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).

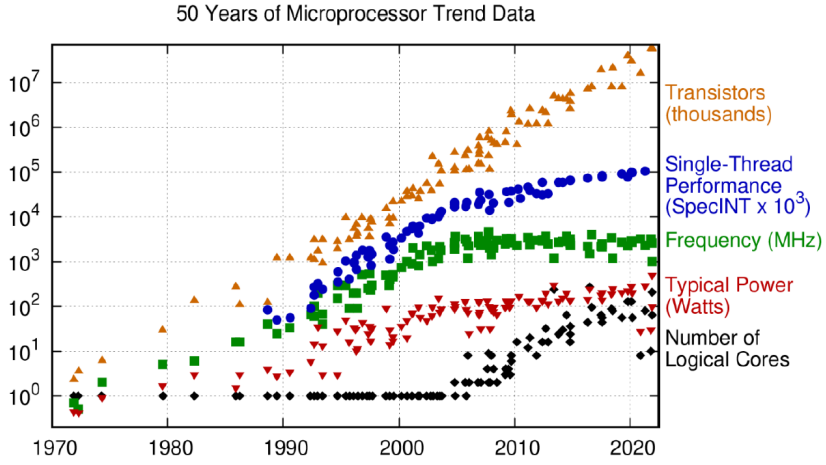


Figure 1.1: Microprocessors in the last 30 years
Single thread performance is increasing slowly, while the Frequency is stable. Moore's law is still valid if we account the number of cores.

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];
         }
      }
      }
```

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%

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

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.

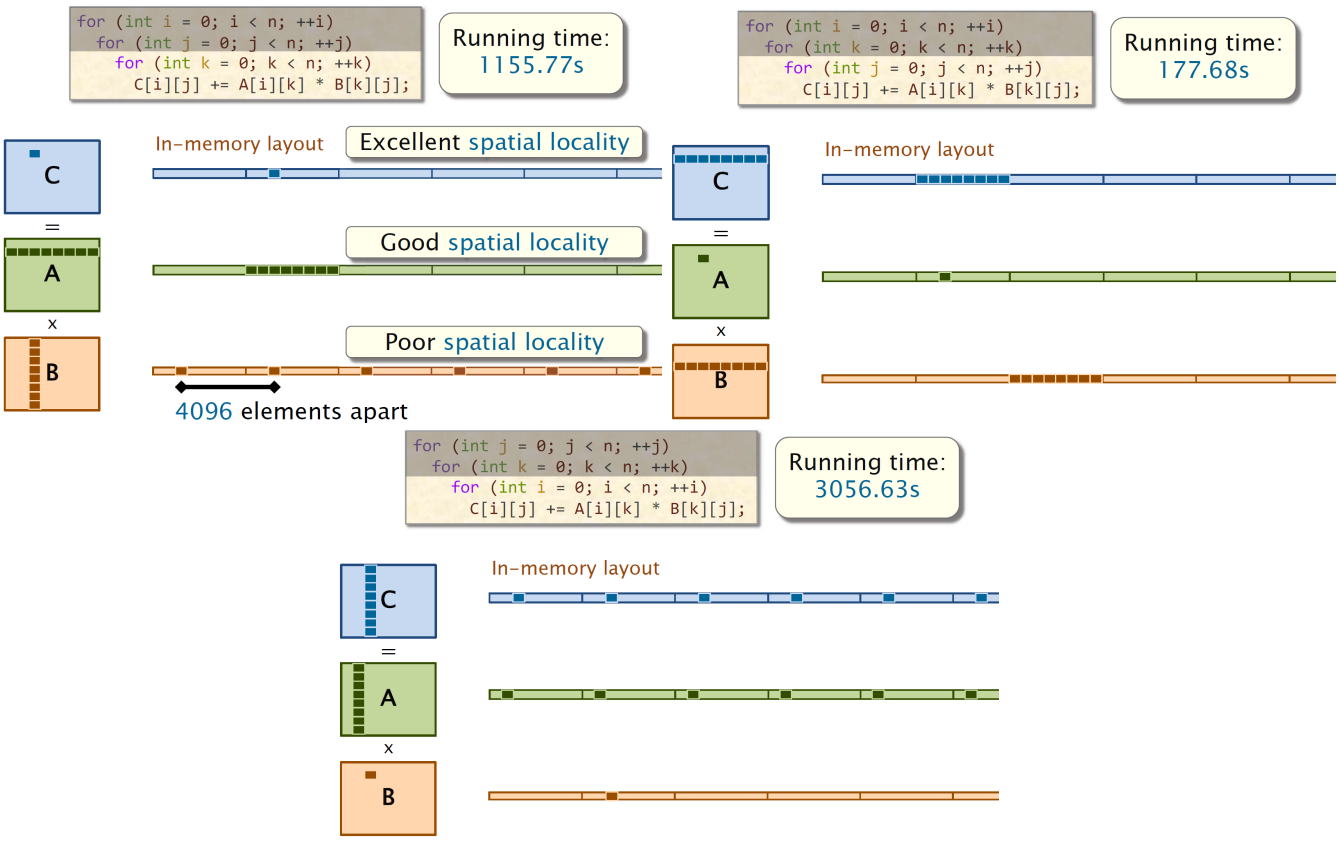


Figure 2.2: Memory layout and spaciality implications

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.

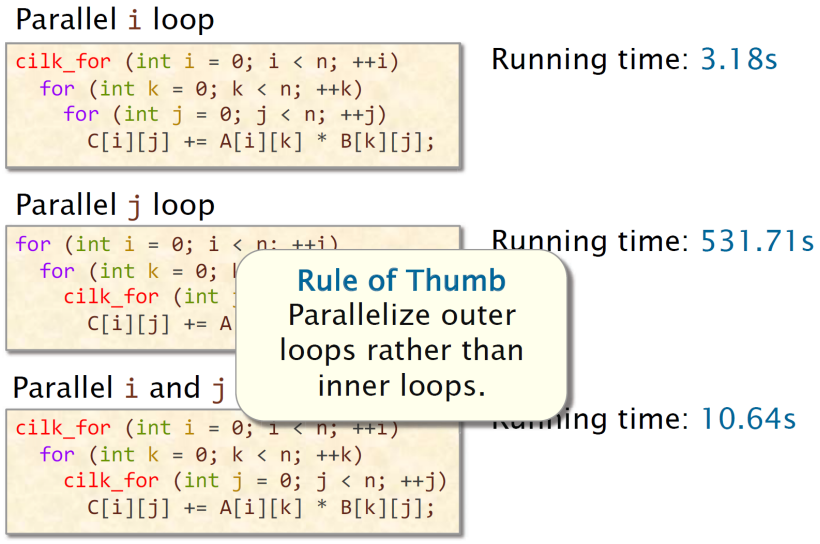


Figure 2.3: Parallelizing only the outermost loop leads to optimal performance

2.5 Tiling

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

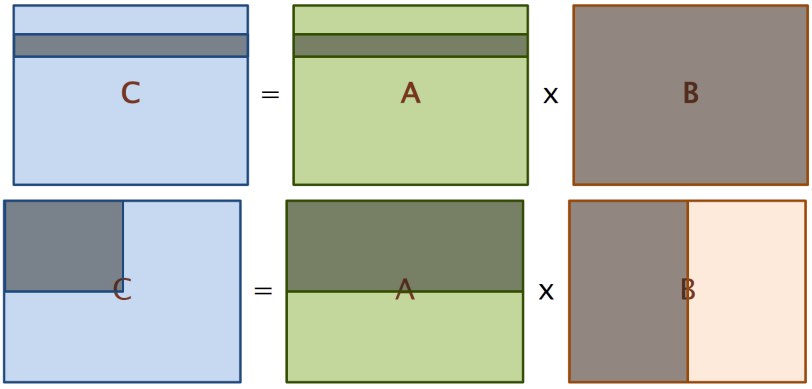


Figure 2.4: Tiling

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

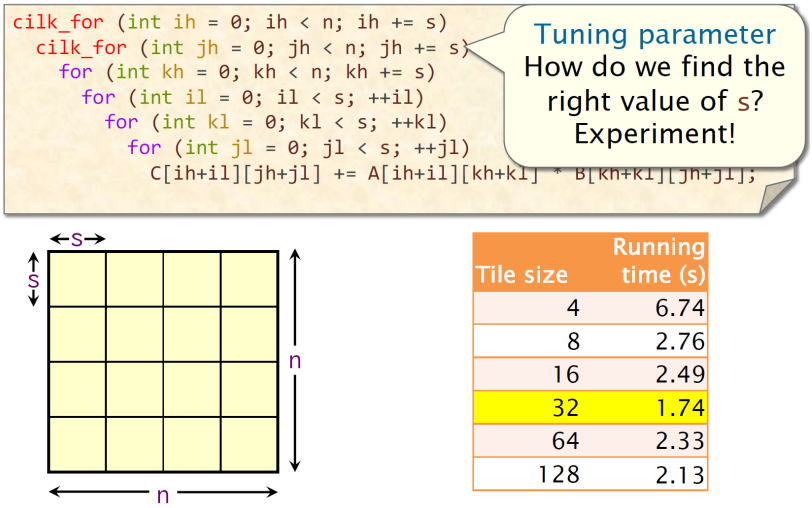
$$4096 \cdot 1 = 4096 \textit{ writes to } C \tag{2.1}$$
$$4096 \cdot 1 = 4096 \textit{ reads from } C \tag{2.2}$$
$$4096 \cdot 4096 = 1.6777216 \cdot 10^6 \textit{ reads from } B \tag{2.3}$$
$$1.6777216 + 4096 + 4096 = 1.6785408 \cdot 10^6 \textit{ total memory accesses} \tag{2.4}$$

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.

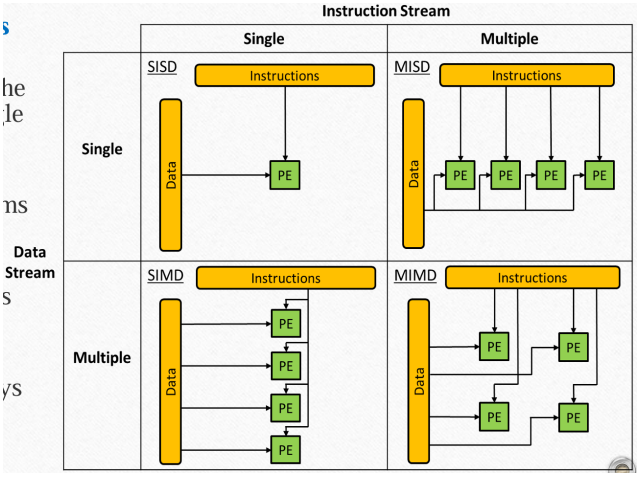


Figure 3.1: Flynn taxonomy

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.

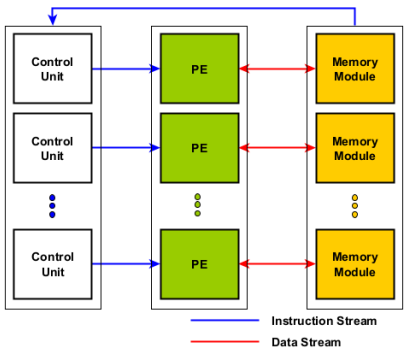


Figure 3.2: MIMD architecture

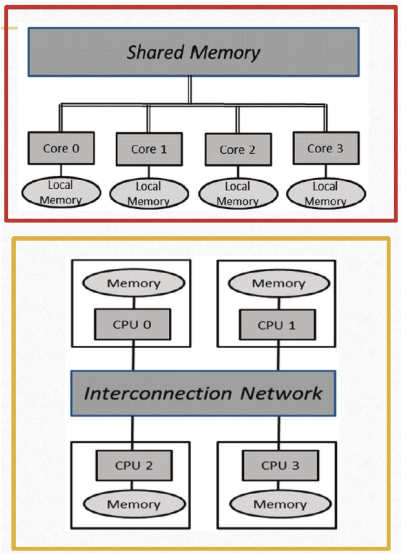


Figure 3.1: Classification based on memory

- **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.*

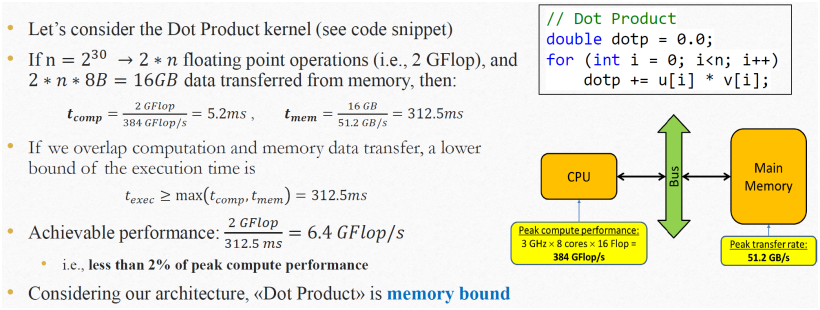


Figure 4.1: von Neumann Bottleneck example

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.

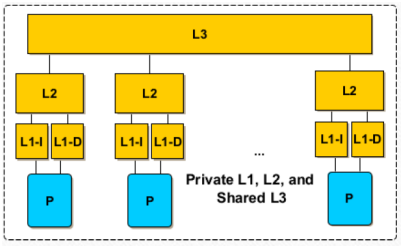


Figure 4.3: Caches hierarchy

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.*

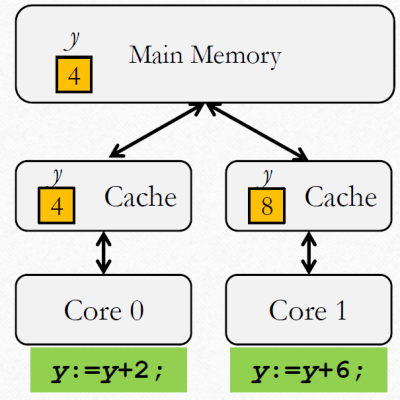


Figure 4.4: Cache incosistency problem

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.

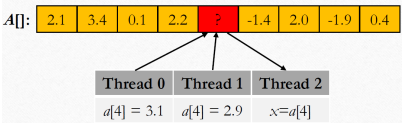


Figure 4.5: Data Race example

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.*

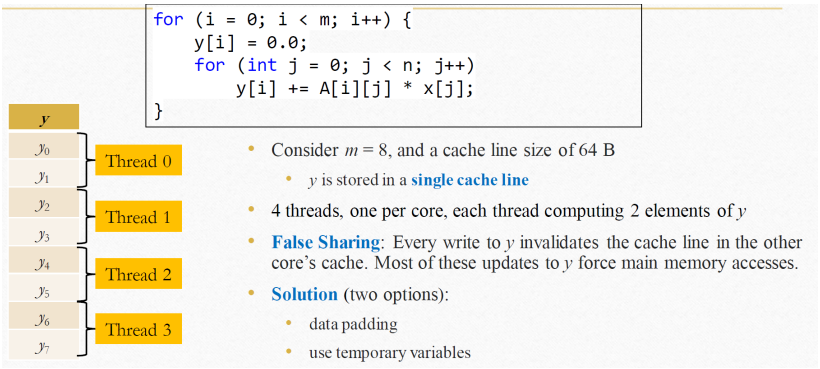


Figure 4.2: False Sharing slide
Different threads access different variables which however reside on the same cache line.

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

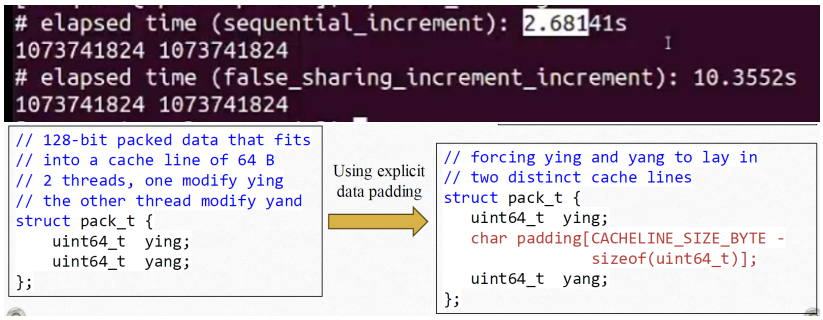


Figure 4.3: False Sharing demo exec demo

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*.

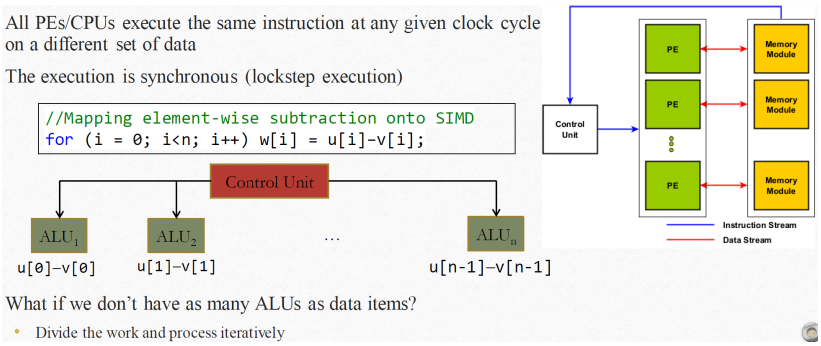


Figure 4.4: SIMD and ALUs

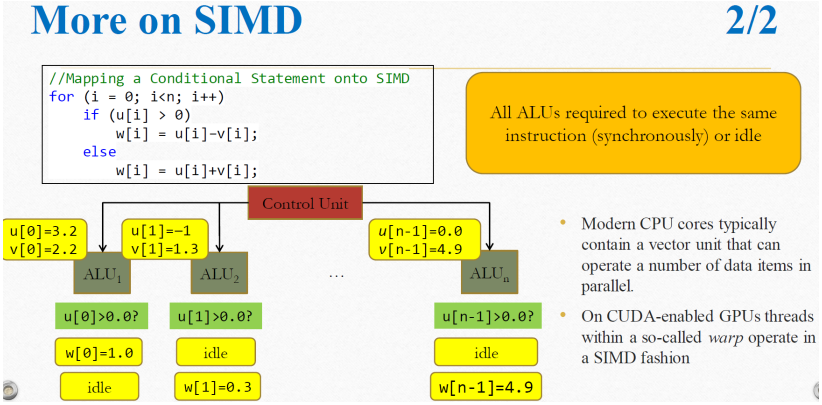


Figure 4.5: Here the ALUs either execute an instruction —another ALU is performing— or stay **idle**.

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.

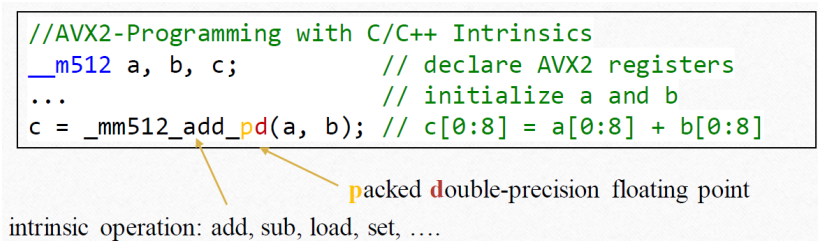


Figure 4.6: Intrinsics explained

There are further notions on intrinsics and vectorization i did not report here

4.7 Vectorizing code

4.7.1 AoS vs SoA

Figure 4.7: AoS vs SoA

- ◇ **AoS** - *Array of Structures*: stores records consecutively in a single array.
  Copilot: "each element of the array is a struct, and the struct contains all the data."
- ◇ **SoA** - *Structure of Arrays*: uses one array per dimension. Each array stores only the values of the associated element dimension
  Copilot:"each element of the array is a single data type, and the data is stored in different arrays."

Typically AoS does not allow for vectorization, since the data is not contiguous in memory, while SoA does, since the data is contiguous, allowing to perform operations on up to `#registers` triplets of data.

4.7.2 Loops - Compiler Optimization

The compiler is able to optimize the code and vectorize it, but it is not always possible. In particular, there are some guidelines concerning loops to allow the compiler to vectorize the code:

- ◇ The loop count must be known at entry to the loop.
- ◇ Conversely, the exit condition must not be data-dependent.
  - `break,goto,switch,return` statements are forbidden
  - `if` is allowed and may be vectorized if it can be implemented as a masked assignment
- ◇ No library functions calls. If a function call may be inlined it is OK.
  What does this mean?
- ◇ No read/write dependencies between iterations. e.g. the following code is *not* vectorizable:

```
A[0] = 0;for(int_i=1; i<N; ++i) A[i]= A[i-1] + 1;
```

- ◇ No pointer aliasing, i.e. no two pointers can point to the same memory location.
  - The type qualifier `_restrict_` tells the compiler that the pointer is the only pointer that points to that memory location.

Chapter 5

Distributed Memory Systems

The first issue to address is the communication between different nodes. The main problem is that the memory is distributed, so we need to find a way to communicate between different nodes.

5.1 Interconnection Networks

Interconnection networks for parallel systems share many technical features of WAN, but they have very different requirements:

- $\mathcal{O}(10^1 \div 10^5)$ nodes
- Distances ranging in $\mathcal{O}(10^0 \div 10^1)$ meters

Key metrics for us are:

- **Latency** - time lapse between the instant a packet starts to be transmitted and the instant it is —entirely— received
  - **No-load** (or *Zero-load*) latency: the latency experienced by a packet when there is no other traffic on the network
  - **Under-load** latency: the latency experienced by a packet when there is other traffic on the network, but below the *saturation point*
    Saturation point is a crucial threshold we will discuss in future.
- (Offered) **Throughput** - the amount of data that transmitted over the network in a given time. The **Saturation Throughput** is the maximum amount of traffic sustained by the network, it is the point at which the network is fully loaded.
- **Bandwidth** - theoretical maximum data transfer rate under ideal conditions on a network path.

Two aspects which are fundamental but which we won't address are:

- **Routing**
- **Flow control**

5.1.1 Terminology

Endpoints, links, switches...

- **Direct Network** aka *static* - nodes are both endpoints and switches
- **Indirect Network** aka *dynamic* - endpoints are connected indirectly through switches

- **Degree** - number of maximum neighbors a node can have
- **Diameter** - longest of shortest paths between any two nodes
  maximum (minimum) distance between two nodes
- **Bisection width** - minimum number of links that must be removed to split the network into two equal halves

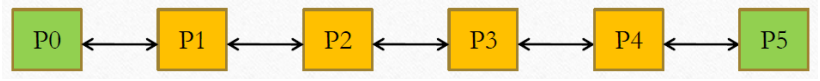


Figure 5.1: Linear Array topology
$degree = 2 \quad Diameter = n - 1 \quad bw = 1$

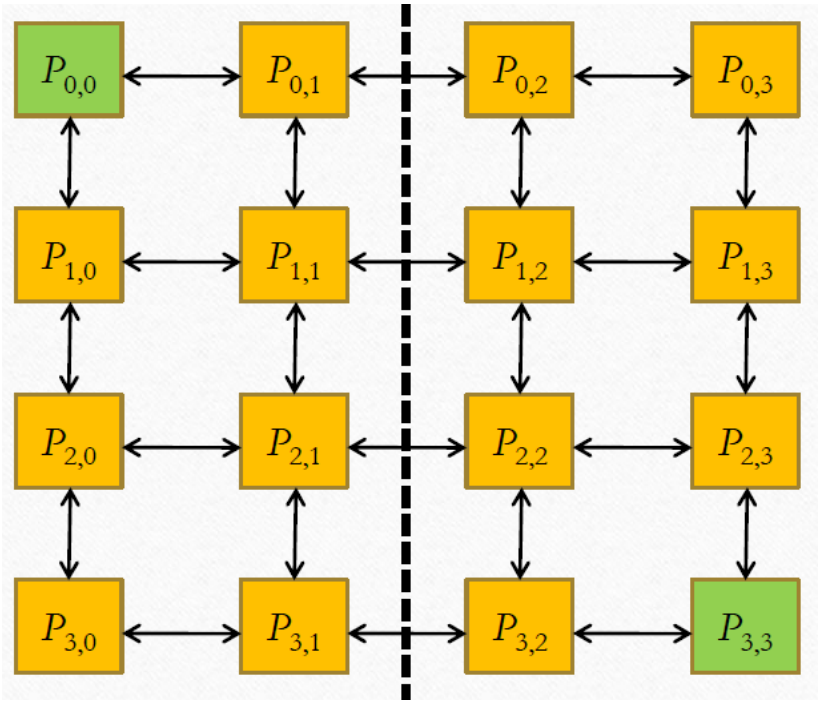


Figure 5.2: 2D Mesh topology
$M(d,d)$ has $n = d^2$ endpoints $degree = 4 \quad Diameter = 2(d-1) = \mathcal{O}(\sqrt{n}) \quad bw = d = \mathcal{O}(\sqrt{n})$

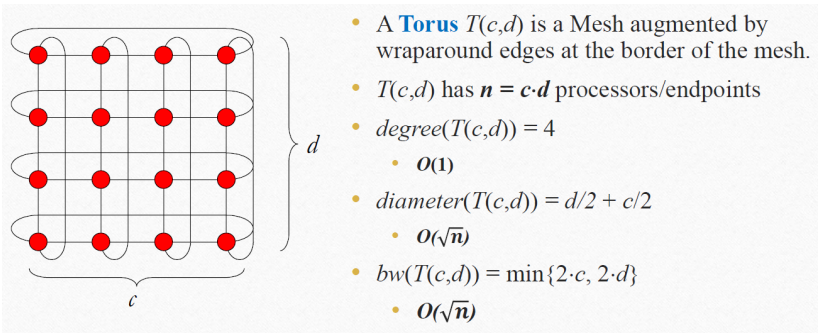


Figure 5.3: Torus topology

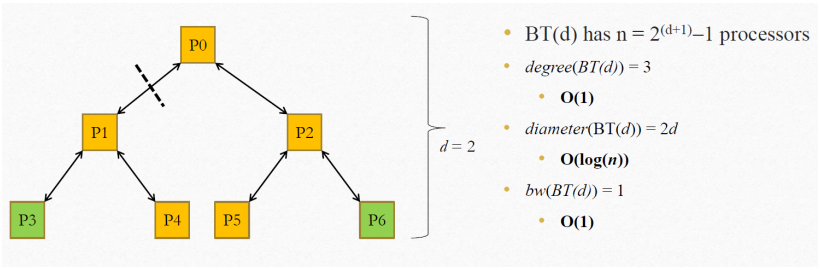


Figure 5.4: Binary Tree topology

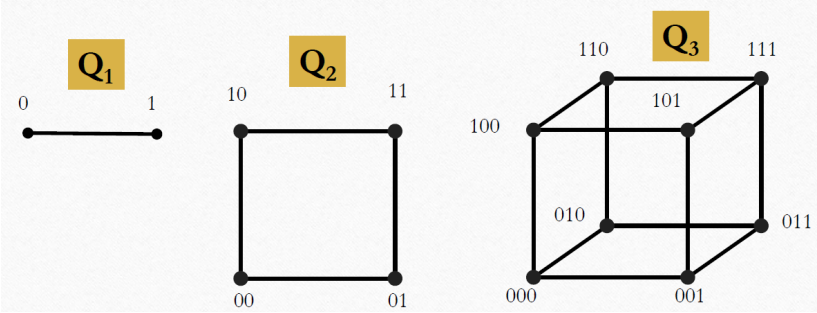


Figure 5.5: Hypercube topology

5.1.2 Examples

5.1.2.1 Hypercube

The hypercube $\mathcal{Q}_d (d \geq 1)$ is the graph that has vertices representing the 2^d binary strings of length d. Two vertices are adjacent if their binary strings differ in *exactly* one bit.

The hypercube is a very interesting topology because it has a very low diameter and bisection width, but it is very expensive to build.

5.1.3 Criteria for Evaluation - Summary

So, what do we want from an interconnection network?

- ◇ *Constant* **degree** - Allows scaling the network without increasing the degree of the nodes
- ◇ *Low* **Diameter** - Allows for low latency
- ◇ *High* **bisection width** - Allows for high bandwidth, but may require to have dynamic degree.

5.1.3.1 Simple Communication cost model

$$T_{comm} = t_0 + n \times s \approx \begin{cases} t_0 & \textit{for small } n \\ n \times s & \textit{for large } n \end{cases} \tag{5.1}$$

- ◇ t_0 start-up time (network and communication runtime setup)
  - – Includes all costs for sending the shortest message. (*latency*)
  - – Its value may be different for different programming models
- ◇ n amount of (e.g. bytes) data to be transferred
- ◇ s transmission cost. Usually $s = \frac{1}{B}$ with B available bandwidth on the path
  - – s il limited by the slowest part of the path between the sender and the receiver processes
  - – s includes both SW agents (e.g. app max output rate) and HW agents (e.g. network max throughput)

5.1.4 More advanced topologies

- ◇ **Fat-tree** - a tree-like topology that has a high bisection width
- ◇ **Dragonfly** - a topology that combines a tree-like structure with a mesh-like structure
- ◇ **Jellyfish** - a topology that is based on a random graph

5.1.4.1 Fat-tree

All leaves are endpoints and all internal nodes are switches. Links higher in the tree have higher bandwidth. The idea is to keep the bandwidth constant across the tree.

The key issue is the cost, which increases with the depth of the tree. To overcome this problem, we can use a **Spine-Leaf** topology, where the tree is split into two parts: the spine and the leaves.

In the slides this is considered as a different implementation of the fat-tree, but it is actually a different topology.

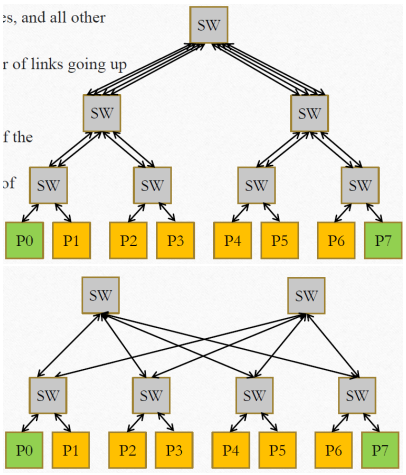


Figure 5.6: Fat tree and Spine Leaf

5.1.4.2 Dragonfly

This is the most used network used in supercomputers. It is divided in three levels:

1. Router
2. Group
3. System

Each router has connections to p endpoints, $a-1$ local channels (to other routers in the same group) and h global channels (to routers in other groups).

A group consists of a routers and group has ap connections to endpoints, and ah connections to other groups.

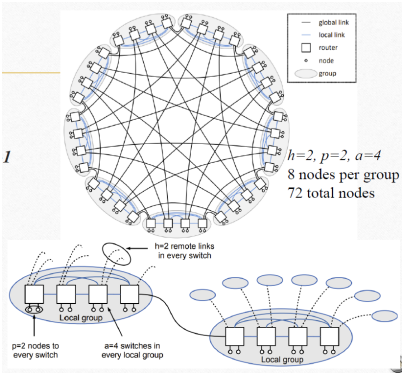


Figure 5.7: Dragonfly

5.2 Computation-to-Communication Overlap

It is always udeful for the sender to execute a **non-blocking send** to the destination process: the NICE executes the data transfer in the background, and then notifies the sender process when the tranmission is completed, so that the sender can continue with its computation. This creates a partial or full *overlap* between computation and communication.

How can we achieve this overlap?

5.2.1 Syncronous vs Asynchronous Communication

The **asynchrony degree** of a channel is the maximum number of messages that can be sent on the channel without waiting for the receiver to start receiving data. Essentially, it is the length of the queue of messages that can be sent on the channel.

- **Synchronous** - the operation comples only after the message has been received/sent. i.e. a peer **blocks** until the other peer completes the operation.
- **Asynchronous** - the communication operation returns immediately, without waiting for the message to be effectively sent/received.
  - The completion/success will be tested later on
  - The number of messages that can be sent depends on the **asynchrony degree** of the channel.

Note also that sometimes **input non-determinism** must be taken into account: the *receive* operation may return a message from any channels in the set, and the message picking algorithm is random and non-predictable.

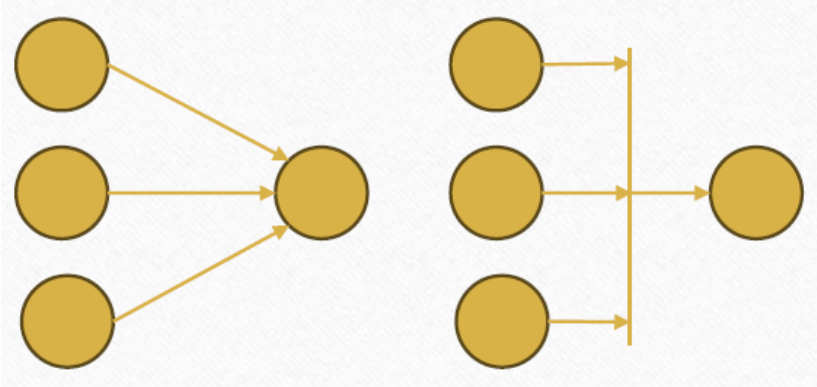


Figure 5.6: Input non-determinism

5.3 Foster's Parallel Algorithm Design Method

There is no one-fits-all recipe for parallelizing sequential programs, however... Foster's **PCAM** approach provides some useful guidelines to keep in mind when designing parallel algorithms.

- ◇ **Partitioning** - divide the problem into a large amount of small fine-grained tasksk that can be executed in parallel
- ◇ **Communication** - determine the required communication pattern between the tasks (*data dependencies*)
- ◇ **Agglomeration** - combine identified tasks into larger coarse-grained tasks to reduce communication by improving data locality
- ◇ **Mapping** - assign tasks to processes and threads to minimize communication, enable concurrency and *balance workload*

5.3.1 Jacobi and MCOP

In Lecture 7, more concepts concerning Jacobi and Matrix Chain Ordering Problem (MCOP) are be discussed. They are not reported here.

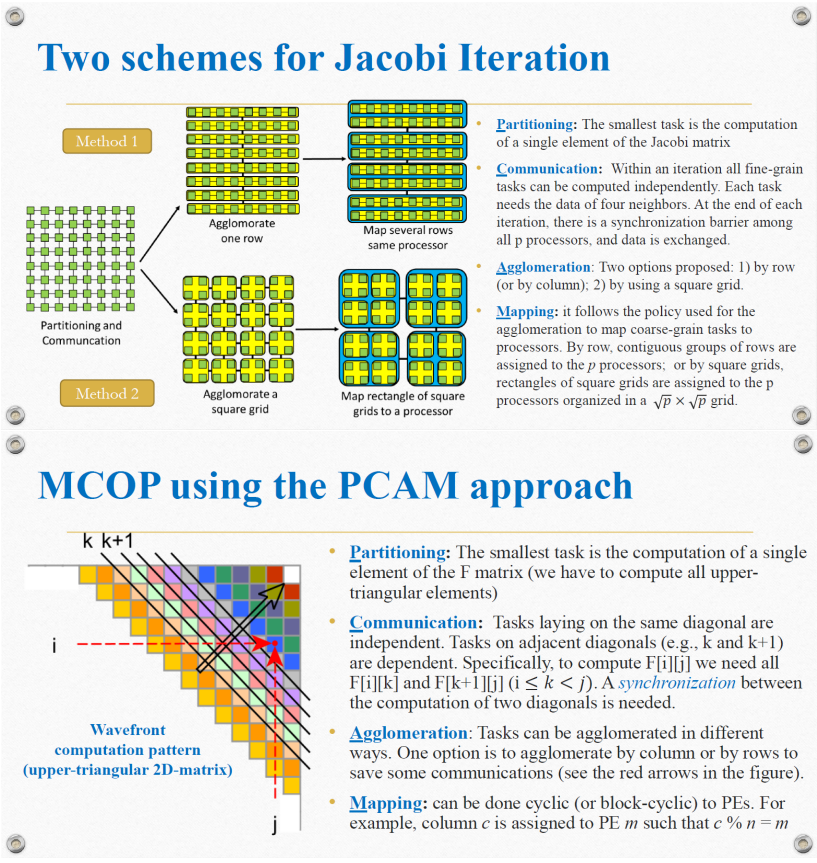


Figure 5.7: Jacobi and MCOP slides

Chapter 6

Laws and Metrics

Starting off with some terminology, we will introduce some basic concepts and metrics that are fundamental to understanding the performance of parallel systems. We will then delve into the laws that govern the performance of parallel systems, and how they can be used to predict the performance of a parallel system.

6.1 Terminology

Definition 6.1 (Speedup) *The speedup (S) is defined as the quotient of the time taken using a single processor ($T(1)$) over the time measured using p processors ($T(p)$)*

$$S = \frac{T(1)}{T(p)} \tag{6.1}$$

The best speedup we can achieve is p, which is the number of processors. This is known as **linear speedup**. Exceptions to this are to as **superlinear speedup**.

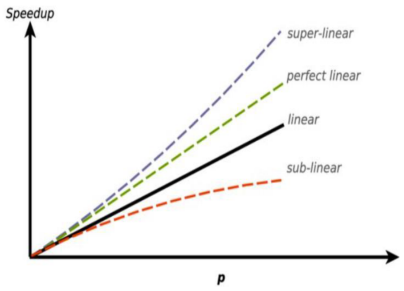


Figure 6.1: Speedup categories

Definition 6.2 (Efficiency) *The efficiency (E) is defined as the speedup divided by the number of processors*

$$E = \frac{S}{p} = \frac{T(1)}{p \cdot T(p)} \tag{6.2}$$

$p \cdot T(p)$ is also called *"cost of parallelization"*, or simply *"cost"*. Clearly, the ideal efficiency is 1.

Scalability and speedup are apparently defined in the same way, but there is a significant difference.

Definition 6.3 (Scalability) *Scalability considers as $T(1)$ the time obtained by executing the **parallel** implementation of the algorithm on a single processor.*
*It is also called **"relative speedup"***

$$Scalability = \frac{T_{par}(1)}{T_{par}(p)} \tag{6.3}$$

Definition 6.4 (Absolute Speedup) *(Absolute) Speedup instead considers $T(1)$ as the time obtained by executing the —best— **sequential** implementation of the algorithm on a single processor.*

$$Speedup = \frac{T_{seq}(1)}{T_{par}(p)} \tag{6.4}$$

6.1.1 Strong and Weak Scaling

- ◇ **Strong-scaling** - the problem size is fixed. Linear scaling is hard to achieve due to *Amdahl's Law.* (discussed later on). It depends on the amount of **serial work**, i.e. the part of the problem that cannot be parallelized.
- ◇ **Weak-scaling** - the problem size is p times bigger in the same amount of time. The problem size remains constant per processor, but Weak scaling assumes that as the size of the problem increases,

– the amount of serial work remains constant (or increases slowly)
– The amount of communication among processors remains constant (or increases slowly)

Consider the following example:

◇ `INPUT` - Array A of n numbers
◇ `OUTPUT` - $\sum_{i=0}^{n-1}$
◇ `Task` - Parallelize this problem by using an array of *processing elements (PEs)*.

Assumptions

◇ **Computation**: Each PE can add two numbers stored in its local memory in 1 sec
◇ **Communication**: A PE can send data from its local memory to the local memory of any other PE in 3 sec (independently of the data size!)
◇ **Input and Output**: At the beginning of the program, the whole input array A is stored in PE #0. At the end, the result must be gathered in PE #0
◇ **Synchronization**: All PEs operate in a lock-step manner; i.e., they can either compute, communicate, or be idle (no computation-to-communication overlap!).

First of all we must establish the sequential runtime ($p = 1$), for example $T(1, n) = n - 1sec$. Then we may evaluate speedup, efficiency, and scalability for different values of p.

In the data displayed in the slides, speedup and efficiency decreases for values of p greater than 2. In fact, in the following figure 6.1 we have a figure summarizing results.

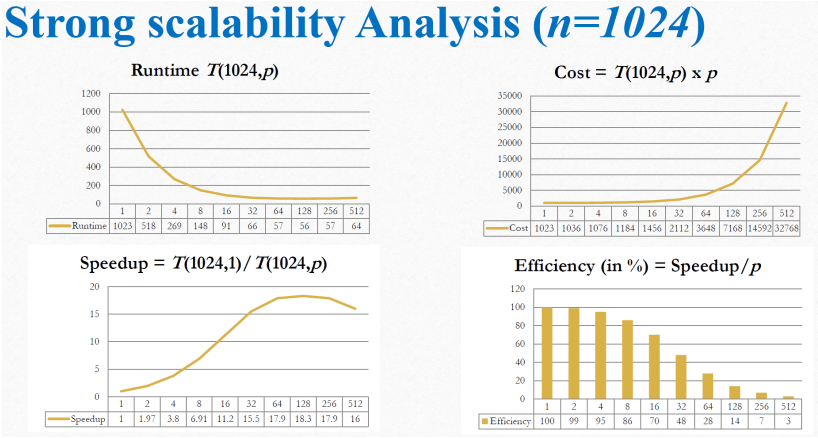


Figure 6.1: Strong scalability analysis

We can determine which is the optimal speedup for a given problem. It is the red line in Fig. 6.2 .

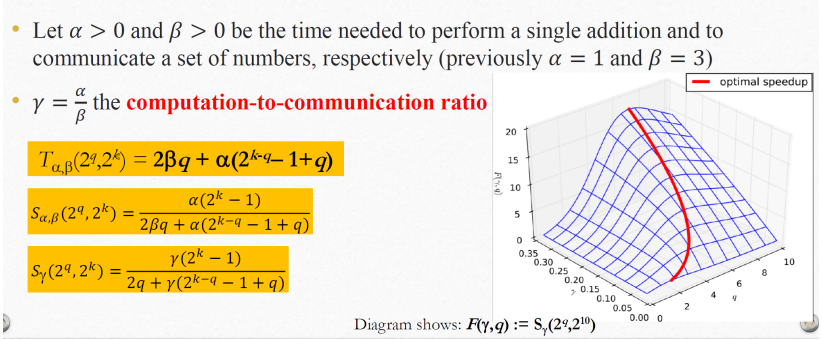


Figure 6.2: Computation-to-communication ratio

Takeaway message

In general the speedup is mostly stable up to a certain point, then it starts to decrease. This is due to the fact that the communication overhead starts to dominate the computation time.

The grater the communication overhead, the lower the speedup.

6.2 Laws of Parallel Systems

6.2.1 Amdahl's Law

It states that no matter how many processors are used in a parallel run, the program's speedup will be limited by its fraction of sequential code.
The Amdahl's law gives an **upper bound** on the speedup that can be achieved. The speedup is limited by the part of the task that cannot be parallelized. Even if you have a large number of processors, the sequential portion of the task becomes the bottleneck. For example, if 90% of the task can be parallelized, the maximum speedup (no matter how many processors) is limited to 10x, because the remaining 10% has to be done sequentially.

Definition 6.5 (Amdahl's Law)

$$S(p) = \frac{1}{f + \frac{1-f}{p}} \tag{6.5}$$

where f is the fraction of the program that is sequential. Conversely, $1 - f$ is the fraction of the program that can be parallelized.

However, note that Amdahl's law is not always applicable. In fact, it is based on the assumption that the problem size is fixed (**strong scalability**), and that the parallel part of the program is perfectly parallelizable.

6.2.2 Gustafson's Law

Gustafson's law takes into account scenarios where the problem size scales with the number of processors (**weak scalability**). Unlike Amdahl's Law, Gustafson's Law assumes that the problem size grows with the number of processors, meaning that parallel processors are used to solve bigger problems rather than trying to reduce the execution time of a fixed problem size.As the problem grows, the parallelizable portion becomes more significant, leading to better utilization of the available processors.
The law asserts that the parallelizable part grows linear in p while the non-parallelizable part remains constant.

$$S(p) \leq f + p(1 - f) = p + f \cdot (1 - p) \tag{6.6}$$

Chapter 7

Models of Computation

7.1 PRAM

Idealized Shared-Memory platform with no caching, no NUMA organization, no synchronization overhead. The PRAM model is a theoretical model that is used to analyze the performance of parallel algorithms. It is a shared-memory model, where all processors can access all memory locations. The PRAM model is a simplification of the real world, and it is used to analyze the performance of parallel algorithms.

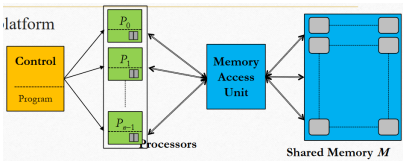


Figure 7.2: PRAM Model

There exist also variations of the PRAM model, such as the EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Exclusive Write), CRCW (Concurrent Read Concurrent Write), and ERCW (Exclusive Read Concurrent Write).

7.2 Prefix Computation

Given a binary associative operation $\circ$ on the set X:

Definition 7.1 (associative operation)

$$(x_1 \circ x_2) \circ x_3 = x_1 \circ (x_2 \circ x_3) \tag{7.1}$$

Examples are sum, product, max, min, string concat, boolean `and/or` etc.

The prefix computation of a sequence $x_1, x_2, \ldots, x_n$ is the sequence $s_1, s_2, \ldots, s_n$ where:

Abstract

$$s_0 = x_0 \tag{7.2}$$

$$s_1 = x_0 \circ x_1 \tag{7.3}$$

$$\dots \tag{7.4}$$

$$s_i = x_0 \circ x_1 \circ \dots \circ x_i \quad i = 0, \dots, n-1 \tag{7.5}$$

$$s_i = s_{i-1} \circ x_i \quad i = 1, \dots, n-1 \tag{7.6}$$

$$s_n = x_1 \circ x_2 \circ \dots \circ x_{n-1} \quad \text{Questo non mi torna molto} \tag{7.7}$$

Definition 7.2 (Prefix computation) *Obtaining $S = s_0, s_1, \ldots, s_n$ from $X = x_0, x_1, \ldots, x_n$ is called prefix computation.*

7.2.1 Parallel Prefix Computation on PRAM

Consider $\circ = +$ and a PRAM model with n processors. The following algorithm computes the prefix sum of an array $A[0 \ldots n-1]$ in $O(nlogn)$ time using n processors..

```
for (j = 0; j<n; j++) do_in_parallel    // each processor j
   reg_j = A[j];                        // copies one value to a local register
for (i = 0; i<ceil(log(n)); i++) do     // sequential outer loop
   for (j = pow(2, i); j<n; j++) do_in_parallel // each proc. j
      reg_j += A[j - pow(2, i)];        // performs computation
      A[j] = reg_j;                     // writes result to shared memory
}
```

$O(nlogn)$ is **not optimal**, we want to reach $O(n)$. How can we do that?

7.2.1.1 Approach 1

Let's use $p = n/logn$ processors and apply the following algorithm:

1. Partition the n input values into chunks of size $log(n)$. Each processor computes local prefix sums of the values in one chunk in parallel (takes time $O(log(n))$).
2. Perform the old non-cost-optimal prefix sum algorithm on the $nlog(n)$ partial results (takes time $O(log(n/log(n)))$).
3. Each processor adds the value computed in step 2 by its left neighbor to all values of its chunk(takes time $O(log(n))$)
   Hence, the algorithm is *cost-optimal*, since we get that

$$C(n) = T(n,p) \times p = O(log(n)) \times \frac{n}{log(n)} = O(n) \tag{7.8}$$

7.2.1.2 Approach 2

ASsume you have a one-dimensional array A where most entries are zero. We can represent the array in a more memory-efficient way by only storing the values of the non-zero entries (in `V`) and their corresponding coordinates (in `C`).

We generate a temporary array (*tmp*) with $tmp[i] = 1$ if $A[i]! = 0$ and $tmp[i] = 0$ otherwise. We then perform a parallel prefix summation on *tmp*. For each non-zero element of A, the respective value stored in *tmp* now contains the destination address for that element in `V` .

Then, we write the non-zero elements of A to `V` using the addresses generated by the parallel prefix summation. The respective coordinates can be written to C in a similar way.

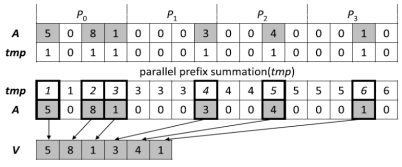


Figure 7.3: Sparse Array Computation

7.3 Bulk Synchronous Parallel (BSP) Model

This is more than a theoretical model, it is closer to reality than the PRAM model. It was proposed as a unified framework for the design, analysis, and programming of general-purpose parallel systems.

It consists of three parts:

- ◇ A collection of processor-memory components
- ◇ A communication network that can deliver point-to-point messages among processors. It is a black box, no assumptions are made about its internal functioning.
- ◇ A facility for global synchronization (barrier) of all processors

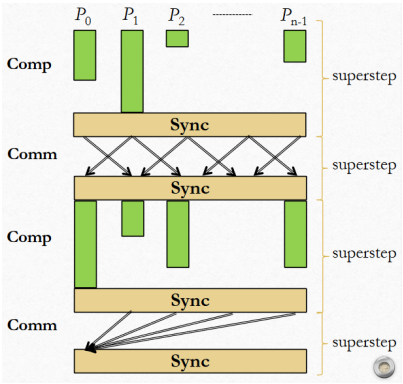


Figure 7.1: Bulk Synchronous Parallel Model

A BSP algorithm consists of a sequence of **supersteps**, which consist of

- ◇ a number of computation or communication steps or both
- ◇ called mixed superstep in case both are executed
- ◇ a *global barrier* synchronization (bulk synchronization)

A ***computation* superstep** consists of many small steps executing operations (e.g., floating point operations), while a ***communication* superstep** consists of many basic communication operations (e.g., a data word transferring).

A communication step hence is means transferring data from a memory to another

An h-relation is a communication superstep in which every processors sends and receives at most h data words. It is the maximum between the data words sent and received by a processor in a communication superstep.

h-relations

The cost of an h-relation is $T(h) = g * h + l$ where:

in general, the cost of a BSP algorithm is the sum of the costs of all supersteps.

$$a + b * g + c * l \tag{7.9}$$

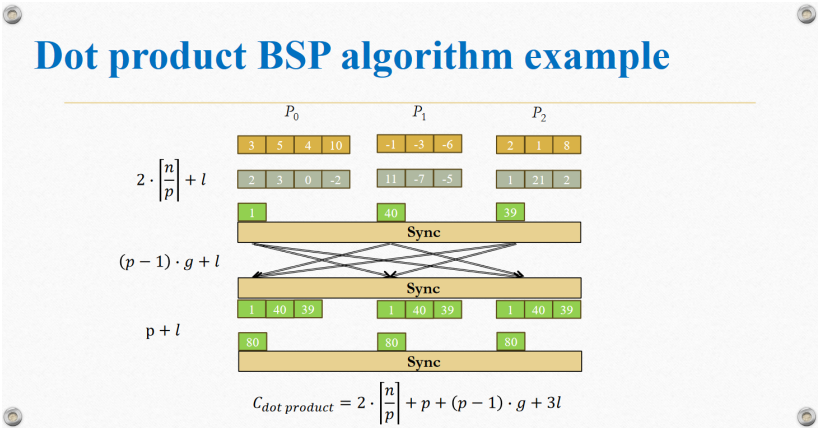


Figure 7.2: BSP algorithm example

Recall that BSP is SPMD (Single Program Multiple Data) model, meaning that all processors execute the same program, but on different data.

7.4 Work-Span Model

The Work-Span model (also called Work-Depth) model is another classic performance model. It provides more strict bounds than those offered by the Amdahl's and Gustafson's laws.

The program's tasks form a DAG (Directed Acyclic Graph; a **task** is a unit of work, i.e., arbitrary sequential code, that can be executed in parallel (using threads or processes) with other program's tasks.

A given graph's task is ready to run iff all its predecessors in the graph have been completed. Hence, the edges represent a **data dependency** between tasks, i.e. the output of a task is the input of another task.

Let's assume:

- ◇ p identical processors, each executing one ready task at a time

◇ The task scheduling is greedy, i.e., whenever there is a ready task (and an available processor), the task is executed immediately

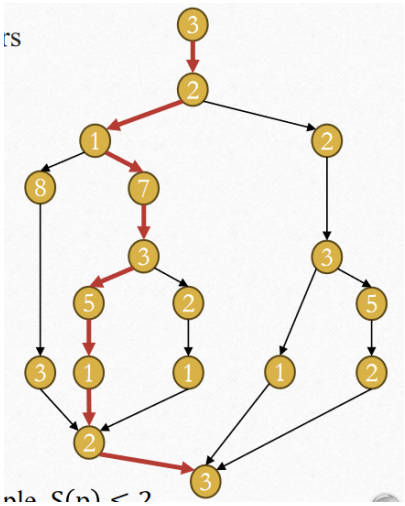


Figure 7.3: Work-Span Model

◇ T_p is the time when executing with a greedy scheduler with p processors
◇ T_1 is called *work*
◇ T_∞ is called *span*, or *critical path*
◇ In the example on the right we have $T_1 = 54$ and $T_\infty = 27$
◇ Let's consider the speedup to derive interesting bounds:

$$S(p) = \frac{T_1}{T_p} \leq \frac{T_1}{\frac{T_1}{T_p}} = p \quad \textit{i.e. it is not possible to have superline} \tag{7.10}$$

$$S(p) = \frac{T_1}{T_p} \tag{7.11}$$

7.5 Brent's Theorem

◇ Assume a parallel computer whose processors can perform a task in unit time with greedy scheduling of tasks
◇ Assume that the computer has enough processors to exploit the maximum concurrency in an algorithm containing T_1 tasks such that it completes in T_∞ time steps.
◇ At each level i of the DAG, there are m_i tasks. We may use m_i processors to compute all results in $O(1)$.

Brent's theorem states that a similar computer with fewer processors p can execute the algorithm with the following upper time limit on T_p:

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty \tag{7.10}$$

7.5.1 Implications

If $T_\infty \ll T_1$, then $T_1 - T_\infty \approx T_1$, therefore

$$T_p \leq \frac{T_1}{p} + T_\infty \quad \textit{if } T_\infty \ll T_1 \tag{7.11}$$

When designing a parallel algorithm, focus on reducing the span, because the span is the fundamental asymptotic limit on scalability. Increase the work only if it enables a drastic decrease in span.
Overall we have:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty \tag{7.12}$$

Additionally, it can be proved that

$$S(p) = \frac{T_1}{T_p} \approx p \textit{ if } \frac{T_1}{T_\infty} \gg p \tag{7.13}$$

Part II

C++

Chapter 8

Concurrency in C++

8.1 Threads

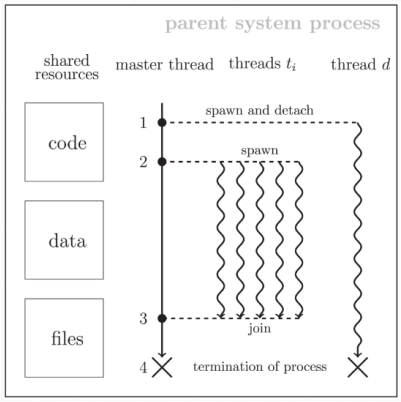


Figure 8.1: Thread schema

- ◇ The master thread can spawn threads, and each thread can spawn threads as well
- ◇ The number of spawned threads should be roughly the amount of cores (i.e., pay attention to oversubscription)
- ◇ Threads share process resources
- ◇ each thread has a separate stack
- ◇ A thread can be joined or detached once
- ◇ A detached thread cannot be joined
- ◇ Joined or detached threads cannot be reused
- ◇ All threads must be joined or detached within the scope of their declaration

8.2 Asynchronicity and Tasks

It may be more convenient to let the threads start asynchronously. This can be achieved by using the `std::async` function. This function returns a `std::future` object that can be used to retrieve the result of the function. The function is executed asynchronously, and the result is stored in the `std::future` object. The function `std::future::get()` can be used to retrieve the result. The function `std::future::wait()` can be used to wait for the result to be available. The function `std::future::valid()` can be used to check if the result is available.

Note however that a call to `std::async` does *not* necessarily imply that a new thread is spawned. The calling thread might execute the task without spawning a thread!
Default behavior depends on the implementation; thus, remember to use `std::launch::async`

```
auto future = std::async(std::launch::async, foo, id);
```

8.2.1 Tasks

The `std::package_task` function that allows you to conveniently construct task objects, which are callable objects (function pointer, functor class, member function pointer, lambda, std::function wrapper, ...) with associated the corresponding future object handling the return value

...

8.3 Mutexes

A mutex is a synchronization mechanism that guarantees mutual exclusion execution of a critical section. Its usage restricts the execution of a critical section to a single thread at a time.

A thread locking a mutex prevents other threads from acquiring the mutex. The other threads wait for its release (passive waiting).

```
#include <mutex>
std: :mutex mutex;
// to be called by threads
void some_function ( ... ) {
   mutex. lock ()
   // this region is only processed by one
       thread at a time
   mutex. unlock() ;
   // this region is processed in parallel
}
```

```
#include <mutex>
std: : mutex mutex;
// to be called by threads
void some_function ( ... ) {
   // here we acquire the lock
   std :: lock_guard<std: :mutex> lock_guard
       (mutex);
   // this region is locked by the mutex
   } // <- here we release the lock
   // this region is processed in parallel
}
```

8.3.1 Different implementations

- ◇ `std :: lock_guard` is a wrapper around `std :: mutex` that acquires ownership of the mutex upon construction and releases it upon destruction.
  - Once constructed, it cannot be explicitly released until it goes out of scope
  - It can only manage a single mutex
- ◇ `std :: scoped_lock` manages multiple mutexes simultaneously, acquiring/releasing all of them simultaneously when entering/exiting the scope.
  - It provides an `unlock ()` method to explicitly release the locks before the end of the scope.
  - Helps avoid deadlock when you need to acquire multiple locks together
  - Pay attention to the fact that it accepts 0 mutexes, resulting in a run-time error

8.3.2 Condition Variables

A CV enables one or more threads to wait (passively) for an event inside a critical section.

Conceptually, a CV is associated with an event or condition. When a thread has determined that the condition is satisfied,it can notify one or more of the threads waiting on the CV to wake them up.

Pay attention to **spurious wake-ups**! The condition must be checked in a loop (typically a while loop).

8.4 Distributing Workload

8.4.1 Static assignment

We may distribute the work in various manners. A possibility is to assign a fixed number of elements to each thread. This is called **static assignment**. Supposing an array of data $b = b_0, b_1, \ldots, b_15$ and $p = 5$ processors, these are three common strategies:

- ◇ **Block distribution**: Each processor gets a block of data. For example, p_0 gets b_0, b_1, b_2, b_3; p_1 gets b_4, b_5, b_6; and so on.
- ◇ **Cyclic distribution**: Each processor gets every p^th element. For example, p_0 gets b_0, b_5, b_{10}, b_{15}; p_1 gets b_1, b_6, b_{11}; and so on.
- ◇ **Block-cyclic distribution** given c block size: A combination of the two above. For example, p_0 gets b_0, b_1, b_{10}, b_{11}; p_1 gets b_5, b_6, b_{12}, b_{13}; and so on.
  $c \cdot p$ is called **stride**. b_i is assigned to $p_{(i/c) \ mod \ p}$

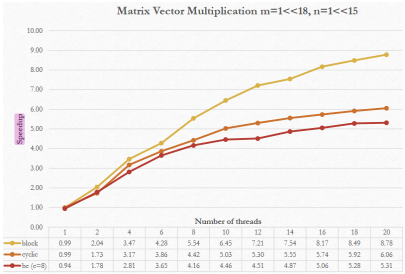


Figure 8.1: Speedup comparison

8.4.2 Dynamic assignment

It may not be possible to know in advance how much work each task will have to do. The size of the problem may be unkown, or the workload may be unbalanced, meaning that some tasks may result in more work than others. Even in the case of multiplying matrices, the workload may be strongly unbalanced if the matrices are sparse.

For such cases, dynamic assignment is more appropriate. The master thread assigns work to the worker threads as they become available. This is done by using a **work queue**.

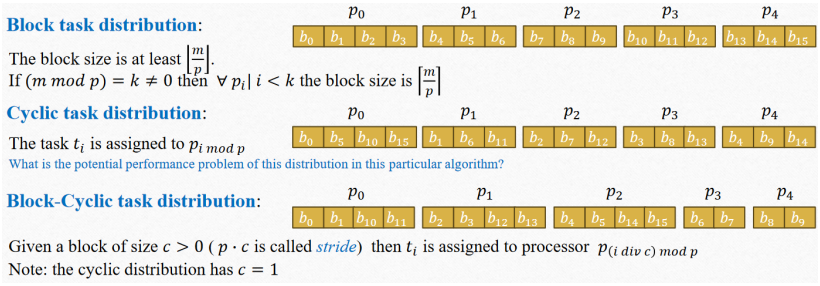


Figure 8.2: Static assignment strategies

All-pairs distance Matrix

- We have a $m \times n$ matrix D_{ij} where i denotes one of the m vectors and j enumerates the n elements of the vector.
- We want to compute the distance (or *similarity*) $d(\cdot, \cdot)$ between all pairs of vectors.
  - $\Delta_{ij} = d(x^{(i)}, x^{(i')}) \quad = \sqrt{\sum_{k=1}^{n}(D_{ik} - D_{jk})^2}$
  - The distance/similarity measure $d(\cdot, \cdot)$ might be a traditional metric such as Euclidean distance or any symmetric binary function that assign a notion of similarity to pair of instances
- We have to comput m^2 distances, so $\mathcal{O}(m^2 n)$ operations
- However note that $\Delta_{ij} = \Delta_{ji}$, i.e. it is a symmetric matrix, so we can compute only the lower triangular part

We statically assign each row to a thread, using a block cycling distribution. This is not ideal, as some rows may take longer to compute than others, resulting in unbalanced workloads.

In Fig. 8.3 we can see the distribution of the workload. Each row is assigned to a thread, and the workload is distributed in blocks size $c = 2$. As c increases, the number of blocks decreases, and the workload becomes more unbalanced.

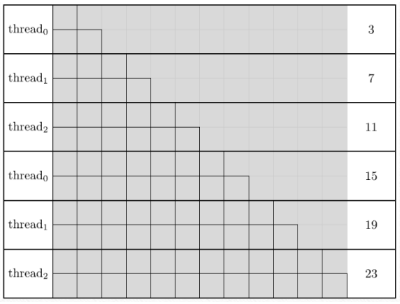


Figure 8.3: Block cyclic distribution

Consider the snippet on the right. Each thread tries to hold a lock on the mutex, and then updates the global variable `lower` with the next row to compute. The global variable `global_lower` is updated by the master thread.

This is a simple example of dynamic assignment, directly performed by thread and automatically balancing the workload. This however comes at the cost of the mutex overhead, which is not negligible.

In general, if the workload is not-so-unbalanced, static assignment is preferable.

Listing 8.1: spm3/all-pairs.cpp

```
// while there are still rows to compute
while (lower < rows) {

   // update lower row with global lower row
   {
      std::lock_guard<std::mutex> lock_guard
         (mutex);
      lower  = global_lower;
      global_lower += chunk_size;
      }
```

8.4.2.1 Producer-Consumer model

Here we have a shared data structure, typically a queue, that is accessed by multiple threads. The producer thread(s) insert data into the queue, while the consumer thread(s) remove data from the queue. The producer and consumer threads are synchronized by a mutex and a condition variable.

8.4.2.1.1 Stop token The `std::stop_token` is a mechanism to signal the threads to stop. The producer thread will keep adding data to the queue until the stop token is requested. The consumer thread will keep consuming data from the queue until the stop token is requested.

It is not always the way-to-go solution to stop threads, it is one of the available options. Another option is to use

a global `std::atomic<bool>` flag, which is more efficient but less flexible. Or, as I did in my Operative Systems project, to insert a *special value* in the queue to signal the end of the data.

Listing 8.1: spm3/prod-cons.cpp

```
std::stop_token stoken = stopSrc.get_token();
// ...
auto producer = [&](const std::stop_token &stoken, int id) {
  uint64_t i=0;
  while (!stoken.stop_requested()) {
    {
      std::lock_guard<std::mutex> lock(mtx);
      dataq.push_back(i++);
    }
    cv.notify_one();
      //...

auto consumer = [&](const std::stop_token &stoken, int id) {
  std::unique_lock<std::mutex> lock(mtx, std::defer_lock);
  for(;;) {
    lock.lock();
    if (cv.wait(lock, stoken, [&dataq] { return !dataq.empty(); })) {
      auto data = dataq.front();
      (void)data;
      dataq.pop_front();
      //std::printf("Consumer%d, data=%lu\n", id, data);
    }
    else {
      if (stoken.stop_requested()) {
        lock.unlock();
        break;
      }
    }
    lock.unlock();
      // ...
```

8.4.2.2 Multiple-Reader Single-Writer

A reader may access the data concurrently with other readers, while a writer needs exclusive access to the shared data. This is a common pattern with a simple —not trivial— solution.

Using a normal `std::mutex` to protect the accesses to the critical section **does prevent** the concurrent access of multiple readers, so it is not a viable solution. The Standard library offers a solution with `std::shared_mutex`, which allows multiple readers to access the data concurrently, while a writer needs exclusive access to the shared data.

Listing 8.2: spm3/reader-writer.cpp

```
auto reader = [&](int count, int id) {
   for(int i=0;i<count; ++i) {
      std::shared_lock<std::shared_mutex>
          lock(smutex);
      std::printf("Reader%d has read %d\n
          ", id, shared_counter);
   }
};
auto writer = [&](int count, int id) {
   for(int i=0;i<count; ++i) {
      std::unique_lock<std::shared_mutex>
          lock(smutex);
      ++shared_counter;
      std::printf("Writer%d has written %d\n
          ", id, shared_counter);
   }
};
```

8.4.2.3 Thread Pool

Instead of creating threads on the fly, we can create a pool of threads that are ready to execute tasks. This is useful when the overhead of creating threads is high, and we want to reuse threads for multiple tasks.

There is no out-of-the-box solution of a Thread Pool (TP) in the Standard Library. There are many libraries that provide thread pools, such as *Boost* or *Intel TBB*, but we can also manage them by ourselves.

The synchronization is handled through mutex and condition variables, in a Producer-Consumer fashion. In the implementation provided (`threadPool.hpp`) the task queue is **unbounded**! If the producers are much faster than the Workers, the tasks queue becomes huge, and the memory blows up ☹.
To avoid this we can either increase the number of Worker threads (if we have enough cores), or we can implement a **bounded queue**.

8.5 Lock-Free Programming

With locks and mutexes, we can ensure that only one thread accesses a critical section at a time. However, this comes at a cost: the overhead of acquiring and releasing the lock, and the possibility of deadlocks.
Aside from the operational overhead, the lock approach sometimes forces to spend a lot of time waiting for the lock to be released, even when the critical section is accessed for a very short time.

8.5.1 Atomic Counting

C++11 introduces atomic data types that can be safely manipulated in a concurrent context without acquiring time-consuming locks. The atomic data types are implemented using hardware instructions that guarantee atomicity.

Listing 8.2: spmcode4/atomic_counting.cpp

```
auto lock_count =
     [&] (volatile uint64_t* counter, const auto& id) {
     for (uint64_t i = id; i < num_iters; i += num_threads) {
         std::lock_guard<std::mutex> lock_guard(mutex);
         (*counter)++;
     }
 };
auto atomic_count =
[&] (volatile std::atomic<uint64_t>* counter, const auto& id) {
for (uint64_t i = id; i < num_iters; i += num_threads)
    (*counter)++;
};

// > elapsed time (mutex_multithreaded): 5.98232s
// > elapsed time (atomic_multithreaded): 1.02452s
```

`std::atomic<T>` is a template class that provides atomic operations on the type `T`. The operations are implemented using hardware instructions that guarantee atomicity.

- `std::atomic` is neither copyable nor movable
- `T` must be copyable and movable
- `T` cannot have a user-defined copy constructor
- Operations (i.e., the methods of the `std::atomic` class)
  - `load` / `store`: to get and set the content of a `std::atomic`
  - `exchange`: stores a new value and returns the old value of the variable
  - `compare` and exchange: it does an atomic exchange only if the value is equal to the provided expected value
  - `wait` / `notify` (from C++20) behavior similar to condition variables

```
// While there are still rows to compute
while (lower < rows) {
   // Update lower row with global lower
       row
   {
      std::lock_guard<std::mutex>
          lock_guard(mutex);
      lower = global_lower;
      global_lower += chunk_size;
   }
}
```

```
// While there are still rows to compute
while (lower < rows) {
   // Update lower row with global lower
       row
   lower = std::atomic_fetch_add(&
       global_lower, chunk_size);
}
```

Not all `std::atomic` operations are supported on all types. For example, `std::atomic<bool>` does not support `fetch_add`, but only `fetch_or`, `fetch_and`, `fetch_xor`, and `exchange`.

Operation	atomic_flag	atomic<bool>	atomic<T*>	atomic<integral-type>	atomic<other-type>
test_and_set	Y				
clear	Y				
is_lock_free		Y	Y	Y	Y
load		Y	Y	Y	Y
store		Y	Y	Y	Y
exchange		Y	Y	Y	Y
compare_exchange_weak, compare_exchange_strong		Y	Y	Y	Y
fetch_add, +=			Y	Y	
fetch_sub, -=			Y	Y	
fetch_or, |=				Y	
fetch_and, &=				Y	
fetch_xor, ^=				Y	
++, --			Y	Y	

Figure 8.4: Atomic operations supported by std::atomic

8.5.1.1 CAS - Compare and Swap

Every C++ atomic data type features a CAS (Compare And Swap) operation for implementing arbitrary atomic assignments. There are two methods for CAS which should *always* be both performed in loops:

- `compare_exchange_weak(T& expected, T desired)`: It may fail spuriously, but it is more efficient i.e. may return false even if the comparison yields true
- `compare_exchange_strong(T& expected, T desired)`: It unlikely fails spuriously, but it is less efficient

1. Compare the value expected with the value stored in the atomic
2. If yes, then it sets the atomic to the value desired, otherwise writes the actual value stored in atomic into expected
3. Return *true* if the swap operation in step 2 was successful, *false* otherwise

Listing 8.3: Finding the max in an array

```
auto false_max = [&] (volatile std::
    atomic<uint64_t> &counter,
                      const
                          auto& id) ->
                          void {
     for (uint64_t i = id; i < num_iters;
          i += num_threads) {
         if(i > counter) counter = i;
 }
 };
 auto correct_max = [&] (volatile std::
     atomic<uint64_t> &counter,
```

The problem with the first uncorrect max implementation is that it is not thread-safe: the `load` (reading `i< counter`) and `store` (writing `counter =i`) operations are atomic per se, but anything may happen in between them.
For this reason we need to use the CAS operation, which is a *read-modify-write* operation.

```
                            const auto& id)
                                ->
                                void {
        for (uint64_t i = id; i < num_iters;
             i += num_threads) {
            auto previous = counter.load();
            while (previous < i &&
                !counter.
                    compare_exchange_weak(
                    previous, i)) {}
        }
    };
```

8.6 Memory Consistency Concepts

The intuition says that a memory read should return *the latest value written* in a memory location. What does "*the latest value written*" mean in a CMP system when executing multiple threads?
It depends on the ordering of memory operations, precisely the order in which memory operations performed by one thread become visible to other threads.

Modern CMPs reorder memory operations in different/strange ways for performance reasons

Memory consistency defines the allowed behavior of reads and writes (i.e., loads and stores) to different addresses in a parallel system.

We should care about memory ordering mostly because it helps to program lock-free concurrent code.

Memory consistency deals with when writes to a variable X propagate to other processors relatively to reads and writes to other addresses.

```
// initially, A = B = flag = 0
// Thread1
Store A = 1;
Store B = 1;
Store flag = 1;

// Thread2
while (Load flag==0); // spin
Load A;
Load B;
print A and B;
```

If there are caches in the system, from the viewpoint of the cache coherence protocol, the printing of `A=B=0` can be perfectly fine, due to the ordering of instructions, even if we expect that Thread2 should print `A=B=1`.

Memory consistency defines correct program behavior and is part of the specification of architectures. **Cache coherence** is a different concept, it deals with the objective of ensuring that the memory system in a parallel computer behaves as if the caches were not present; coherence is *not* directly visible at the software level, but only as a side effect, e.g. false sharing, superlinear speedup...

8.6.1 Memory Ordering

The **program order** defines a sequence of reads and writes.
There are four types of memory operation ordering that can be guaranteed (or not) by the platform:

1. $W_X \rightarrow R_Y$: write to X must commit before the subsequent read of Y
   i.e., when a write comes before a read in program order, the write must commit (i.e, its result must be visible) by the time the read occurs
2. $R_X \rightarrow R_Y$: read from X must commit before the subsequent read from Y
3. $R_X \rightarrow W_Y$: read from X must commit before the subsequent write to Y
4. $W_X \rightarrow W_Y$: write to X must commit before the subsequent write to Y

8.6.1.1 Sequential Consistency - SC

The most intuitive memory model is **Sequential Consistency** (SC). It is the most restrictive memory model; it was introduced by Leslie Lamport in 1979.

Definition 8.1 (Sequential Consistency) *A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

Alternatively, *There exists a **total** order of all loads and stores across all threads such that the value returned by each load is equal to the value of the most recent store to that location*

SC maintains all four memory operation orderings.

8.6.1.2 Relaxed Memory Consistency

The disadvantage to SC is that the firmware may not apply optimizations in the code, which can lead to a significant performance loss. In all modern systems, some reordering and optimizations are enforced.

Relaxed memory consistency models allow the compiler and hardware to violate certain memory ordering rules to improve performance.
Ideally, it aims to gain performance by hidinh memory latency, i.e. overlapping memory access operations with other operations when they are independent.

Modern architectures exploit *Write Buffers*, which allow to write to a buffer instead of directly to memory, and *Read Buffers*, which allow to read from a buffer instead of directly from memory.
The $W_X \rightarrow R_Y$ ordering is relaxed to hide write latency, hence the processor k may move its own reads ahead of its own writes. This is fine from the viewpoint of a single-thread control flow.

Total Store Order (TSO) and *Processor Consistency* (PC) relax $W_X \rightarrow R_Y$ in different ways.

Partial Store Ordering (PSO) relaxes also $W_X \rightarrow W_Y$. The processor might reorder write operations in the Write Buffer: one write might be a cache miss, while the other might be a cache hit whose management costs less.

```
Thread1
on P0
A = 1;
flag=1;

Thread2
on P1
while (flag==0) ; //spin
print A;
```

This is a valid optimization if a program consists of a single instruction stream

Under PSO, P1 may observe the flag change before the write to A is visible, hence printing 0. This behavior would not be allowed by TSO and SC.

Even more aggressive reordering may allow to overlap multiple reads and writes, Execute reads as early as possible and writes as late as possible to hide memory latency. An example is the *Weak Ordering* (WO) model, which relaxes all orderings. `ARM` and `POWER` architectures are based on a relaxed memory model.

8.6.1.3 Solving this nightmare

Every architecture provides synchronization primitives to make memory ordering stricter. **Memory barrier** instructions (also "fences") prevent reordering, but they are expensive:

- ◇ All memory operations must be completed before any memory operation after the barrier can begin
- ◇ There are different flowers of fences: load fence (RMB), store fence (WMB), mem fence (MB)

Besides, there are also the synchronization primitives discussed before which enable ordering on a specific memory address, such as `test_and_set` , `compare_and_swap` , and all `read-modify-write` (RMW) operations.

However, recall also that if a program is *Data-Race-Free* (DRF), then its behavior is sequentially consistent, because the reordering won't affect program correctness. Properly syncronized programs via mutexes, barriers, atomics, etc. are **DRF**.

"SC does not guarantee DRF!" — prof. Torquati

8.6.1.4 Programming Languages

We talked about the underlying architecture as the culprit of reordering, but the programming language can also play a role in this. The **compiler** may reorder instructions for performance reasons independengly from the low- level architectures.

It is generally accepted that a compiler can reorder ordinary reads from and writes to memory almost arbitrarily, provided the reordering cannot change the observed single-threaded execution of the code.