Parallel and Distributed Systems - Appunti

Francesco Lorenzoni

September 2024

Contents

Part I

Introduction to SDC

Chapter 1

Basic Concepts

Fun fact: SPM stands for *Software Paradigms and Models*, the historical name of the course

1.1 Parallel Computing

Definition 1.1 (Parallel Computing) *the practice of using multiple processors in parallel to solve problems more quickly than with a single processor. It implies the capability of:*

- ◇ *identifying and exposing parallelism in algorithms and software systems*
- ◇ *understanding the costs, benefits, and limitations of a given parallel implementation*

1.1.1 Current usages

The motivation for parallel computing is the need to solve larger and more complex problems in less time, typically *simulation* ones, but not only. Besides, today, even from the single machine perspective, there exists no more the single processor architecture, so parallel addresses also exploiting the multiple cores available in a single machine.

- ◇ Big Data Analytics (BDAs)
- ◇ HPC and/for AI

Besides also the *Moore's law* indicates another motivation:

Definition 1.2 *Gordon Moore, co-founder of Intel, observed that the number of transistors on a chip doubles every 18-24 months, leading to a doubling of the performance of the chip.*

However, even if the number of transistors on a chip continues to increase, we started to face the problem of powering simultaneously all the transistors, leading to the *power wall* problem. It was estimated in the early 00s that the *power density* of a chip would reach the power density of a nuclear reactor by 2020, and then the power density of the sun in a while. This was the main reason for the shift from single-core to **multi-core** chips (**CMP**s).

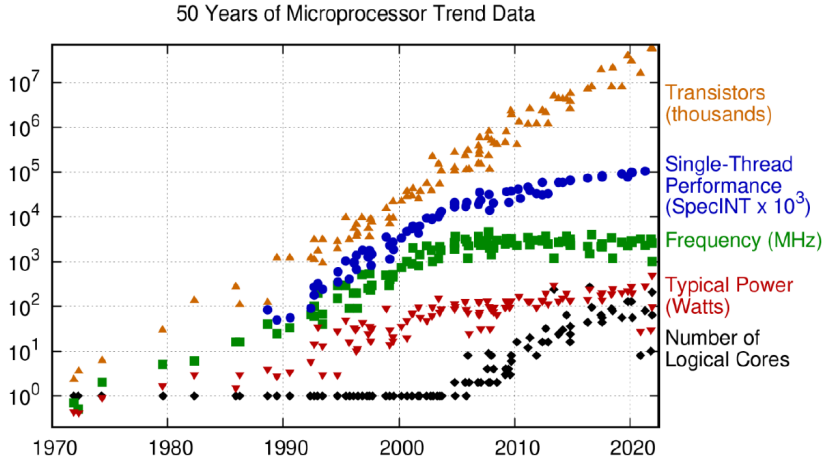


Figure 1.1: Microprocessors in the last 30 years
Single thread performance is increasing slowly, while the Frequency is stable. Moore's law is still valid if we account the number of cores.

Multicore processors help reducing power for this reason:

1. Doubling the number of cores doubles the performance, but also power ☺
2. Doubling the number of cores and *halving* Voltage and Frequency, leaves the same performance unaltered, but the power consumption is reduced by a factor of 4. ☺

To fully exploit the potential of multicore processors, programmers need to **parallelize** our software.

There also forms of parallelization under-the-hood, which make the parallelization transparent to the developer. There also libraries that help in parallelizing the code, such as *OpenMP* or *FastFlow*.

There also Heterogeneous CMPs which integrate different processor cores in a single chip, but they are more complex to handle. Common examples are the integration of a GPU in the chip, or the integration of a *big.LITTLE* architecture, which integrates high-performance cores with low-power cores. Real-world uses are some ARM processors, or the Apple M1.

Chapter 2

Compilation - Leiserson MIT

2.1 Interpreters vs Compilers

Interpreted languages are more versatile, but much slower.

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

This code executed using Clang/LLVM 5.0 takes 1156s (19m 16s) to execute, about **2x** times faster than Java and **18x** times than python

2.2 Cache

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

We can change the order of the loops without changing the result, but the performance can change.

Loop order (outer to inner)	Running time (s)	Last-level-cache miss rate
i, j, k	1155.77	7.7%
i, k, j	177.68	1.0%
j, i, k	1080.61	8.6%
j, k, i	3056.63	15.4%
k, i, j	179.21	1.0%
k, j, i	3032.82	15.4%

Figure 2.1: Performance against loop order

As you can see, there is a huge difference in the running time of the loop depending on the loops ordering. This is due to **caching**, which consists in storing in a fast-access memory previously accessed memory lines.

Figure 2.1: Memory layout for matrix rows

Matrices are stored in memory in row-major order, so the first loop should iterate over the rows of the matrix, to exploit the cache.

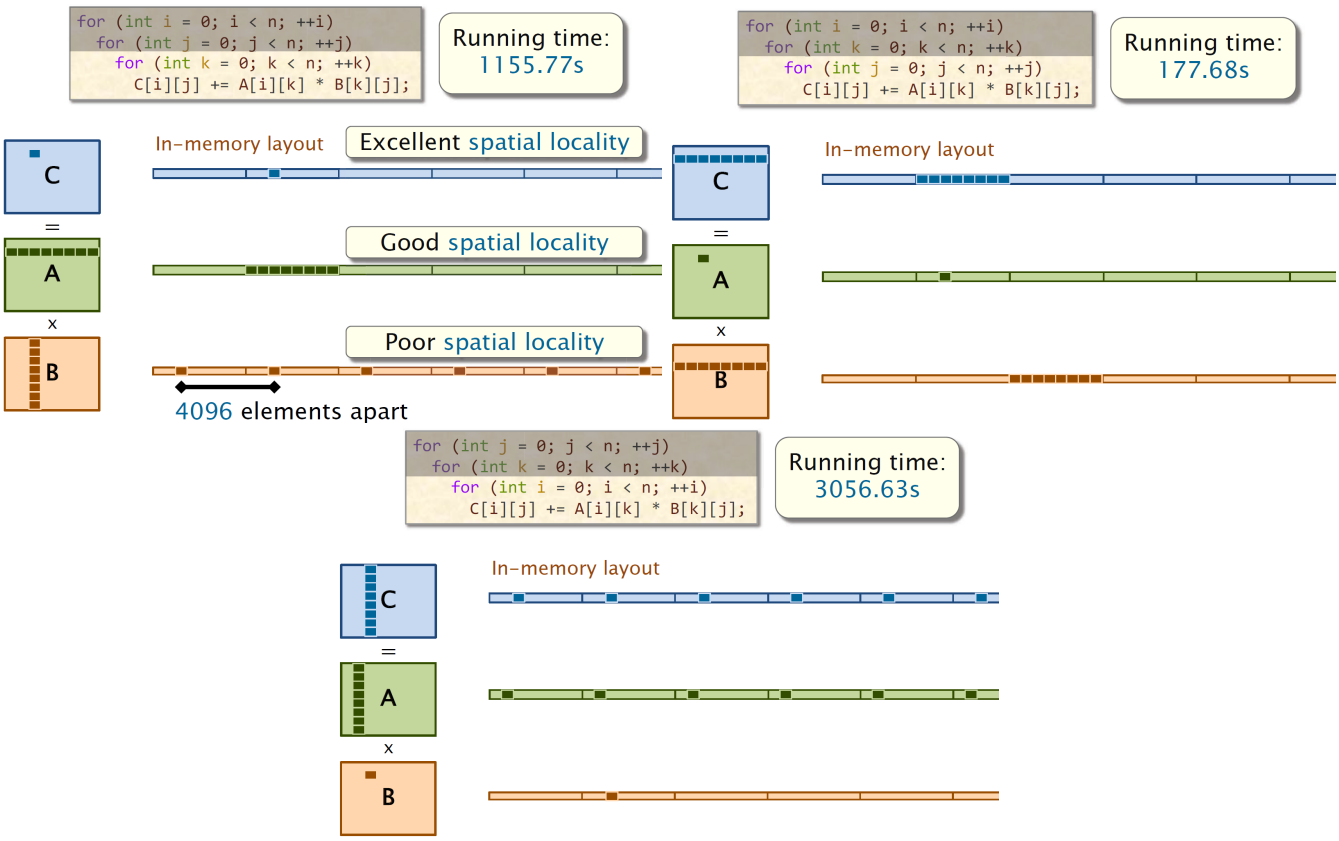


Figure 2.2: Memory layout and spaciality implications

2.3 Compiler Optimization

Clang offers a lot of optimization flags, like `-O3` which enables all the optimizations. The compiler can also unroll loops, which means that it can execute multiple iterations of the loop in parallel. This can be done only if the number of iterations is known at compile time. There are also `-Os` which optimizes for size, and `-Og` which generates debug information. There's plenty of them, for various uses.

Opt. level	Meaning	Time (s)
-O0	Do not optimize	177.54
-O1	Optimize	66.24
-O2	Optimize even more	54.63
-O3	Optimize yet more	55.58

Figure 2.3: Optimization flags and relative performance

2.4 Parallelizing

Even after all these tweaks, we are still using only one of the 9 cores of the CPU. So...

```
cilk_for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        cilk_for (int j = 0; j < n; j++) {
            C[i][j] += A[i][k] +B[k][j];
        }
    }
}
```

We don't have to know what's behind the `cilk_for` keyword, but it will parallelize the `for` loop execution.

But which `for` loops should we parallelize?

Parallelizing all three would cause multiple threads to access the same memory, which would be messy.

A **rule of thumb** is to parallelize the **outermost** loop, which is the one that iterates over the rows of the matrix.

This is demonstrated by the following slide.

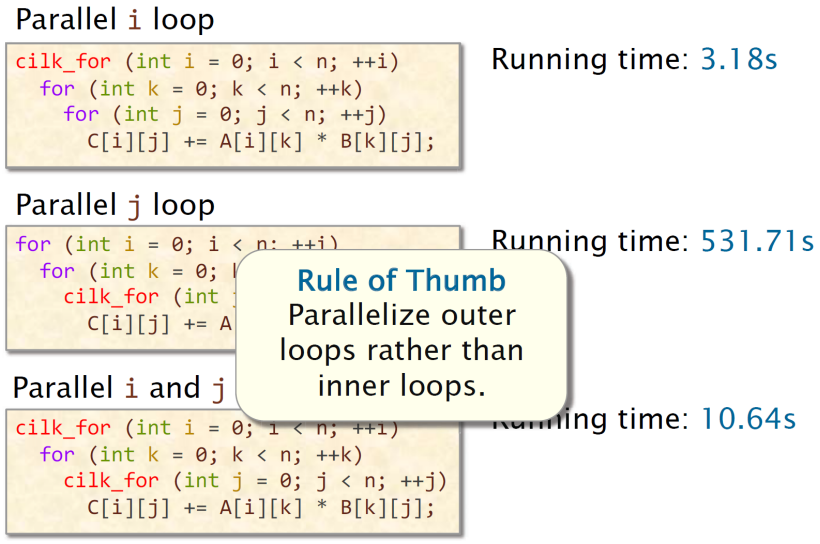


Figure 2.3: Parallelizing only the outermost loop leads to optimal performance

2.5 Tiling

Well, the possible optimizations ain't over ☺. Consider the first picture and let's dig into some math. How many

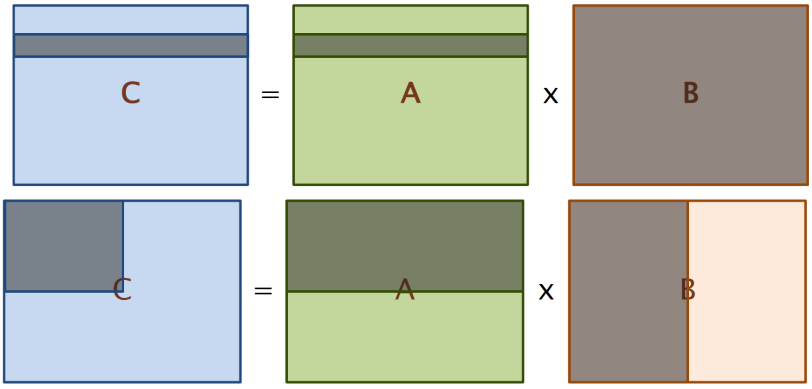


Figure 2.4: Tiling

memory accesses must the looping perform to fully compute 1 row of C?

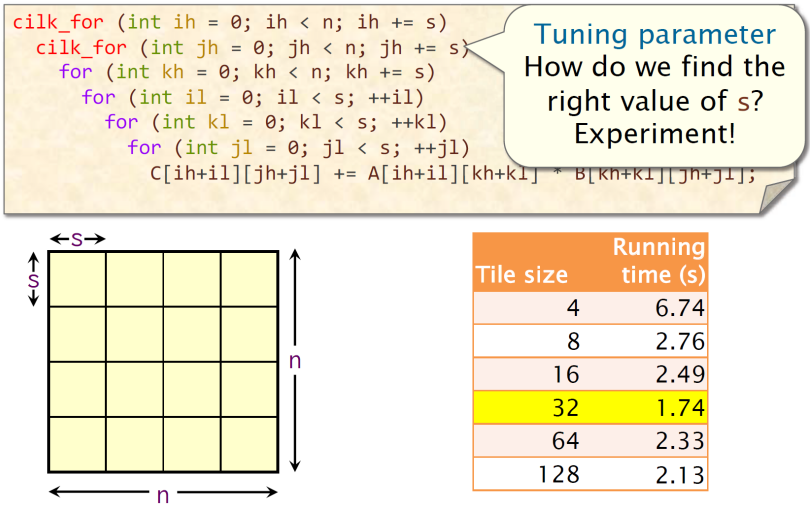
$$4096 \cdot 1 = 4096 \textit{ writes to } C \tag{2.1}$$
$$4096 \cdot 1 = 4096 \textit{ reads from } C \tag{2.2}$$
$$4096 \cdot 4096 = 1.6777216 \cdot 10^6 \textit{ reads from } B \tag{2.3}$$
$$1.6777216 + 4096 + 4096 = 1.6785408 \cdot 10^6 \textit{ total memory accesses} \tag{2.4}$$

But if we consider instead computing a 64×64 block of C we can shrink down the number of memory accesses to half a million:

$$64 \cdot 64 = 4096 \textit{ writes to } C \tag{2.5}$$
$$64 \cdot 4096 = 262144 \textit{ reads from } A \tag{2.6}$$
$$4096 \cdot 64 = 262144 \textit{ reads from } B \tag{2.7}$$
$$262144 + 262144 + 4096 = 528384 \textit{ total memory accesses} \tag{2.8}$$
$$\tag{2.9}$$

But in general, which would the optimal block size be? The only way is to experiment.

Why?



Tile size	Running time (s)
4	6.74
8	2.76
16	2.49
32	1.74
64	2.33
128	2.13

Figure 2.5: Tile size

2.6 Where have we gotten so far - Further optimizations

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184

Implementation	Cache references (millions)	L1-d cache misses (millions)	Last-level cache misses (millions)
Parallel loops	104,090	17,220	8,600
+ tiling	64,690	11,777	416

The tiled implementation performs about 62% fewer cache references and incurs 68% fewer cache misses.

Figure 2.6: Comparison of the various optimizations

2.6.1 Recursion in Tiling

Tiling may be also implemented as a divide-and-conquer algorithm exploiting recursion. This yields slightly better performance, but requires to tune the recursion base case **threshold**. Having a too small threshold would lead to a lot of overhead, due to many function invocations.

2.6.2 Vectorization Flags

There may be also flags to enable instructions specific of a given architecture:

- `-mavx`: Use Intel AVX vector instructions.
- `-mavx2`: Use Intel AVX2 vector instructions.
- `-mfma`: Use fused multiply-add vector instructions.
- `-march=<string>`: Use whatever instructions are available on the specified architecture.
- `-march=native`: Use whatever instructions are available on the architecture of the machine doing compilation. Due to restrictions on floating-point arithmetic, additional flags, such as `-ffast-math`, might be needed for these vectorization flags to have an effect

You could also use `AVX Intrinsic Instructions` that provide access to hardware vector operations. They are available in C and C++. software.intel.com/sites/landingpage/IntrinsicsGuide.
These may help further more, but we are getting very closer to the hardware.

Chapter 3

Parallel Architectures

3.1 Flynn's Taxonomy

There are various classifications possible for parallel architectures, but the most common one is the one based on the **Flynn's Taxonomy**.

This taxonomy is based on the number of **instructions** and **data** streams

- ◇ SISD (Single Instruction, Single Data): the classic Von Neumann architecture, with a single processor executing a single instruction on a single data stream.
- ◇ SIMD (Single Instruction, Multiple Data): a single instruction is broadcasted to multiple processors, each of which operates on a different data stream. This is the architecture of GPUs.
- ◇ MISD (Multiple Instruction, Single Data): multiple processors execute different instructions on the same data stream. This is not common in practice.
- ◇ MIMD (Multiple Instruction, Multiple Data): multiple processors execute different instructions on different data streams. This is the most common architecture for parallel systems.

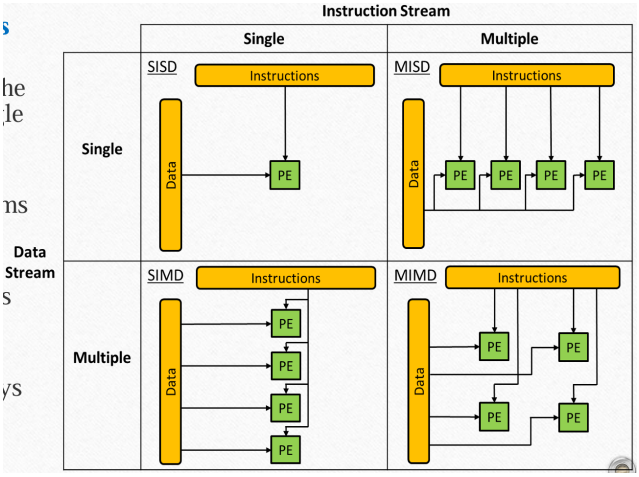


Figure 3.1: Flynn taxonomy

3.1.1 MIMD Architectures

A set of PEs (Processing Elements) simultaneously execute different instructions on different data streams. Each processor can execute all instructions. This is the most common architecture for parallel systems.

This architecture can be further classified considering memory organization and interconnection (between PE and MM) topologies.

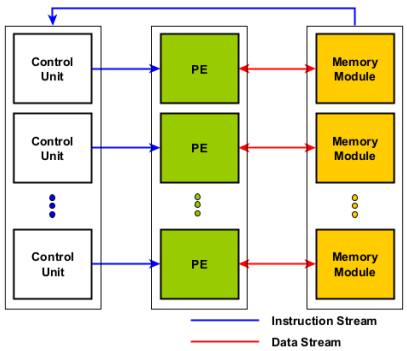


Figure 3.2: MIMD architecture

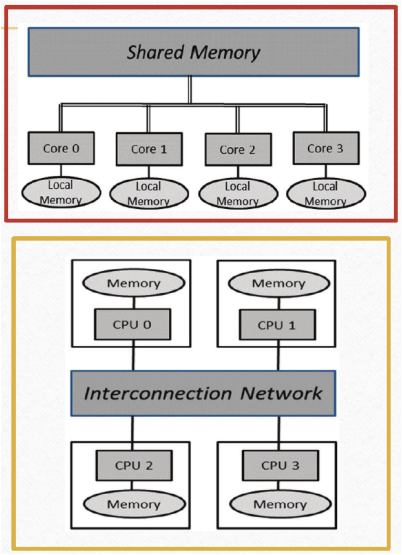


Figure 3.1: Classification based on memory

- **Shared Memory MIMD**: all processors share the same memory space, and can access it directly. This is the most common architecture for multi-core processors (*"multiprocessors"*).
  - **UMA (Uniform Memory Access)**: all processors access memory with the same latency.
    SMP - Symmetric Multiprocessor
  - **NUMA (Non-Uniform Memory Access)**: processors access memory with different latencies.
    Distributed Memory architectures are inherently NUMA
- **Distributed Memory MIMD**: each processor has its own memory space, and communicates with other processors through messages. This is the most common architecture for clusters.
  Interconnection may be based on Ethernet, InfiniBand, etc.
  Historically called *"multicomputers"*

3.2 Classifying on cores count

- $\mathcal{O}(10^1 \div 10^2)$ cores, for a single multiprocessor chip (CMP)
- $\mathcal{O}(10^2 \div 10^3)$ cores, for a Shared Memory tightly-coupled multiprocessor
- $\mathcal{O}(10^3 \div 10^5)$ cores, for a Distributed Memory loosely-coupled multiprocessor (small to large compute clusters)
- $\mathcal{O}(10^5 \div 10^6)$ cores, top supercomputers (e.g. *Leonardo @cineca*)

3.3 Programming Parallel Architectures

3.3.1 Shared-Memory

In *Shared-Memory* systems the emphasis is on the memory organization, meaning the memory hierarchy and processor-memory interconnections, aiming to reduce the *"von Neumann bottleneck"*. Critical aspects are cache coherence, memory consistency and thread synchronization.

Programming models are based on *thread-level parallelism* to exploit the phyisical shared memory by means of **Shared Variables** programming models (e.g. OpenMP, Pthreads, Cilk).

Caches shared among cores are used to reduce the latency of memory accesses, and to exploit the *spatial locality* of data accesses, however they introduce the problem of *cache coherence*, with respective issue (more on this later).

3.3.2 Distributed-Memory

In *Distributed-Memory* systems the emphasis is on the interconnection network, aiming to reduce the *"communication bottleneck"*, so reducing latency and increasing bandwidth. Critical aspects are messaging protocols/libraries, routing.

Here the only parallelism exploitable is *process-level parallelism*, so the programming models are based on **message-passing** (e.g. MPI, POSIX socket, PVM).

Nowdays, each node is a CMP. Depending on the network and some other aspects, we may further classify these systems in Clusters, Cloud, geographical distributed systems, etc.

We are interested in systems with high performance network topologies and homogeneous nodes, like compute clusters.

A common example of application is the **Stencil Computation**, which is a common pattern in scientific computing, where each element of a matrix is computed as a function of its neighbors.

3.3.3 Summary

Distributed Memory systems are more scalable, costly and less energy efficient.

From the programming perspective, Shared Memory systems are easier to program and the physical shared memory can be used for fast communication between threads. However, locking and synchronization are critical points deserving

attention.

For what concerns Distributed Memory systems, the most important aspect is to reduce as much as possible the cost of communication (i.e. I/O), for instance overlapping computation and communication, reducing memory copies for I/O, using fast messaging protocols (e.g. RDMA).

3.4 Suggested Readings

- ◇ Chapter 1 - Section 1.2 - “Parallelism Basics” of Parallel Programming Concept and Practice book
- ◇ Chapter 3 - Section 3.2 - “Flynn’s Taxonomy” of Parallel Programming Concept and Practice book

Chapter 4

Shared Memory Architectures

Shared Memory Architectures are a type of MIMD architecture where all processors share the same memory space, and can access it directly. This is the most common architecture for multi-core processors (*"multiprocessors"*).

They mirror the Von Neumann architecture, with multiple processors sharing the same memory space.

4.1 Von Neumann Bottleneck

Definition 4.1 (von Neumann Bottleneck) *The von Neumann bottleneck is a limitation on throughput caused by the standard personal computer architecture. The term is named for John von Neumann, who is credited with developing the von Neumann architecture, in which programs and data are stored in the same memory. The bottleneck refers to the limited data transfer rate between a computer's CPU and memory compared to the amount of memory.*

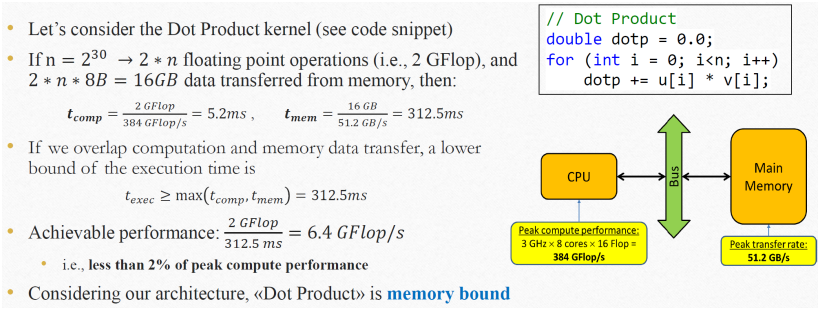


Figure 4.1: von Neumann Bottleneck example

In the example in Figure 4.1, the CPU is waiting for the data to be loaded from memory, which is a slow operation, leading to exploiting only the 2% of the CPU capabilities.

4.1.1 Caches

Back in the day, the solution was to *"move the data closer to the CPU"*, introducing **memory hierarchy** and **caches**.

Usally, L1 and L2 are private to each core, while L3 is shared among all cores.

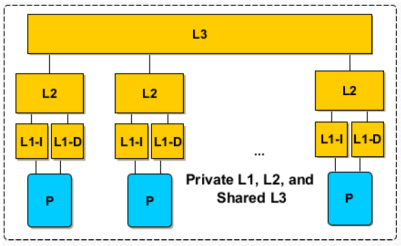


Figure 4.3: Caches hierarchy

If a matrix of the previous example in Fig. 4.1 is entirely stored in cache, then the achievable performance is 223 GFLOPS, which is about 60% of the peak power.

If the matrices do not fit in the cache, the performance drops since we need to trap to the main memory to fetch

missing data.
So, the problem shifts to understand how to exploit the cache to its fullest.

4.2 Locality Principle

The locality principle is the driving force that makes the cache work. Locality increases the probability of reusing data blocks that were previously moved from level n to level $n-1$.

- ◇ **Temporal Locality**: if a data is accessed, it is likely to be accessed again soon.
  Cache mapping strategy (Direct/associative) and the replacement policy (LRU, FIFO, Random etc.) are crucial to exploit temporal locality.
- ◇ **Spatial Locality**: if a data is accessed, it is likely that nearby data will be accessed soon.

4.2.1 Measuring CPU time with caches

$$CPU_{time} = ClockCycles \cdot ClockCycleTime = IC \cdot ClockCycleTime \tag{4.1}$$

- ◇ IC: Instruction Count (number of instructions executed)
  - $IC = IC_{CPU} + IC_{MEM}$
- ◇ CPI: Cycles Per Instruction
  - $CPI = \dfrac{ClockCycles}{IC}$
  - $CPI = \dfrac{IC_{CPU}}{IC} \cdot CPI_{CPU} + \dfrac{IC_{MEM}}{IC} \cdot CPI_{MEM}$ where CPI_{CPU} are the average cycles per ALU instruction and CPI_{MEM} are the average cycles per memory instruction.
  - Considering that each memory instruction may generate a cache hit or miss with a given probability, and naming $HitRate$ the probability of a cache hit, we can write

$$CPI_{MEM} = HitRate \cdot CPI_{MEM-Hit} + (1 - HitRate) \cdot CPI_{MEM-Miss} \tag{4.2}$$

4.2.2 Cache Algorithms

1. *What do we load from main memory?*
2. *Where do we store it in the cache?*
3. *Cache is full, what should we evict?*

At the beginning of the second half of the 4^{th} lecture, the professor displays how to VPN in the unipi and then ssh to the servers.

4.2.3 Cache mapping and eviction strategies

- ◇ **Direct-mapped** cache: each memory block can be placed in only one cache line.
- ◇ **n-way** set associative cache: each memory block can be placed in n cache lines.
- ◇ **Least Recently Used** (LRU) cache: the block that has been accessed the least recently is evicted.

Transposing Matrices

Suppose you have to multiply matrix A and B. If you access A as rows, you'll access B as columns.
Since matrices are stored in row-major order, you won't exploit spatial locality on B, and for each element of A you'll have a cache miss on B, creating the need to evict a line and load another one in cache.
Transposing B would solve the problem, since you would access B as rows, exploiting spatial locality.

Prof. Torquati displayed that transposing and then multiplying the matrices would lead to a 2x speedup.

17s vs 37s

4.2.4 Cache Write Policies

Data in cache may be inconsistent with the value in memory, leading to the need to write back the data to memory. There are two policies:

- **Write-through**: data is written to both cache and memory. It is simple but slow.
- **Write-back**: data is written only to the cache, and then to memory when the block is evicted. It is faster but more complex.
  - Caches mark data in the cache as dirty (Dirty bit)
  - When a dirty line is evicted, it is written in main memory
  - A store write buffer is generally used to reduce the cost of cache writes

4.2.5 Cache Coherence

With private caches per core, it is possible to have several copies of shared data in distinct caches, each cache stores a different value for a single address location.

Definition 4.2 (Cache incosistency) *Two caches store different values for the same variable.*

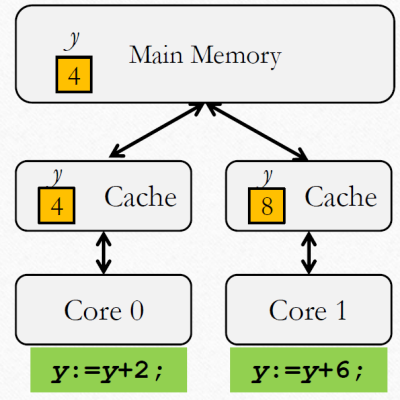


Figure 4.4: Cache incosistency problem

Hardware firmware automatically solves this problem, but it is important to know that it exists. The algorithms responsible for this are called **cache coherence protocols**, a famous one is the **MESI** protocol, but we ain't going to study it. Note however that the cache coherence protocol granularity is the *cache line*, not the *single variable*.

4.3 Suggested readings

- Chapter 3.1 of the Parallel Programming Concept and Practice book
- Cache-coherence protocols: "A Primer on Memory Consistency and Cache Coherence" Daniel J. Sorin, Mark D. Hill, and David A. Wood

4.4 Advanced Processors and Technologies

4.4.1 Superscalar CPUs

Superscalar CPUs are designed to execute multiple instructions from a single process/thread simultaneously to improve performance and CPU utilization

The processor fetches multiple instructions concurrently in a single clock cycle and executes them out-of-order, exploiting instruction-level parallelism, Results are then re-ordered to ensure they are written back to the register file or memory in the correct program order

However, in sequential programs, the number of instructions that are independent are small thus, the exploited parallelism is low. To overcome this limitation, **SMT** (*Simultaneous Multi-Threading*) has been added in superscalar processors to execute multiple instructions from multiple threads of control simultaneously.

Hyperthreading is Intel's implementation of SMT

4.4.2 HW Multithreading

HW multithreading enablesa single core to execute multiple threads concurrently. There are two main types of HW multithreading:

- **Fine-grained multithreading**: the processor switches between threads at each cycle (instruction level).

- ◇ **Coarse-grained multithreading**: the processor switches between threads only when the thread in execution causes a stall.

Each thread has its own set of registers and program counter; the processor maintains the context of each thread to quickly switch between them. However, they share the same cache and execution units. For the OS, each context is seen as a logical core.

4.5 Programming Shared Memory Systems

4.5.1 Threads are the way to go

Thread creation is more lightweight and faster (from 3x to 5x) than process creation[1], and threads share the same memory space, so they can communicate easily. Creating a thread takes $\mathcal{O}(10^4)$ cycles in C/C++.

4.5.2 Data-race

Definition 4.3 (Data Race) *Scenario that occurs when two threads access a shared variable simultaneously and at least one of the accesses is a write, and the accesses are not guarded by a synchronization operation.*

[1] A `fork` system call requires copying (`memcpy`) more data, e.g. page table

DRs produce non-deterministic results, and are hard to debug, since they depend on the thread scheduling. To avoid this issue, we may use *mutexes, condition variables, semaphores, atomic operations, etc.*

We will discuss all four, except *semaphores*, which are more common in processes synchronization.

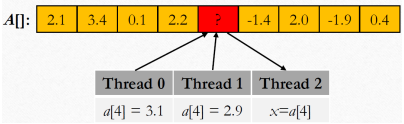


Figure 4.5: Data Race example

4.5.3 False Sharing

Caches are organized in cache lines, and cache coherence is managed at the cache line granularity.

Definition 4.4 (False Sharing) *Scenario that occurs when two threads access different variables that reside on the same cache line. The cache coherence protocol will invalidate the cache line, so actually the two threads won't share anything.*

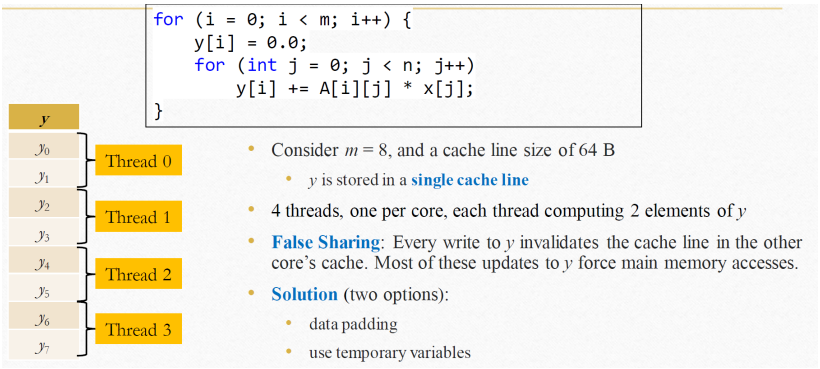


Figure 4.2: False Sharing slide
Different threads access different variables which however reside on the same cache line.

When False Sharing occurs, the performance of the system is degraded, since the cache line is invalidated and the cache coherence protocol keeps the cache line consistent among all copies.

Looking at Fig. 4.2, suppose that T_1 accesses y_2: the entire y cache line is —marked as?— **invalidated**, so when T_1 accesses y_1, it will have a cache miss, and the cache line will be ~~reloaded from memory~~ "moved from one core to another"[2]

.

[2] Prof. Torquati said this, not sure if saying it gets "reloaded from mem" is correct

4.5.3.1 Padding

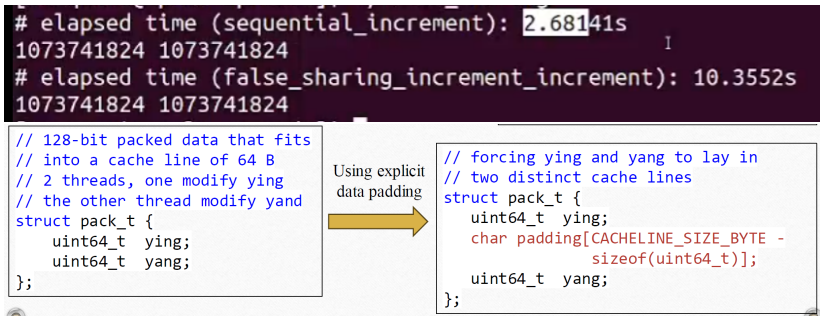


Figure 4.3: False Sharing demo exec demo

Prof. Torquati displayed that the exec time of a dummy program which increments two distinct variables in a struct went from 2.6s to 10.3s, when going from sequential to parallel execution. The overhead is due to false sharing.

Fig. 4.3 also displays a possible solution to the problem: **padding** the struct with a dummy variable, so that the two variables are placed on different cache lines. Even though looks a bit hardcoded, it works indeed! Exec time, went from 2.6s to 2.9s, basically the same time, since the two variables are now on different cache lines; there is only some overhead due to the threads creation.

How can i understand if false sharing is happening in a complex program?

Test.

4.5.3.2 Local variables

Actually the better solution would be to use **local variables**, since they are stored in registers, and each thread has its own stack, so there is no sharing at all.

```
for (i = 0; i < m; i++) {
    float _y = 0.0;
    for (int j = 0; j < n; j++)
        _y += A[i][j] * x[j];
    y[i]= _y;
}
```

Figure 4.4: Local variable stored in registers

Compiler optimization

Modern compilers are able to optimize the code and avoid false sharing. If the example in Fig. 4.3 is compiled with `g++ -O3` (instead of `-O0`), the exec time goes back to normal.

However, it is not always possible to rely on the compiler, and in fact many times it does not work. So, it is better to pad the struct or use local variables, or avoid false sharing in general.

4.6 SIMD and Vectorization

SIMD - *Single Instruction, Multiple Data.* A parallel computing model that exploits data-level parallelism. There are two limitations which must be kept in mind:

1. ALUs are *limited in number*, so the number of operations that can be executed in parallel is limited.
2. All ALUs must execute the *same instruction*.

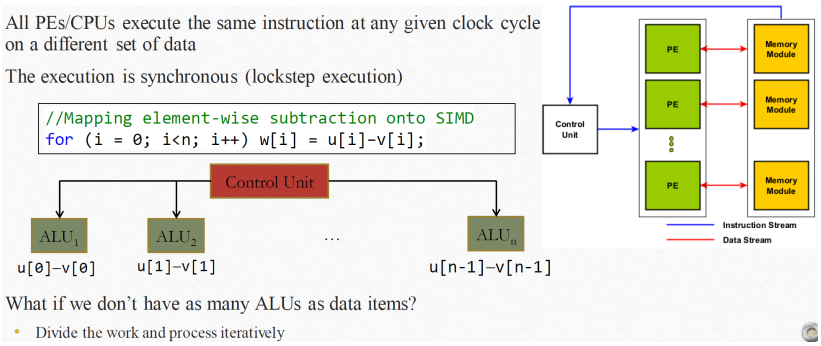


Figure 4.4: SIMD and ALUs

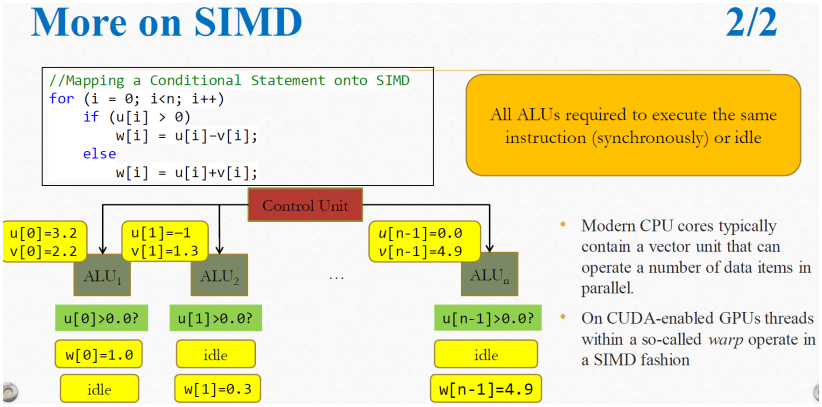


Figure 4.5: Here the ALUs either execute an instruction —another ALU is performing— or stay **idle**.

4.6.1 AVX registers

Listing 4.1: Intrinsics

```
//AVX-Programming with C/C++ Intrinsics
__m256 a, b, c; // declare AVX registers
... // initialize a and b
c = _mm256_add_ps(a, b); // c[0:8] = a[0:8] + b[0:8]
// or c = _mm512_add_pd(a, b); c[0:8] = a[0:8] + b[0:8]
```

Intrinsics are assembly-coded functions that can be used in C/C++ to exploit SIMD parallelism. They are provide a light abstraction from assembly. Exploiting intrinsics as in Lst. 4.1 may lead to a 8x speedup, they are very powerful.

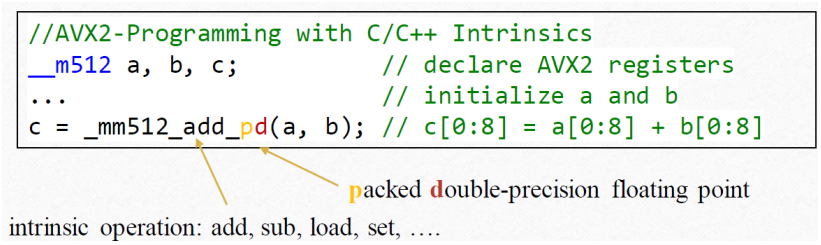


Figure 4.6: Intrinsics explained