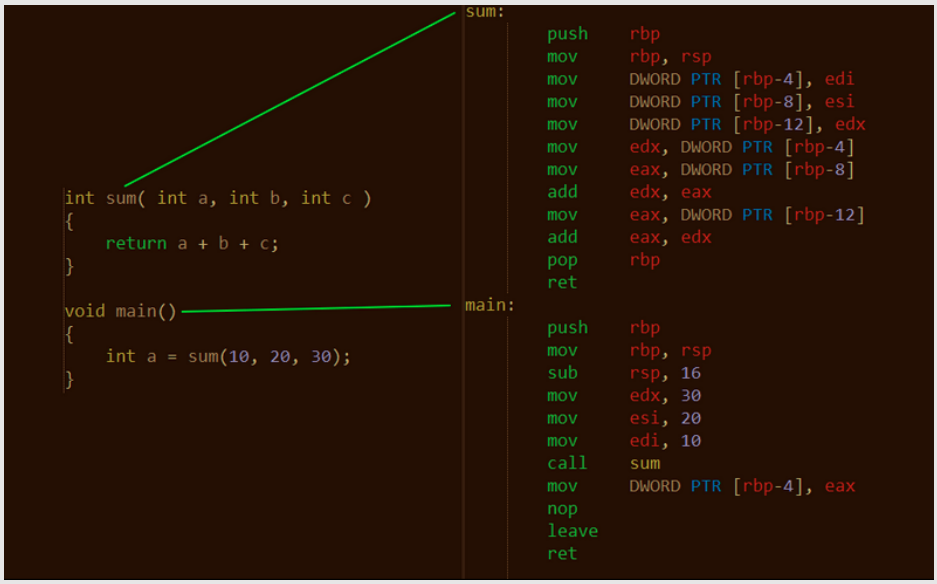
- **Stack Pointer (SP):** points to the top (the end) of the stack.
- **Base Pointer (BP):** points to the bottom (the beginning) of the stack.
- **Instruction Pointer (IP):** points to the next instruction to be executed by the CPU.

The heap is a dynamically allocated memory. You don't have control over where within the memory it will be allocated – the operating system takes care of that. However, the heap grows upwards. Let's say we have a block of heap memory starting at address 1000 with a size of 200 bytes. This block of memory will occupy the addresses between 1000 and 1200 (decimal). The stack, however, grows in the opposite direction. Therefore the stack base pointer (RBP) is set at an address higher than the tip of the heap memory and grows downwards. For example, if the allocated memory ends at address 1200, the stack Base Pointer could be pointing at 1300 and it will grow down to address 1200 (meaning that the stack is 100 bytes big). These are just imaginary numbers and examples to illustrate the way it works. The important thing you need to remember is that the stack "grows" from higher addresses to lower addresses as items are pushed onto it.

When we enter a function, pass arguments to it, declare a local variable or leave a function, the stack is used by the CPU in one way or another.

Example: *Function call and stack usage*

Consider the following sample code:



When the `call` instruction is executed, the value of the RIP (instruction pointer) register is pushed onto the stack which saves the address before entering the `sum()` function, to which the program should return execution after leaving the function. After the `call` instruction, the CPU jumps to the address of the `sum()` function and starts executing its instructions. The 3 arguments are moved to 3 32-bit general purpose registers (RDI, RSI, RDX) as per the x86-64 calling convention.
In the `sum()` function, the first thing that happens is the creation of a stack frame (a portion of the stack that will be available only for the current function), that will be destroyed when the function returns. This is done by pushing the current value of the RBP register onto the stack (to save the previous stack frame) and then moving the value of the RSP register to the RBP register (to set the new stack frame base). To use the arguments saved in the registers, they are copied to the stack frame. They are an offset relative to the RBP.
After doing its job, the function `sum()` prepares to return to its caller, the `main()`. The code `add eax,edx` adds the value of the third argument (in EDX) to the value of the first argument (in EAX) and stores the result in EAX. The `pop rbp` instruction restores the previous stack frame by popping the value of the old RBP from the stack and restoring the previous ”bottom” of the stack frame. The `ret` instruction transfers control to the return address located on the stack (the RIP) pushed to the stack before calling the `sum()` function.

Why the stack is faster than the heap?

- Stack is allocated when the thread starts - this is really the first and foremost thing. You do not have to ask your OS for memory; instead, you get a preallocated chunk. For ARM64, x86, and x64 machines, the default stack size is 1 MB.
- Stack variables do not require allocating on the heap, which is managed by a memory allocator, which is a slow process. Stack variables do not need to be “freed”, which is also a slow process. These are the major reasons why people say the “stack” is faster than the “heap”, even though they all come from the same memory place physically. Also, stack variables are put one next to another. This plays well with the CPU cache.

- Deallocating is a simple matter of moving the Base Pointer and Stack Pointer - given that we are really just moving up and down inside our own 1MB of space, all we need to do in order to allocate and wipe out data is to move a pointer that tells us where's our stack bottom.
- Addresses can be precalculated at compile time, while the heap is allocated dynamically at run time and retrieves its address after the allocation.
- When the CPU requests data from a certain address in the memory it retrieves a whole cache line, which is 64 bytes in x64 systems (128 in Apple's M2), instead of simply the value it will work with. Once the data is retrieved from the RAM it is stored in the CPU cache L1 which is an extremely fast memory. The only memory faster than the L1 Cache is the registers. If the stack variables are stored one after another, requesting the value of one variable leads to retrieving everything after it up to 64 bytes. This leads to caching the following stack variables in the L1 cache making them available for fast access.

Warning:

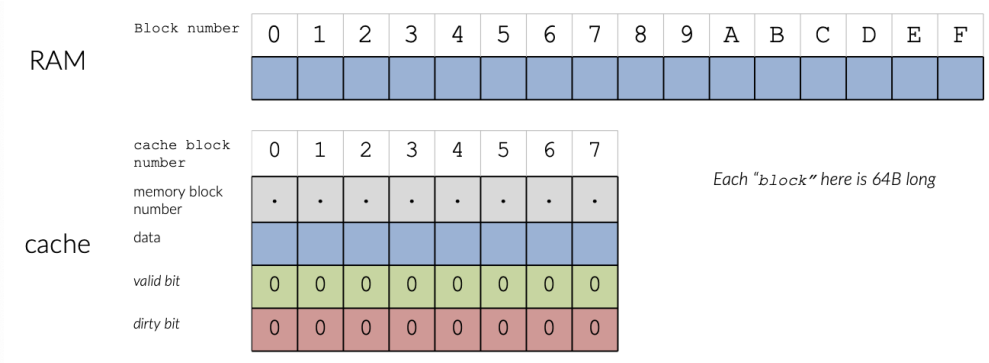
However, not always the stack is faster than the heap. Accessing an array on the heap in the fastest way will possibly generate less instructions than accessing an array on the stack. This is because the compiler can optimize the access to the heap array better than the access to the stack array. So, depending on how things are set-up, the stack can be faster or slower than the heap. The only thing that is certain is that the stack is limited in size, while the heap is not.

3.5 Cache Optimization

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

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

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



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

Table 3.3: Cache Mapping Strategies

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

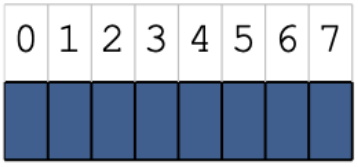


Figure 3.12: Full mapping

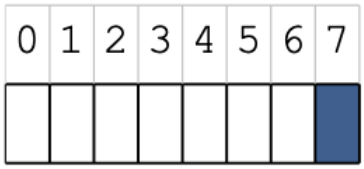


Figure 3.13: Direct mapping

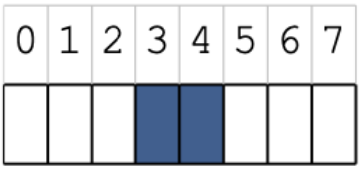


Figure 3.14: n-way associative

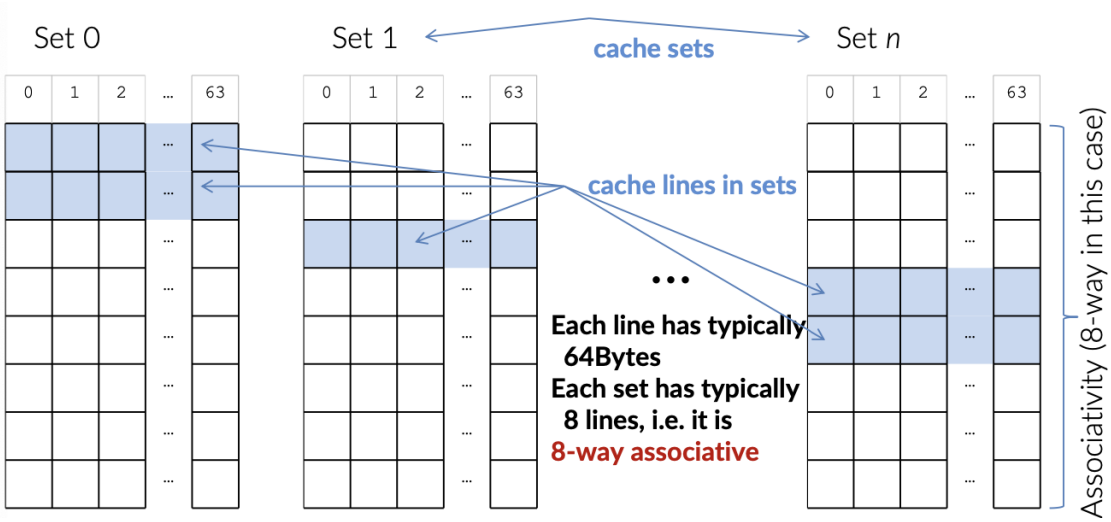


Figure 3.15: A typical today cache

Advanced Concept: *Cache mapping*

How a byte is actually mapped into a cache location?

The elemental unit of the cache is a **line**, composed by 64 bytes. Thus, 64 consecutive bytes in the RAM are mapped in the same line of the cache. Let's suppose that we have 256B of memory. Then 8 bits are sufficient to address our memory since $2^8 = 256$.

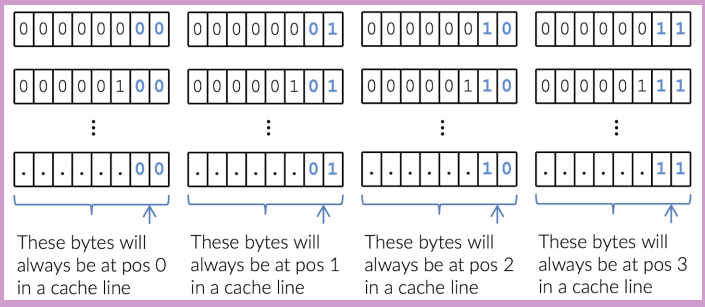


Figure 3.16: Cache mapping

This is possible considering the 6 least significant bits, which will cycle (faster than the others) and determine a cycle over 64 bytes. So, the least significant bits decides the position of a given address in a cache line, by group of 64. The 2 remaining bits (8-6=2) are used to address to which cache line the address belongs to. In this case, we have only 4 lines ($2^2 = 4$).

In general, if we have a cache size of **c** bytes in total and it is **w**-way associative with lines of size **L** bytes, then there are **C/(LW)** = 2^s sets. In our example:

- $L = 64$B
- $c = 256$B
- $w = 2$
- $C/(LW) = 256/(64*2) = 2 = 2^1$

As you can see in Figure 3.17, the least $b+s$ significant bits uniquely determines the position of the byte in the line (b) and to which line it belongs to (s). The rest of the bits (t) are used to uniquely identify the address in the RAM. This is why the space used by the address is 48-52 bits, even if we are in a 64 bit architecture.

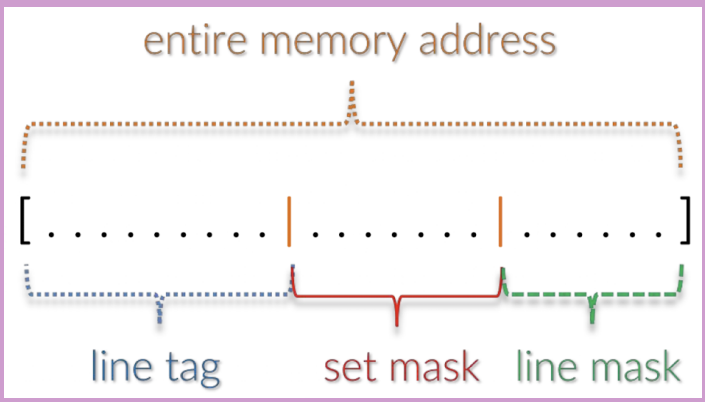


Figure 3.17: Entire memory address

The memory access pattern

Consider a simple direct mapped **16 byte data cache** with **two cache lines**, each of size 8B.

Consider the following code sequence, in which the array `X` is cache-aligned (i.e., `x[0]` is always loaded into the beginning of the first cache line) and accessed twice in consecutive order:

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

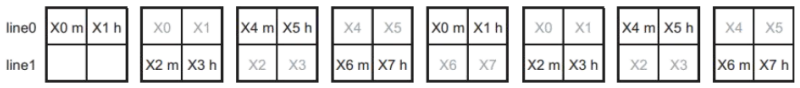


Figure 3.18: The hit-miss pattern is: MH MH MH MH MH MH MH MH, the miss-rate is 50% (the first miss is compulsory miss)

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

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

Figure 3.19: The hit-miss pattern now is: MM MM MM MM MM MM MM MM, the miss-rate is 100%

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

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

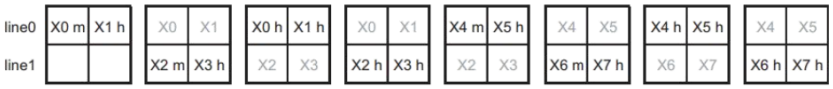


Figure 3.20: The hit-miss pattern now is: MH MH HH HH MH MH HH HH, the miss-rate is 25%

Definition: ***Compulsory cache miss***

A **cache miss** is an event that occurs when the data requested by the CPU is not found in the cache memory. This results in a delay as the data must be fetched from the main memory or a lower level cache. A **compulsory miss** occurs when data is accessed for the very first time and is not yet in the cache. It's called "compulsory" because this miss is unavoidable - no matter how large your cache is or how it's organized, you cannot avoid this miss since the data has never been loaded before. In the examples above, the first access to `X[0]` results in a compulsory miss because it is the first time this data is being accessed and it is not yet in the cache. It is then loaded into the cache for future accesses.

1. **Initial state:** The cache is empty
2. **First access to X[0]:** The cache controller looks for this memory address in the cache
3. **Cache miss:** Since the cache is empty, it results in a cache miss
4. **Load cache line:** The cache controller fetches the data from the main memory and loads the entire cache line containing `X[0]` into the cache
5. **Subsequent accesses:** Future accesses to `X[0]` will result in cache hits, as the data is now in the cache

Thus, **memory access pattern is of primary importance**.

Strided Access

Let's now consider a common problem: the transposition of a matrix.

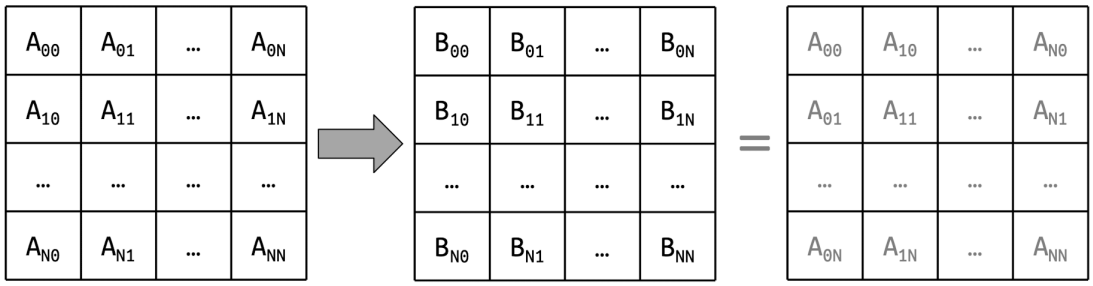


Figure 3.21: Matrix transposition

The naive approach is the following:

```
1 for (int i = 0; i < N; i++) {
2     for (int j = 0; j < N; j++) {
3         B[j][i] = A[i][j];
4     }
5 }
```

In this case, the access to `A[i][j]` is **row-major**, while the access to `B[j][i]` is **column-major**. This means that the access to `B` is strided, and thus it will lead to a high number of cache misses.

Observation:

When we modify the content of a variable, that change amounts to overwrite a memory location. When the variable is loaded in the cache memory, what is loaded is a **copy** of the variable, that maintains the original location in the main memory. So, if we modify its value, shall we modify it in the cache only or in the main memory too?
There are two main strategies to deal with this problem:

- **Write-through:** every time a variable is modified in the cache, it is also modified in the main memory. This strategy is simple to implement, but it is slow, since every write operation requires a write to the main memory.
- **Write-back:** every time a variable is modified in the cache, it is marked as dirty, but it is not immediately written to the main memory. The dirty cache line is written to the main memory only when it is evicted from the cache. This strategy is faster, since it reduces

the number of write operations to the main memory, but it is more complex to implement, since it requires a mechanism to keep track of dirty cache lines.

Going back to our matrix transposition problem, since strided access is unavoidable, is it better to have it on **read** or **write**? Due to write-allocate transactions in the cache, strided writes are more expensice than strided loads. Thus, it is better to have strided access on read. Thus, swapping the loops is a good idea:

```
1 for (int j = 0; j < N; j++) {
2     for (int i = 0; i < N; i++) {
3         B[j][i] = A[i][j];
4     }
5 }
```

And we can compare the two approaches:

Table 3.4: Matrix Transpose Performance Comparison

Matrix Size	Strided BW (MB/s)	Contiguous BW (MB/s)	Ratio
1023	6573.67	5571.80	1.18
4095	2214.94	2067.50	1.07
8191	3143.78	2583.68	1.22
16383	2176.67	1177.84	1.85

Traversing data to enhance locality

A common technique in matrix-matrix multiplication or in matrix inversion is to process the matrices by blocks instead of traversing entire rows or columns. This way, if the block size is choosen wisely, that could enhance the possibility that all the data needed is already in the cache. We should pursue the generation of blocks in a ”natural” way, without hard-coding or fine tuning for a given platform, so that the code is portable and maintainable.

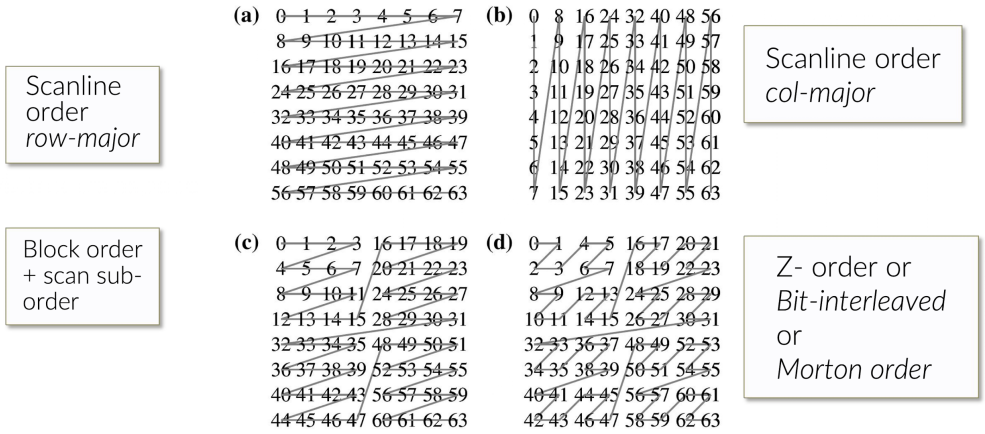


Figure 3.22: Traversing orders

Advanced Concept: *The z-Order*

Let's say we want to map a linear access order $(1,2,3,\ldots,n)$ on some spatially-distributed data with integer coordinates (e.g. a 2D grid). First, rewrite the linear index in binary form $(0000,0001,0010,\ldots)$, then define the coordinates of the data points in binary form too $(00,01,10,11)$. Then, interleave the bits of the coordinates to get the linear index. This is called **z-order** or **Morton order**.

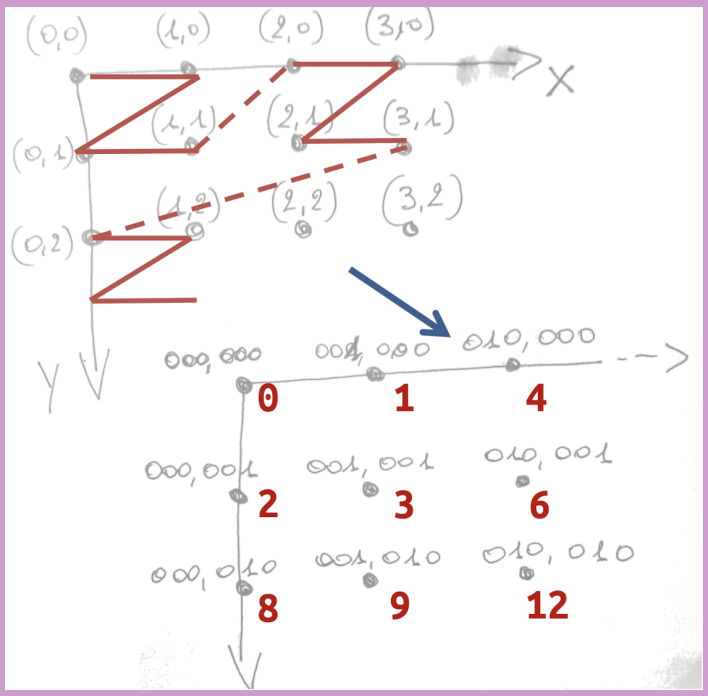


Figure 3.23: Z-order traversal

The result of this interleaving is a mapping from the 2D plane to the 1D line known as Z-line which is one of the **plane-filling curves** discovered by Peano in 1890.

Organizing the data to enhance locality

Whenever the memory bandwidth is limited, **data locality optimization** can play a strong role. Re-organizing data in "space" so that the access pattern is optimal for a given algorithm is related to such locality optimization.

1. **Hot and Cold fields:** they refer to the frequency of access of different fields within a data structure. Hot fields are those that are accessed frequently, while cold fields are those that are accessed infrequently. By organizing data structures so that hot fields are stored together and cold fields are stored separately, we can improve cache performance and reduce memory access times.

   As an example, consider the following data structure:

```
1 struct node {
2   double jey;
3   char data[300];
4   node *next;
5 }
```

If we traverse a linked list of such nodes, we will access the `key` and the `next` fields frequently, while the `data` field will be accessed infrequently. Thus, we can reorganize the structure as follows:

```
1 struct node {
2   double key;
3   node *next;
4   char data[300];
5 }
```

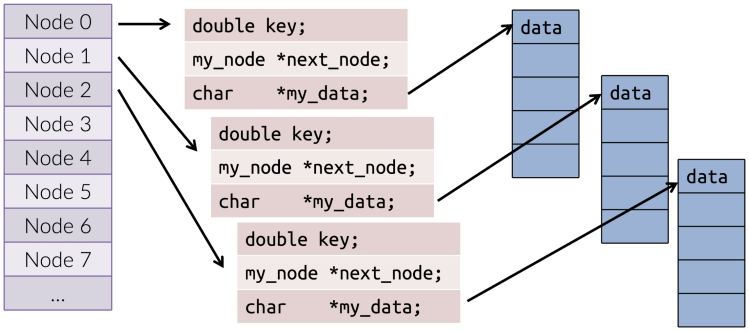


Figure 3.24: Hot and Cold fields

2. **Space filling curves:** they are a way to map multi-dimensional data to one-dimensional data while preserving locality. By organizing data in a space-filling curve, we can improve cache performance and reduce memory access times.

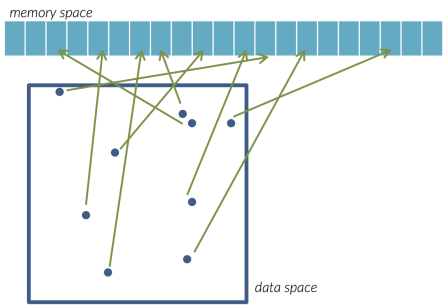


Figure 3.25: Data space

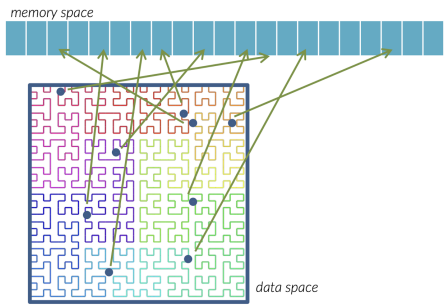


Figure 3.26: Sort data according to the index

3.6 Branches Optimization

Definition: ***Conditional execution***

Whenever either **(i)** the sequence of operations that must be executed or **(ii)** the sequence of data to be processed depends on some condition, i.e. on the outcome of a test performed on some data or result, we have a **conditional execution**.

On modern architectures, conditional executions are implemented in two low-level ways:

1. **Modifying the control flow:** this uses **branch instructions** to change the sequence of instructions executed by the CPU. Branch instructions can be either conditional or unconditional. Conditional branch instructions change the control flow based on the outcome

of a test, while unconditional branch instructions always change the control flow to a specific address. At machine level, a **jump instruction** is used to implement a branch. `jmp` is an unconditional jump, while `je`, `jne`, `jl`, `jle`, `jg`, `jge` are conditional jumps that depend on the status of the flags in the FLAGS register.

Observation: *True branch*

Note that the **true branch** is the closest to the test condition, while the **false branch** is reached upon a jump. When coding, if possible pay attention to what is most likely to be true, to preserve **code locality**.

However, control flow modification is quite inefficient on modern CPUs.

2. **Modifying the data flow:** can only be done on cases with simple values involved, but yields better performance. This is done using **conditional move instructions** that move data from one location to another based on the outcome of a test. The most common conditional move instruction is `cmov`, which can be used to move data from one register to another based on the status of the flags in the FLAGS register.
   - two registers contain the two possible values
   - the conditional move checks the result of cmp
   - the content of one of the two registers is moved to the destination register

We saw that modern processors achieve great performance thanks to the **pipelines** and **out-of-order execution**. However, that requires the pipelines to be always full, otherwise penalties in terms of wasted cpu cycles is to be paid. To achieve this, the scheduler has to **predict** in advance what will be the sequence of instructions to be executed. How can this be done? Modern cpus use a **branch predictor**, which is an internal unit of highly sophisticated logic that guesses whether a jump instruction will succeed or not. This is why the conditional change of data flow is preferred whenever possible and the compiler will try to use it as much as possible.

Tip:

Conditional branches should be **avoided** as much as possible inside loops.

Basically, there are two possible cases:

- Variables tested in the conditions **do not** change during the loop
- Variables tested in the conditions **change** during the loop

We will now see 4 examples of how to revise our code to optimize branches:

1. **Clean loops from branches**

```
for (int i=0; i<N, i++){
  if (case1==0){
    // do something
  } else {
    // do something else
  }
}
```

can be rewritten as:

```
if (case1==0){
  for (int i=0; i<N, i++){
    // do something
  }
} else {
  for (int i=0; i<N, i++){
    // do something else
  }
}
```

Normally, the compiler should be able to do this automatically, but it is not always the case. If you do it manually, remember to define a specialized function for each case and set a function pointer before and outside the loop to point to the right function.

In cases where the conditions are more complex, then having multiple for loops may be highly unpractical. In this cases it is better to use `switch` constructor instead.

```
switch (case){
  case 0:
    for (int i=0; i<N, i++){
      // do something
    }
    break;
  case 1:
    for (int i=0; i<N, i++){
      // do something else
    }
    break;
  ...
}
```

In fact, the switch constructor is translated in a static table of code pointers that can be addressed directly.

2. **Unpredictable datastreams**

```
// generate random numbers
for (cc=0; cc<SIZE; cc++){
  data[cc] = rand() % TOP;
}

// take action depending on their value
for (ii=0; ii<SIZE; ii++){
  if (data[ii] < PIVOT){
    sum += data[ii];
  }
}
```

can be rewritten as:

```
// generate random numbers
for (cc=0; cc<SIZE; cc++){
  data[cc] = rand() % TOP;
}

```

```
qsort(data, SIZE, sizeof(int), compare);

// take action depending on their value
for (ii=0; ii<SIZE; ii++){
  if (data[ii] < PIVOT){
    sum += data[ii];
  }
}
```

of course, an overhead is added, however we should focus here on how in general it is better to avoid conditionals inside loops.

We can do even better, by rewriting the last loop as:

```
for (ii=0; ii<SIZE; ii++){
  t = (data[ii] - PIVOT - 1) >> 31;
  sum += ~t & data[ii];
}
```

3. **Sorting two arrays**

   Let's say we have two arrays A and B, and we want to swap their elements so that `A[i] ≥ B[i]` for all i. A simple approach should be:

```
for (int i=0; i<N; i++){
  if (A[i] < B[i]){
    t = B[i];
    B[i] = A[i];
    A[i] = t;
  }
}
```

   However, that implementation suffers exactly of the same problem we have just discussed. An alternative implementation is:

```
for (int i=0; i<N; i++){
  int min = A[i] > B[i] ? B[i] : A[i];
  int max = A[i] > B[i] ? A[i] : B[i];
  A[i] = max;
  B[i] = min;
}
```

Observation:

Note that there are other smarter implementations, but we won't discuss them here. The standard implementation relies on the ability of your CPU's branch predictor to guess the correct data pattern. When it is successful, it is really so and exhibits the lowest CPE (cycles per element) and IPE (instructions per element).
However, whenever the data pattern is unpredictable things quickly become really weird. Writing the code differently may make you loosing something in terms of CPE/IPE but not really in terms of time-to-solution. And, above all, the code behaviour is stable with both predictable and unpredictable patterns.

4. **Are design and simplicity the best move?**

Just changing the point of view sometimes can help. The following code initializes a NxM matrix so that the elements in top-right triangle are set to 1.0, the entries in the diagonal $i = j$ are set to 0.0 and the bottom-left part is set to 1.0.

```
for (int j=0; j<N; j++){
  for (int i=0; i<M; i++){
    if (i < j){
      matrix[i][j] = 1.0;
    } else if (i == j){
      matrix[i][j] = 0.0;
    } else {
      matrix[i][j] = 1.0;
    }
  }
}
```

can be easily rewritten without conditional evaluations at all:

```
for (int j=0; j<N; j++){
  for (int i=0; i<M; i++){
    int i;
    for (int i=0; i<j; i++){
      matrix[i][j] = 1.0;
    }
    matrix[j][j] = 0.0;
    for (int i=j+1; i<M; i++){
      matrix[i][j] = 1.0;
    }
  }
```

3.7 Loops Optimization and Prefetching

Definition: *Arithmetic Intensity*

Arithmetic Intensity is the ration between the number of floating point operations and the number of memory operations.

$$A_I = \frac{f(n)}{n}$$

There are three main types of loops:

- **Linear loops** $O(N)/O(N)$: they have low arithmetic intensity, since they perform a number of floating point operations proportional to the number of memory operations. An example is the vector addition. Here, optimization come from avoiding **unnecessary operations** and/or **repeated memory accesses**, and increasing **data reuse**.
- **2-level loops** $O(N^2)/O(N^2)$: they have low arithmetic intensity, since they perform a number of floating point operations proportional to the number of memory operations. An example is the matrix addition. Here, optimization comes again from increasing **data reuse**, exploiting **locality** and **avoiding unnecessary loads / stores**.
- **3-level loops** $O(N^3)/O(N^2)$: they have high arithmetic intensity, since they perform a number of floating point operations proportional to the square of the number of memory operations. An example is the matrix-matrix multiplication. Here, more optimization can be exploited, and is at

the core of the **Linpack** library.

Let's now dive deeper into each of these types of loops.

1-level loops

Examples for this type of loops are scalar products, vector sums, sparse matrix vector multiplications. There is an inevitable memory bound for very large N. Consider the following code on **loop fusion**:

```
for (int i=0; i<N; i++){
  A[i] = B[i] x C[i];
}
for (int i=0; i<N; i++){
  Q[i] = B[i] + D[i];
}
```

can be rewritten as:

```
for (int i=0; i<N; i++){
  A[i] = B[i] x C[i];
  Q[i] = B[i] + D[i];
}
```

In the optimized version, the array `B` is loaded only once instead of twice.

2-level loop

Examples for this type of loops are dense matrix-vector multiplications, matrix transposition, matrix addition. There are three main strategies to optimize these loops:

- **Avoid unecessary loads / stores**

```
for (int i=0; i<N; i++){
  for (int j=0; j<N; j++){
    C[j] += A[i][j] x B[i]; // 3xNxN memory accesses
  }
}
```

  can be rewritten as:

```
for (int j=0; j<N; j++){
  c_temp = C[j]; // C[j] is loaded and stored only once
  for (int i=0; i<N; i++){
    c_temp += A[i][j] x B[i]; // 2x(NxN + N) memory accesses
  }
  C[j] = c_temp;
}
```

- **Loop unrolling**: Unroll outer loop and fuse in the inner loop; there is potential for vectorisation

```
  for (int i=0; i<N; i++){
    for (int j=0; j<N; j++){
      C[i] += A[i][j] x B[j];
    }
```

```
5    }
```

can be rewritten as:

```
1  for (int i=0; i<N; i+=m){
2    for (int j=0; j<N; j++){
3      b_temp = B[j];
4      C[i] += A[i][j] x b_temp;
5      C[i+1] += A[i+1][j] x b_temp;
6      ...
7      C[i+m-1] += A[i+m-1][j] x b_temp; // NxNx(1+1/m)+N memory accesses
8    }
9  }
```

Warning: *Unrolling and register spill*

Using a too large **m** in the previous example while the target CPU does not have enough registers to keep all the needed operands results in a **code bloating** and in a **register spill**, which is a situation where the CPU has to temporarily store some register values in memory (usually in the stack) because there are not enough registers available to hold all the necessary data. This can lead to a significant performance degradation, as accessing data from memory is much slower than accessing data from registers. Thus:

- Learn to inspect the compiler's log
- Hand code effort to clarify the code
- Hints/directives to the compiler

Sometimes no magic wand can cure the fact that you have to access N^2 memory locations. Then:

- **Unroll and Jam** strategy can bring benefits as long as the cache can hold N lines. An L_c-way unrolling is too much aggressive and may easily result in register pressure.
- **Loop tailing (or blocking)** strategy does not save memory loads but increase dramatically the cache hit ratio.

- **Locality of referenced data:** cut TLB misses by accessing 2D arrays by blocks.

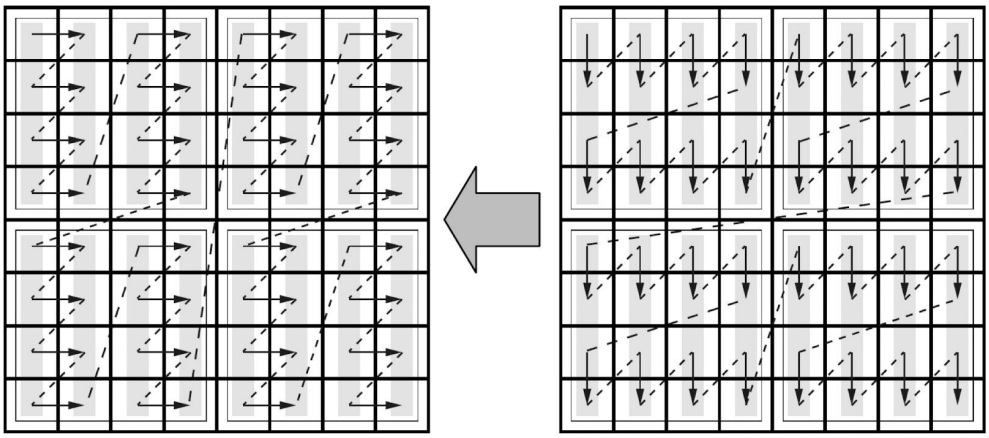


Figure 3.27: Loop blocking

Loop unrolling and vectorization

Loop unrolling is a fundamental code transformation which usually helps significantly in improving code performance:

- Reduces the loop overhead (counter update, branching)
- Exposes critical data path and dependencies
- Helps in exploiting ILP, especially in case of memory aliasing

Consider the following code:

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

Using optimization level `-O3`, the compiler is able to unroll the loop, managing much better the loop overhead. However, it does not optimize the Floating Point operations, since they are not associative. Using `int`, in fact, the compiler opts for the complete **vectorization** of the loop, with a significant performance boost.

Our aim, then, is to to reorganize the code so that the compiler could exploit the CPU's ILP.

First, we can unroll the loop manually:

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

Observation:

Note that when unrolling, we always have to care about the final iterations that would be left behind. A common way to do this for an unrolling factor U (usually in the range $[2 \dots 16]$) is:

```
1 int N_ = (N/U)*U // largest multiple of U <= N
2 for (int i = 0; i < N_; i+=U)
3   // unrolled loop body
4 for (int i = N_; i < N; i++)
5   // clean-up loop body
```

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]);
```

Observation: ***Why this is better?***

The reason why this is better is that now the two additions of `A[i]` and `A[i+1]` are independent, thus they can be executed in parallel. The compiler is now able to vectorize the code, using SIMD instructions.

and finally we can separate partial results in multiple accumulators:

```
for (int i=0; i<N-2; i+=2){
S0 = S0 + A[i];
S1 = S1 + A[i+1];
}
S = S0 + S1;
```

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

Observation:

The unrolling is expressed in general as $n \times m$ where n refers to the number of iterations that are unrolled and m refers to the number of accumulators that are being used.

3-level loops

These algorithms like matrix-matrix multiplication or dense matrix diagonalization are very good candidates for optimizations that lead flop/s performance close to the theoretical peak. Techniques like Tailing, unrolling and jamming, vectorization and reorganization of operations to exploit CPU's pipelines and out-of-order capability are all used by extremely specialized libraries like **BLAS** and **LAPACK**.

Tip:

Whenever you have to deal with these kind of problems, use these libraries instead of reinventing the wheel.

Let's now dive deeper into the **matrix-matrix multiplication** task, which is very common in HPC.

Definition: ***Matrix-matrix multiplication***

Given two matrices A and B having respectively (m,n) and (n,p) rows and columns, the matrix-matrix multiplication is defined as:

$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} \times B_{k,j}$$

where $C_{i,j}$ is the resulting matrix.

The naive implementation is the following:

```
for (int i=0; i<m; i++){ // traverse A's and C's rows
  for (int k=0; k<p; k++){ // traverse B's rows (A's columns = B's rows)
    for (int j=0; j<n; j++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

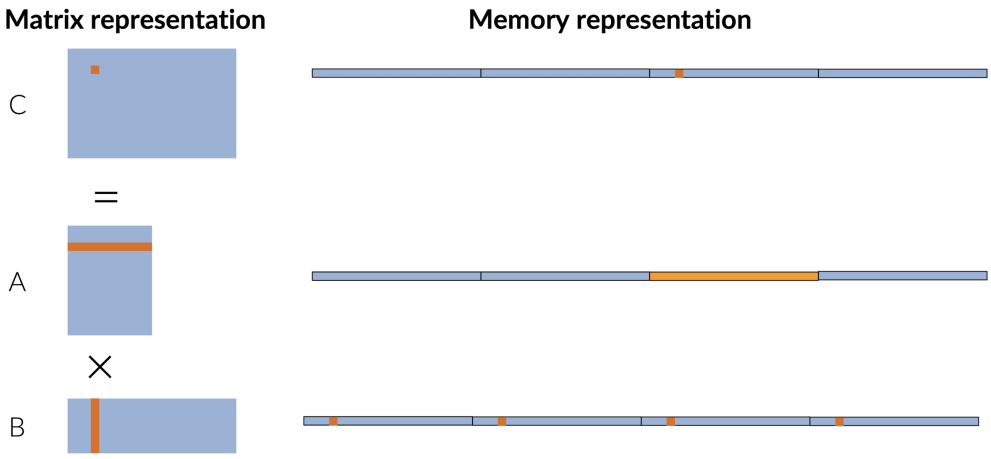


Figure 3.28: Matrix-matrix multiplication

This implementation is inefficient, since it has an issue with data locality for large enough matrices. For each C's element, possibly all accesses to B results in a cache miss. Then, the total number of expected misses is:

$$\text{misses} = \underbrace{\frac{mp}{L}}_{\text{C is traversed once}} + \underbrace{\frac{mnp}{L}}_{\text{A is scanned p times}} + \underbrace{mnp}_{\text{B is accessed sparsely p times}}$$

where m, n and p are the dimensions of the matrices and L is the cache line size.

How to fix this?

Transposing the matrix B before entering the loop is a good idea, although the transposition requires some additional work. The new code is:

```
1 for (int i=0; i<m; i++){ // traverse A's and C's rows
2   for (int k=0; k<p; k++){ // traverse B's columns (A's columns = B's
        rows)
3     for (int j=0; j<n; j++){
4       C[i][j] += A[i][k] * B[j][k];
5     }
6   }
7 }
```

Now we expect to have:

$$\text{misses} = \underbrace{\frac{mnp}{L}}_{\text{running over C}} + \underbrace{\frac{mnp}{L}}_{\text{running over A}} + \underbrace{\frac{mnp}{L}}_{\text{running over B}}$$

which is a significant improvement ($\approx mnp$ less cache misses).

We can do even better using **loop tailing** (or blocking). The idea is to cut the matrices in smaller blocks that fit in the cache, and then perform the multiplication on these blocks.

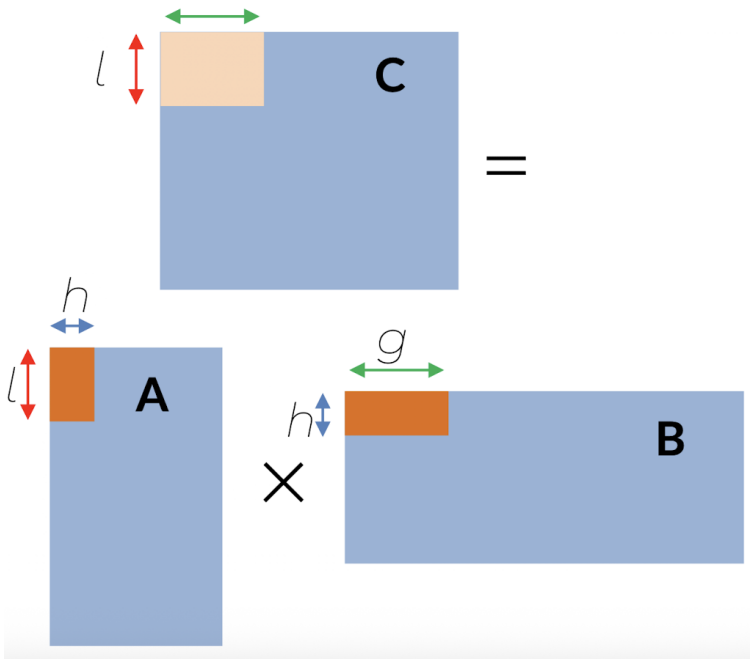


Figure 3.29: Matrix-matrix multiplication with tailing

The idea is to keep in the cache a segment of the A's line, re-using it against the columns of B (or better against a columns section tall as the line segment of A). This will reduce the number of cache misses by a factor $\frac{L}{l \times h \times g}$, where l is the size of the data type, h is the height of the block and g is the width of the block. The optimal values for h and g depend on the cache size and on the cache line size.

The code is the following:

```
1  for (int i=0; i<m; i+=h){ // traverse A's and C's rows
2    for (int k=0; k<p; k+=g){ // traverse B's columns (A's columns = B's
       rows)
3      for (int j=0; j<n; j+=l){
4        // multiply the blocks A[i:i+h][k:k+g] and B[j:j+l][k:k+g]
5        for (int ii=i; ii<i+h; ii++){
6          for (int kk=k; kk<k+g; kk++){
7            for (int jj=j; jj<j+l; jj++){
8              C[ii][jj] += A[ii][kk] * B[jj][kk];
9            }
10         }
11       }
12     }
13   }
14 }
```

Advanced Concept: *Prefetching*

Waiting for data and instructions is a major performance killer. Thus, modern CPUs have the capability of pre-emptively bring from memory into cache levels data that **will be needed shortly afterwards**. They can do this by following some speculative algorithms based on the current execution flow and assuming spatial locality and temporal locality.

Both data and instructions can be pre-fetched.

(It can be both hardware-based and software-based.) There are two possible ways to deal with prefetching:

- **Explicit:** The programmer explicitly inserts a pre-fetching directive. This is difficult since the directive must be inserted timely but not too early (data eviction) or too late (load latency).
- **Induced:** The programmer consciously organizes the code and execution flow so that the compiler can insert pre-fetching directives automatically. This is easier and more portable.

4
Parallel Computing

Why do we need parallel computing? There are two main reasons:

- To solve problems faster and/or bigger problems in the same time
- To solve problems that could not fit in the memory addressable by a single computational unit

In general, 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 (tiny fraction in the real world). If a single processing unit was applied to the problem's solution, performing the task T for each data point subsequently, we would have a **serial** solution that would take $N \times \delta t$ run-time, where N is the size of our data set and δt is the same needed for a single data-point. Applying more than one processing unit, each one would solve the problem for a data point, and the overall run-time would be $\frac{N}{p} \times \delta t$, where p is the number of processing units. The speedup would be $S = \frac{N \times \delta t}{\frac{N}{p} \times \delta t} = p$. The efficiency would be $E = \frac{S}{p} = 1$. This is the ideal case.

   Observation: ***Concurrency vs Parallelism***

   People often confuse **concurrency** with **parallelism**. The first one is achieved with expensive many-cores CPUs and large amount of RAM, launching many separate instances of code as data points. The second, instead, is achieved when the code execution runs on more than one processing unit, tackling the entire data set and managing the processing units to cooperatively solve the problem.

2. **All the rest**

Let's now take into consideration the fact that the run-time for each data point is **not constant**. Then, how we spread the data points among the processing units may strongly affect the overall run-time, since **the overall run-time is determined by the slowest processing unit**.

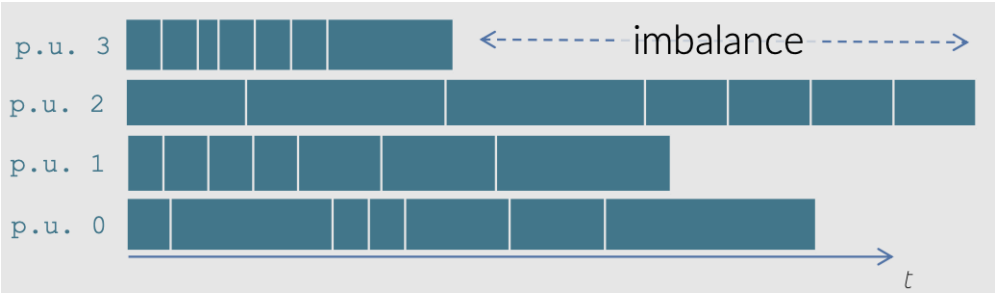


Figure 4.1: Load balancing

From that it descends that ideally the slowest must progress at the same time than the fastest, i.e., we do not have any **imbalance**.

Tip: ***Work-imbalance***

The **work-imbalance** is the measure of how different the computational load is among our players (it is 0 when the work is perfectly spread, almost never the case). It is a fundamental metrics to characterize the performance of our code, or the different sections of it, so one should always instrument his code to track it.

Domain decomposition

How data is distributed among the processing units may be sub-optimal. How we decide which data point are processed by each processing unit is called **domain decomposition** and depends on the nature of the data and their availability. However, there is not a general optimal way to perform it: it strongly depends on the nature of both the computational problem and of the data.

- When the work-load is presumably uniform among the data points, just go for the simplest decomposition possible to minimize the overhead due to the decomposition itself. **Trivial random decomposition** may serve quite well in these cases.
- When the work-load is dependent on the properties of the data and those properties change in time during the computation, there is not a general rule. One may, as an example, sort the data by computational intendity and distribute them to different players so to achieve an even work-load with a minimum imbalance; then, repeating the procedure as often as needed while the data evolves during a multi-step computation.

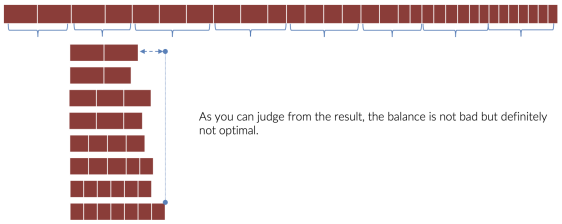


Figure 4.2: Sorting data by computational load.

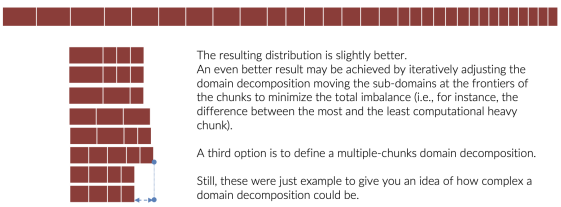


Figure 4.3: Round-robin distribution of data.

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.

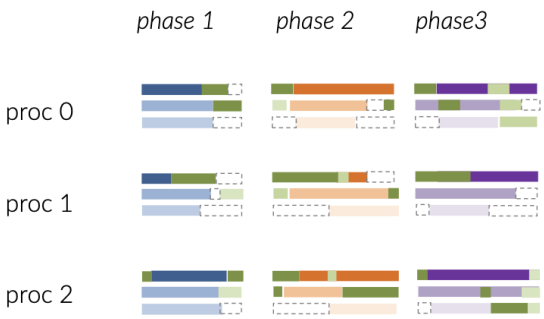
Functional decomposition

Often, our 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.

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.

Almost always, an optimal choice is a mix of both domain and functional decomposition. At least to fit the data in the memory, data are distributed among tasks so that to find the best trade-off among the desired work-load and the constraints of memory-load. Then, some phases require a distributed parallelism (the computation on the data requires a communication that involves all data). Once all the communication among players has been done, every player performs the computation on its data, possibly by distributing tasks among sub-players. Think to sub-players as the threads and to players as MPI tasks, which leads to the concept of **hybrid parallelism**.



Let's suppose the code has 3 algorithms (colors) and that each player has 3 sub-players (color intensity) that run different tasks on the data. Dark green indicates inter-player communication and light green indicates inter-task data exchange or synchronization. White dotted segments indicates idle spinning of a task.

Shared vs Distributed Paradigms

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 travels 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.

The two programming paradigms can easily be fused together in a hybrid approach, where each node runs an MPI process and each process spawns multiple threads to exploit the cores of the node.

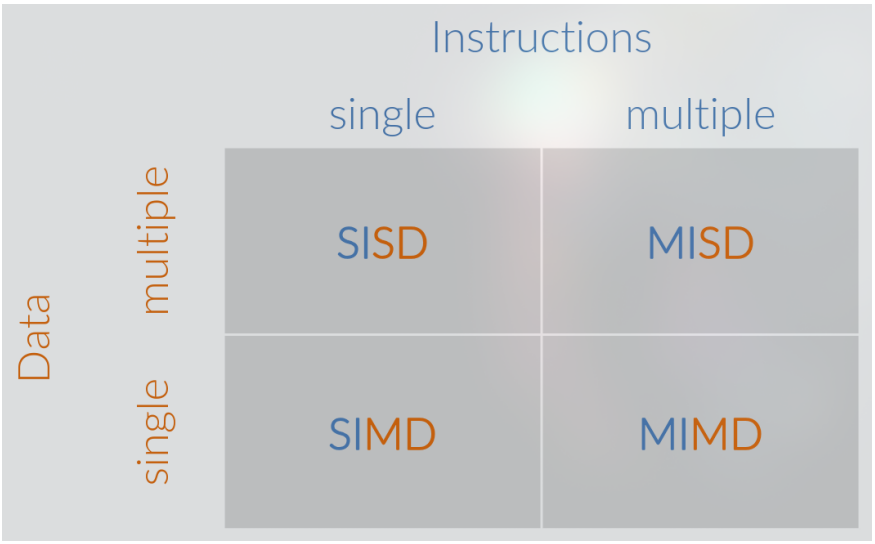
Tip: ***MPI or OpenMP?***

Opt at the beginning for a distributed memory paradigm (MPI) since it is more general and practical. It has the advantage that it can be used on a single node as well as on multiple nodes, while OpenMP is limited to a single node. Moreover, MPI can be combined with OpenMP if needed. Of course, it will cost some effort. Later also OpenMP can be exploited to further speed up the code.
For the cases in which you need a quick impact on your code's performance for runs that fit on single-node, then OpenMP is a good choice. Multi-threading for many problems offers an easier parallel point of view and if properly implemented it may deliver very good performance boosts.

	OpenMP	MPI
Standard specifications	openmp.org	mpi-forum.org
Implementations	the C/C++/Fortran compilers implement themselves the standard up to some version. You switch on the OpenMP with a command-line option (e.g. `-fopenmp` for *gcc*)	There are many implementations of the MPI, basically are a compiler wrapper and a library that interacts with lower-level library that manages the network.
Usage	Compile the code with `cc -fopenmp -o executable code.c`	Compile the code with `mpicc -o executable code.c` and run it with `mpirun -n <num_processes> ./executable`

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 4.4: 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

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 and multiple tasks or threads operating on the same data
SIMD	Any vector-capable hardware, the vector register on a core, a GPU, a vector processor, an FPGA, ...	data parallelism through vector instructions and operations
MIMD	Every multi-core/processor system; on a superscalar CPUs, different ports executing different ops on different data	ILP on different data and multiple tasks or threads executing different code on different data

HPC components

We can now refine our HPC definition:

Definition: *HPC*

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.

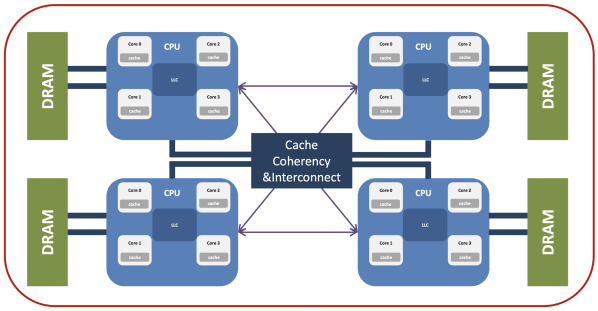
The essential elements of a supercomputer are the following:

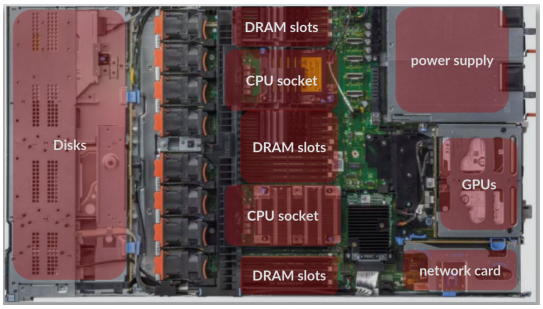
- a large number of **computational nodes**;
- one or more **switch-based networks** that interconnect the nodes together;
- a **storage**: usually there are 3 storage areas that are home, a huge long-term resident storage and a fast parallel FS for production;
- some **login nodes** to access the system;
- some **parallel schedulers** to manage the jobs.

The node topology

In old times all the cores of the 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”

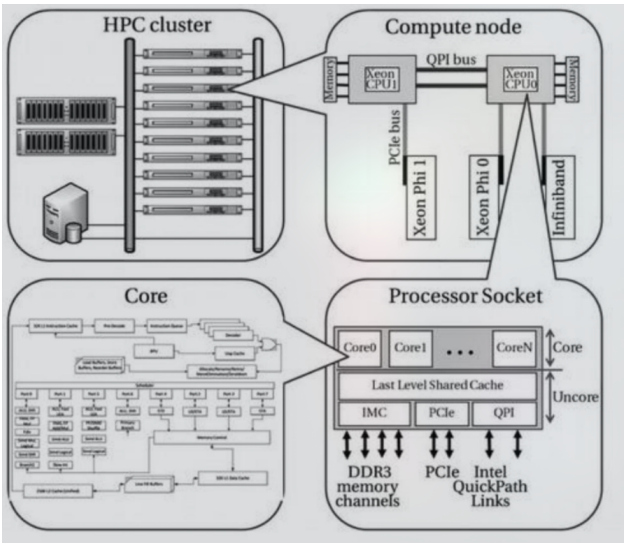




The overall topology

Many ($\approx 10 - 10^5$) 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 4.5: 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**. The processes running on those cpu share a unique physical address space that is mapped on memory physically distributed on the memory banks. Moreover, read and write are simple memory accesses and the process communicate using the 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**. The physical address space is separated and private to the processes that run on a given cpu. The processes running on different cpu communicate through messages that travel on a linking network.

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

There are two fundamental types of shared-memory:

1. if the access to every RAM location is equally costly for all CPUs, the system is **Uniform Memory Access (UMA)**;

2. if the cost of RAM access depends on the location, then the system is said to be **Non-Uniform Memory Access (NUMA)**.

In **UMA** systems, all the CPUs are connected to the DRAM banks through a common connection and for each CPU the cost fr accessing any DRAM memory location is independent on the memory location itself.

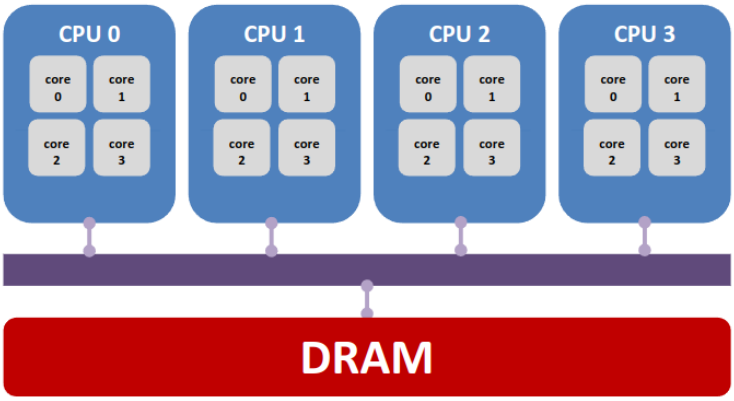


Figure 4.6: UMA architecture

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.

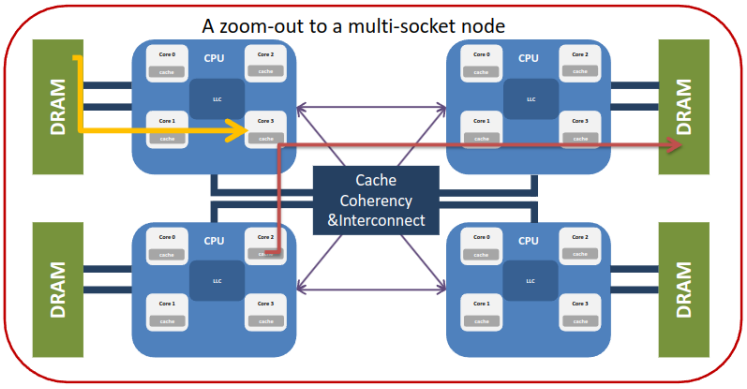


Figure 4.7: NUMA architecture

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.

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.

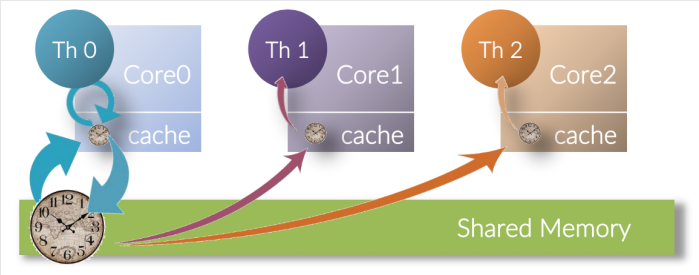


Figure 4.8: MESI protocol example

		Time (in secs)	Action	cache status		
				Core0	Core1	Core2
M	Modified					
E	Exclusive	0	-	I	I	I
S	Shared	1	Th0 reads	E	I	I
I	Invalid	2	Th0 writes[0]	E	I	I
		2.3	Th1 reads[1]	S	S	I
		2.7	Th2 reads	S	S	S
		3	Th0 writes[2]	M	I	I
		4	Th0 writes	M	I	I
		4.4	Th2 reads[1]	S	I	S
		5	Th0 writes	M	I	I
		...	...	...	...	...

- [0] Core 0 is the only one using the value, that is then **E**xclusive;
- [1] A signal is issued to "the memory", which recognizes that the only valid copy is in the Core 0 cache. That value is copied back into the shared memory, and from there it is copied in the cache. Everybody now has a valid copy, then **S**hared;
- [2] A signal about the change is issued to all the interested actors because their values are now **I**nvalid. 0's copy is instead **M**odified.

Parallel Performance

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

If f is the fraction of the code which is intrinsically sequential, the speedup is

$$S_p(n,t) \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

i.e. the serial fraction is severely limiting the achievable speedup.

Is that a problem? Yes, if f is not small enough. Actually, tens of years ago there was a huge debate about whether or not a huge effort towards developing massive parallelism was in order or not.

Warning:

There are some significant issues in the Amdhal's law:

- No matter of how many processes p are used, the maximum achievable speedup is determined by f (and is quite low for ordinary problems);
- The problem size n is kept fixed when estimating the possible speedup while the number of processes increases (strong scaling);
- The parallel overhead $k(n,t)$ is not considered, which leads to an optimistic estimate of the speedup, and usually $\frac{p(n)}{t} > k(n,t)$;
- The fraction of sequential part may decrease when the problem size increases.

Gustafson-Barsis' Law

Normally, when you increase the problem's size, the parallelizable part increases way more than the sequential part. If we consider the workload as the sum

$$w = a + b$$

where a and b being the serial and the parallel work, and we assign the same amount of workload to every process, that would amount to a serial run-time

$$T_s \propto a + p \times b$$

while it still takes

$$T_p \propto a + b$$

using p processes. Hence the speedup is $\frac{[a+p\times b]}{[a+b]}$, which if $f_n = \frac{a}{a+b}$ we can rewrite as the Gustafson's law for the speedup:

$$S_{p_G}(n,p) = p - (p-1)f_n \leq p$$

Observation: *Parallel overhead*

Both Amdahl's and Gustafson's laws lead us to two different concepts of **scalability**, which is the ability of a parallel system to increase its efficiency when the number of processes and/or size of the problem get larger.

- **Strong scalability** is the ability to reduce the time-to-solution for a fixed-size problem by increasing the number of processes. Amdahl's law applies to this case.
- **Weak scalability** is the ability to solve larger problems in the same time by increasing both the problem size and the number of processes proportionally. Gustafson's law applies to this case.

In parallel computing, there may be several sources of overhead due to the parallelization itself. Hence, if $k(n,t)$ is the overhead of some kind, t_s and t_p the run-time for the serial and the parallel part, the parallel run-time can be written as

$$T_p(n,p) = t_s + \frac{t_p}{p} + k(n,p)$$

A simple way to measure the parallel overhead is to consider the distance between the code's scaling and the perfect scaling.

$$T_s(n) - T_p(n,p) = t_s + t_p - t_s - \frac{t_p}{p} - k(n,p)$$

which becomes for large $p \approx t_p - k(n,p)$.

5 OpenMP

5.1 Introduction

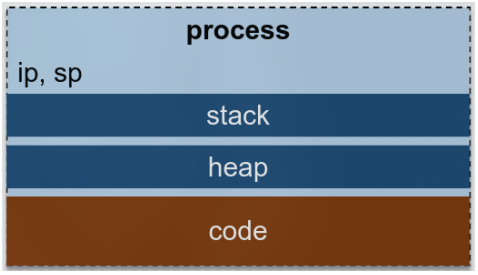
OpenMP is a standard API for shared-memory parallel programming. The name stands for **Open specifications for MultiProcessing**. It enables the development of multi-threaded programs with consistent, portable behavior by using a set of compiler directives embedded in the source code:

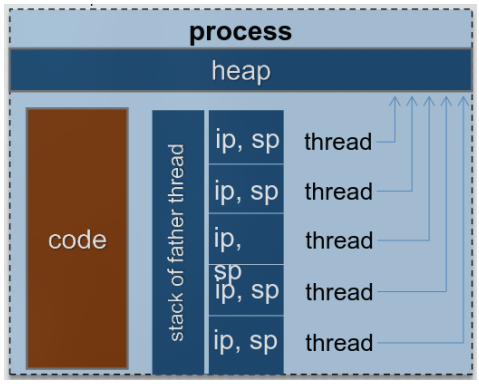
- **Pragmas** (`#pragma`) in C/C++
- **Specially formatted comments** in Fortran

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

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.

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).

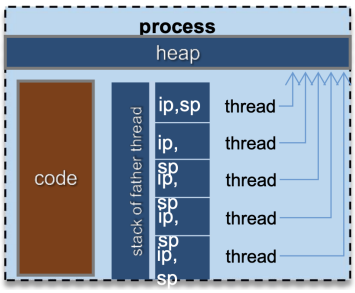


Figure 5.1: Shared-Memory Architecture

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.

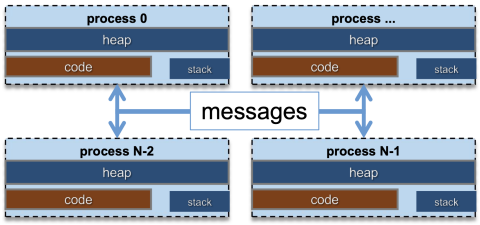


Figure 5.2: Distributed-Memory Architecture

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.

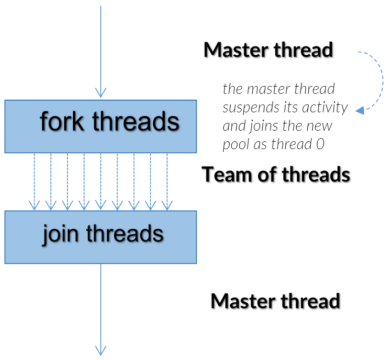
OpenMP programming model

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.

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

```
1 #pragma omp parallel
2 {
3     // This code block will be executed by multiple threads in parallel
4     ...
5 }
```

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.

```
1 // static extent:
2 #pragma omp parallel
3 {
4     double *array;
```

```
5       int N;
6       ...
7       sum = foo(array, N);
8       ...
9   }
10
11  // dynamic extent:
12  double foo(double *A, int N) {
13      double mysum = 0;
14      #pragma parallel for reduction(+:sum) // "orphan" directive
15      for (int ii = 0; ii < N; ii++) {
16          mysum += A[ii];
17      }
18      return mysum;
19  }
```

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.

OpenMP directives can be **orphan**, i.e. they can appear in a dynamic extent without being enclosed in a parallel region. In this case, the directive applies to the current team of threads.

Warning: *Orphan Directives*

Orphan directives must be used with care, as they can lead to unexpected behavior if not properly managed within the context of the existing thread team.

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

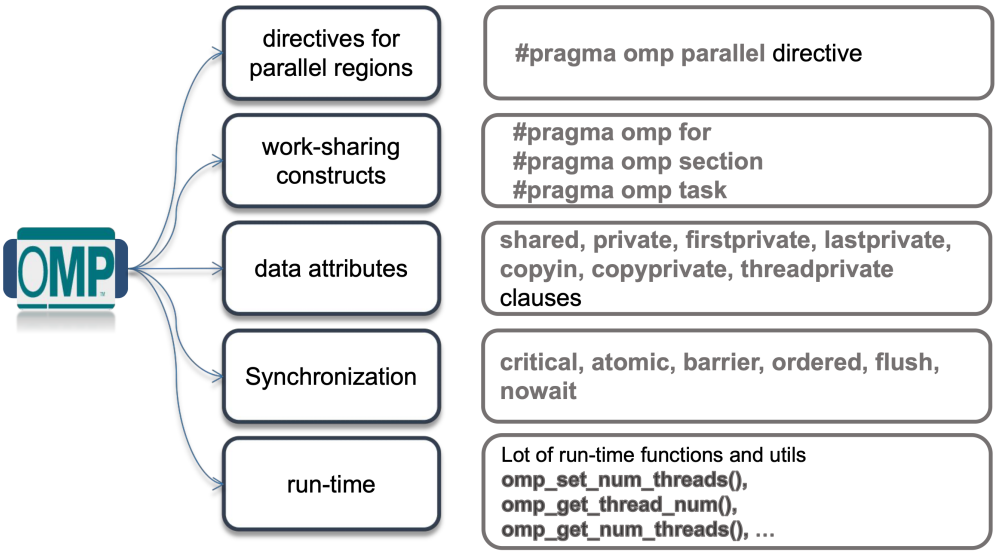


Figure 5.3: OpenMP Toolbox

Tip: ***Conditional Compilation***

By default, when the compiler is instructed to activate the processing of OpenMP directives (`gcc -fopenmp` for example), it defines a macro that let you to conditionally compile sections of the code:

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

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.

5.2 Parallel Regions

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.

The threads factory

When we create a parallel region, a pool of threads is created. Each one receives an ID, ranging from 0 to $n-1$, with n threads. Their stack and IP are separated from the others' ones and from the father-process's one.

- **How large is the stack of each thread?** The default value is system dependent. However, we can control the size of the threads' stack using the environmental variable `OMP_STACKSIZE: export OMP_STACKSIZE=N`, where N is a number followed by a letter ('B', 'K', 'M', or 'G').
- **How many threads can be created by a single process?** Also this is system dependent. On Linux it depends on the amout of total physical memory: basically, the maximum number of threads is the amount of physical memory divided by the memory amount needed to describe and run a thread, times a factor that accounts for the fact that you don't want all the memory allocated only to make the threads alive. One can change the behaviour of the OpenMP program by using the env variable `OMP_THREAD_LIMIT`.

Warning:

When the threads are created, the space needed for their stack is also allocated. If that space is quite large, you may run out of memory. Conversely, if the stack is not large enough, you may incur into a segmentation fault error due to stack overflow.

`parallel` directive

The `parallel` directive creates a parallel region, i.e. a team of threads is created and each thread executes the code within the structured block.

```
1 int nthreads;
2 #pragma omp parallel
3 {
4     int myid = omp_get_thread_num();
5     #pragma single
6     int nthreads = omp_get_num_threads();
7     #pragma barrier
8     printf("Hello from thread %d of %d\n", myid, nthreads);
9 }
```

One may also want to open a parallel region **only if** some condition is met. The condition must result in an integer value

- $0 \rightarrow$ false, do not open the parallel region
- $\neq 0 \rightarrow$ true, open the parallel region

```
1 int nthreads;
2 #pragma omp parallel if(nthreads > 1)
3 {
4     int myid = omp_get_thread_num();
5     #pragma single
6     int nthreads = omp_get_num_threads();
7     #pragma barrier
8     printf("Hello from thread %d of %d\n", myid, nthreads);
9 }
```

or even open it with a given number n of threads

```
1 int nthreads;
2 #pragma omp parallel num_threads(nthreads)
3 {
4     int myid = omp_get_thread_num();
5     #pragma single
6     int nthreads = omp_get_num_threads();
7     #pragma barrier
8     printf("Hello from thread %d of %d\n", myid, nthreads);
9 }
```

Warning:

The `barrier` directive is the most brutal synchronization concept in parallelism: when a thread meets a barrier, it stops and waits for all the other threads to meet the same barrier.

Only when all the threads have reached the barrier, they are all released and can continue their execution.

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)

```
1  int i, j, k;
2  double *array;
3
4  #pragma omp parallel
5  {
6      // i,j,k and array
7      double *array;
8
9      #pragma omp parallel
10     {
11         // i,j,k and array
12         // are shared here
13         ...
14     }
15 }
```

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.

```
1  int i, j, k;
2  double *array;
3
4  #pragma omp parallel private(i,k)
5  {
6  // j and array are shared here i and k are unique to each thread, they
       live in each thread's stack.
7
8  // They are different than the original j and k, and array does NOT
       point to the region pointed by the original array pointer
9      ...
10 }
```

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.

Tip: *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.

```
int i, j, k;
double *array;
array = (double*)malloc(...);
#pragma omp parallel
firstprivate(array)
{
// 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.

// As such, now array can be used to access the previously allocated
   memory.
...
}
```

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.

```
double last_result;

#pragma omp parallel
lastprivate(last_result)
{
...
    #pragma omp for
    for( int j = 0; j < some_value; j++ )
    last_result = calculation( j, ...);
    ...
}

other_calculations( last_result, ... );

// 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.

```
int myN;
double *array;
#pragma omp threadprivate(myN, array)

#pragma omp parallel
{
    // array does exist and 'its private
    // to each thread
    array = ... allocate memory...;
}
// .. something serial here..
#pragma omp parallel
{
    // array does exist here and
    // the allocated memory is available
}
```

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.

```
int golden_values[3];
#pragma omp threadprivate(golden_values)

for( int i = 0; i < N; i++ )
{
    get_golden_values();
    #pragma omp parallel copyin(golden_values)
    {
        // each thread uses golden_values[]
    }
}
```

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
}
```

Controlling the number of threads

By default the number of threads spawned in a parallel region is the number of **cores** available. However, you can vary that number in several ways:

- `OMP_NUM_THREADS` environmental variable
- `#pragma omp parallel num_threads(n)` clause
- `omp_set_num_threads(n)` function

The last two work if `OMP_DYNAMIC` variable is set to TRUE, otherwise the number of threads spawned is strictly equal to `OMP_NUM_THREADS`. This setting can be changed through `omp_set_dynamic(true)`.

Specializing execution in parallel regions

- The `critical` directive ensures that only a thread at a time executes the block. All the threads will execute it, although in unspecified order. A barrier is present at the entry point and no one at

the end point;

- The `atomic` directive is like critical, but limited to a single update operation on a single variable (a single line);
- The `single` directive ensures that only a single thread executes the block. The other threads skip it. There is an implicit barrier at the end of the block, unless `nowait` clause is used. There is an implicity synchronization at the end and only one thread is inside the region: all the others are waiting at the end of the region;
- The `master` directive ensures that only the master thread (i.e. the thread with ID 0) executes the block. The other threads skip it. There is no implicit barrier at the end of the block. There is no explicit synchronization at all: only the thread 0 is inside the region, the others can be everywhere else.

Observation: `mask` *directive*

Since the `master` directive is now deprecated, it is recommended to use the `mask` directive instead.

```
#pragma omp parallel
{
    #pragma omp master
    {
        // code executed only by the master thread
    }

    #pragma omp masked [filter (integer expression)]
    {
        // code executed by all threads except the master thread
    }
}
```

The `filter` clause allows to select which thread must execute the associated code block. No implicit barrier is set at the exit of the region.

Tip: `atomic` *vs* `critical`

When you deal with shared variables, enduring that the workflow maintains the semantic correctness of your code is fundamental. For instance, the assignment

$$a+ = b$$

(where either a, b or both are shared) is meaningful only if the value of a, b or both does not change in the middle of the instruction itself. To avoid that, it is necessary to **protect** the sensible regions.
An `atomic` directive has a much lower overhead than a `critical` one and must be preferred in the appropriate cases:

- **read**: causes an atomic read of the location designated by the expression;
- **write**: causes an atomic write of the location designated by the expression;
- **update**: causes an atomic read-modify-write operation on the location designated by the expression;
- **capture**: combines an atomic read-modify-write operation with a separate assignment of the original value to a variable.

In OpenMP exists a unique global `critical` section. Hence, when we define a `critical` section, it is logically considered to be part of the global one. As a consequence only one thread can be inside any of the unnamed `critical` sections, which of course limits the performance when more than one region is present.
However, that can be cured by the **named regions**, i.e. by giving a name to the critical section:

```
#pragma omp critical
{
    // 'A' critical section
}
#pragma omp critical
{
    // 'B' critical section
}
```

In this case, two different critical sections are defined and two threads can be inside them at the same time.
Finally, since `atomic` protects memory regions while `critical` protects code regions, they are not mutually **exclusive**: a thread may be executing the critical region while another may be executing the atomic.

The ordering of threads

The order by which the threads execute the code inside a parallel region is undefined. Consider the following example:

```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    printf("Hello from thread %d\n", myid);
}
```

How can we order the output of the threads so that they print their ID in increasing order?

The solutions is using a `critical` directive, since it ensures that only one thread at a time executes the associated block and so creates a sort of ”ordering”.

```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    #pragma omp critical
    {
        printf("Hello from thread %d\n", myid);
    }
}
```

However, the order is still undefined, since the order by which the threads enter in the critical section is undefined. A `while` loop can be used to enforce the order:

```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    int done = 0;

    while( !done ) {
        #pragma omp critical
        {
            if( myid == next ) {
                printf("Hello from thread %d\n", myid);
                next++;
                done = 1;
            }
        }
    }
}
```

A third implementation is possible, using the clause `ordered` in a `for` construct:

```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    #pragma omp master
    int nthreads = omp_get_num_threads();
    #pragma omp barrier  // without this barrier, nthreads may be
       undefined

    #pragma omp for ordered
    for( int i = 0; i < nthreads; i++ ) {
        #pragma omp ordered
        {
            printf("Hello from thread %d\n", myid);
        }
    }
}
```

5.3 Working with Loops

Loops are one of the most common work structures in HPC, and vast amount of compute-intensive code resides in loops. In fact, up to version 2.x, OpenMP was essentially about quickly and effectively parallelizing loops without much effort.

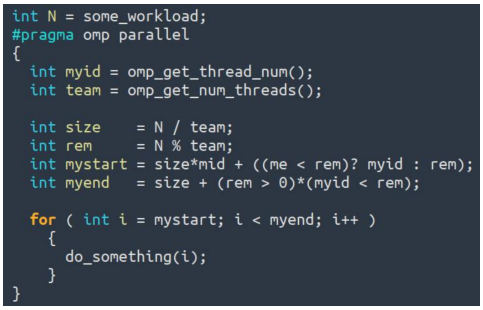


Figure 5.4: Loop Parallelism

Splitting the work of a `for` loop amont the threads could easily be achieved by directly assigning the boundaries of the loop to each thread. However, OpenMP has dedicated constructs that offer easier and more flexible mechanisms to share the work within a `for` loop.

Let's start with a very simple loop:

```
double *a;
double sum=0;
int N;

#pragma omp parallel for implicit(none) shared(a,sum,N) private(i)
for (int i=0; i<N; i++){
    sum += a[i];
}
```

where:

- `for` is a **work-sharing construct** that splits the iterations of the loop among the threads in the team;
- `implicit(none)` forces the programmer to explicitly specify the sharing attributes of all variables in the parallel region;
- `shared(a,sum,N)` specifies that the variables `a`, `sum`, and `N` are shared among all threads;
- `private(i)` specifies that the loop variable `i` is private to each thread.

Tip: *`implicit(none)` clause*

The default poilcy for memory regions is actually that all the variables defined in serial regions at the moment of entering in the parallel region are shared. However, that is a **very** common source oof error since when you have lots of variables, you forgot what is what in the code. It may be a good practice to add `implicit(none)` to all your constructs so that to spot any error alike.
Moreover, it is a good practice to declare the loop counter inside the `for` declaration, so that it is private by default.

How is the work assigned to the single threads?

```
#pragma omp parallel for schedule(schedule-type)
for (int i=0; i<N; i++){
    sum += a[i];
}
```

- `schedule(static, chunk-size)`: the iteration is divided in chunks of size chunk-size (or in equal-size) distributed to threads in circular order;
- `schedule(dynamic, chunk-size)`: the iteration is divided in chunks of size chunk-size (or 1) distributed to threads in no given order;
- `schedule(guided, chunk-size)`: the iteration is divided in chunks of minimum chunk-size (or 1) distributed to threads in no given order like before. The chunk size is proportional to the number of still unassigned iterations divided by the number of threads;
- `runtime-default`: the policy is set at runtime via the environmental variable `OMP_SCHEDULE` or to a default implementation-dependent value.

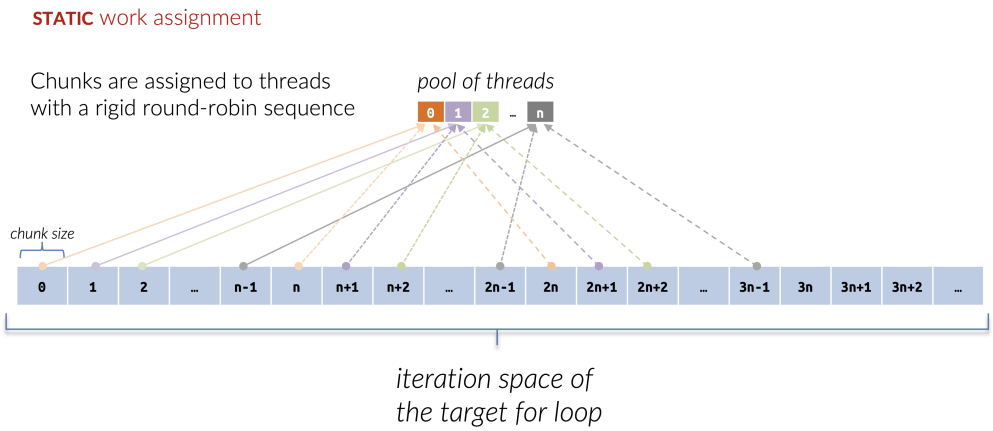


Figure 5.5: `static` work assignment

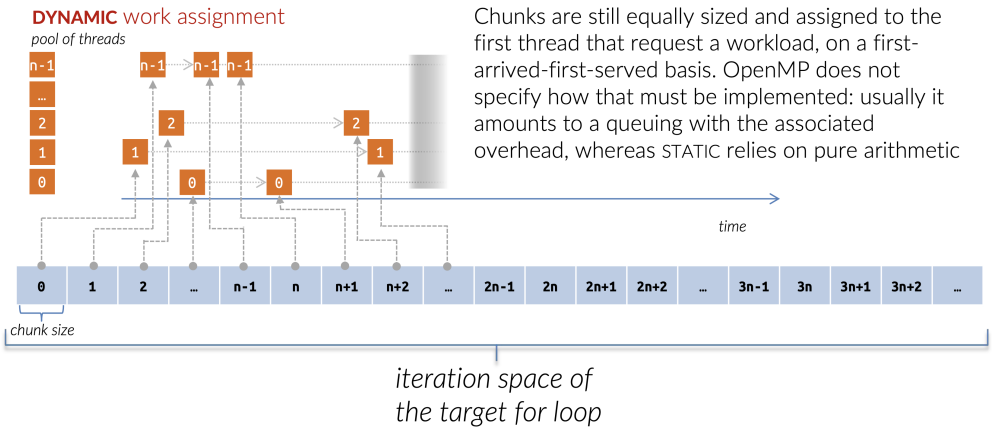


Figure 5.6: `dynamic` work assignment

Observation: `static` *vs* `dynamic`

- `static` assignment is (supposed to be) more effective when each iteration brings the same computational work, since the direct and predictable assignment has a smaller overhead;
- in `dynamic` assignment chunks are assigned to the firsst free thread, so it is supposed to be more effective in the opposite case, i.e., when the computational load of each iteration is unpredictable.

However, the overhead of the dynamic assignment is much larger than the static one, so that it may be convenient to use a chunk size larger than 1.

Clauses for `for` construct

The `for` construct accepts the following clauses:

- `private(list)`: vars in the list will be private to each thread; despite their name is the same our of the parallel region, they have different memory locations and die with the parallel region;
- `firstprivate(list)`: the vars in the list are private and are initialized at the value that shared variables have in the begin of the parallel region;
- `lastprivate(list)`: the vars in the list are private and are not initialized; at the end of the parallel region, the original vars (i.e. the ones in the serial region) will have the value that the

private vars have at the end of the last iteration of the loop;

- `reduction(op:list)`: possible operators are: `+,-,*,&,|,,&&,||, max, min`; the initial value of vars is taken into account at the end of the parallel `for`; at the befin of the for, initialization values are 0 for `+,-`, 1 for `*`, false for `&&,||`, all bits 0 for `&,|,^`, the minimum representable value for `max` and the maximum representable value for `min`;
- `collapse(n)`: enable parallelization of multipla loops level (must be perfectly nested);
- `nowait`: no implicit barrier at the end of the for;

Anatomy of a data race

```
1 #include <omp.h>
2 double *a, sum=0;
3 int N;
4
5 #pragma omp parallel for
6 for (int i=0; i<N; i++){
7     sum += a[i];
8 }
```

In the code above, without the `atomic` directive, the assignment `sum += a[i];` determines a **data race**: between two synchronization points at least one thread writes to a data location from which another thread may read or write. A data race happens when at least two memory accesses

- point to the same memory location;
- are performed concurrently by different threads;
- are not sync ops;
- at least one is a write.

A **race condition** is a semantic error in the code. Due to the random ordering of events, it leads the fact that its behaviour is non-deterministic and the result is not correct.

Let's now say that we solved the data race by using the `atomic` directive. **Does it scale? Surely not! Why?**

```
1 #include <omp.h>
2 double *a, sum=0;
3 int N;
4
5 #pragma omp parallel for
6 for (int i=0; i<N; i++){
7     #pragma omp atomic
8     sum += a[i];
9 }
```

Because the above solution makes the threads to wait for each other to frequently. A critical region has **synchronization points** at the start and at the end of critical regions, meaning that threads have to communicate with each other and decide who's waiting and who's not.

Other solutions are available obviously:

1. use a `reduction` clause:

```
#include <omp.h>
double *a, sum=0;
int N;

#pragma omp parallel for reduction(+:sum)
for (int i=0; i<N; i++){
    sum += a[i];
}
```

2. declaring a shared array for sum:

```
#include <omp.h>
double *a, sum=0;
int N;

#pragma omp master
int nthreads = omp_get_num_threads();

double sum[nthreads];

#pragma omp parallel
{
    int me = omp_get_thread_num();
    #pragma omp for
    for (int i=0; i<N; i++){
        sum[me] += a[i];
    }
}
```

This scales **hardly**, since the values of `sum[nthreads]` reside in the same cache lines; hence, when a thread access and modify its location, to maintain the coherence the cache must write-back and reflush every time. This is the so called **false sharing** problem.

Definition: *False Sharing*

False sharing occurs when each thread explicitly accesses a memory location that is different than any other thread in the parallel pool BUT at least some of those memory locations are mapped on the same cache line. The effect of this is a very poor efficiency when the threads run on different cores.
Note that false sharing is an issue when it happens a large amount of times. Having an array that stores values peculiar for each thread so that they are exposed to all the other threads that access them only once a while, is something very common and not an issue.

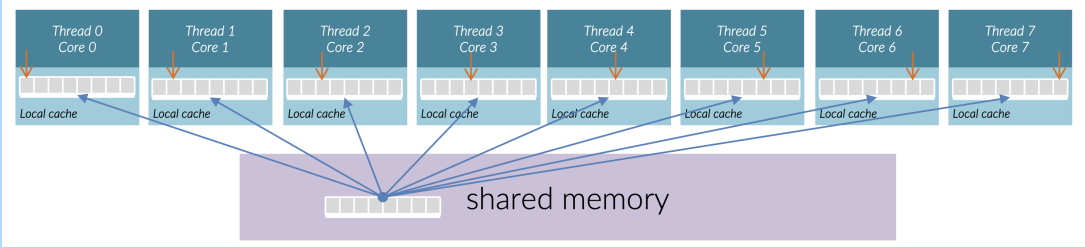


Figure 5.7: False Sharing

If we concentrate only on 2 threads:

(a) both threads read their target memory location;
(b) the entire line is loaded up in the cache of both cores;
(c) the line is flagged as ”S” (shared) in both caches;
(d) thread 0 writes its target location;
(e) the line is flagged as ”M” (modified) for thread 0 and ”I” (invalid) for all the other threads;
(f) thread 0 could write again on its target location without modifying the situation;
(g) thread 1 wants to write its target location;
(h) the entire cache line must be re-flushed to enforce cache-coherence because it is flagged as ”I” for thread 1;
(i) once thread 1 has written its target location, the entire cache line is flagged ”M” for thread 1 and ”I” for all the other threads.

3. a final way that scales better is the following:

```
#include <omp.h>
double *a, sum=0;
int N;
int nthreads;

#pragma omp master
nthreads = omp_get_num_threads();

double sum[nthreads*8];

#pragma omp parallel
{
    int me = omp_get_thread_num();
    #pragma omp for
    for (int i=0; i<N; i++){
        sum[me*8] += a[i];
    }
}
```

where the factor 8 is used to ensure that each `sum[me*8]` resides in a different cache line (assuming a cache line of 64 bytes and a double of 8 bytes). However, this uses much more memory than needed and is hard-coded, hence not portable.

Observation:

When you declare an `omp for` inside an existing parallel region, there is a fundamental difference between the two codes here below. In the snippet A, the `for` is declared without the `parallel`, whereas in snippet B it is declared with the `parallel for`.

- in A, the `for` is shared among the threads that form the pool of the outer `parallel` region;
- in B, **every thread of the pool** is creating a new `parallel` region and inside each of the new `parallel` regions the new pools of threads will execute the `for`. So, in case B, if there are n threads in the outer `parallel` you will have n `for` cycles executed.

```
A
#pragma omp parallel
{
    #pragma omp for
    for( int i = 0; i < N; i
        ++ ) {
        ...
    }
}
```

```
B
#pragma omp parallel
{
    #pragma omp parallel for
    for( int i = 0; i < N; i
        ++ ) {
        ...
    }
}
```

5.4 Threads Affinity

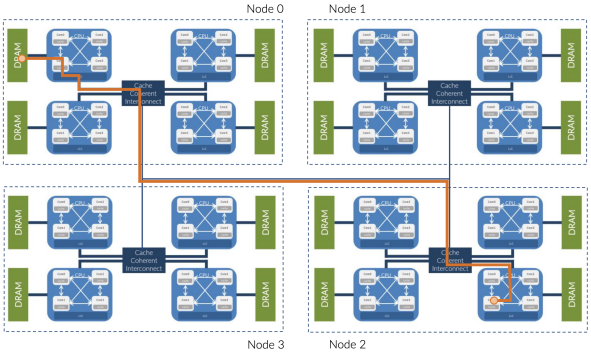
As already mentioned, in modern architectures a unique central memory with a fixed bandwidth would be a major bottleneck in a system with a fast growing number of cores. The problem is solved by physically disjointing the memory in separated units (**memory banks**) each of which is connected to a socket; 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 memory bandwidth grows with the number of sockets, although the latency to access a memory location may vary depending on the core that is trying to access it and on the memory bank where that location resides. In fact, the major drawback is that the access time is no more **uniform**, with severe consequences on how to write and run the codes.

OpenMP and the OS offer the capability to decise where each thread have to run, i.e. on which core and/or how the threads have to distribute on the available cores.

Observation: *SMT*

Each core may have the capability of running more than one thread, which is called **Simultaneous Multithreading (SMT)**. This allows multiple threads to be executed in parallel on a single core, improving resource utilization and overall performance. We can either call **strands** or **hwthreads** the different threads that a physical core could run, as opposed to **swthreads** that indicates the OpenMP threads.

The placemenet of OpenMP threads on the available cores is called **threads affinity**. The default behaviour of OpenMP is to let the OS decide where to place each thread. However, that may lead to a non-optimal placement, with consequent performance degradation. As usual, what ”efficient” means depends on the details of the specific case: for example if n threads work on shared data, it would be more efficient if they share the L2 cache (run on the same socket), so that frequenty used data are at hands for all of them. Conversely, if n threads work on independent memory segments, the most efficient choice is to maximize the memory bandwidth over the shortest core-to-ram-path.



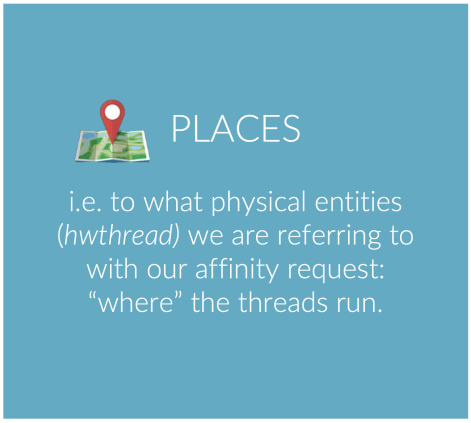
The aim is to have as few **remote memory accesses (RMA)** as possible. It depends on:

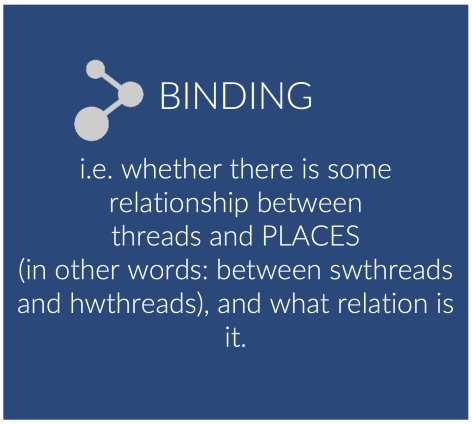
- **Where:** i.e. in what memory bank the data resides;
- **Who:** which thread is accessing the data;
- **What:** how is the workload distributed among the threads.

In principle, one wants to be able to distribute the work in an optimal way, i.e. without any resource contention, and to place the threads in an optimal way, i.e. minimizing the RMA. To do that, one must be able to place each OpenMP swthread to a dedicated computational resource and to grant it the fastest possible access to its own data.

- Placing the swthreads **distant** from each other:
  1. increases the aggregate bandwidth if the data are placed accordingly;
  2. results in a better utilization of each core's cache, since each core has its own cache;
  3. may worsen the performance of synchronization constructs.
- Placing the swthreads **close** to each other:
  1. decreases the latency of synchronization constructs;
  2. decreases the aggregate bandwidth if the data are placed accordingly;
  3. may worsen the cache performance depending on what operations are performed on the data.

OpenMP offers the following mechanisms to control the threads affinity:



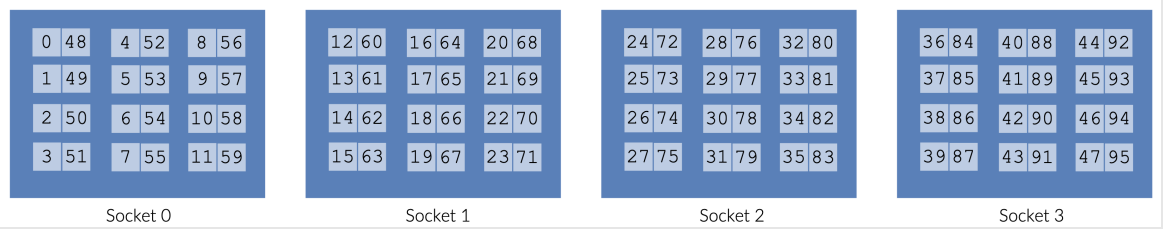


PLACES

Places are where swthreads run. Places can be:

- **threads:** each place corresponds to a hwthread, or strand, on cores;
- **cores:** each place is a core;
- **sockets:** each place is a socket, with its multiple cores;

Example:



On a node the computational resources are identified as the physical threads numbered in a round-robin way. If there are $n_{sockets}$ with $n_{cores-per-socket}$ then there are

$$n_{cores} = n_{sockets} \times n_{cores-per-socket} n_{threads} = n_{cores} \times n_{hwthreads-per-core}$$

The $n_{threads}$ are the computational resources available on the node; in this example we do refer to these IDs.
To pass to OpenMP the places definition, the easiest way is through the environmental variable `OMP_PLACES`:

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

A **place** can be defined by an **unordered** set of comma-separated non-negative numbers enclosed in braces (the numbers are the IDs of the smallest unit of execution on that hardware, i.e. the hwthreads). For example:

- `0,1` defines a place made by hwt 0 and hwt 1; in the previous example, it corresponds to cores 0 and 1 of socket 0;
- `0,48` defines a place made by hwt 0 and hwt 48; in the previous example, it corresponds to hwt and SMT hwt on core 0 of socket 0;
- `0,12,24,36` defines a place made by hwt 0, 12, 24, and 36; in the previous example, it corresponds to hwt on cores 0 of sockets 0,1,2,3;
- `0,1,1,49` a list with two places;

`OMP_PLACES` can be defined as an explicit **ordered** list of comma-separated places. Intervals can be used, specified as `start:counter:stride`:

- `0,48,1,49` sets `OMP_PLACES` to two places;
- `0:4:12` sets `OMP_PLACES` to four places, each made by a single hwthread, corresponding to hwthreads 0, 12, 24, and 36; same as `0,12,24,36`;

The `!` operator can be used to exclude intervals. The places are **static**, meaning that there is no way to change it while the program is running.
Finally, to pass to OpenMP the places definition:

```
export OMP_PLACES = "{0,1},{1,49}"
```

BINDINGS

The **binding** defines how the swthreads are mapped onto the places. The binding can be:

- `NONE:` the placement is up to the OS;

- `CLOSE:` the swthreads are placed onto places as close as possible to each other (assigned to consecutive places in a round-robin way);
- `SPREAD:` the swthreads are placed onto places as evenly as possible, then the places are filled in a round-robin fashion;
- `MASTER:` the swthreads run onto the same place as the master thread;

The binding can be set through the environmental variable `OMP_PROC_BIND`:

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

where `false` is equivalent to ask no policy (i.e. ”none”) and to allow the OS to migrate the threads, while `true` is the same but forbids the migration.

The binding can be set also through the clause `proc_bind` in a non-persistent way:

```
#pragma omp parallel proc_bind(policy)
{
    ...
}
```

where `policy` is one of the above.

`CLOSE`

- $T \leq P$: there are sufficient places for a unique assignment. Swthreads are assigned to consecutive places by their thread IS. The first one is the master's place;
- $T > P$: at least one place will execute more than one swthread. Swthreads are splitted in P subsets (S_{t_i}), so that
$$floor(\frac{T}{P}) \leq S_{t_i} \leq ceiling(\frac{T}{P})$$

`SPREAD`

- $T \leq P$: place list is splitted in T subsets; each one contains at least $floor(\frac{P}{T})$ and at most $ceiling(\frac{P}{T})$ consecutive places. A thread is assigned to a partition then, starting from the master thread. Then, assignment proceeds by thread ID, and the threads are placed in the first place of the next subpartition;
- $T > P$: place list is splitted in P subsets, each of which contains only 1 place and S_{t_i} threads with consecutive IDs. The number of threads in each subpartition is chosen so that
$$floor(\frac{T}{P}) \leq S_{t_i} \leq ceiling(\frac{T}{P})$$
At least one place has more than one thread assigned to it. The first subset with S_{t_0} contains thread 0 and runs on the place that hosts the master thread.

extbfScope	**Function**
extbfSetting the affinity	`omp_bind_clause()`
extbfGetting the affinity	`omp_get_proc_bind()`

extbfGetting details on places	`omp_get_num_places()`
	`omp_get_place_num()`
	`omp_get_place_num_procs()`
	`omp_get_place_proc_ids()`

extbfDisplaying the affinity	`omp_display_affinity()`
	`omp_get_affinity_format()`
	`omp_set_affinity_format()`
	`omp_capture_affinity()`

Table 5.1: OpenMP Affinity Library Functions

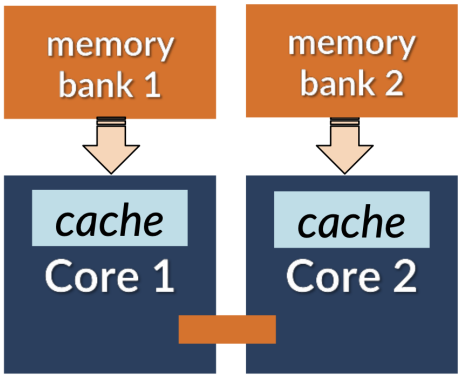
places / binding	THREADS	CORES	SOCKETS
CLOSE	swt are placed on close hwt, saturating all the SMT hwt in each core before using new cores	swt are placed on close hwt, using 1 hwt/core before starting to use SMT	swt are placed round-robin per socket, 1/core; after saturation, SMT is used by round-robin +1 hwt/socket
SPREAD	swt are placed round-robin sockets, onto free cores in sockets	similar to ← SMT is avoided until saturation	similar to ← swt are placed by round-robin sockets and hwt
MASTER	all swt are placed on the same hwt on the same core on the same socket	all swt are placed on the same core on the same socket, using all its hwt	all swt are placed on the same socket, saturing all hwt starting from SMT ones

Figure 5.8: Threads Affinity Examples

The OpenMP standard offers several `omp_` library functions to deal with the affinity:

Memory allocation

It is possible to control on what physical memory the data will reside by carefully **touching data**.



Suppose that you are operating on a SMP system similar to the one in Figure 5.9. Each socket is physically connected to a RAM bank, and then physically connected to other sockets. This way, the memory access is **not-uniform**: the bandwidth for a core to access a memory bank not physically connected to it is likely to be significantly smaller than that to access the closest bank.

Figure 5.9: Touching Data in SMP Systems

The matter is: **who owns the data?**

```
1  double *a = (double*)calloc(N, sizeof(double));
2
3  for (int ii=0; ii<N; ii++){
4      a[ii] = initialize(ii);
5  }
6
7  #pragma omp parallel for reduction(+:sum)
8  for (int i=0; i<N; i++){
9      sum += a[i];
10 }
```

In this way, all the data are physically paged in the memory bank of the core on which the master thread runs; its cache is also warmed-up; the other thread must access the memory bank 1 which is not the most suited for the bandwidth. This way, the cache of the thread that initialize (first touch) the data is warmed-up **and the data are allocated in the memory connected to it**.

Observation:

In the **touch-first** policy, the data pages are allocated in the physical memory that is the closest to the physical core which is running the thread that access the data first. If a single thread is initializing all the data, then all the data will reside in its memory and the number of remote accesses will be maximized.

In the **touch-by-all** policy, the cache of each thread is warmed-up with the data it will use afterwards **and the data are allocated into each thread's memory**.

```
1  double *a = (double*)malloc(N, sizeof(double));
2
3  #pragma omp parallel for
4  for (int ii=0; ii<N; ii++){
5      a[ii] = initialize(ii);
6  }
7
8  #pragma omp parallel for reduction(+:sum)
9  for (int i=0; i<N; i++){
10     sum += a[i];
11 }
```

Observation: `malloc` *vs* `calloc`

- `malloc` notifies that the required amount of memory will be used, and the memory occupancy of the process in the heap is grown accordingly; however, the actual mapping of the memory pages into the physical memory does not happen until the pages are actually touched (read or written). Moreover, the mapping is done only for the touched pages, not for the entire amount of memory.
- `calloc` same as `malloc`, but with two differences:
  1. the memory is required to be physically contiguous, hence entirely on the same physical location;
  2. all the memory is initialized to zero as a way to immediately touch it so that it is mapped onto a physical bank as soon as it is required.

If each thread touches as first the data it will operate on subsequently, those data are allocated in the physical memory that is the closest. Hence, each thread will have its data placed in the most convenient memory and the remote access will be minimized.

Tip: ***Discover your topology***

There are some tools usually used on HPC platforms to discover the hardware topology of the node you are working on. The most common ones are:

- `numactl --hardware`: shows the NUMA nodes and the CPUs associated to each node;
- `lscpu`: shows the CPU architecture;
- `lstopo`: shows a graphical representation of the hardware topology;
- `hwloc-ls`: shows a textual representation of the hardware topology.

6 MPI

6.1 Introduction

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 how is MPI structured? It is organized into several key components:

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

```
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 */
)
```

we'll see it more in detail later. 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

```
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 */
)
```

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**

Let’s create the first and most basic MPI program.

```
#include <mpi.h>
int main( int argc, char **argv ) {
    MPI_Init( &argc, &argv ); /* Initialize MPI */

    MPI_Finalize( ); /* Terminate MPI */
    return 0; /* explicitly return as best practice */
}
```

The second most simple MPI program is obviously ”hello MPI world”!

```
#include <mpi.h>

int main( int argc, char **argv ) {
    int Myrank, Nprocs;

    MPI_Init( &argc, &argv ); /* Initialize MPI */

    MPI_Comm_rank( MPI_COMM_WORLD, &Myrank ); /* Get my rank */
    MPI_Comm_size( MPI_COMM_WORLD, &Nprocs ); /* Get number of processes
        */

    printf( "Hello from process %d of %d\n", Myrank, Nprocs );

    MPI_Finalize( ); /* Terminate MPI */
    return 0; /* explicitly return as best practice */
}
```

Tip: *The man page*

There exist a wonderful thing, invented by smart people, called **man page**. It is a documentation of the functions provided by the system and by libraries. To access it you can use the command

```
1          man MPI_$FUNCTION$
```

where `$FUNCTION$` is the name of the function you want to know more about.

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".

Figure 6.1: Communicator and groups

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.

Observation:

Always create a separated **context** for the application you're writing.

```c
#include <mpi.h>

int main(int argc, char **argv) {
    int Myranlk, Ntasks;
    int mpi_provided_thread_level;
    MPI_Comm myCOMM_WORLD;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &
       mpi_provided_thread_level);
    MPI_Comm_dup(MPI_COMM_WORLD, &myCOMM_WORLD); /* create a
       separated context */

    MPI_Comm_size(&Ntasks, myCOMM_WORLD);
    MPI_Comm_rank(&Myranlk, myCOMM_WORLD);
    ...
}
```

6.2 Point-to-point communications

Building blocks: Send and Receive

Recalling the previous section, MPI provides two basic functions for sending and receiving messages between processes:

```c
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 */
)
```

```c
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 */
)
```

We can say that the message consists of a **body** (`buf`, `count`, `datatype`) and an **envelop** (`dest`,

`tag`, `comm`). 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 6.1: 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*

```
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);
}
```

Why the `MPI_STATUS` in `MPI_Recv`?

- The `MPI_Recv`'s argument count provides the maximum number of elements that the call expects to receive. If the message exceeds that count, an error is thrown. Hence, you don't know whether `MPI_Recv` has got $n \leq count$ elements.
- Valid values for the source and tag arguments are `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, so that the task could receive messages from anybody with any tag.
- However, it may be that, once received, you need to know who was the sender, which was the tag and what is the size of the received message. Use `status.TAG, status.MPI_SOURCE` and `MPI_Get_count(&status, MPI_type_used, &count)` to get this information.

You may also want to know whether a message is arriving, from who and how large before to actually get it.

```
MPI_Status status;

// Probe for an incoming message from whatever process and whatever tag
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

source = status.MPI_SOURCE; // Get the source
tag = status.MPI_TAG;       // Get the tag
MPI_Get_count(&status, MPI_BYTE, &number_amount); // Get the size
char *buffer = (char *)malloc(number_amount); // Allocate the buffer
MPI_Recv(buffer, number_amount, MPI_BYTE, source, tag, MPI_COMM_WORLD, &
    status); // Now receive the message
...
// Probe for an incoming message from process zero with a precise tag
MPI_Probe(0, 123, MPI_COMM_WORLD, &status);
...
```

Observation: *Safety first*

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. However, as a more general discussion, there are two main protocols:

1. **Eager protocol**

   Here, `MPI_Send` returns before the data actually reach the destination. More correctly, it returns without knowing whether that happened already or not. The MPI library copies the sent data on local buffer, either on the sender size or the receiver size. The actual transfer will complete afterwards and the `MPI_Send` returns.
   This protocol is used for **small messages** usually.

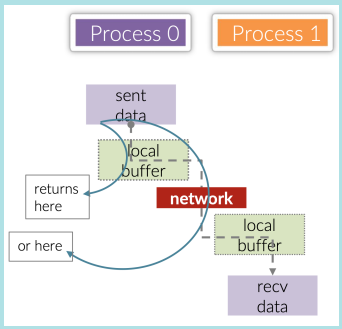


Figure 6.2: Eager protocol

2. **Rendezvous protocol**

   Here, the sender first asks the agreement to the receiver. Once it gets the acknowledgment, the data transfer starts. At the end of it, the `MPI_Send` and `MPI_Recv` returns.
   This protocol is used for **large messages** usually, for which the bufferization would require too much memory.

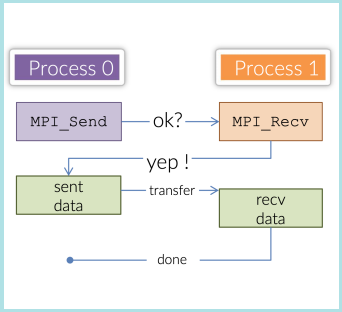


Figure 6.3: Rendezvous protocol

As a matter of fact, `MPI_Recv` does not complete until the buffer is full and `MPI_Send` does not complete until the buffer is empty. As such, the completion of `MPI_Send/Recv` depends on the size of the message and the size of the buffer provided by MPI.
This in general may lead to potentially dangerous situations, such as **deadlocks** or **unsafe code**, which appears to run smoothly just because of the system's bufferization.

Warning: ***Deadlocks and unsafe code***

Unsafe code rely on the system's bufferization to run correctly. Solutions to cure or to avoid these situations are:

- design more carefully the communication pattern;
- check the runnability by substituting `MPI_Send` with `MPI_Ssend`;
- use `MPI_Sendrecv`;
- supply explicitly buffer with `MPI_Bsend`;
- non-block operations.

The order of the `MPI_Send/Recv` must be crafted so that a `MPI_Send` is always matched by a corresponding `MPI_Recv` that is posted before the `MPI_Send` is executed, with the same **source, tag, communicator**.
Deadlocks happen when two or more tasks are waiting for each other to send or receive messages, resulting in a standstill where none of the tasks can proceed. This situation typically arises from circular dependencies in communication patterns.

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 ); }
```

```
if (Rank == 0) {
  Recv ( from 1 );
  Send ( to 2 ); }
else if (Rank == 1) {
  Recv ( from 2 );
  Send ( to 1 ); }
else if (Rank == 2 ) {
  Recv ( from 0 );
  Send ( to 0 ); }
```

MPI does not ensure any ordering of messages from different sources:

- Messages from the same source to the same target with the same tags are ensured to arrive in issuing order;
- Messages from different sources to the same target with the same tags may arrive in any order;

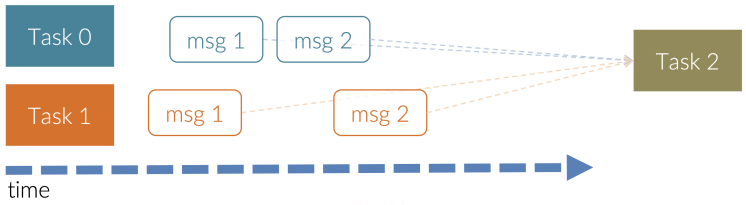


Figure 6.4: Message ordering

Moreover, MPI does not ensure any fairness in case of message starvation.

If two messages from two sources match a `MPI_Recv` on a task (two `MPI_Send` with same size, tag, comm; one `MPI_Recv` with that size, `MPI_ANY_SOURCE` and either the same tag or `MPI_ANY_TAG`) then it is undefined which one will be received.

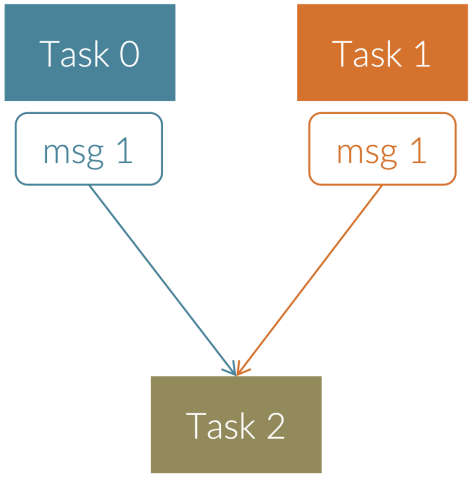


Figure 6.5: Message starvation

The `MPI_Barrier` ensures a synchronization among the MPI threads.

```c
int MPI_Barrier(MPI_Comm comm)
```

It is a **collective call**, that completes when all the MPI ranks in the group have entered the barrier.

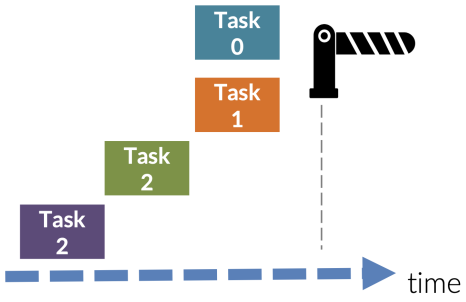


Figure 6.6: MPI Barrier

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 6.2: Different MPI send and receive routines

`MPI_Ssend`

```c
int MPI_Ssend(
    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 */
)
```

It has the same signature of `MPI_Send`, the only difference is that `MPI_Ssend` always apply the synchronous (i.e. rendez-vous) protocol, and hence it returns when the actual data sending starts. That is the reason why it can be used to spot unsafe communication patterns.

Requester	Server
`MPI_Irecv`	←
`MPI_Send`	`MPI_Recv` ↑
	`MPI_Rsend` ↑
`MPI_Wait`	

`MPI_Bsend`

```c
int MPI_Bsend(
    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 */
)
```

It has the same signature of `MPI_Send`, the only difference is that `MPI_Bsend` uses a buffer that must have been attached previously and is detatched afterwards.

1. allocate room for the buffer. Clearly its size must be the max data size that will be sent. Note that `MPI_BSEND_OVERHEAD` bytes must be added per every call posted;
2. notify to MPI that that area is the buffer to be used. That is said **to attach**;
3. when all the `MPI_BSEND`s issued that use that buffer have completed, the buffer can be detatched;
4. the detatch will not return until the send has been completed.

`MPI_Rsend`

```c
int MPI_Rsend(
    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 */
)
```

It has the same signature of `MPI_Send`, the only difference is that `MPI_Rsend` requires that the matching `MPI_Recv` was already posted because it skips all the protocols and immediately starts the communication. It may be really performant but it must be used with **extreme caution**. A typical pattern is:

Observation:

Quite often, there is a combined `MPI_Send & MPI_Recv` pattern between pairs of processes, for instance when performing domain decomposition when an all-to-all exchange take place, or a shift on a chain on processes. `MPI_Sendrecv` offers this combined call that executes exactly that pattern.

```c
int MPI_Sendrecv(
    const void *sendbuf,    /* starting address of send buffer */
    int sendcount,          /* number of elements in send buffer */
    MPI_Datatype sendtype,  /* datatype of each send buffer element
        */
    int dest,               /* rank of destination */
    int sendtag,            /* send message tag */
    void *recvbuf,          /* starting address of receive buffer
       */
    int recvcount,          /* number of elements in receive buffer
        */
    MPI_Datatype recvtype,  /* datatype of each receive buffer
       element */
    int source,             /* rank of source */
    int recvtag,            /* receive message tag */
    MPI_Comm comm,          /* communicator */
    MPI_Status *status      /* status object */
)
```

With the `MPI_Sendrecv_replace` variant, the same amount of data is sent for each process and the same buffer is used for sending and receiving.

```c
int MPI_Sendrecv_replace(
    void *buf,              /* starting address of send and receive
        buffer */
    int count,              /* number of elements in send and
       receive buffer */
    MPI_Datatype datatype,  /* datatype of each send and receive
       buffer element */
    int dest,               /* rank of destination */
    int sendtag,            /* send message tag */
    MPI_Comm comm,          /* communicator */
    MPI_Status *status      /* status object */
)
```

Non-blocking communication

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:

```
MPI_Test ( MPI_Request *, int *flag, MPI_Status *)
MPI_Wait ( MPI_Request *, MPI_Status *)
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:

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

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               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.

The `MPI_Request` variable is used ot 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 which sets the `flag` variable to 0 (not completed) or 1 (completed):

```
int MPI_Test(
    MPI_Request *request,   /* communication request */
    int *flag,              /* true if operation completed */
    MPI_Status *status      /* status object */
)
```

A call to `MPI_Test` returns `flag=true` if the operation identified by request is complete. In such case, the status object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is

set to `MPI_REQUEST_NULL`. The call returns `flag=false`, otherwise. In this case, the value of the status object is undefined. `MPI_Test` is a local operation.

In case the `MPI_Status` variable is not needed, `MPI_STATUS_IGNORE` can be used instead.

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

- for **one** communication:

```
1 MPI_Test ( MPI_Request *, int *flag, MPI_Status *)
2 MPI_Wait ( MPI_Request *, MPI_Status *)
```

- for **multiple** communications:

```
1 MPI_Testall (int count, MPI_Request array_of_req[], MPI_Status
     array_of_st[] )
2 MPI_Waitall (int count, MPI_Request array_of_req[], MPI_Status
     array_of_st[] )
```

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 6.3: Comparison of blocking and non-blocking MPI communication routines

Example: *Non-blocking communication example*

```
MPI_Recv( data_bunch, from prev_proc );
int flag_send = 1;

while( data_bunch != no_data ){
    int flag_recv;
    MPI_Request req_recv, req_send;

    MPI_Irecv( next_data_bunch, &req_recv );
    process( data_bunch ); // do something with the data

    do {
        if( flag_send ){
            MPI_Isend( data_bunch, to_next_proc, &req_send );
        } else {
            MPI_Test( &req_send, &flag, MPI_STATUS_IGNORE );
        }
    } while( flag == 0 );

    MPI_Test( req_recv, &flag, MPI_STATUS_IGNORE );
    while( flag != true ){
        do_something_else();
        MPI_Test( req_recv, flag );
    }
};
```

Non-blocking communications avoid to get stuck in non-returning communication when the involved processes are non synchonized. If there are other independent tasks that the processes can perform, the non-blocking can be used instead. Quite often it is possible to re-design the workflow so that the communications are ”pre-emptively” issued while some calculations is performed on a previous data bunch.

6.3 Collective communication

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:

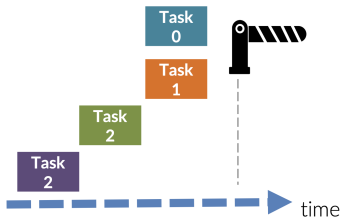
1. **Synchronization**: These operations are used to synchronize processes within a group. The most common example is `MPI_Barrier`, which blocks each process until all processes in the communicator have called it.
2. **Data Movement**: These operations involve distributing, gathering, or re-arranging 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.

Figure 6.7: Types of collective operations

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

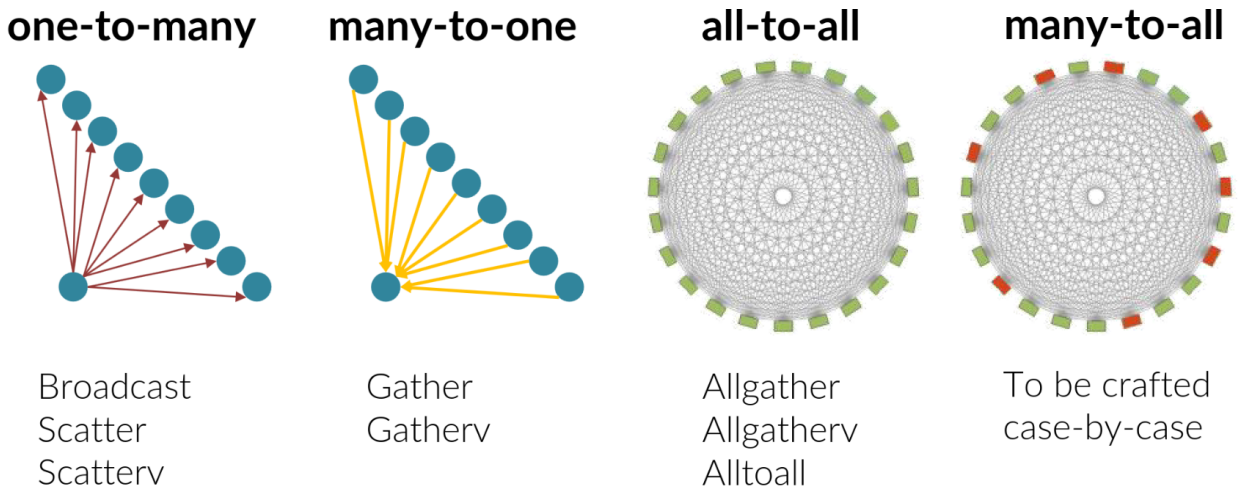
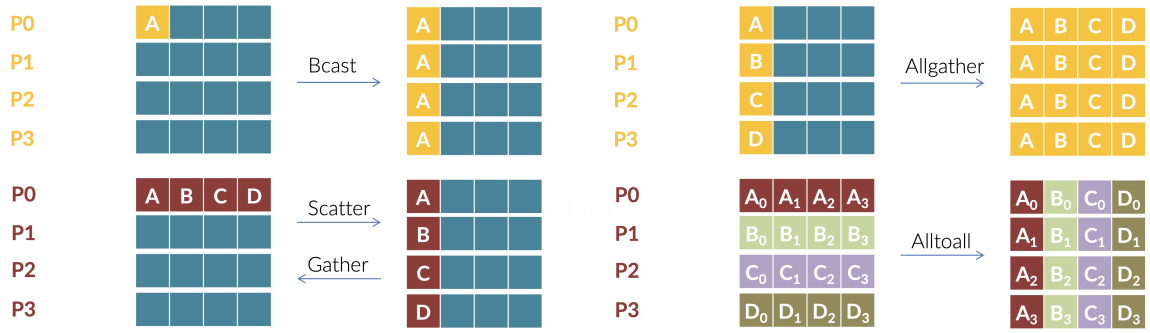


Figure 6.8: Data movement operations



where `Allgather` is equivalend to `Gather` followed by `Bcast` . `Allgather` algorithms can be faster. `Alltoall` is equivalent to a transpose of the data.

Observation: *"v" versions of collectives*

The collective we saw deal with the same amount for each process. There are several cases in which you may want to deal with a **variable amount of data per process**. MPI provides the **v** version of the routines ("v" stands for "vector") for these cases. Even if efficient algorithms exist, those are not as efficient as the ordinary fixed-size ones.

Collective computations combine communications with computation:

- **Reduce:** all-to-one, combined with an operation;
- **Scan:** the prefix-sum, all prior ranks to all combined with an operation;
- **Reduce_scatter:** all-to-all, combined with an operation.

The performed operations can be either the predefined ones or an user-defined one.

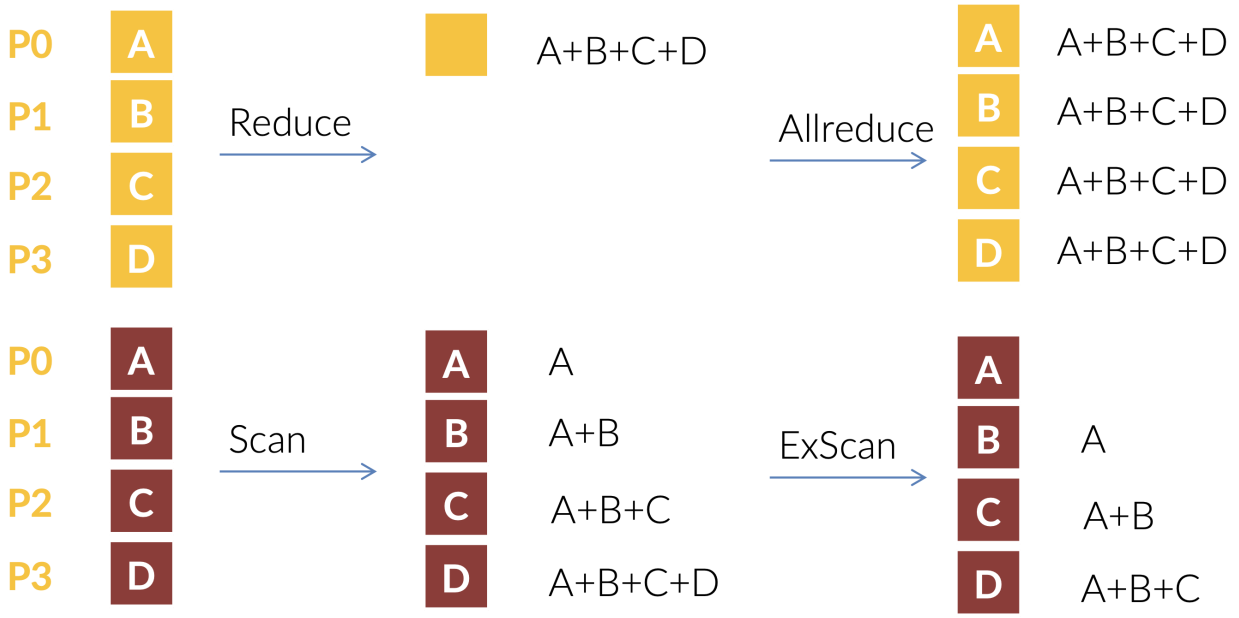
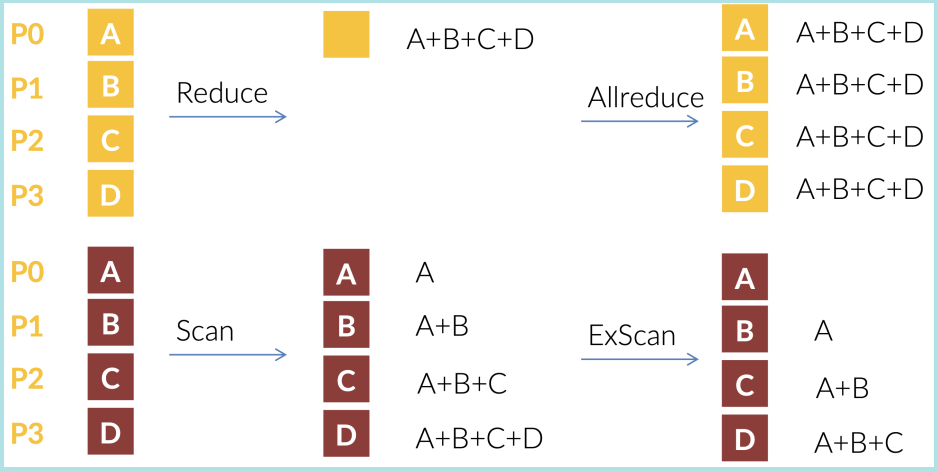


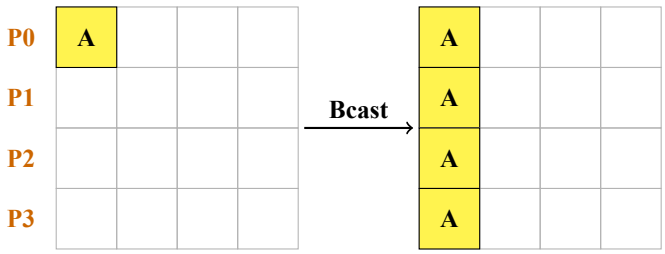
Figure 6.9: Collective computation operations

Observation:

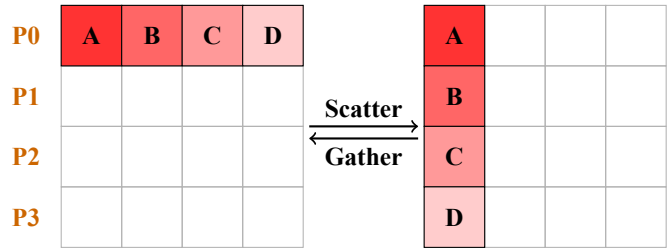
Note: there are many possible algorithms for collective communications. A `Broadcast`, for instance, may be implemented either as



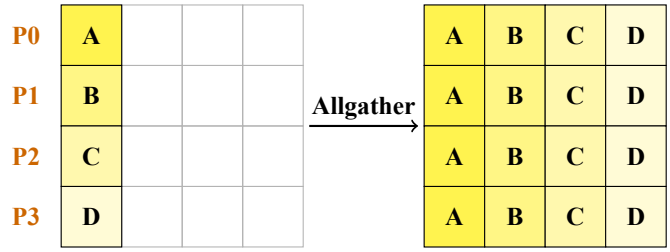
- **Broadcast:** The `MPI_Bcast` function is one of the most fundamental collective operations in MPI. It allows a single process (the root) to send the same data to all other processes in the communicator. The root process (P0) holds the data (A) and broadcasts it to all other processes. After the broadcast,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



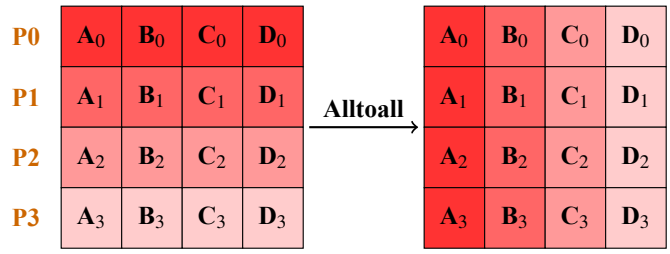
- **Scatter and Gather:** The `MPI_Scatter` and `MPI_Gather` functions are complementary collective operations that distribute and collect data among processes. `MPI_Scatter` takes data from a single process (the root) and distributes it among all processes, while `MPI_Gather` does the opposite - it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



Groups and communicators

The situation in which a task is preparing data and distributing the work among the other tasks is called **director/orchestra paradigm**. The final reduce must happen only among the workers, excluding the director task. That would not be a problem, since it would suffice that the director had a zero-valued variable to participate to the reduce. However, this is the case to illustrate how to create new groups and communicators.

When you want to derive a group from an existing communicator, as first you need to ”extract” the group of tasks associated with the communicator:

```
1 MPI_Group group;
2 MPI_Comm_group( communicator, &group );
```

Then, you manipulate the group. For example by selecting/excluding some of the tasks:

```
MPI_Group_excl( group, n, ranks, &newgroup );
MPI_Group_incl( group, n, ranks, &newgroup );
MPI_Group_range_excl( group, n, ranges, &newgroup );
MPI_Group_range_incl( group, n, ranges, &newgroup );
MPI_Group_union( group1, group2, &newgroup );
```

Finally, you create a new communicator from the new group:

```
MPI_Comm newcomm;
MPI_Comm_create( communicator, newgroup, &newcomm );
MPI_Group_free( &group ); /* free the old group */
MPI_Group_free( &newgroup ); /* free the new group */
```

Observation:

A communicator holds an internal reference to the group, we don't need the groups anymore after the communicator has been created.

Alternatively, it is also possible to operate on the original communicator, subdividing it in two sections:

```
MPI_Comm new_comm;
MPI_Comm_split( communicator, color, key, &new_comm );
```

then you'll need to know what are the ranks in the new communicator

```
MPI_Comm_size( new_comm, &new_size );
MPI_Comm_rank( new_comm, &new_rank );
```

The root task will get `MPI_INVALID` as `Myrank_new` and `new_comm_size` and `MPI_COMM_NULL` as `new_comm` .

7 Debugging

A **bug** is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Hence, **debugging** is the process of finding and resolving bugs within a computer program that prevent correct operation of computer software or a system. What is the best way to proceed?

1. Do not insert bugs → well, obviously you can say :), but not that easy as you may think;
2. Add `printf` statements everywhere → highly discouraged, but sometimes works and is the first approach you may think of;
3. Use a debugger → the best way to proceed, but you need to learn how to use it.

7.1 GDB

The GNU Project Debugger (`gdb`) is a portable debugger that runs on many Unix-like systems and works for many programming languages, including C, C++, and Fortran. It allows you to see what is going on 'inside' another program while it executes or what another program was doing at the moment it crashed. There are three basic usages of `gdb` :

- Debugging a code (best is compiled using the `-g` option);
- Inspecting a code crash through a core file;
- Debugging/inspecting a running code.

Tip:

It is highly advised to learn **keyboard commands**.

Debugging a code

In order to include debugging information in your code, you need to compile it with the `-g` family options (read the `gcc` manual for complete info):

- `-g` : produce debugging information in the operating system's native format (e.g. DWARF, STABS, etc.);
- `-ggdb` : produce debugging information in the format used by `gdb` ;
- `-g3` : produce even more debugging information, including macro definitions.
- `-O0` : disable optimizations (highly recommended when debugging).
- `-ggdb(level)` : produce debugging information in the format used by `gdb` , with different levels of detail (0-3).

You just start your code under `gdb` control:

```
gdb ./my_program
```

You can define the arguments needed by your program already at invocation

```
gdb --args ./my_program arg1 arg2 ...
```

or you can define the arguments from within `gdb` using the `set args` command:

```
gdb ./my_program
Reading symbols from my_program...done.
(gdb) set args arg1 arg2 ...
(gdb) run
```

To run the code (to spot a segmentation fault, for example) you just need to use the `run` command. If your program crashes, `gdb` will stop the execution and will show you where the crash happened. Or, you may want to stop it from the beginning to have full control over each step:

```
(gdb) break main
(gdb) run
```

Or you may already know where the bug is and set a breakpoint there:

```
(gdb) break myfile.c:linenumber
(gdb) run
```

Observation: ***Breakpoints***

Breakpoints are a key concept in debugging. They are stopping points at which the execution interrupts and the control is given back to you, so that you can inspect the memory contents (vars, register values, ...) or follow the subsequent execution step by step. A breakpoint can be defined in several ways:

- at the current position:

```
(gdb) break
```

- at offset lines after/before the current line:

```
(gdb) break +N
(gdb) break -N
```

- at linenum of a file:

```
(gdb) break filename:linenum
```

- at the entry point of a function:

```
(gdb) break function_name
```

- more options in the `gdb` manual.

A breakpoint may be defined as dependent on a given condition:

```
(gdb) break linenum if var==value
```

This sets a breakpoint at line `linenum` that will stop the execution only if the variable `var` is equal to `value` .
You can also define a list of commands to be executed when the breakpoint is hit:

```
(gdb) break linenum
(gdb) commands
> command1
> command2
> end
```

This sets a breakpoint at line `linenum` that will execute `command1` and `command2` when the breakpoint is hit, before stopping the execution and giving the control back to you.

When you have the **control** of the program execution, you can decide how to proceed:

- `(gdb) cont [c]` : continue until the end/next stop
- `((gbd) count-ignore)` : continue ignoring the next count-ignore stops (for instance, a bp)
- `(gdb) next [n] | count` : continue to the next src line in the current stack frame
- `(gdb) step [s] | count` : conutnue to the next src line, entering in called functions
- `(gdb) until [u] | count` : continue until a src line past the current one is reached in the current stack fr
- `(gdb) advance location` : continue until the specified location is reached

Examining the **stack** is often of vital importance. With `gdb` you can have a quick and detailed inspection of all the stack frames.

```
backtrace   [args]  /* print the backtrace of the whole stack */
            n       /* print only the n innermost frames */
            -n      /* print all but the n innermost frames */
            full    /* print local variable values */
```

Accessing the content of **memory** is fundamental when debugging:

- `(gdb) print var` : print the value of variable `var` ;
- `(gdb) x/FMT address` : explore memory starting at address `address` , where `FMT` is a format specifier (see the `gdb` manual for details);
- `(gdb) display expr` : add expr to the list of expressions to display each time your program stops.

Examining **registers** is also possible:

- `(gdb) info registers` : print the value of all registers;
- `(gdb) info vector` : print the content of vector registers;

- `(gdb) print &rsp` : print value of the stack pointer.

If there are macros in the code, they can be expanded, provided that you compiled the code with the appropriate option (e.g. `-g3 [gdb3][-dwarf-4]`):

- `(gdb) macro expand macro` : shows the expansion of macro; expression can be any string of tokens;
- `(gdb) info macro [-a|-all] macro` : shows the current (or all) definition(s) of macro;
- `(gdb) info macros location` : shows all macro definitions effective at location.

Finally, you can set **watchpoints** (aka "keep an eye on this and that") instead of breakpoints, to stop the execution whenever a value of an expression/variable/memory region changes:

- `(gdb) watch variable` : keep an eye onto variable;
- `(gdb) watch expression` : stops when the value of expression changes;
- `(gdb) watch -l expression` : interpret expression as a memory location to be watched;
- `(gdb) rwatch expression` : stops when expression is read.

Inspecting a code crash

It happens that you have code. crashes in conditions not easily reproducible when you debug the code itself. However, the OS can dump the entire program status on a file, called the **core file**. In order to allow it to dump the code, you have to check/set the core file size limit:

```
1 ulimit -c [size limit in KB]
```

Once you have a core, you can inspect it with `gdb` :

```
1 gdb ./executable ./corefile
```

The first thing to do, normally, is to unwind the stack frame to understand where the program crashed:

```
1 (gdb) bt full /* print the backtrace of the whole stack with local
      variable values */
```

Debugging a running process(es)

In order to debug a running process, you can simply attach `gdb` to it:

```
1 gdb
2 (gdb) attach PID
```

and start searching it to understand what is going on.

Observation: *GUI*

You can start `gdb` with a text-user-interface:

```
gdb -tui
```

or you can activate/deactivate it from within `gdb` itself:

- `Ctrl-x a` : toggle the TUI mode (press Ctrl and x together, then a);
- `Ctrl-x o` : change focus;
- `Ctrl-x 2` : shows assembly windows;
- `layout src` : shows the source code window (type at the gdb prompt);
- `layout asm` : shows the assembly window;
- `layout split` : shows both source code and assembly windows.

7.2 Debugging parallel codes

Debugging in parallel is much more complex since the fundamental additional challenge is the simultaneus execution.

Shared memory paradigm (OpenMP, pthreads, ...)

- Multiple threads running;
- Shared vs private memory regions;
- Race conditions.

Message-passing paradigm (MPI)

- Multiple independent processes (+ possible multithread);
- Communication;
- Deadlocks.

```
gcc -g -o my_program my_program.c -lpthread
gdb ./my_program
```

It is necessary to explicitly set up `gdb` for multi-thread debugging:

```
(gdb) set paginating off
(gdb) set scheduler-locking On
(gdb) set non-stop [on|off]
```

In `all-stop` mode, whenever the execution stops, all the threads stop (wherever they are). Whenever you restart the execution, all the threads restart. However, `gdb` can. not single-step all the threads in the steplock. Some threads may execute several instructions even if you single-stepped the thread under focus with `step` or `next` commands. `non-stop` mode means that when you stop a thread, all the other ones continue running until they finish or they reach some breakpoint that you pre-defined.

```
(gdb) thread thread_no /* change focus to thread_no */
(gdb) info threads /* shows info of active threads */
(gdb) thread apply [thread_no] [all] args /* apply a command to a list
   of threads */
(gdb) break <...> thread thread_no /* inserts a break into a list of
   threads */
```

Tip:

- If possible, write a code natively parallel but able to run in serial, which means with 1 MPI task or 1 thread;
- Profile, debug and optimize that code in serial first;
- If multi-threaded, test and debug thread sync/races with 1 MPI task;
- Deal with communications, synchronization and race/deadlock conditions on a small number of MPI tasks;
- Profile, debug and optimize communications on a small number of MPI tasks;
- Finally, try the full-size run.

Unfortunately, some times bugs or improper design issues arise only with large number of processes or threads.

It is still possible to use `gdb` directly, called from mpirun:

```
mpirun -np <NP> -e gdb ./my_program
```

Warning:

However, depending on your system that may not work as expected.

The simplest way to use `gdb` with a parallel program is

```
mpirun -np <NP> xterm -e gdb ./my_program
```

which launches `<NP>` xterm windows with running gdb processes in which you can run each parallel process.

```
(gdb) run <arg_1> <arg_2> ...
```

Observation:

Normally on an HPC facility you do that while running an **interactive session** and in several occasions this will not work since HPC environments are hostile to X for several reasons (remember to connect with `-X` or `-Y` switch of ssh).

Another possibility is to open as many connections as processes on different terminals on your local machine, and attach `gdb` to the already running MPI processes, even though it is not so practical:

```
mpi run -np <NP> ./my_program <arg_1> <arg_2> ...

/* followed by, on each terminal */
gdb -p <PID_of_MPI_task_n>
```

There are still two **issues**:

1. **Where** to run `gdb`, if **xterm** is not available and you don't want to use it in multi-thread mode? Consider using `screen`;
2. **How** the MPI tasks should be convinced to wait for `gdb` to step in?

A possible issue for attaching `gdb` to a running process is that you **may not have the capability to do that** on a Linux system. Solutions:

- Get the capability from the system administrator;
- As root, type `echo 0 > /proc/sys/kernel/yama/ptrace_scope`;
- Set the `kernel.yama.ptrace_scope` variable in the file `/etc/sysctl.d/10-ptrace.conf` to 0.

The last solution turns off the security measure permanently, it is not a good idea.

We are left with the problem of attaching the `gdb` to a running process. There is a classical trick, that requires to insert some small additional code in your program:

```
int wait=1;
while (wait)
    sleep(1);
```

The MPI processes will wait indefinetly until the value of wait does not change, which you can do from inside `gdb` attached to each process.

Bibliography

[1] David Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48.

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