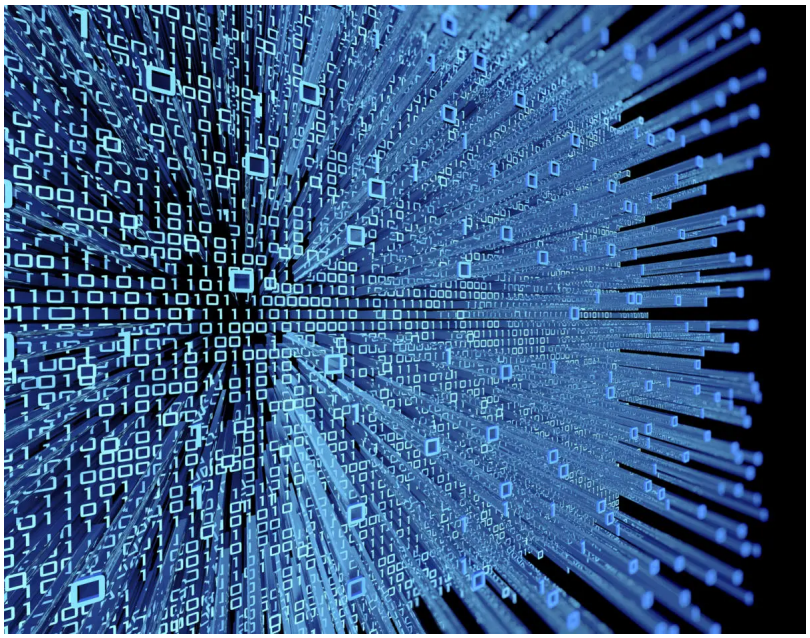


LINMA2710 Scientific Computing

Issambre L'Hermite Dumont
Simon Desmidt

Academic year 2024-2025 - Q2

UCLouvain

Contents

Shared-Memory Multiprocessing

1.1 How memory works

1.1.1 Memory hierarchy

To store the data that it will use, the CPU uses memory. Memory is hierarchical like a pyramid. The higher it is, the faster it goes, but the less space there is.

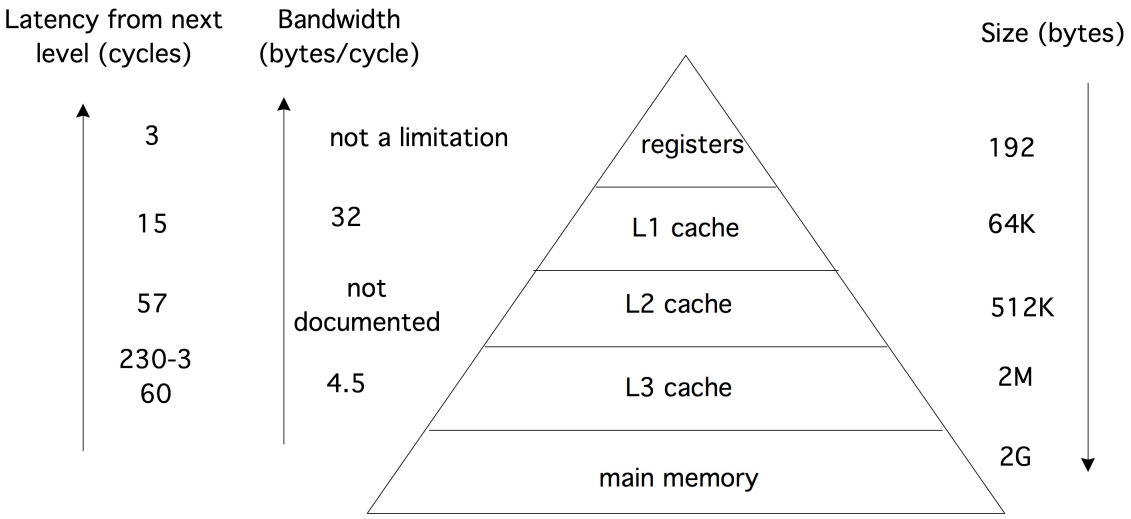


Figure 1.1: Memory hierarchy

First we need to define a cycle . It's an unit of time that is defined like this: $cycle = \frac{1}{\text{CPU freq}}$. For example, for the CPU *AMD Ryzen 5 5600X*, the maximum frequency is $4.6GHz$, so the cycle is $cycle = \frac{1}{4.6GHz} \approx 0.217ns$. The **bandwidth** is the number of bytes that can be transferred in one cycle. And the **latency** is the number of cycles required to access a level of memory. There's also the **latency** but for a number of bytes, its formula is $\alpha + \beta n$ where α is the level latency, β is the inverse of the bandwidth and n is the number of bytes.

Level	Level Latency $[cycle]$	Bandwidth $[bytes/cycle]$	Size	What is stored	example
Register	3	No limit	$\pm 192B$	"Immediate" data for the CPU	Results of addition, memory address
Cache L1	15	$32-64$	$\pm 64KB$	Instructions and "Immediate" data	Local variable
Cache L2	57	$16-32$	$\pm 512KB$	Data used recently	Data struct, code part
Cache L3	$230-360$	$4-10$	$\pm 64MB$	Data shared between core	Global variable
RAM	$300-500$	$9-50GB/s$	$\geq 4GB$	Running programs	Running software, open document
Disks	$\geq 10^5$	Usually $\leq 3GB/s$	$\geq 128GB$	Persistent data	Document, OS, etc

1.1.2 Caches lines and prefetching

A **cache line** is a small fixed-size contiguous block of memory, usually 64 or 128 bytes. It's not necessarily stored in the cache. We use them to organize the memory, because it is easier to deal with fixed size block. When the CPU need to access a memory location, it loads the entire cache line into the cache of the CPU. If the wanted data is not in the cache, there will be a **cache miss**. After that the CPU will load the entire cache line into the cache.
The **prefetching** is the fact that the CPU will load the cache line that is next to the one that is needed. It's because of the spacial locality. Spacial locality is the reason why we use cache lines. For example, if we store an array of data, it may use some space greater than one cache line so for precaution, the CPU will load the next cache line too. And so we save time, by anticipating.
The data locality is important for at least the two following reasons:

- **Temporal locality**: If a data is used frequently, we will keep it in the cache.
- **Spacial locality**: If a data is used, the data next to it could be useful too.

1.1.3 Arithmetic intensity

The **arithmetic intensity** is a concept in performance analysis for memory-bound and compute-bound programs. Let's consider a program that does o arithmetic operations and m memory operations, we define:

- **Arithmetic intensity**:
$$a = \frac{o}{m} \tag{1.1}$$
It helps to find if the program is limited by a compute-bound or memory-bound.

- **Arithmetic time**:
$$t_{arith} = \frac{o}{\text{CPU freq}} \tag{1.2}$$
It's the time needed to perform o operations.

- **Memory transfer time**:
$$t_{mem} = \frac{m}{\text{bandwidth}} = \frac{o}{a \times \text{bandwidth}} \tag{1.3}$$
It's the time needed to do m memory operations.

The overall performance of a program is thus defined by the worst component of the PC, and so we get the **time per iteration**:

$$\min\left(\frac{t_{arith}}{o}, \frac{t_{mem}}{o}\right) \tag{1.4}$$

With some algebra, we can find the **number of operations per second**:

$$\max\left(\text{CPU freq}, a \times \text{bandwidth}\right) \tag{1.5}$$

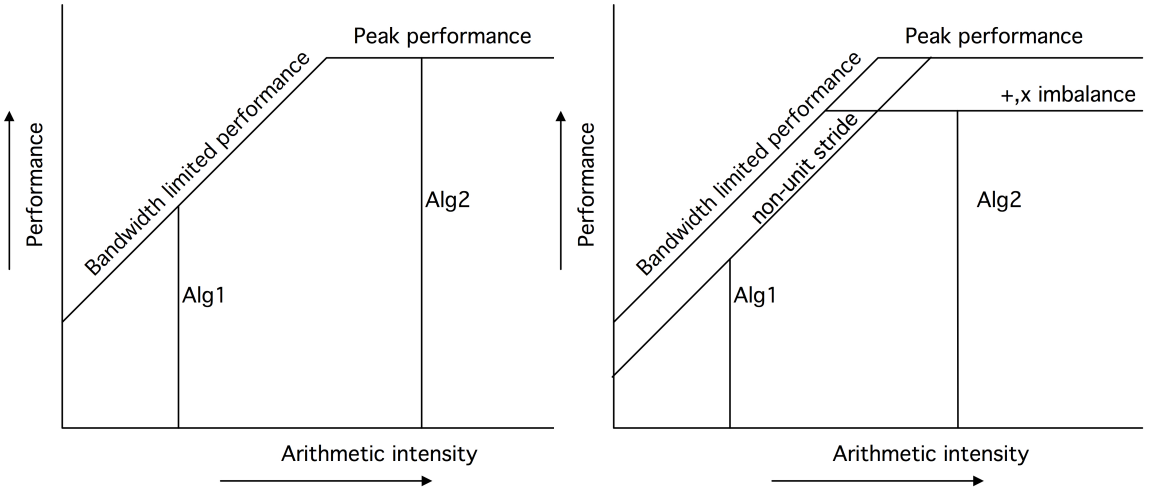
To resume:

- If a is low then the performance is memory-bounded $\Rightarrow$ **Need cache optimization**.
- If a is high then the performance is compute-bounded $\Rightarrow$ **Need more CPU cores or vectorization**.

The arithmetic intensity can be represented with the **roofline model** which we'll talk now.

1.1.4 The roofline model

The two following graph link the performances of a program with the arithmetic intensity. The "performances" means the number of operations per second and the "arithmetic intensity" is the number of operations per byte of memory transferred. The right graph is the case where we consider a perfect usage of the PC, and the left graph is the case where we consider a bad usage of the PC.



(a) Roofline model (b) Model with worse performances

As we can clearly see, we have two limitations:

- Bandwidth limit (slopped line)
- Computing limit (flat line)

These two limits lines are upper bound for performances and are limited from above by the hardware (we can't do better than what our CPU can do), but they can be lower if the program is not optimized. For example the slopped line can be lower if we don't use the cache correctly. The flat line can be lower if we don't use the CPU correctly (e.g. using SIMD).

1.1.5 Cache hierarchy for a multi-core CPU

In the case where we have a multi-core CPU, the organization of the memory isn't always the same, it depends of the CPU architecture. For example, we can have a hierarchy like this:

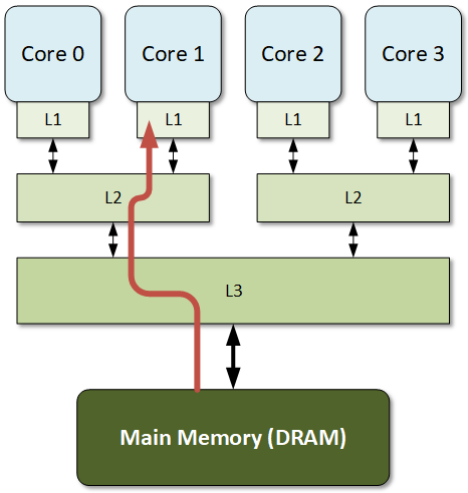


Figure 1.3: Cache hierarchy for a multi-core CPU

Here the L1 cache is private to each core, the L2 cache is shared between two cores and the L3 cache is shared between all cores, as well as the RAM.

1.2 Parallelism

In this course, we use *OpenMP* to parallelize our code instead of *lpthread*. The advantages of *OpenMP* are detailed in the chapter **useful tools** at the section 4.1.
Parallelism is a very useful tool to optimize the performances of a program. It allows to use multiple cores of the CPU to do multiple tasks at the same time. But sometimes it may arise some problems of performances or computational error.

1.2.1 computational error

When using multi-threading, we may modify and use some variables at the same time which could cause computational error. So to avoid that we can either protect the variable (with some adapted tools) but this option is not very efficient because it takes some time to *lock* and *unlock* and to wait for the variable to be available. Or we can make each thread use its own version of the variable, this introduces the next problems.

1.2.2 Race condition

The race condition is a problem that occurs when two or more threads try to modify the same variable at the same time, here there are no computational error possible. For example if all the threads want to modify a variable a the same time, they will need to wait for the variable to be available, and it is a waste of time. The solution for that, as we said before is to make each thread use its own version of the variable. Let's say they need to compute the sum of an array. The solution is to make each thread compute the

sum of a part of the array and then by "reduction" sum all the sums. We will see this in more detail in the next chapter.

1.2.3 False sharing

As we say before, in the section about cache lines 1.1.2, the CPU loads the entire cache line into the cache when it need to use some data that is on the cache line. So here is the problem, let's say that all of the partial sum of the array are stored in the same cache line, when a thread modify its partial sum, the entire cache line is loaded into the cache of the CPU, and so the other threads will have to wait for the cache line to be available. The solution is to make each thread use its own cache line. We can solve this by adding some padding (shift the data) to the data structure.

1.3 Amdahl's law

First let's define the **speedup** of a program, it represents the improvement of running time while using p threads. It is defined by:

$$S_p = \frac{T_1}{T_p} \tag{1.6}$$

Where T_1 is the time needed to run the program with one thread and T_p is the time needed to run the program with p threads.
Then let's define the **efficiency** of a program, is mesures how well the parallelization is done. It is defined by:

$$E_p = \frac{S_p}{p} \tag{1.7}$$

Where S_p is the speedup of the program with p threads. We get 3 cases:

- If $E_p = 1 \Rightarrow$ **Ideal case**, the parallelization is perfect.
- If $E_p < 1 \Rightarrow$ **Realistic case**, there's some inefficiency.
- If $E_p > 1 \Rightarrow$ **Unrealistic case**, would imply a super-linear speedup.

We can now define **Amdahl's law**, which explain the limitation of parallelization due to the presence of a sequential portion in a program. It is defined like this:

$$T_p = F_s T_1 + \frac{(1 - F_s)T_1}{p} \tag{1.8}$$

F_s is the percentage of the program that is sequential (impossible to parallelize). With that we can redefine the **speedup** and the **efficiency** of a program:

$$S_p = \frac{T_1}{F_s T_1 + \frac{(1-F_s)T_1}{p}} = \frac{1}{F_s + \frac{1-F_s}{p}} \Rightarrow \lim_{p\to\infty} S_p = \frac{1}{F_s} \tag{1.9}$$

And

$$E_p = \frac{S_p}{p} = \frac{1}{pF_s + 1 - F_s} = \frac{1}{F_s(p-1) + 1} \tag{1.10}$$

1.3.1 Application of Amdahl's law to parallel sum

Let's consider the sum of an array, we want to parallelize it. The time without parallelization is n and with parallelization is $\frac{n}{p} + \log_2(p)$, the log term is the time needed to sum all the partial sums. So the speedup is:

$$S_p = \frac{n}{\frac{n}{p} + \log_2(p)} = \frac{1}{\frac{1}{p} + \frac{\log_2(p)}{n}} \tag{1.11}$$

And the efficiency is:

$$E_p = \frac{S_p}{p} = \frac{1}{1 + \frac{p}{n}\log_2(p)} \tag{1.12}$$

We clearly see that if we use more than n processes the efficiency will decrease. Because if $p \geq n$ then we have $T_p = \log_2(n)$ and $\lim_{s\to\infty} S_p = \frac{1}{\log_2(n)}$ and $F_s = \log_2(n)$

Single Instruction Multiple Data (SIMD)

Maybe to improve

SIMD (Single Instruction Multiple Data) is a parallel computing architecture used to process multiple data points simultaneously with a single instruction.
The idea of SIMD is that instead of executing the same instruction separately for each data element, SIMD processes multiple elements in parallel within a single CPU cycle.
To simplify SIMD takes advantages of **vectorization**, to optimize the performances of the program. It uses register to store the vector on which it does the computing.
Let's do a little example, consider the following code:

```
int a[] = {...};
int b[] = {...};
int c[size];
for(int i = 0; i < size; i++){
  c[i] = a[i] + b[i];
}
```

Here we use `int` which is 32 bytes so if we use SIMD with AVX2 registers, we can store $(512/32 =)16$ integers in a register. And so SIMD with that type of registers will automatically translate the previous code to:

```
int a[] = {...};
int b[] = {...};
int c[size];
for(int i = 0; i < size; i+=16){
  __m512_add_epi16(c + i, a + i, b+i);
}
```

We can write the code with the intrinsics (`__m512_add_epi16`) but it's not necessarily. The compiler will do it for you.
To use SIMD, you need to give to your compiler some flags, for example:

- `-O2` or `-O3` for optimization
- `-mavx` for AVX
- `-mavx2` for AVX2
- `-mavx512` for AVX512

Distributed Computing with MPI

3.1 Single Program Multiple Data (SPMD)

Single Program Multiple Data means that we make multiple processes execute the same program but operate on different data. It's a kind of parallel computing that we call distributed computing.

3.1.1 Message Passing Interface (MPI)

MPI is a standardized and portable message-passing system used to enable parallel computing across multiple nodes. Here are some key steps with MPI:

- Initialization: `MPI_Init(&argc, &argv);`, initialize MPI and the processes.
- Getting the number of processes: `MPI_Comm_size(MPI_COMM_WORLD, &nprocs);`, determines how many processes are running (`nprocs` is the same for all processes.)
- Getting the rank of the process: `MPI_Comm_rank(MPI_COMM_WORLD, &procid);` Give a unique ID (rank) to each process, which allows them to perform different tasks despite running the same program.
- Finalizing MPI: `MPI_Finalize();`, finalize MPI and free the ressources.

3.2 Collectives

Lets consider a system that can support 4 processes for the explcit examples (tables). And for the bounds consider as usual p proccess and the variables defined for the latency 1.1.1 (α, β, n) and an arithmetic intensity parameter (γ). Here is some illustration of how collectives work:

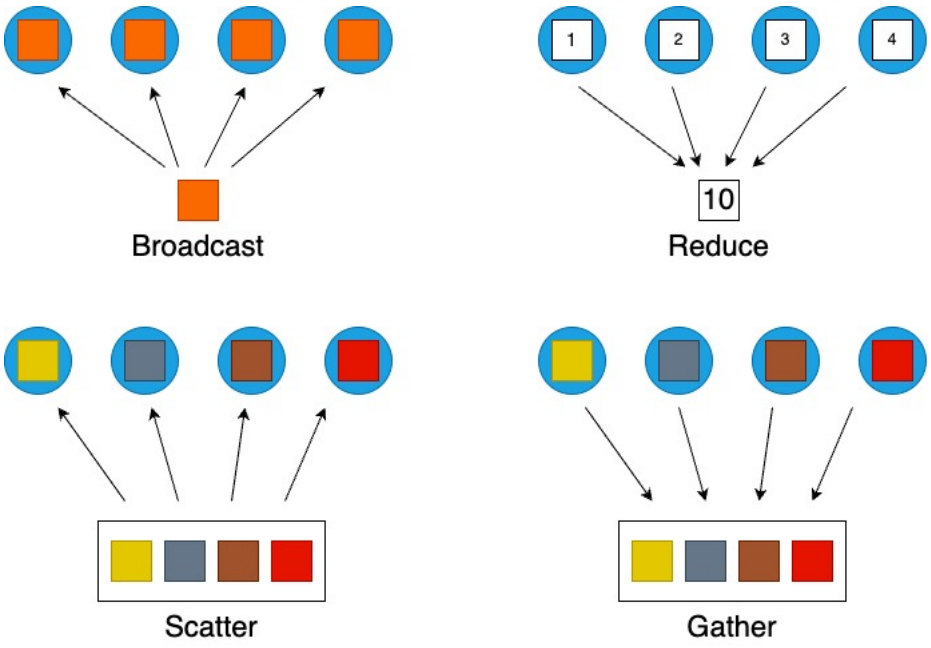


Figure 3.1: Collectives

3.2.1 Broadcast

Broadcast makes a process gives a variable to the other processes.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x						x	x	x	x

Using *spanning tree algorithm*, we can **broadcast** in $\log_2(p)(\alpha + \beta n)$.

3.2.2 Gather

Gather makes a process gather the variables of the other processes.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0						x_0			
		x_1					x_1			
			x_2				x_2			
				x_3			x_3			

Using *spanning tree algorithm*, we **gather** in $\log_2(p)\alpha + \beta n$.

3.2.3 Reduce

Reduce sums up all the values of the variables on the differents processes and store the sum on one process.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0	x_1	x_2	x_3			$\sum_{i=0}^{3} x_i$			

Using *spanning tree algorithm*, we can **reduce** in $\log_2(p)(\alpha + \beta n) + \log_2(p)\gamma n$. To simplify we have: $\log_2(p)(\alpha + (\beta + \gamma)n)$.

3.2.4 All gather

All gather is a sort of combination of **gather** and **broadcast**. It gathers all the variables of the processes and broadcast them to all the processes but more efficiently.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0						x_0	x_0	x_0	x_0
		x_1					x_1	x_1	x_1	x_1
			x_2				x_2	x_2	x_2	x_2
				x_3			x_3	x_3	x_3	x_3

Using *spanning tree algorithm*, we can **all gather** in $\log_2(p)\alpha + \beta n$.

3.2.5 Reduce scatter

Reduce scatter is a sort of combination of **reduce** and **scatter**. It reduces the variables of the processes and scatter them to all the processes, but more efficiently.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$			$\sum_{j=0}^{3} x_{0,j}$			
	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$				$\sum_{j=0}^{3} x_{1,j}$		
	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$					$\sum_{j=0}^{3} x_{2,j}$	
	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$						$\sum_{j=0}^{3} x_{3,j}$

Using *spanning tree algorithm*, we can **reduce scatter** in $\log_2(p)\alpha + (\beta + \gamma)n$

3.2.6 All reduce

All reduce is a sort of combination of **reduce** and **broadcast**.

Before					$\Longrightarrow$	After				
procid	0	1	2	3		procid	0	1	2	3
	x_0	x_1	x_2	x_3			$\sum_{i=0}^{3} x_i$	$\sum_{i=0}^{3} x_i$	$\sum_{i=0}^{3} x_i$	$\sum_{i=0}^{3} x_i$

We explained above that we can do a combination of **reduce** and **broadcast**, but we can **all reduce** more efficiently doing **reduce scatter** then **all gather**, and we can do it in $\log_2(p)\alpha + \beta n$.

Useful tools

4.1 OpenMP

TODO: Improve
OpenMP is a library that allows to parallelize code, it is better than *lpthread* for multiple reasons:

- Easy to add to an existing code, because of the compilation directives (`#pragma omp`). It's an implicit gestion of the threads unlike *lpthreads* that require an explicit gestion. It allows to parallelize a code without changing the whole structure of the code.
- Automatic gestion of the threads, for example if you want to parallelize a loop, you don't have to create the threads, the library will do it for you and optimize the number of threads.
- Simplified synchronization with tools. *OpenMP* provides tools to synchronize the threads like `critical`, `atomic`, `barrier`, etc. They help to protect the shared data between the threads.
- Optimized use of the cache
- Compatible with *SIMD*
- Portability: it is compatible with *Windows*, *Linux*, *MacOS*. With multiple CPU architectures like *Intel*, *AMD*, *ARM*, etc.