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 (**CMPs**).

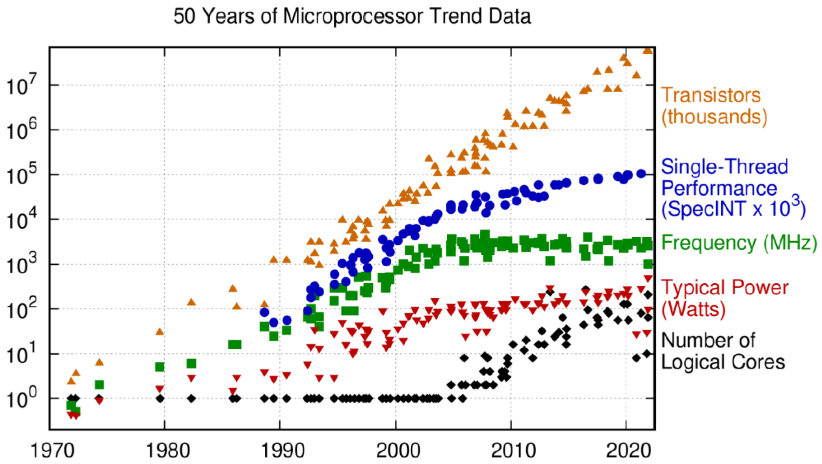

Figure 1.1: Microprocessors in the last 30 years
Single thread performance is increasing slowly, while the Frequency is stable. Moore's law is still valid if we account the number of cores.

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];
      }
   }
}
```

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

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

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.

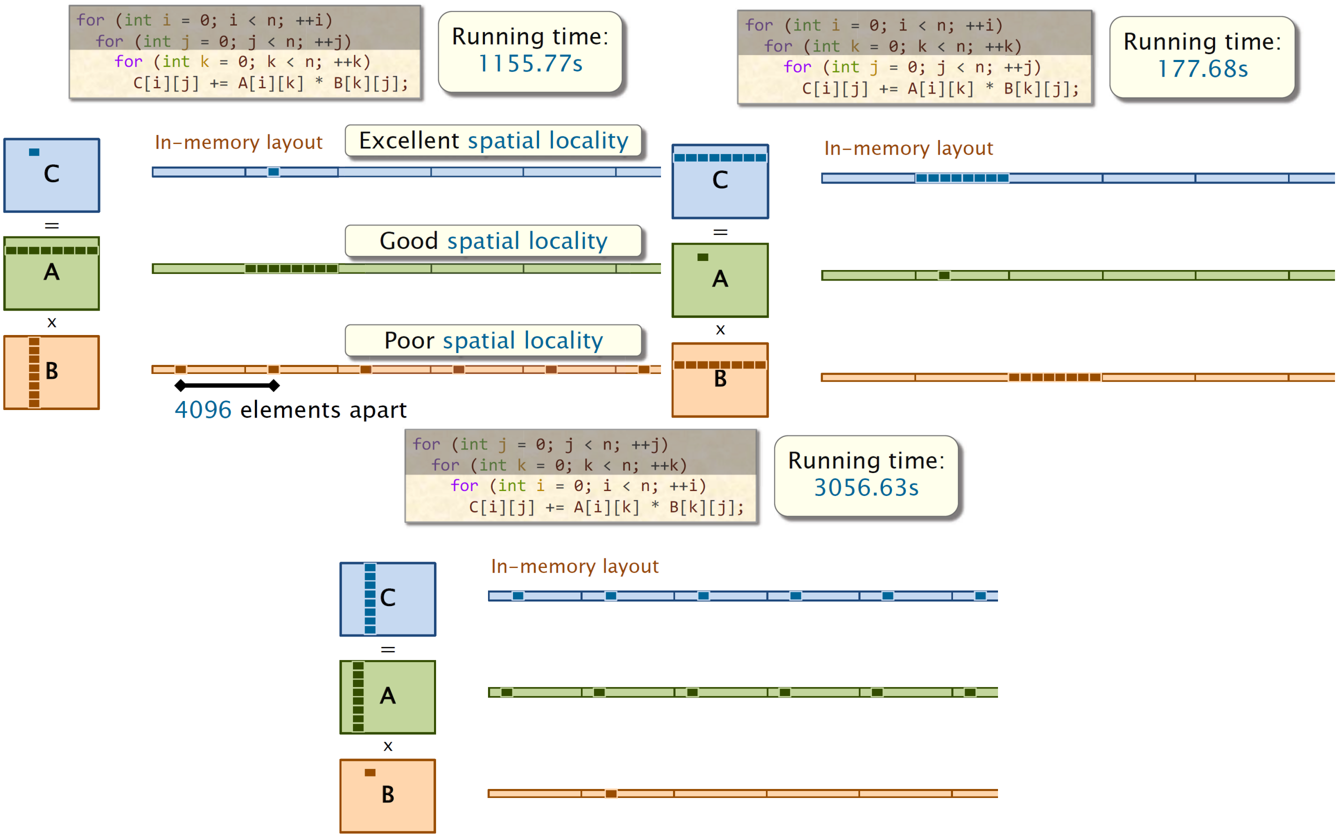

Figure 2.2: Memory layout and spaciality implications

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.

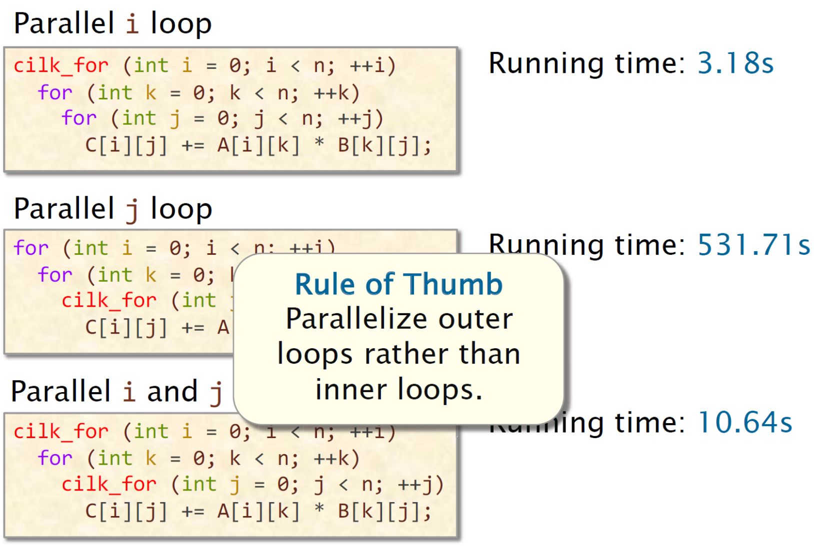

Figure 2.3: Parallelizing only the outermost loop leads to optimal performance