UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Introduction

Parallelism vs. Concurrency

- A system is said to be **concurrent** if it can support two or more actions *in progress* at the same time.

- A system is said to be **parallel** if it can support two or more actions executing at the same time.

- The key concept and difference between these definitions is the phrase “in progress”.

Parallelism vs. Concurrency

- A **concurrent application** will have two or more threads in progress at some time.
  E.g.: two threads that are being swapped in and out by the operating system on a single core processor. These threads will be “in progress”—each in the midst of its execution—at the same time.

- A **parallel application** will have two or more threads executing simultaneously if the computation platform has multiple cores available.
  E.g.: two or more threads could each be assigned a separate core and would be running simultaneously.

Parallelism vs. Concurrency

- Parallelism ⊆ Concurrency

- A concurrent program becomes parallel if there are enough cores to execute its multiple threads or processes.

- Concurrent programming: composition of independently executing processes.

- Parallel programming: programming as the simultaneous execution of (possibly related) computations.

Parallelism vs. Concurrency

- **Concurrency** is about dealing with lots of things at once.

- **Parallelism** is about doing lots of things at once.

- These are not the same thing, but they are related.

- **Concurrency** is about structure, **Parallelism** is about execution.

- **Concurrency** provides a way to structure a solution to solve a problem that may (but not necessarily) be **parallelizable**.

Parallelism vs. Concurrency

- During this lecture I have to deal with concurrent things: advancing the slides checking that the projector works, delivering the lecture and perhaps breaking to answer a question that arises from the lecture.

- I’m doing only one of these things at a certain time.

- If I’m helped by one or two assistants that check the projector and answer the questions while I’m lecturing then the process will become concurrent and parallel.

- If the assistants and I perform a different task independently, e.g. changing the slides without caring if I’m talking about something different, then the process is parallel.

Parallelism vs. Concurrency

- During this lecture I have to deal with concurrent things:
  advanci
  deliverir
  questior
- I'm doin
- If I'm he
  projecto
  the proc
- If the as
  indepen
  talking a
  parallel.

A **concurrent** program has **multiple logical threads of control**. These threads may or may not run in parallel.

A **parallel** program potentially runs more quickly than a sequential program by executing different parts of the computation simultaneously (in parallel). **It may or may not have more than one logical thread of control**.

hen

l'm

Sequential algorithm

- Most of today's algorithms are sequential: they specify a sequence of steps in which each step consists of a single operation.

- It has worked so far because of the free performance improvement provided by CPUs and their increase in clock and their pipelining…

- … but this free ride is over. Chip manufacturers are not working on improving clock speed.

Parallel algorithm

- A parallel algorithm is designed to execute multiple operations at the same step

- The parallelism in an algorithm can yield improved performance on many different kinds of computers.

- For example, on a parallel computer, the operations in a parallel algorithm can be performed simultaneously by different processors or cores.

- Even on a single-processor computer the parallelism in an algorithm can be exploited by using multiple functional units, pipelined functional units, or pipelined memory systems.

Parallel algorithm

- A parallel algorithm is designed to execute multiple operations at the same step

- The parallelism in an algorithm can yield improved performance on many different kinds of computers.

- For example, on a parallel computer, the operations in a parallel algorithm can be performed simultaneously by different processors or cores.

In parallel programming we identify and expose parallelism in algorithms, expressing this in our software, and understanding the costs, benefits, and limitations of the chosen implementation.

Concurrent algorithm

- A concurrent algorithm is an algorithm structured so that it is possible to execute its steps independently.
- A concurrent algorithm can be executed serially.
- There is need of some communication to coordinate the independent executions.

Concurrent algorithm

- A concurrent algorithm is an algorithm structured so that it is possible to execute its steps independently.

- A concurrent algorithm can be executed serially.

Since multicore CPUs are now available everywhere we will talk about parallel execution of concurrent code.

We will use the term “parallelization” when dealing with translation of serial code to concurrent (but this should be “concurrentization”)

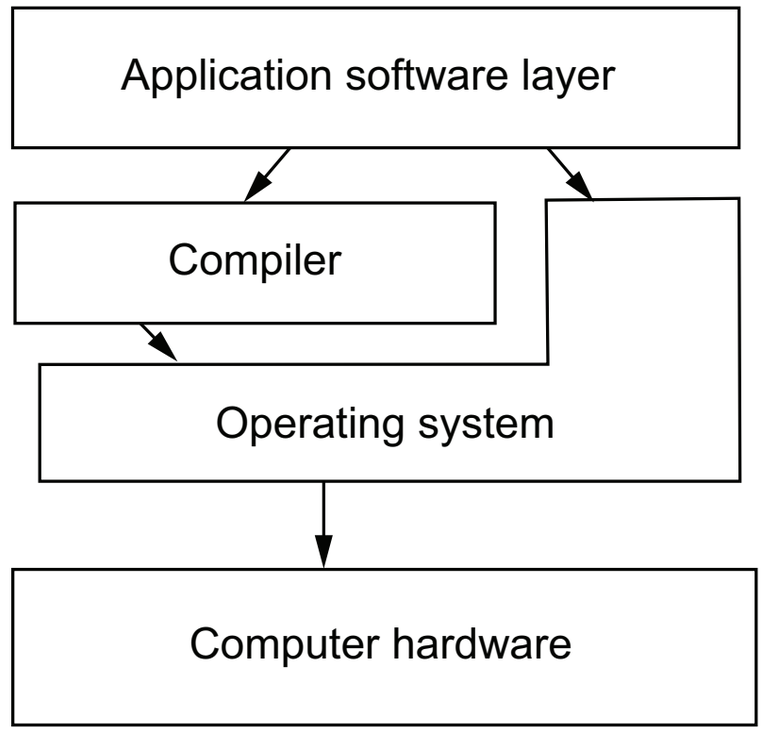
Parallel Programming

- As a developer, you are responsible for the application software layer, which includes your source code.

- In the source code, you make choices about the programming language and parallel software interfaces you use to leverage the underlying hardware.

- Additionally, you decide how to break up your work into parallel units.

- A compiler is designed to translate your source code into a form the hardware can execute. With these instructions at hand, an OS manages executing these on the computer hardware.



Motivations

- It is not always obvious that a parallel algorithm has benefits, unless we want to do things ...
  - faster: doing the same amount of work in less time
  - bigger: doing more work in the same amount of time
- Both are good outcomes of program parallelization: they improve the results of the program.

Motivations

- Consider a 16-core CPU with hyperthreading and a 256 bit-wide vector unit. How many parallel operations can it perform ?
  - 16 (cores) × 2 (hyperthreads) × 256 (bit-wide vector unit)/64 (bit-wide double) = 128-way parallelism

- Using only 1 serial path out of 128 parallel paths means that we are using 1/128 = 0.008 or 0.8% of the available computational capabilities.

Motivations

- Consider a 16-core CPU with hyperthreading and a 256 bit-wide vector unit. How many parallel operations can it perform ?
  - 16 (cores) × 2 (hyperthreads) × 256 (bit-wide vector unit)/64 (bit-wide double) = 128-way parallelism

- Using only 1 serial path out of 128 parallel paths means that we are using 1/128 = 0.008 or 0.8% of the available computational capabilities.

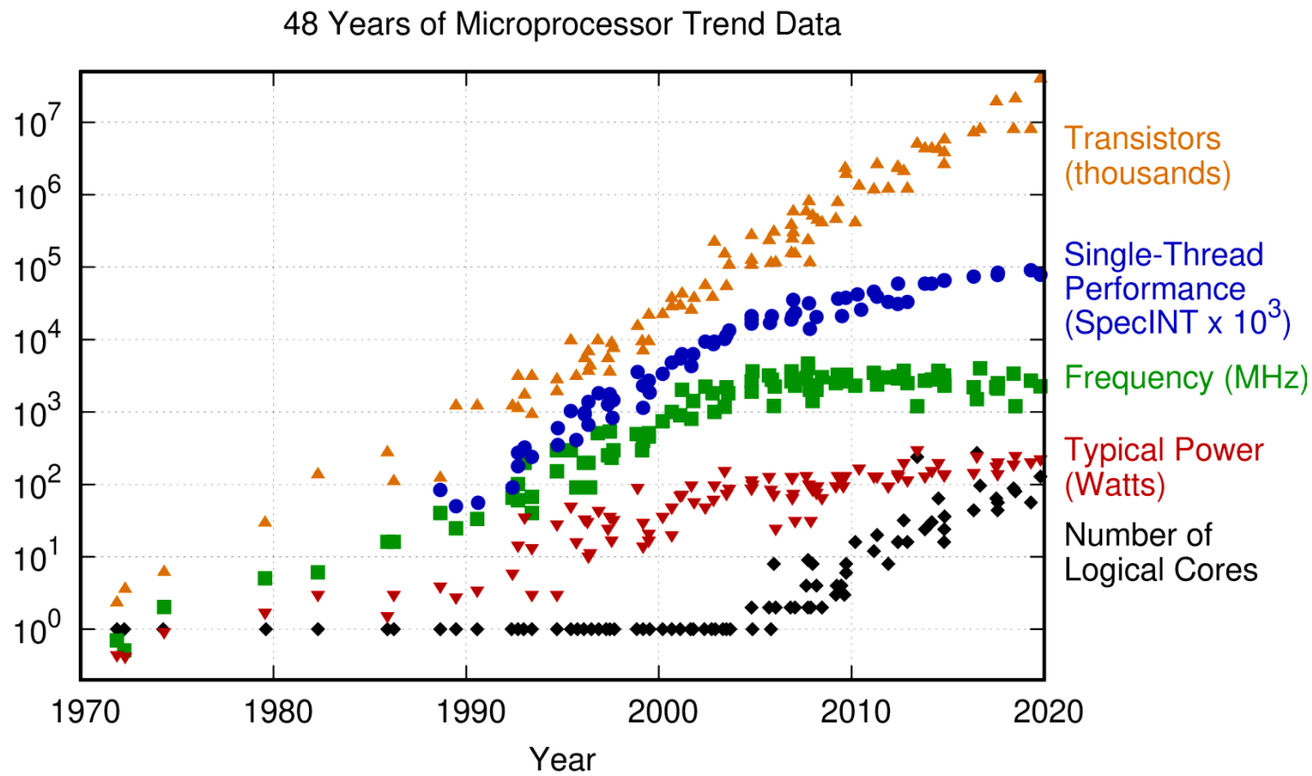
Note that this calculation doesn't take into account many real-world constraints… it's hard to really exploit that 128-way parallelism

CPUs…

- The clock speed of a processor cannot be increased without overheating

But…

- More and more processors can fit in the same space
- Multicores are everywhere



…and all the rest

- Grid computing: collection of computer resources from multiple locations to reach a common goal.

- Cluster computing: loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system.

- Cloud computing: a model for enabling ubiquitous network access to a shared pool of configurable computing resources.

…and all the rest

- Grid computing: collection of computer resources from multiple locations to reach a common goal.

Grids are a form of distributed computing whereby a “super virtual computer” is composed of many networked loosely coupled computers acting together to perform large tasks.

computers that work together so that, in many respects, they can be viewed as a single system.

- Cloud computing: a model for enabling ubiquitous network access to a shared pool of configurable computing resources.

…and all the rest

- Grid computing: collection of computer resources from multiple locations to reach a common goal.

Grids are a form of distributed computing whereby a “super virtual computer” is composed of many networked loosely coupled computers acting together to perform large tasks.

computers that work together so that, in many respects, they can be viewed as a single system

The components of a cluster are usually connected to each other through LAN, with each node (computer used as a server) running its own instance of an operating system.

- Cloud computing: a model for enabling ubiquitous network access to a shared pool of configurable computing resources.

…and all the rest

- Grid computing: collection of computer resources from multiple locations to reach a common goal.

Grids are a form of distributed computing whereby a “super virtual computer” is composed of many networked loosely coupled computers acting together to perform large tasks.

computers that work together so that, in many respects, they can be viewed as a single system

The components of a cluster are usually connected to each other through LAN, with each node (computer used as a server) running its own instance of an operating system.

- Cloud computing: a model for enabling ubiquitous network access to a shared pool of configurable

Cloud computing and storage solutions provide users and enterprises with various capabilities to store and process their data in third-party data centers.[2] It relies on sharing of resources to achieve coherence and economies of scale, similar to a utility (like the electricity grid) over a network.

Parallel vs. Distributed

- **Parallel computing**: provide performance.
  - In terms of processing power or memory;
  - To solve a single problem;
  - Typically: frequent, reliable interaction, fine grained, low overhead, short execution time.
- **Distributed computing**: provide convenience.
  - In terms of availability, reliability and accessibility from many different locations;
  - Typically: interactions infrequent, with heavier weight and assumed to be unreliable, coarse grained, much overhead and long uptime.

Parallel vs. Distributed

- **Parallel computing**: provide performance.
  - In terms of processing power or memory;
  - To solve a single problem;
  - Typically: frequent, reliable interaction, fine grained, low overhead, short execution time.
- **Distributed computing**: provide convenience.

> We will not deal with distributed computing, in this course.

  - Typically: interactions infrequent, with heavier weight and assumed to be unreliable, coarse grained, much overhead and long uptime.

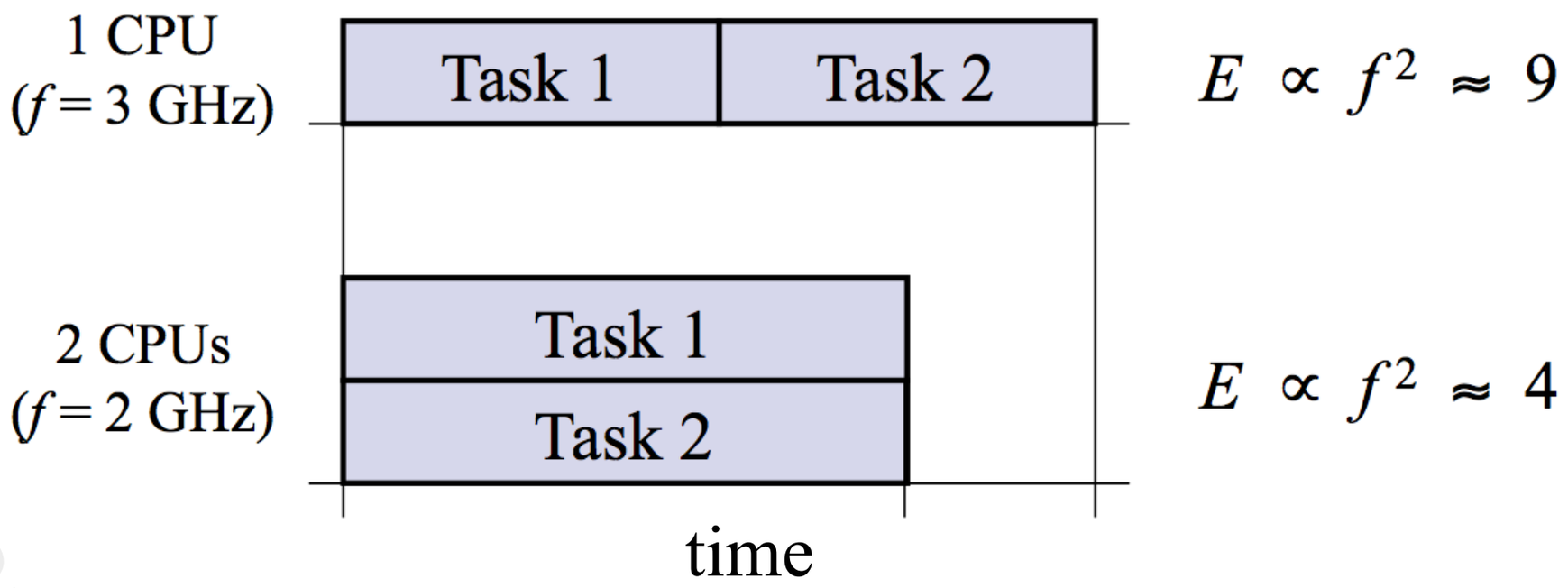
Power consumption

- The power consumption of a CPU has become increasingly important.
- Energy is more suited to compare 2 CPUs (a faster one may consume more power but for much less time, resulting in less energy consumption…)
- In a CPU most of the energy is consumed switching transistor states. This so called *dynamic energy* is proportional to the square of the voltage *V* and frequency *f*, so power consumption is: $P \propto f V^2$
- The voltage required to operate the CPU at given frequency is, roughly proportional to the frequency ($P \propto f^3$).
- Increasing the frequency, the same amount of work can be finished earlier (~ inversely to the frequency), so the required energy is: $E \propto f^2$

Physical limitations

- As noted by Intel CTO in 2004: “[without revolutionary new technologies…] increased power requirements of newer chips will lead to CPUs that are hotter than the surface of the sun by 2010.”

- Parallelism can be used to conserve energy



Energy: CPUs vs GPUs

- GPUs need lot of power but thanks to increased parallelism they can reduce consumption.
- Example:
  - Intel's 16-core Xeon E5-4660 processor has a thermal design power of 120 W. Suppose we need 20 of these processors for 24 hours to run to completion. The estimated energy usage for your application is
    - P = (20 Processors) × (120 W/Processors) × (24 hours) = 57.60 kWh
  - Suppose we can port the application on 4 NVIDIA Tesla V100 GPUs running the same program in 24 hours. NVIDIA's Tesla V100 GPU has a maximum thermal design power of 300 W. The estimated energy usage for your application is
    - P = (4 GPUs) × (300 W/GPUs) × (24 hrs) = 28.80 kWh

Energy: CPUs vs GPUs

- GPUs need lot of power but thanks to increased parallelism they can reduce consumption.

There's also a cost reduction: maintaining a server with 4 GPUs is probably easier than keeping 5 servers with 4 CPUs

  - Intel's 16-core Xeon E5-4660 processor has a thermal design power of 120 W. Suppose we need 20 of these processors for 24 hours to run to completion. The estimated energy usage for your application is
    - P = (20 Processors) × (120 W/Processors) × (24 hours) = 57.60 kWh
  - Suppose we can port the application on 4 NVIDIA Tesla V100 GPUs running the same program in 24 hours. NVIDIA's Tesla V100 GPU has a maximum thermal design power of 300 W. The estimated energy usage for your application is
    - P = (4 GPUs) × (300 W/GPUs) × (24 hrs) = 28.80 kWh

Physical limitations

- Transistors are getting smaller (somehow the Moore's law is still valid), but they are not getting faster.
  - Reducing the gate size in planar MOSFET transistors leads to "leakage". There's need of new geometries (e.g. FinFET, Nanosheet).
- Signalling rates on chip are limited by the resistance and capacitance of the wires connecting the transistors. The smaller they get the thinner the wires (thus higher resistance) and the higher the capacitance.
  - The time delay for a wire depends on what is called the "RC time constant", which is the resistance multiplied by the capacitance.

Physical limitations

- Transistors are getting smaller (somehow the Moore's law is still valid), but they are not getting faster.
  - Reducing the gate size in planar MOSFET transistors leads to "leakage". There's need of new geometries (e.g. FinFET, Nanosh

It is much more convenient to add cores, rather than increasing speed

- Signalling rates or [illegible] capacitance of the wires connecting the transistors. The smaller they get the thinner the wires (thus higher resistance) and the higher the capacitance.
  - The time delay for a wire depends on what is called the "RC time constant", which is the resistance multiplied by the capacitance.

Physical limitations

- Suppose we have to calculate in one second

```
for (i = 0; i < ONE_TRILLION; i++)
        z[i] = x[i] + y[i];
```

- Then we have to perform $3x10^{12}$ memory moves per second
- If data travels at the speed of light ($3x10^8$ m/s) between the CPU and memory, and *r* is the average distance between the CPU and memory, then *r* must satisfy
  - $3\times10^{12}\ r = 3\times10^8$ m/s × 1 s which gives $r = 10^{-4}$ meters
- To fit the data into a square so that the average distance from the CPU in the middle is *r*, then the length of each (square) memory cell will be
  - 2×10^{-4} m / ($\sqrt{3}\times10^6$) = 10^{-10} m which is the size of a relatively small atom!

Instead, the computation can be parallelized to complete it within the required time limit

Important factors

- Important considerations in parallel computing
  - Physical limitations: the speed of light, CPU heat dissipation
  - Scalability: allows data to be subdivided over CPUs to obtain a better match between algorithms and resources to increase performance
  - Memory: allow aggregate memory bandwidth to be increased together with processing power at a reasonable cost
  - Economic factors: cheaper components can be used to achieve comparable levels of aggregate performance

Sometimes parallel is bad

- Bad parallel programs can be worse than their sequential counterparts:
  - Slower: because of communication overhead
  - Scalability: some parallel algorithms are only faster when the problem size is very large
- Understand the problem and use common sense!
- Moreover: not all problems are amenable to parallelism

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Types of parallelism

Bit-Level Parallelism

- It is a form of parallel computing based on increasing processor word size.
- If an 8-bit computer wants to add two 32-bit numbers, it has to do it as a sequence of 8-bit operations. By contrast, a 32-bit computer can do it in one step, handling each of the 4 bytes within the 32-bit numbers in parallel.
- Though, it’s unlikely that we’ll have 128-bit computers soon.

Instruction-Level Parallelism

- Modern CPUs are highly parallel, using techniques like *pipelining*, *out-of-order execution*, and *speculative execution*.
  - *Pipelining*: execution of multiple instructions can be partially overlapped.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

  - *O-o-O execution*: instructions execute in any order that does not violate data dependencies. Its benefit grows as the pipeline gets deeper (and speed difference between CPU and RAM/Cache increases).
  - *Speculative execution*: used to reduce cost of conditional branch instructions in pipelined CPUs. Instructions are scheduled ahead of the determination of the need of their execution. It may use *branch prediction*.

Instruction-Level Parallelism

- Modern CPUs are highly parallel, using techniques like *pipelining*, *out-of-order execution*, and *speculative execution*.
  - On modern (multicore) CPUs instructions will not be executed in the same order in which you write them

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

  - *O-o-O execution*: instructions execute in any order that does not violate data dependencies. Its benefit grows as the pipeline gets deeper (and speed difference between CPU and RAM/Cache increases).
  - *Speculative execution*: used to reduce cost of conditional branch instructions in pipelined CPUs. Instructions are scheduled ahead of the determination of the need of their execution. It may use *branch prediction*.

Instruction-Level Parallelism

- Modern CPUs are highly parallel, using techniques like *pipelining*, *out-of-order execution*, and *speculative execution*.
  - On modern (multicore) CPUs instructions will not be executed in the same order in which you write them

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

  - *O-o-O execution*: instructions execute in any order that does not violate data dependencies. Its benefit grows as

```
DIVD f6,f2,f4 ; slow… latency
ADDD f0,f6,f8 ; must wait for f6
SUBD f12,f8,f14 ; could be executed before DIVD or ADDD
```

  - *Speculative execution*: used to reduce cost of conditional branch instructions in pipelined CPUs. Instructions are scheduled ahead of the determination of the need of their execution. It may use *branch prediction*.

Data Parallelism

- Also known as SIMD (Single Instruction, Multiple Data)

- This architecture performs same operations on a large quantity of data in parallel.

  - Multimedia processing is a good candidate for this type of architecture (i.e. GPUs).

Task-Level Parallelism

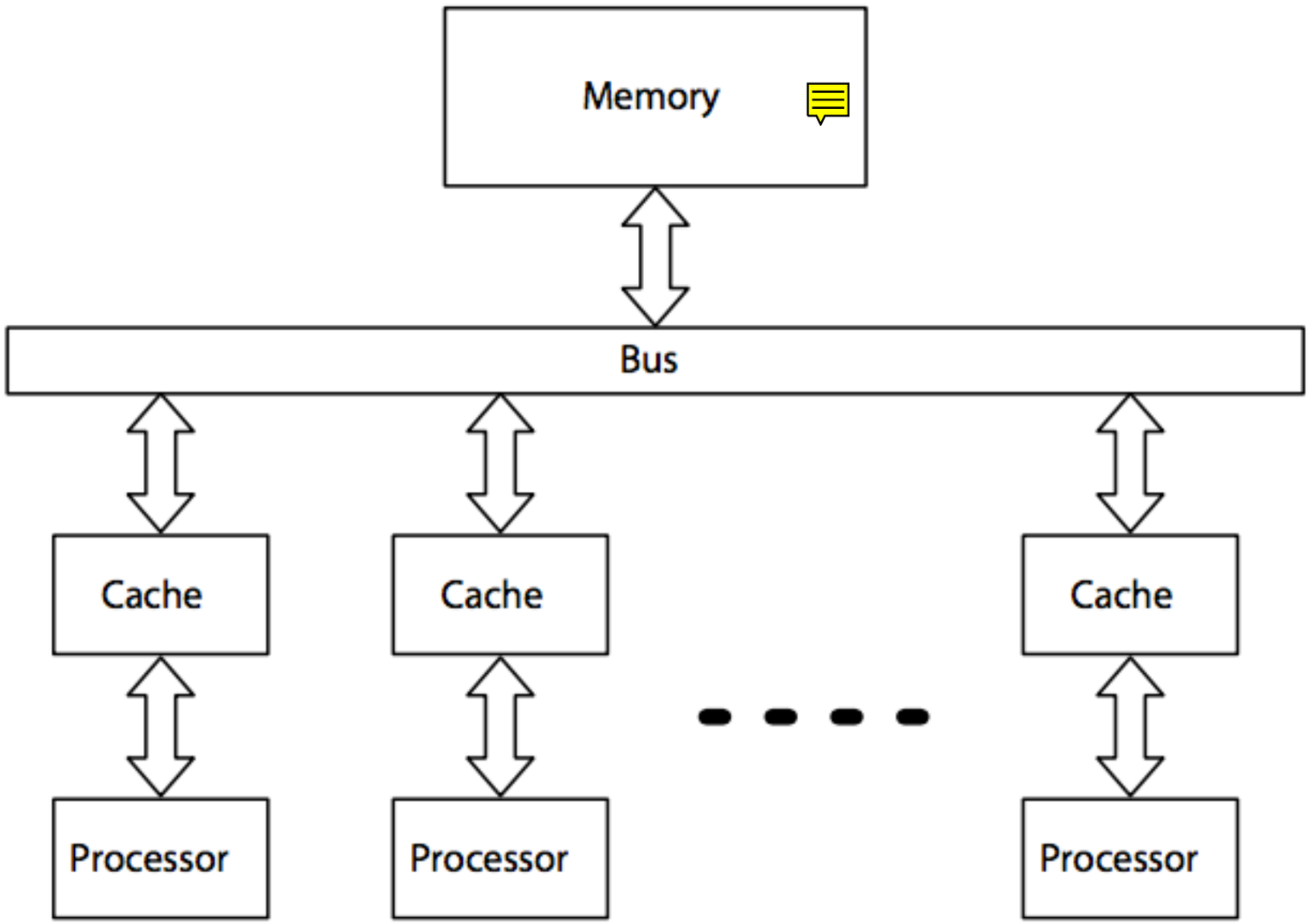
- There are two important variations, related to the underlying memory model: shared vs. distributed.

- **Shared memory** multiprocessors: each CPU accesses any memory location, IPC is done through memory.

- **Distributed memory** system: each CPU has its own local memory, IPC is done through a network.

- Shared memory systems are simpler but do not scale after a certain number of processors: Distributed systems are the way to go for fault-tolerant systems.

Shared memory system



- Natural extension of sequential computer: all memory can be referenced (single address space). Hardware ensures memory coherence.

- Good:

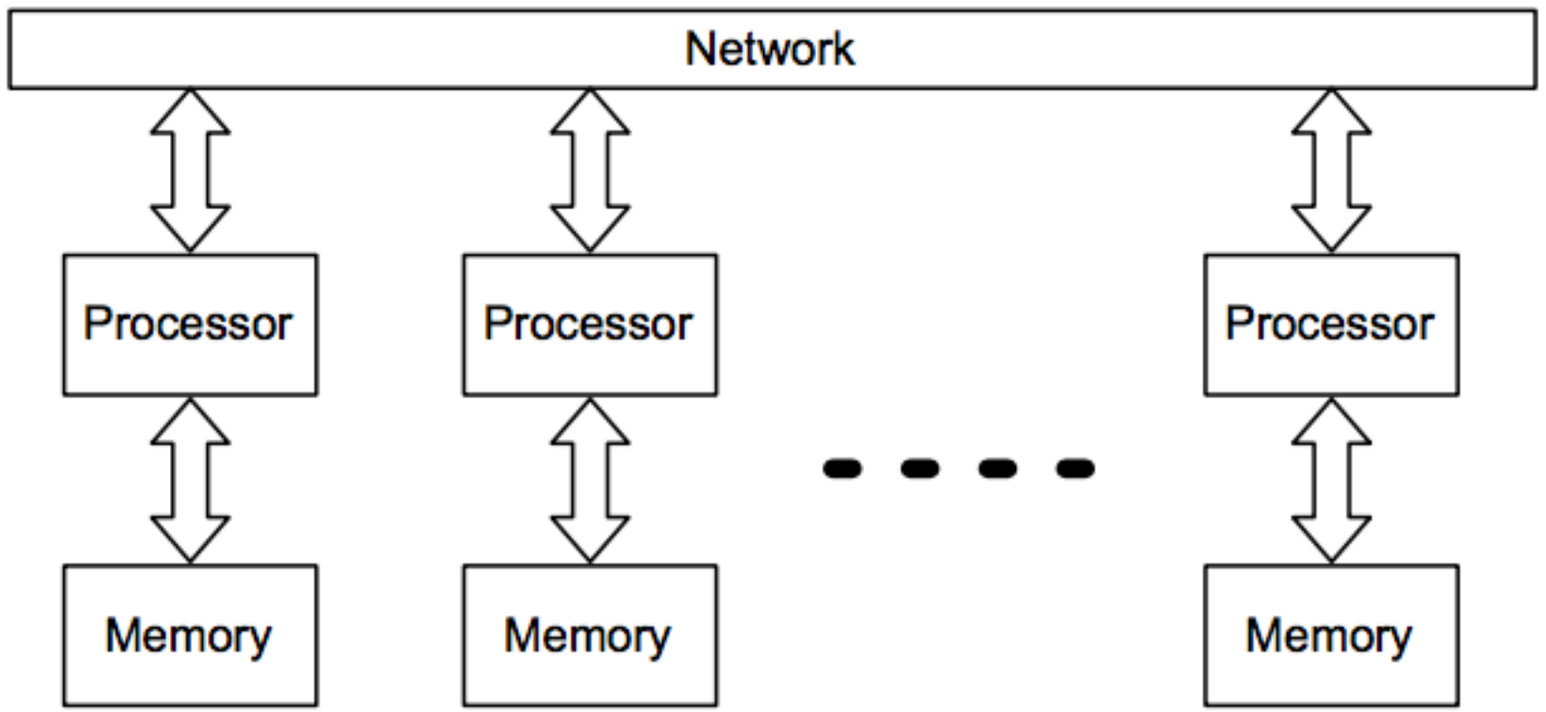
  - Easier to use
    Through multi-threading

- Bad:

  - Easier to create faulty programs
    Race conditions

  - More difficult to debug
    Intertwining of threads is implicit

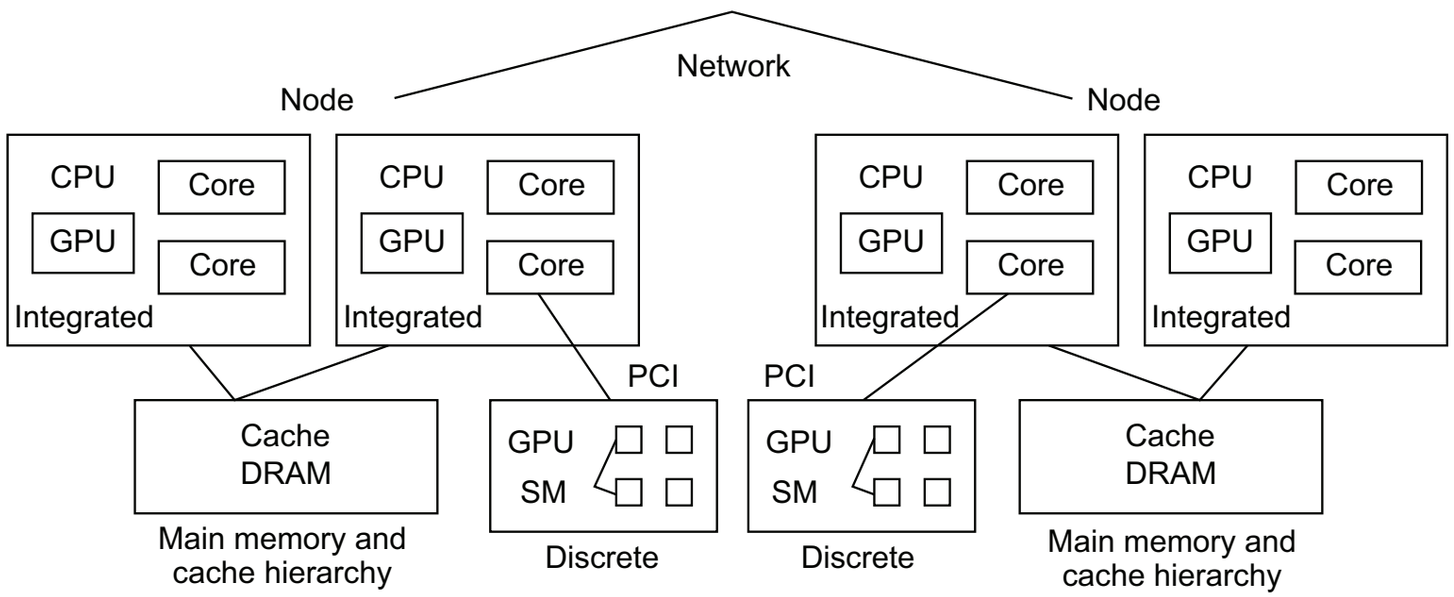
Distributed memory system



- Processors can only access their own memory and communicate through messages.
- Requires the least hardware support.
- Easier to debug.
  - Interactions happens in well-defined program parts
  - The process is in control of its memory!
- Cumbersome communication protocol is needed
  - Remote data cannot be accessed directly, only via request.

Heterogeneous systems

- Today we commonly use heterogeneous parallel systems that combine:
  - Shared memory system
  - Distributed memory system
  - Accelerator devices (i.e. distributed systems within a single unit)



Flynn's taxonomy

		Data Streams	
		Single	**Multiple**
Instruction Streams	**Single**	**SISD** e.g. Intel Pentium 4	**SIMD** SSE instructions of x86
	Multiple	**MISD**	**MIMD** e.g. Intel Xeon Phi

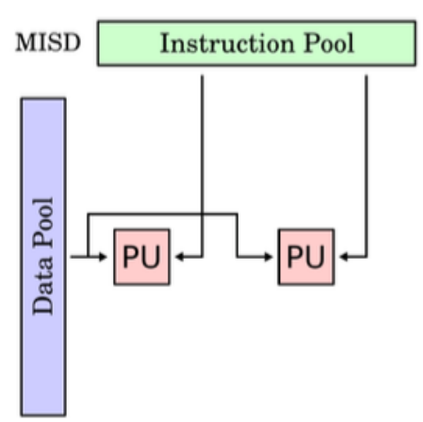
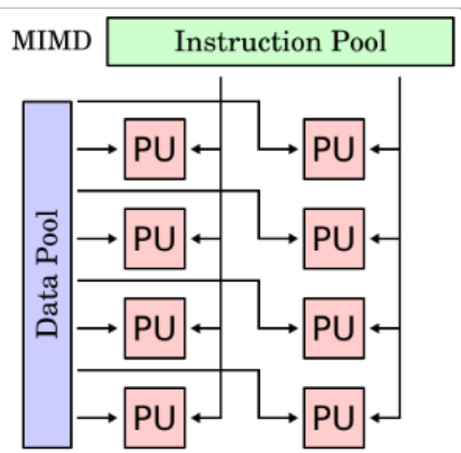
Flynn's taxonomy

- The taxonomy has been expanded with **SPMD** (single program, multiple data).

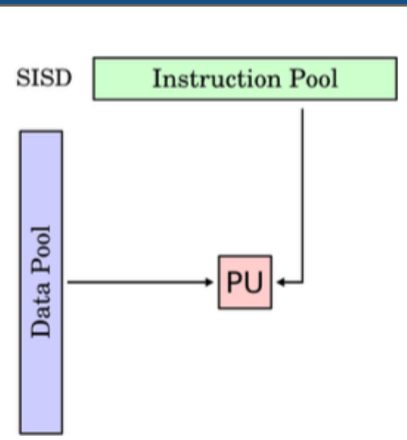
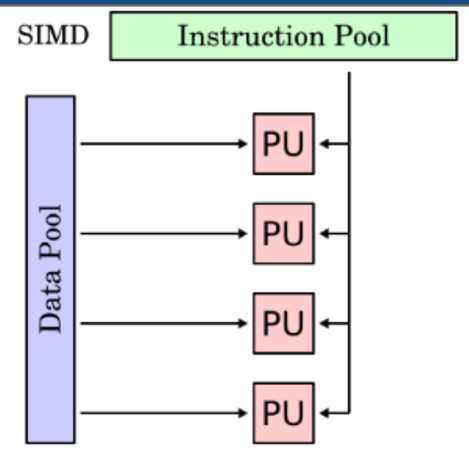
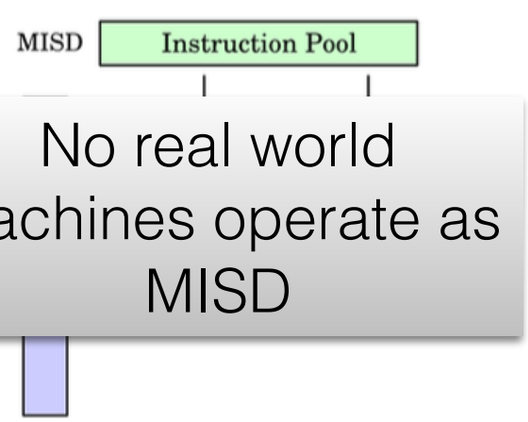
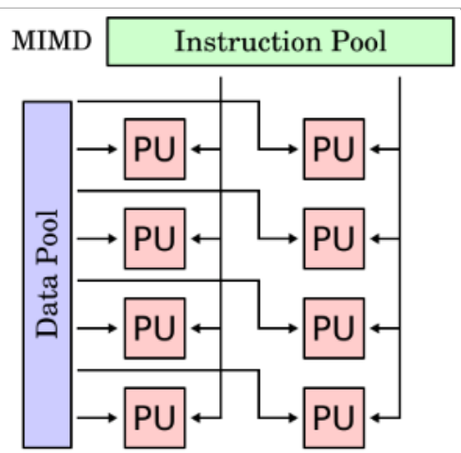
- Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.

- SPMD could be considered a parallel program on a MIMD computer.

Flynn's taxonomy

		Data Streams: Single	Data Streams: Multiple
Instruction Streams	Single		
	Multiple		

Flynn's taxonomy

		Data Streams: Single	Data Streams: Multiple
Instruction Streams	Single		
	Multiple	 No real world machines operate as MISD	

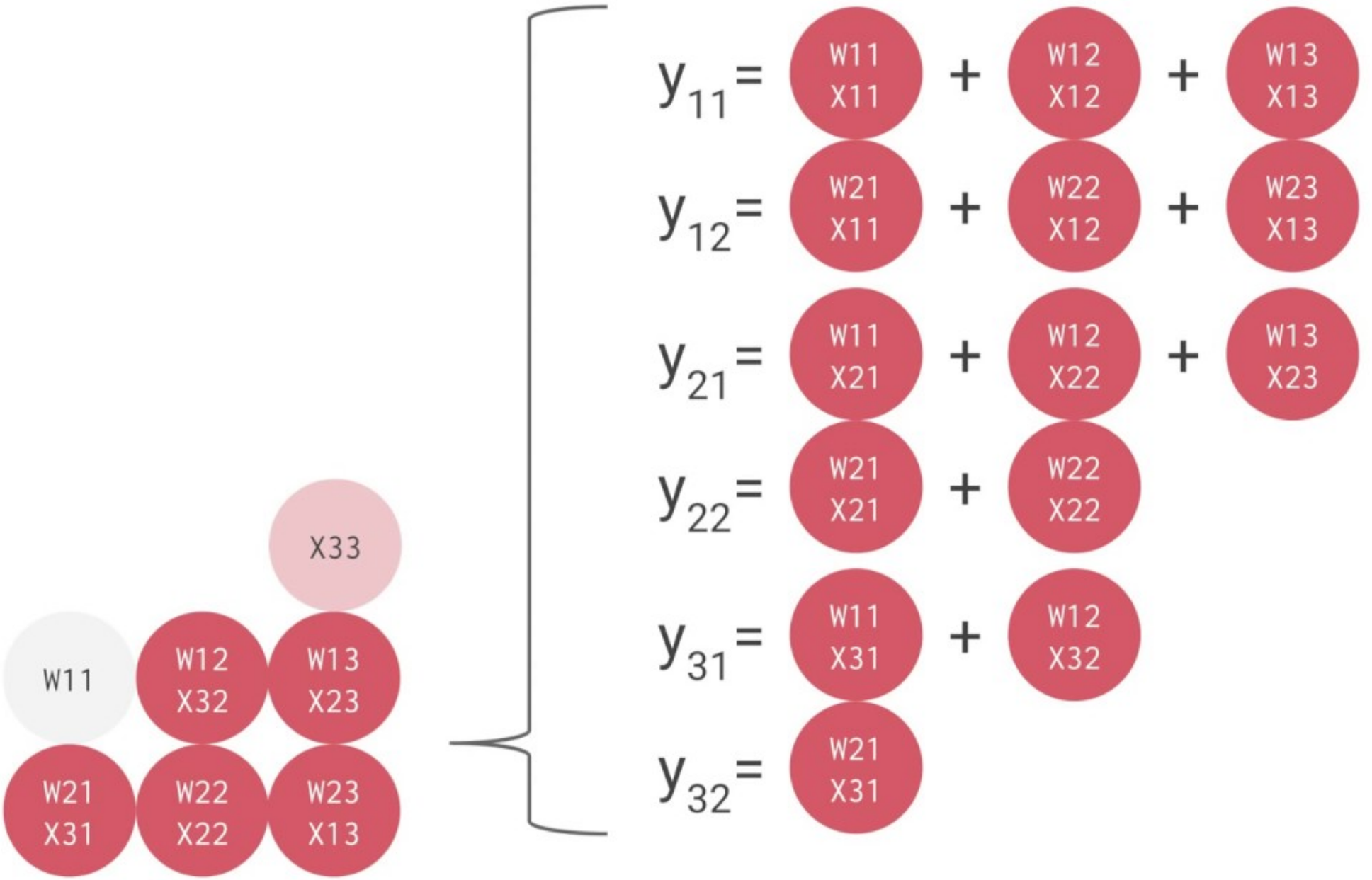
Systolic array

- A systolic array typically consists of a large monolithic network of primitive computing nodes (data processing units - DPUs) which can be hardwired or software configured for a specific application.
- Parallel input data enters this architecture flowing through the nodes that process it as it processes through the node as *waves*. The name of the architecture was picked as it resembles the systolic movement of the heart.
- Sometimes it is referred as an example of MISD but this is questionable as the nodes operate on different data.

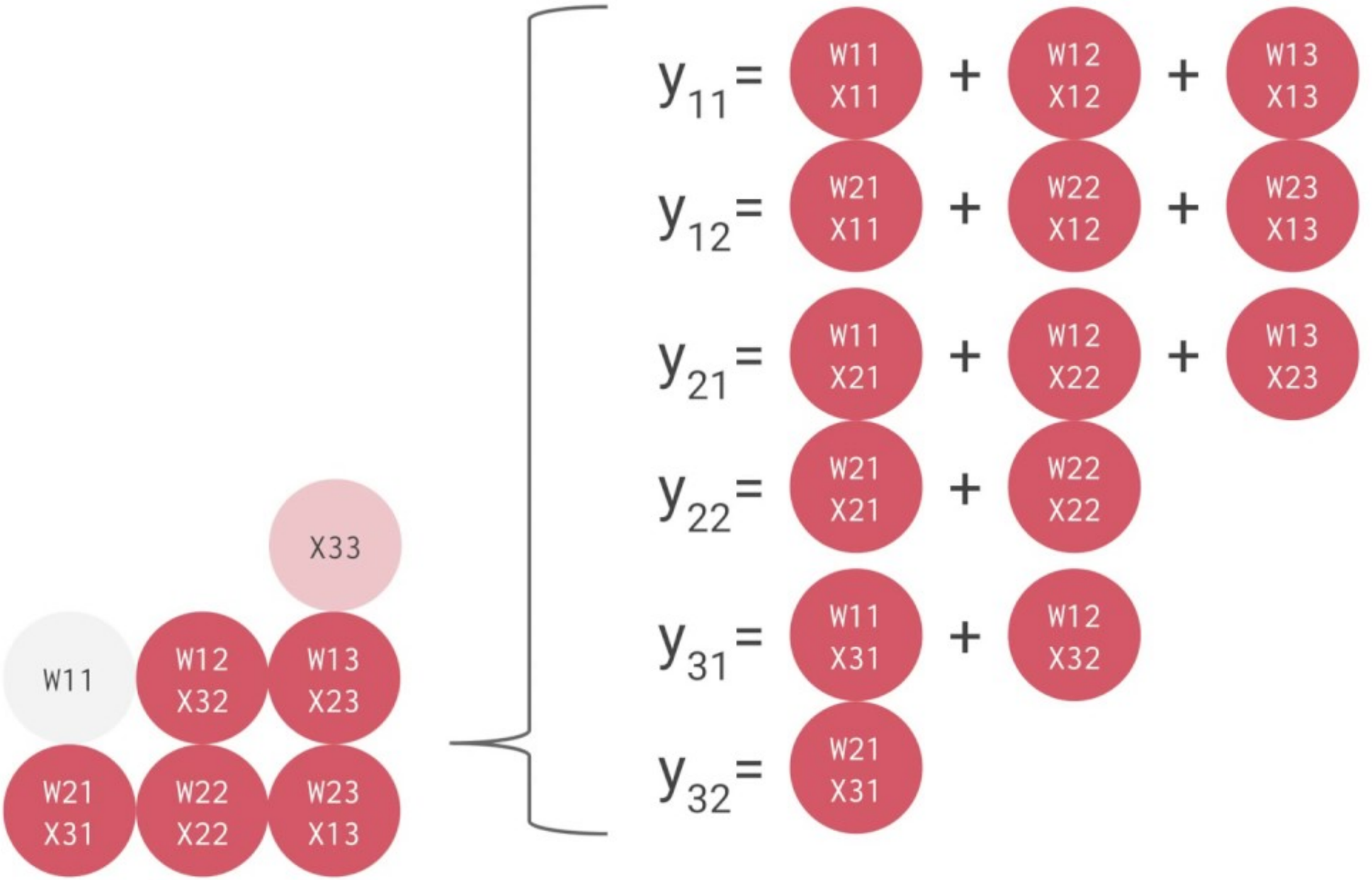
Systolic array

- Now commonly used to accelerate deep learning and computer vision tasks, implementing matrix-matrix multiplications
- Google TPUs use systolic arrays
- NVIDIA tensor cores (sort of)

Systolic array



Systolic array



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel architectures

RAM model

- Random Access Machine: is an abstract model for a sequential computer
- It models a device with an instruction execution unit and unbounded memory.
  - Memory stores program instructions and data.
  - Any memory location can be referenced in ‘unit’ time
  - The instruction unit fetches and executes an instruction every cycle and proceeds to the next instruction.
- Today’s computers depart from RAM, but function as if they match this model.

PRAM model

- Parallel Random Access Machine: abstract model for parallel computer

- It models a device with an unspecified number of instruction execution units and global memory of unbounded size that is uniformly accessible to all processors

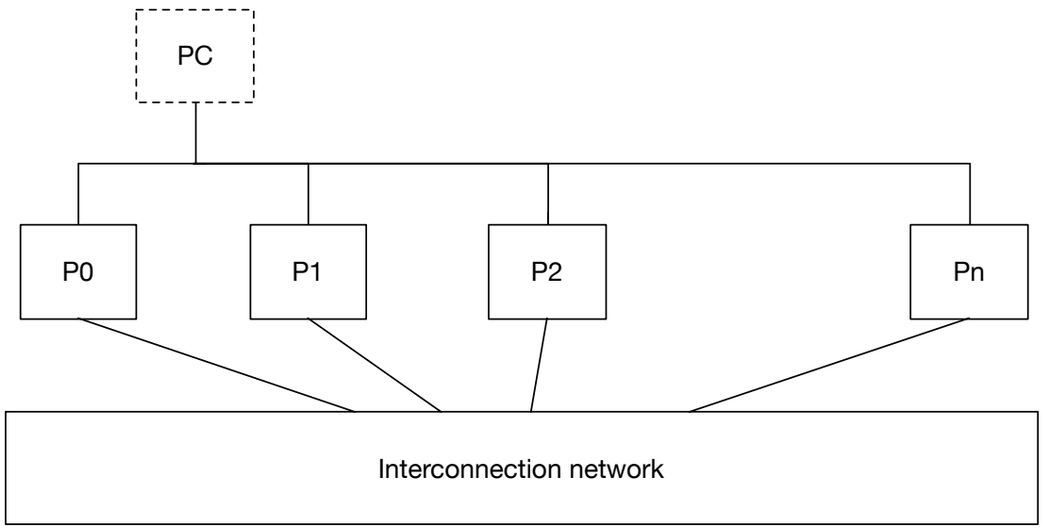
  - It fails by misrepresenting memory behavior.

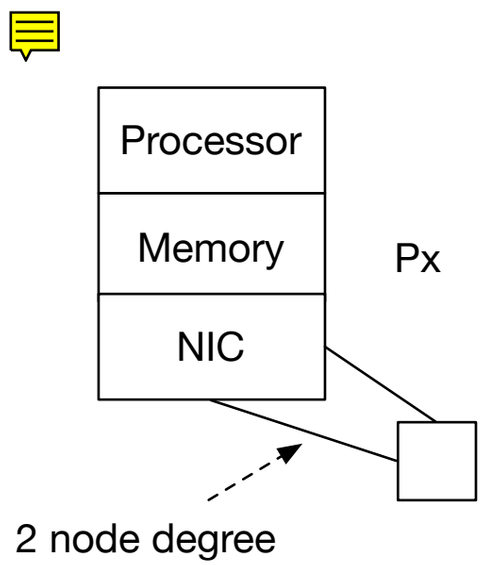
  - Impossible to realize the unit-time single memory image when multiple exec. units access the same location

- Bad memory modeling leads to wrong evaluation of algorithms: PRAM's performance predictions are not observed in real computers !

CTA model

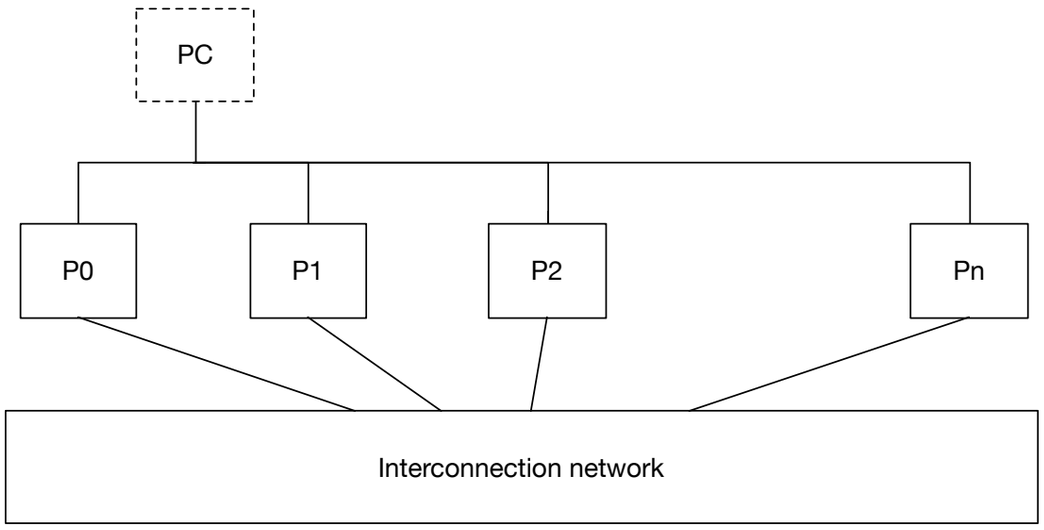
- Candidate Type Architecture: abstract model for parallel computer
- Explicitly separates two types of memory references: inexpensive local references and expensive non-local references

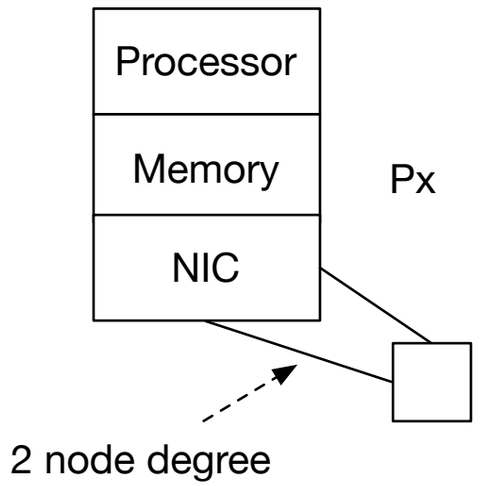




Both shared and distributed memory architectures described before are correctly modeled in this way

- Candidate Type Architecture: abstract model for parallel computer
- Explicitly separates two types of memory references: inexpensive local references and expensive non-local references





Memory Latency λ

- Memory Latency = delay required to make a memory reference. Non local memory latency is indicated with λ.

- Relative to processor's local memory latency, ≈ unit time ≈ one word per instruction

  - Variable, due to cache mechanisms etc.

- λ has values 2-5 orders of magnitude larger than local memory reference

- Sometimes it is better to recalculate globals locally (e.g. random number)

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Metrics for performance evaluation

- An application may operate on values organized as **streams** or as **single data values**.
  - A stream is a possibly infinite sequence of values with the same type (e.g. matrices of known size representing images)
- **Service time** is the average throughput of the stream, or the inverse of **processing bandwidth**.
- **Completion time** is the mean time needed to complete the computation on all the stream elements.
- **Latency** is defined as the mean time to process one stream element.
- When dealing with single data values completion time is meaningful, while service time is not meaningful.

Sequential module: service time, bandwidth, completion time

- Let us consider a single sequential module Σ operating on a stream
  - Σ implements *k* operations, each one with mean service time t_i and occurrence probability p_i
  - **service time** (mean) of Σ is $\mathbf{t} = \sum_{i=1,k} p_i\, t_i$
  - **processing bandwidth** (throughput) is the average number of operations executed by Σ in the time unit: **B** = 1 / **t**
  - **completion time** (mean) if the stream has length *m* is $\mathbf{t_c} = m\, t$
  - **latency** is the mean time needed to process one stream element. Since Σ is sequential then latency coincides with service time.

Parallel module: parallelism degree, latency, bandwidth, completion time

- Let us consider a parallel transformation of Σ into a parallel version Σ^n of n modules
- the **parallelism degree** of Σ^n is ***n***, independently from the effective capability of all modules to work in parallel at any time
  - parallelization consists in transforming Σ^1 into a Σ^n that is functionally equivalent
- **latency** is the mean time needed by Σ^n to process a single stream element
- **service time** $\mathbf{t}^n$ is the average time interval between the beginning of the executions on two consecutive stream elements.
- **processing bandwidth** is the average number of operations per time unit
  $\mathbf{B}^n = 1 / \mathbf{t}^n$
- **completion time** $\mathbf{t_c}^n$ is the average time to complete the execution of all stream elements. For m ≫ n the relation with service time is $\mathbf{t_c}^n \cong \boldsymbol{m\ t}^n$

Parallel module: parallelism degree, latency, bandwidth, completion time

While for a sequential system latency and service time are equal, **in parallel they differ**: the service time measures the average time interval after which Σ^n can accept a new input stream element without waiting for the output result of the previous element, i.e. without waiting the latency time.

Depending on the parallelism paradigm used, latency may increase w.r.t. sequential approach. The important parameter is service time, since it impacts on the completion time.

For example pipelining (as in the CPU) typically increases latency (because of the overhead in managing the pipeline stages) but a larger number of instructions per second are executed.

Scalability

- **Scalability** provides a measure of the relative speed of the *n*-parallel computation w.r.t the same computation with parallelism equal to 1:

- $S_{\Sigma}^{n} = B^{n} / B^{1} = t^{1} / t^{n}$

- It's hard to measure it because numerator and denominator should be evaluated in the same conditions. Sometimes t^{1} is evaluated using t^{s}, i.e. the corresponding sequential algorithm… but even if $t^{1} \cong t^{s}$ there is need to use the same parallel architecture used to execute t^{n}, and still there may be side effects of the architecture on the evaluation.

Speedup

- If it's not possible to guarantee that the same computational and architectural conditions can be applied consistently for scalability evaluations, then we use a weaker concept, called **speedup**:

- $S_P = t_s / t_P$

- where *P* is the number of processors, t_s is the completion time of the sequential algorithm and t_p is the completion time of the parallel algorithm

Speedup

- The speedup is perfect to **ideal** if $S_P = P$
- The speedup is **linear** if $S_P \cong P$
- The speedup is **superlinear** if, for some P, $S_P > P$

- $S_P = t_s / t_P$

- where P is the number of processors, t_s is the completion time of the sequential algorithm and t_p is the completion time of the parallel algorithm

Superlinear speedup

- Cache effects: when data is partitioned and distributed over *P* processors, then the individual data items are (much) smaller and may fit entirely in the data cache of each processor

- For an algorithm with linear speedup, the extra reduction in cache misses may lead to superlinear speedup

  - This is also one of the motivations for with scalability is harder to evaluate than speedup

Speedup limitations

- Several factors can limit the speedup
  - Processors may be idle
  - Extra computations are performed in the parallel version
  - Communication and synchronization overhead

Relative speedup

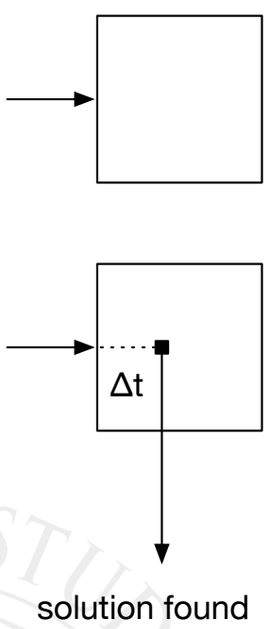
- The **relative speedup** is defined as $S^1_P = t_1 / t_P$ where t_1 is the execution time of the parallel algorithm on one processor

- Similarly, $S^k_P = t_k / t_P$ is the relative speedup with respect to k processors, where $k < P$

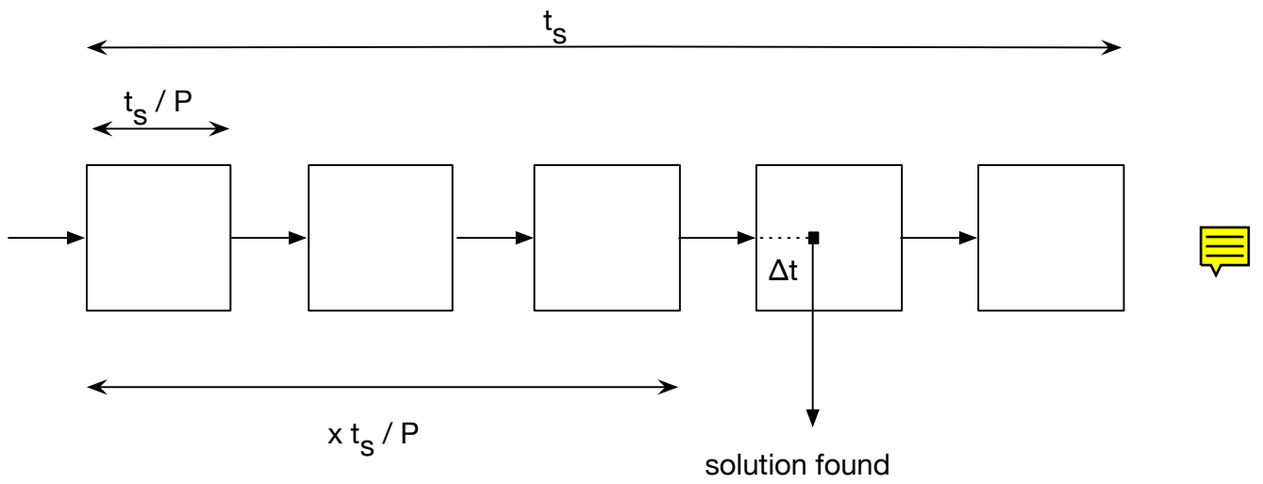
- The relative speedup S^k_P is used when k is the smallest number of processors on which the problem will run

Speedup: example

- Search in parallel by partitioning the search space into *P* chunks
- $S_P = ((x \times t_s/P) + \Delta t) / \Delta t$
- Worst case for sequential search (item in last chunk): $S_P \longrightarrow \infty$ as Δt tends to zero
- Best case for sequential search (item in first chunk): $S_P = 1$



Parallel search

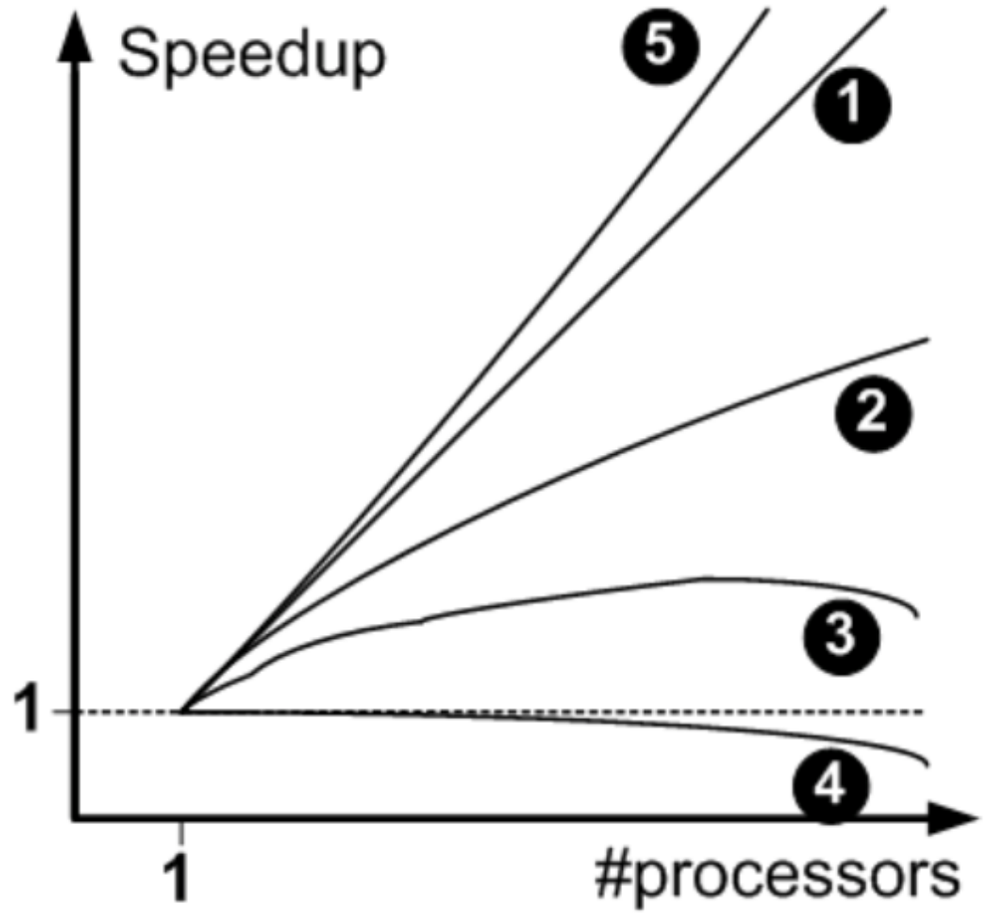


Sequential search

Speedup types

1. Ideal, linear speedup

2. Increasing, sub-linear speedup



3. Speedup with an optimal number of processors

4. No speedup

5. Super-linear speedup

Efficiency

- the efficiency of an algorithm using P processors is

  $E_P = S_P / P$

- Efficiency estimates how well-utilized the processors are in solving the problem, compared to how much effort is lost in idling and communication/synchronization

- Ideal (or perfect) speedup means 100% efficiency $E_P = 1$

- Many difficult-to-parallelize algorithms have efficiency that approaches zero as *P* increases

Work

- The work of an algorithm corresponds to the total number of primitive operations performed by an algorithm.

- If running on a sequential machine, it corresponds to the sequential time.

- On a parallel machine, however, work can be split among multiple processors and thus reduce the time.

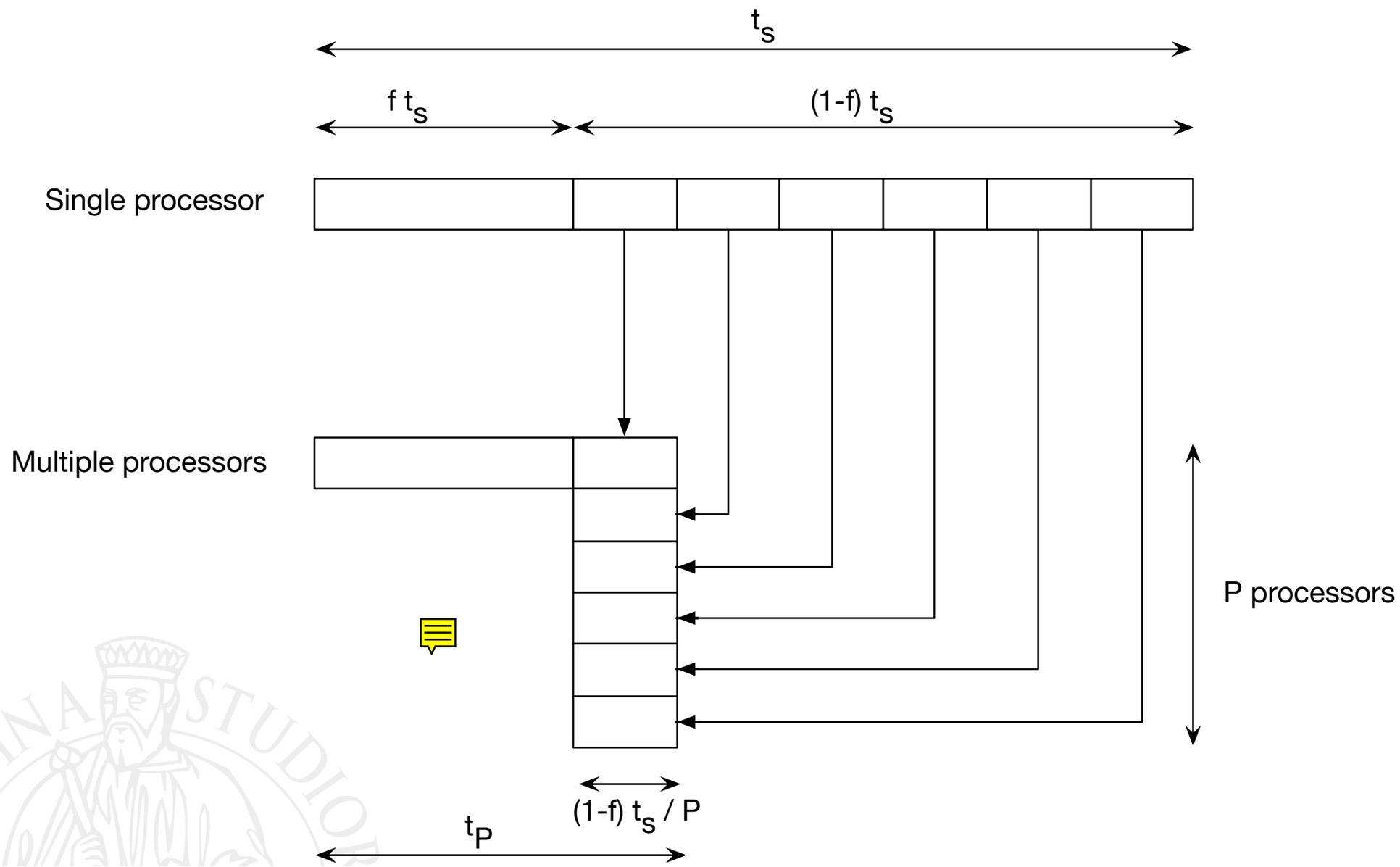
Amdahl's Law

- In 1967, Gene Amdahl pointed out that every algorithm consists of a part that can be done in parallel and a part that cannot be, usually due to things such as data dependencies.

- I.e. there's a limited amount of parallelism in a computer program: once it has been exploited there's no additional benefit in adding more parallelism: a program can never run more quickly than its sequential part.

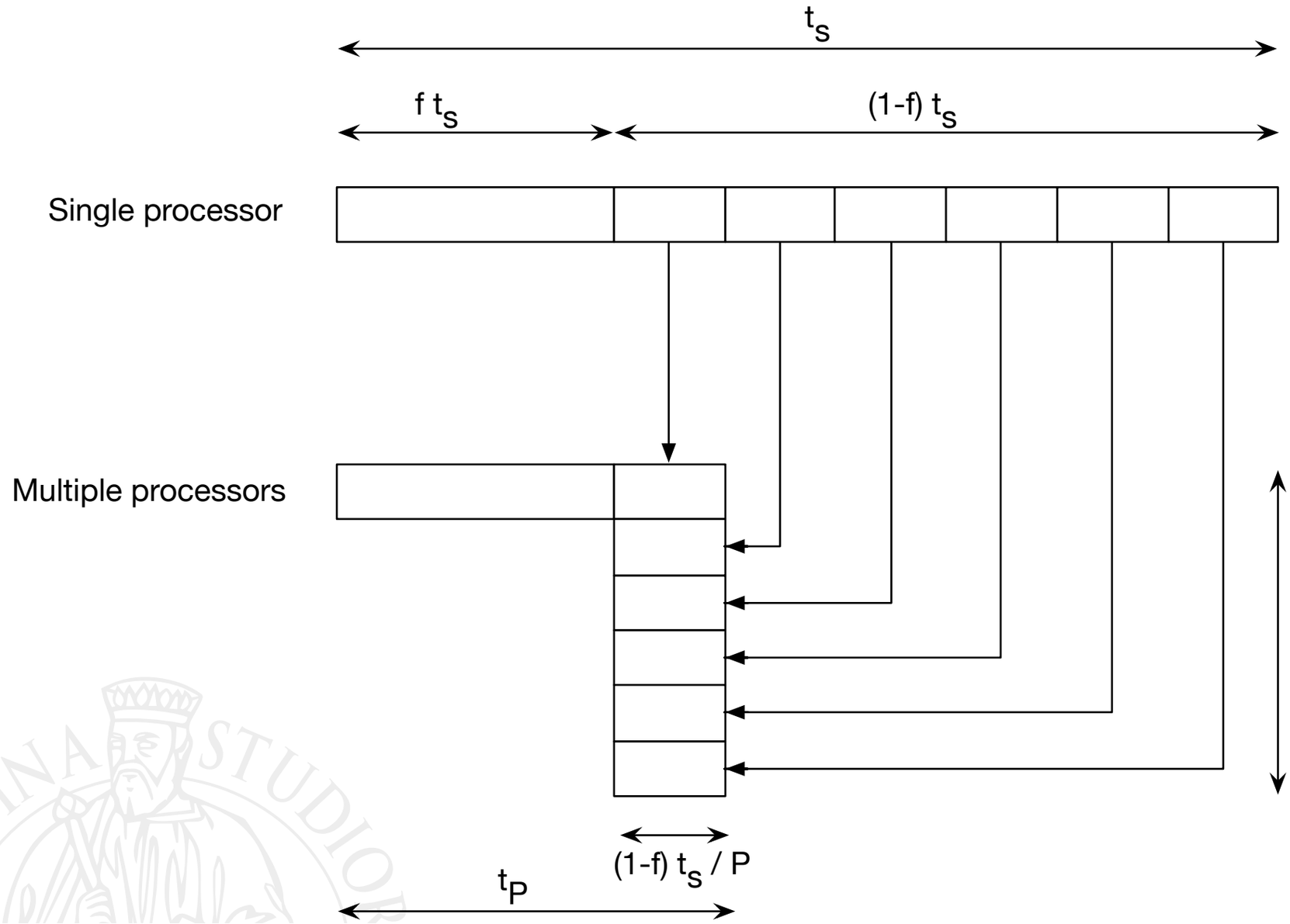
Amdahl's Law

- Let *f* be the fraction of the computation that is sequential and cannot be divided into concurrent tasks



Amdahl's Law

- Let *f* be the fraction of the computation that is sequential and cannot be divided into concurrent tasks



$(1-f)\ t_s / P$ is an underestimate: the parallel code may require more instructions than its sequential counterpart

Amdahl's Law

- Amdahl's law states that the speedup given *P* processors is
  $S_P = t_s / (f \times t_s + (1-f)t_s / P) = P / (1 + (P-1) f)$

- As a consequence, the maximum speedup is limited by $S_P \rightarrow f^{-1}$ as $P \rightarrow \infty$

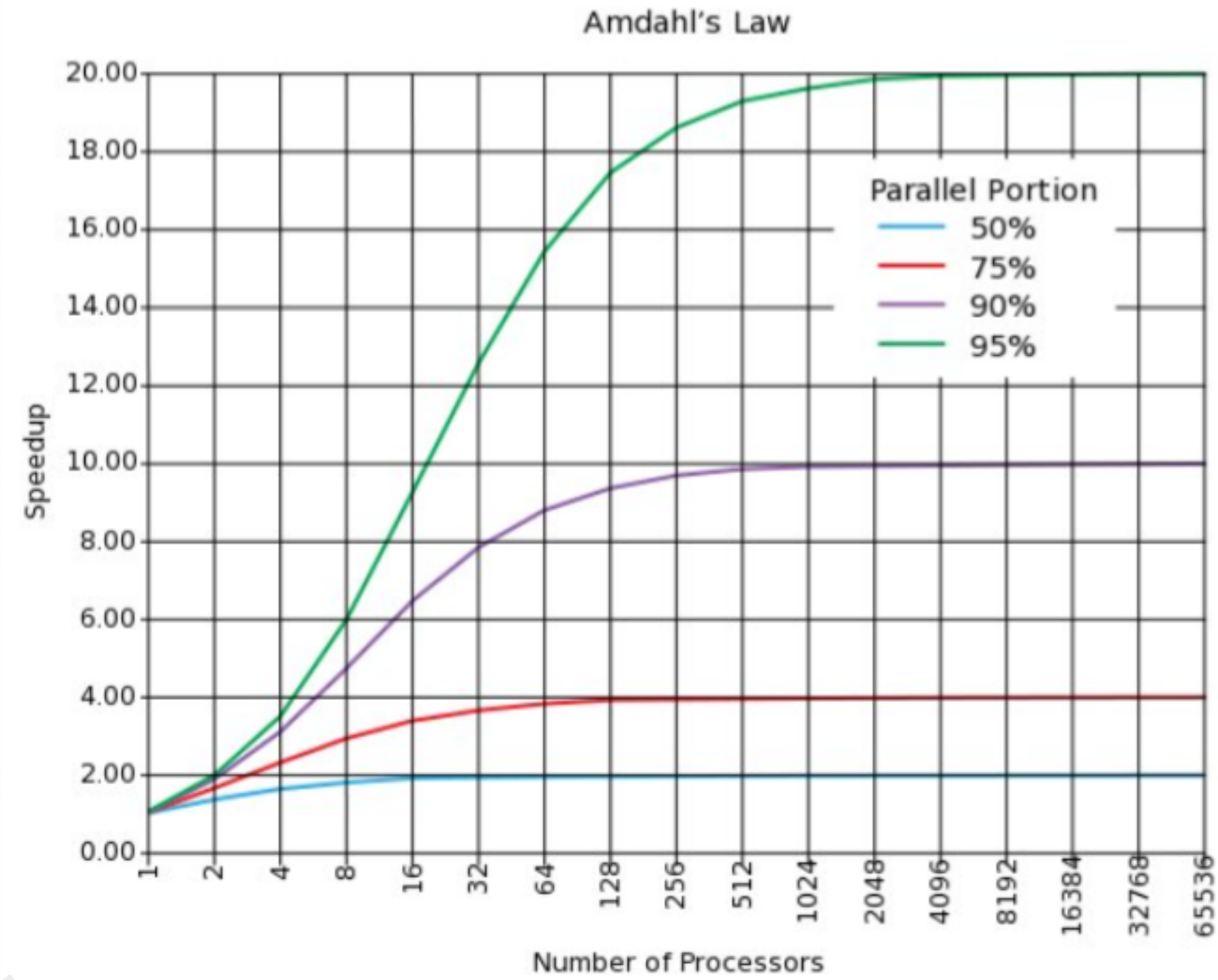
- Even if 95% of a program is parallelisable, you will never see a speed-up of more than 20 times.

Amdahl's Law

- Amc
  proc
  S_P =

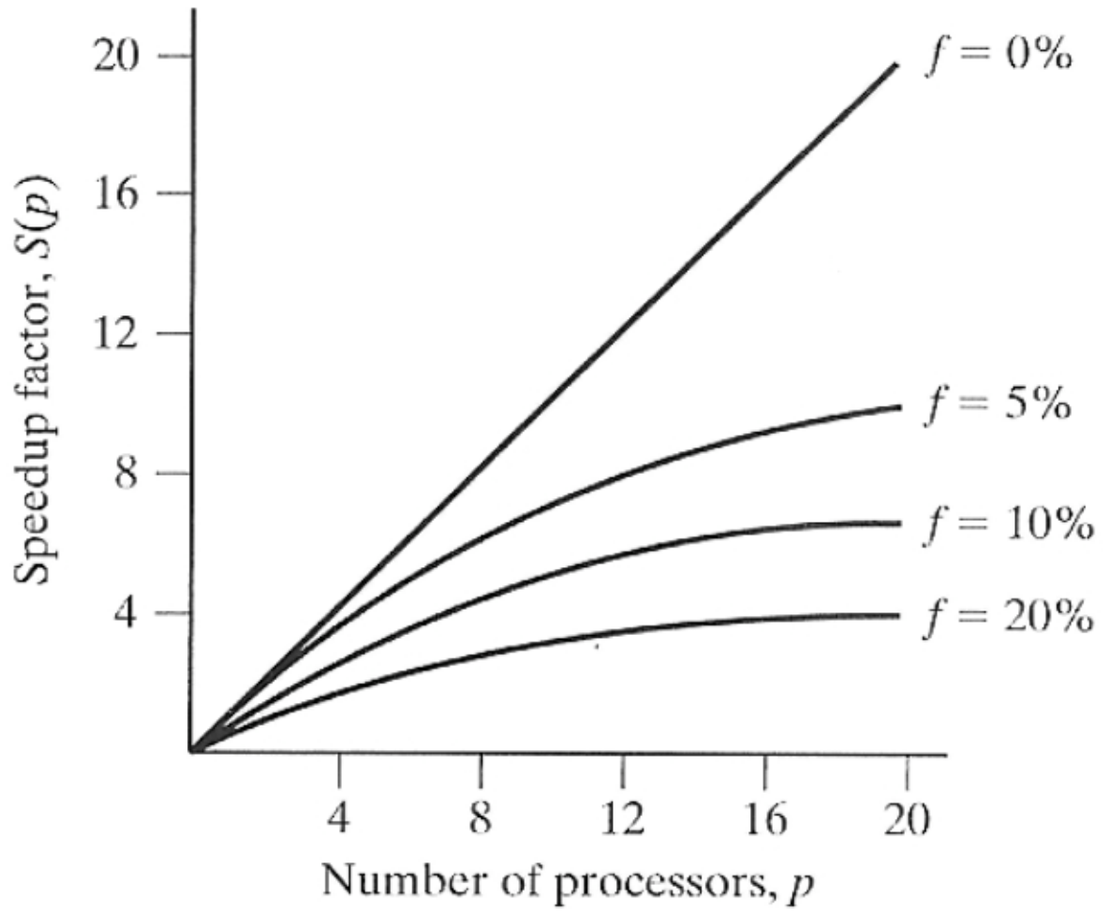
- As a
  limit

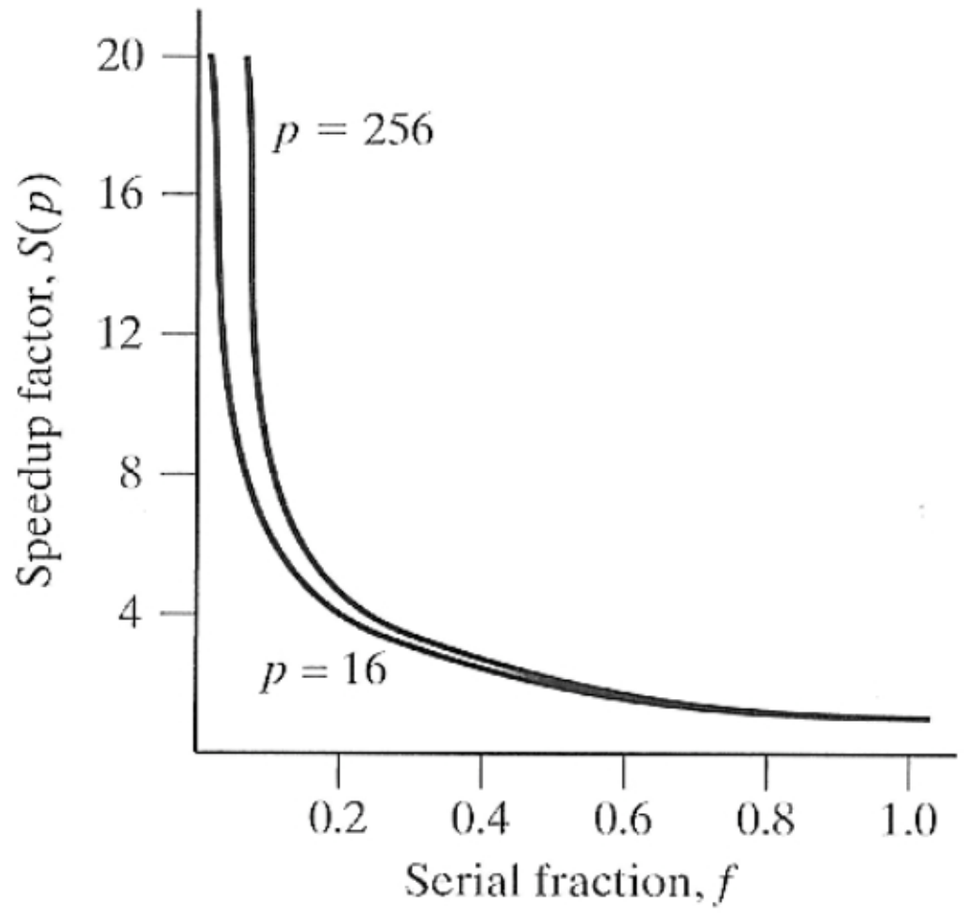
- Eve
  neve



u will

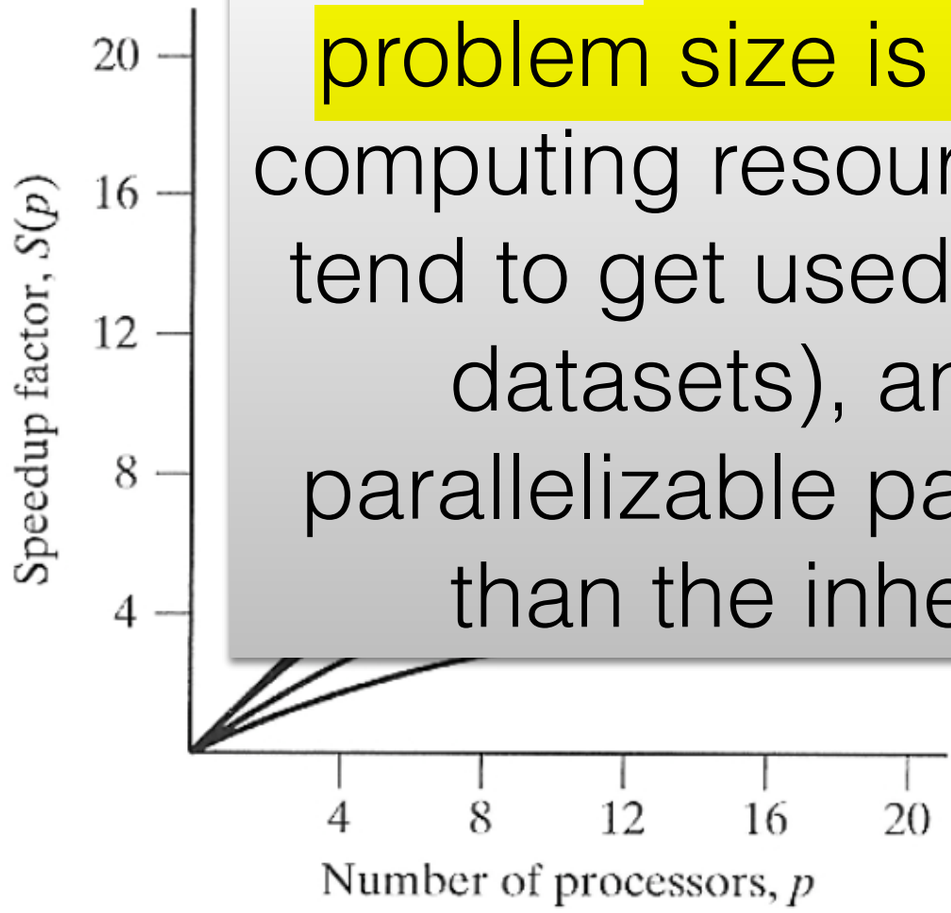
Amdahl's Law

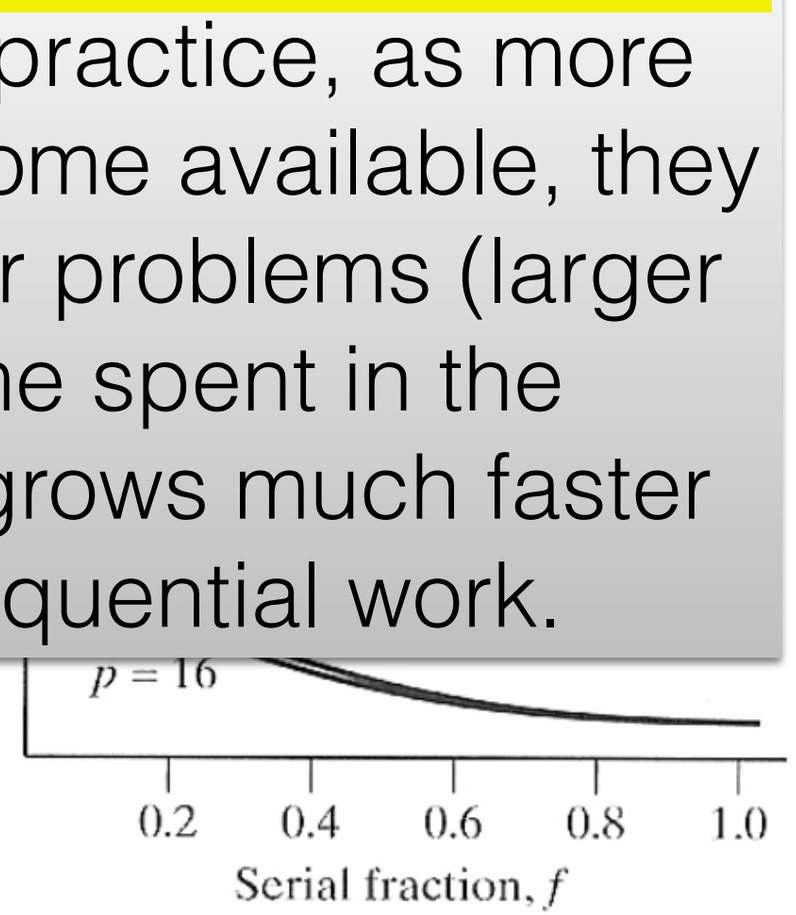

20
16
12
8
4
Speedup factor, S(p)
f = 0%
f = 5%
f = 10%
f = 20%
4
8
12
16
20
Number of processors, p


20
16
12
8
4
Speedup factor, S(p)
p = 256
p = 16
0.2
0.4
0.6
0.8
1.0
Serial fraction, f

Amdahl’s Law

Amdahl's law only applies to cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently sequential work.





Scaling example

- Workload: sum of 10 scalars, and10×10 matrix sum
  - Single processor: Time = (10+100)×t_{add}
- Speed up from 10 to 100 processors
  - 10 processors: Time = 10×t_{add} + 100/10×t_{add} = 20×t_{add}
    Speedup = 110/20 = 5.5 (55% of potential)
  - 100 processors: Time=10×t_{add} +100/100×t_{add} = 11×t_{add}
    Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling example

- Workload: sum of 10 scalars, and10×10 matrix sum
  - Single processor: Time = $(10+100)\times t_{add}$
- Speed up from 10 to 100 processors
  - 10 processors: Time = $10\times t_{add} + 100/10\times t_{add} = 20\times t_{add}$
    Speedup = 110/20 = 5.5 (55% of potential)
  - 100 processors: Time=$10\times t_{add} +100/100\times t_{add} = 11\times t_{add}$
    Speedup = 110/11 = 10 (10% of potential)

Strong scaling represents the time to solution with respect to the number of processors for a fixed total size.

Amdahl's Law

- Amdahl's Law describes a fact that applies to an instance of a computation. Once we have fixed the instance it considers the effects of increasing parallelism.

- Most parallel computations fix the parallelism and expand the size of the instance: in this case the proportion of sequential code often diminishes as larger instances are considered.

- Increasing the problem size may increase the sequential portion negligibly, making a larger part of the problem amenable to parallelism.

Amdahl’s Law

- Amdahl’s Law describes a fact that applies to an instance of a computation. Once we have fixed the

More than two decades after the appearance of Amdahl's Law, John Gustafson noted that several programs at Sandia National Labs were speeding up by over 1000X. Clearly, Amdahl's Law could be evaded.

- Most parallel computations fix the parallelism and expand the size of the instance: in this case the proportion of sequential code often diminishes as larger instances are considered.

- Increasing the problem size may increase the sequential portion negligibly, making a larger part of the problem amenable to parallelism.

Scaling example - cont

- What if matrix size becomes 100×100?
- Single processor: Time = $(10+10000)\times t_{add}$
- 10 processors: Time = $10\times t_{add} + 10000/10\times t_{add} = 1010\times t_{add}$
  Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors: Time = $10\times t_{add} + 10000/100\times t_{add} = 110\times t_{add}$
  Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Scaling example - cont

- What if matrix size becomes 100×100?
- Single processor: Time = $(10+10000)\times t_{add}$
- 10 processors: Time = $10\times t_{add} + 10000/10\times t_{add}$ = $1010\times t_{add}$
  Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors: Time = $10\times t_{add} + 10000/100\times t_{add}$ =$110\times t_{add}$
  Speedup = 10010/110 = 91 (91% of potential)

Weak scaling represents the time to solution with respect to the number of processors for a fixed-sized problem per processor.

Scalable parallelism

- Scalable parallelism in software refers to the ability to efficiently distribute and process tasks across multiple computing units (e.g. CPU cores or GPUs) while maintaining or improving performance as the number of processing units increases.

- If we can make use of additional processors to handle larger problems we have scalable parallelism.

Scaling and Efficiency

- Suppose we have a program that can be parallelized, but with a 20% overhead that can not be parallelized. If the sequential computation time for the whole algorithm is t_S then the parallel computation time for P processors is:

$t_P = t_S / P + 0.2\ t_S$

and the efficiency is

$E_P = t_P / t_S / P$

Efficiency for 10 processors is 0.33 and for 100 processors is 0.047 !

Scaling and Efficiency

- Suppose we have a program that can be parallelized, but with a 20% overhead that can not be parallelized. If the sequential computation time for the whole algorithm is t_S then the parallel cc
is:

$t_P = t_S / P + 0.2\ t_S$

and the efficiency is

$E_P = t_P / t_S / P$

Efficiency for 10 processors is 0.33 and for 100 processors is 0.047 !

Marginal benefit of adding processors decreases as the # of processors increases.

Solutions:

- reduce overhead
- use slower cores: the marginal benefit of improving processors' speed is minimal. IBM BlueGene has thousands of CPUs but with relatively limited clock rate.

Scaling and Efficiency

- Suppose we have a program that can be parallelized, but with a 20% overhead that can not be parallelized. If the sequential computation time for the whole algorithm is t_S then the parallel cc
is:

$t_P = t_S / P + 0.2\ t_S$

and the efficiency is

$E_P = t_P / t_S / P$

Efficiency for 10 processors is 0.33 and for 100 processors is 0.047 !

Marginal benefit of adding processors decreases as the # of processors increases.

Solutions:

- reduce overhead
- use slower cores: the marginal benefit of improving processors' speed is minimal. IBM BlueGene has thousands of CPUs but with relatively limited clock rate.

Typically efficiency figures are low

Gustafson's Law

Amdahl's law is based on a fixed workload or fixed problem size per processor, i.e. analyzes constant problem size scaling

Gustafson's law defines the **scaled speedup** by keeping the parallel execution time constant (i.e. time-constrained scaling) by adjusting P as the problem size N changes

$$S_{P,N} = P + (1\text{-}P)\alpha(N)$$

If $\alpha(N)$ is small then the speedup is almost P !!

where $\alpha(N)$ is the non-parallelizable fraction of the *normalized* parallel time $t_{P,N} = 1$ given problem size N

To see this, let $\beta(N) = 1\text{-}\ \alpha(N)$ be the parallelizable fraction (overhead is ignored) and keep the parallel execution time constant:

$$t_{P,N} = \alpha(N) + \beta(N) = 1$$

then, the scaled sequential time is

$$t_{s,N} = \alpha(N) + P\ \beta(N)$$

$\beta(N)$ is being executed sequentially so the parallel part takes P$\beta(N)$ to execute

giving

$$S_{P,N} = \alpha(N) + P\ (1\text{-}\ \alpha(N)) = P + (1\text{-}P)\alpha(N)$$

…speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size.
— John Gustafson

Amdahl's law is based on a fixed workload or fixed problem size per processor, i.e. analyzes constant problem size scaling

Gustafson's law defines the **scaled speedup** by keeping the parallel execution time constant (i.e. time-constrained scaling) by adjusting P as the problem size N changes

$$S_{P,N} = P + (1\text{-}P)\alpha(N)$$

If $\alpha(N)$ is small then the speedup is almost P !!

where $\alpha(N)$ is the non-parallelizable fraction of the *normalized* parallel time $t_{P,N} = 1$ given problem size N

To see this, let $\beta(N) = 1\text{-}\ \alpha(N)$ be the parallelizable fraction (overhead is ignored) and keep the parallel execution time constant:

$$t_{P,N} = \alpha(N) + \beta(N) = 1$$

then, the scaled sequential time is

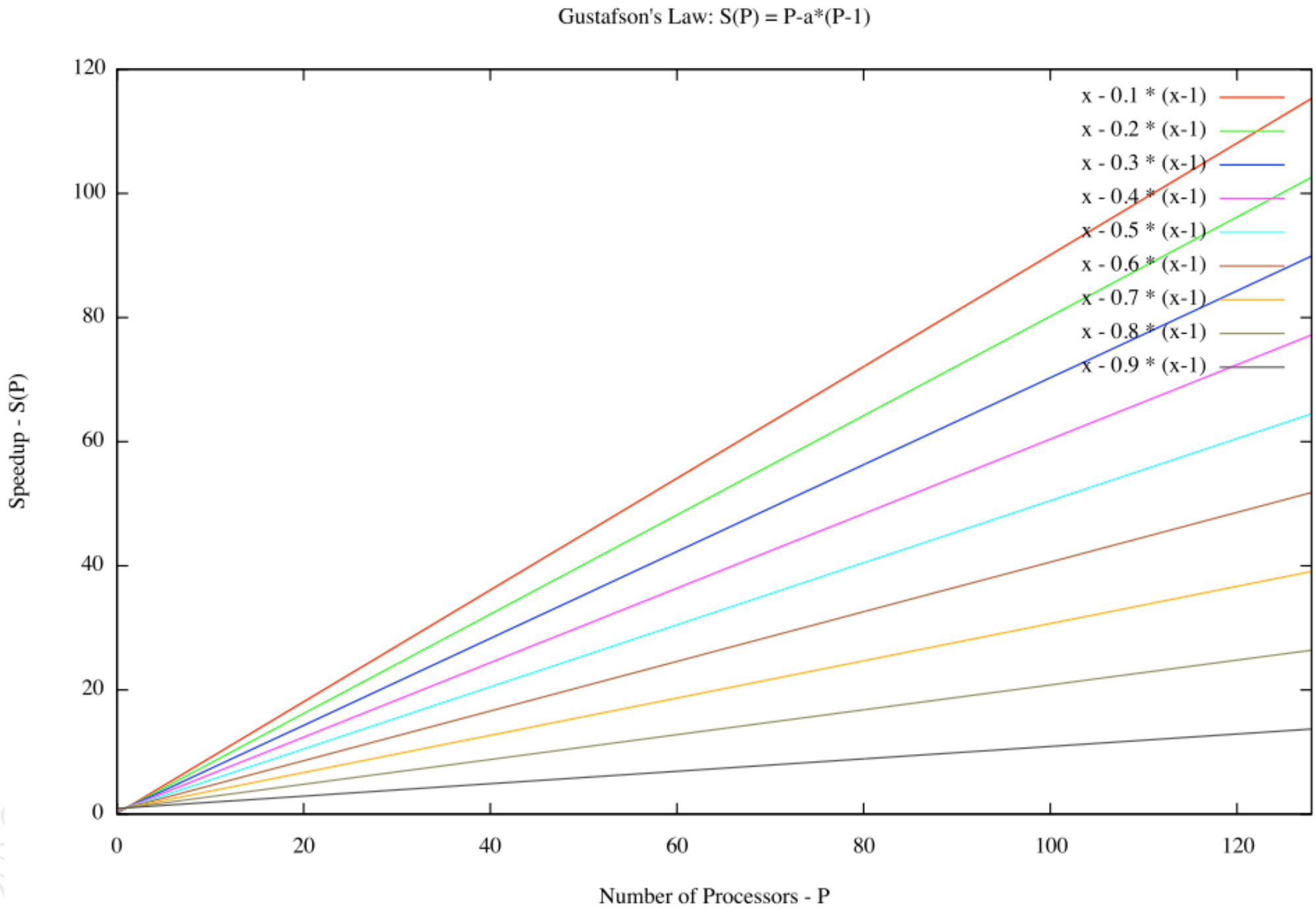
$$t_{s,N} = \alpha(N) + P\ \beta(N)$$

$\beta(N)$ is being executed sequentially so the parallel part takes P$\beta(N)$ to execute

giving

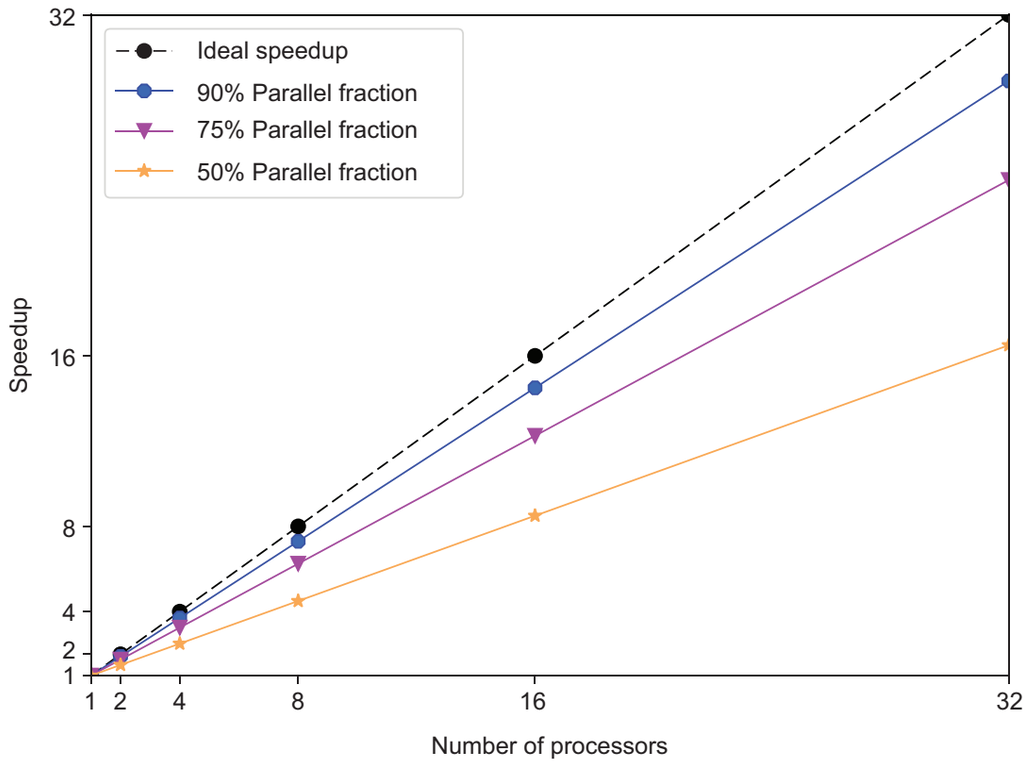
$$S_{P,N} = \alpha(N) + P\ (1\text{-}\ \alpha(N)) = P + (1\text{-}P)\alpha(N)$$

Gustafson's Law



Gustafson's Law

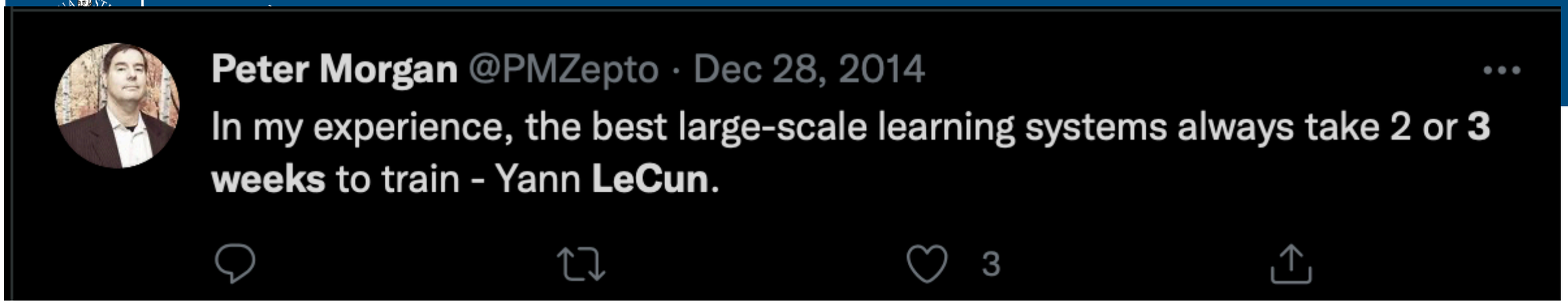
- Let's see another view of Gustafson's law:
- If the size of the problem grows proportionally to the number of processors, *N* is the number of processors, and *S* is the serial fraction of the code



- $S_N = N - S\,(N - 1)$

Amdahl vs. Gustafson

- The goal for parallelization may be to
  - make a program run faster with the same workload (reflected in Amdahl's Law)
- or
  - to run a program in the same time with a larger workload (Gustafson's Law).



- The goal for parallelization may be to
  - make a program run faster with the same workload (reflected in Amdahl's Law)
- or
  - to run a program in the same time with a larger workload (Gustafson's Law).

Peter Morgan @PMZepto · Dec 28, 2014

In my experience, the best large-scale learning systems always take 2 or **3 weeks** to train - Yann **LeCun**.

3

Yann LeCun @ylecun · Jul 13

ML researchers:
Late 1990s: "Method X is worthless because the Matlab code takes more than 20 minutes to converge"
Early 2020s: "Method X is great because with <favorite_DL_framework>, I can train it on 10 billion samples using 1000 {GPU,TPU}s in less than a week."

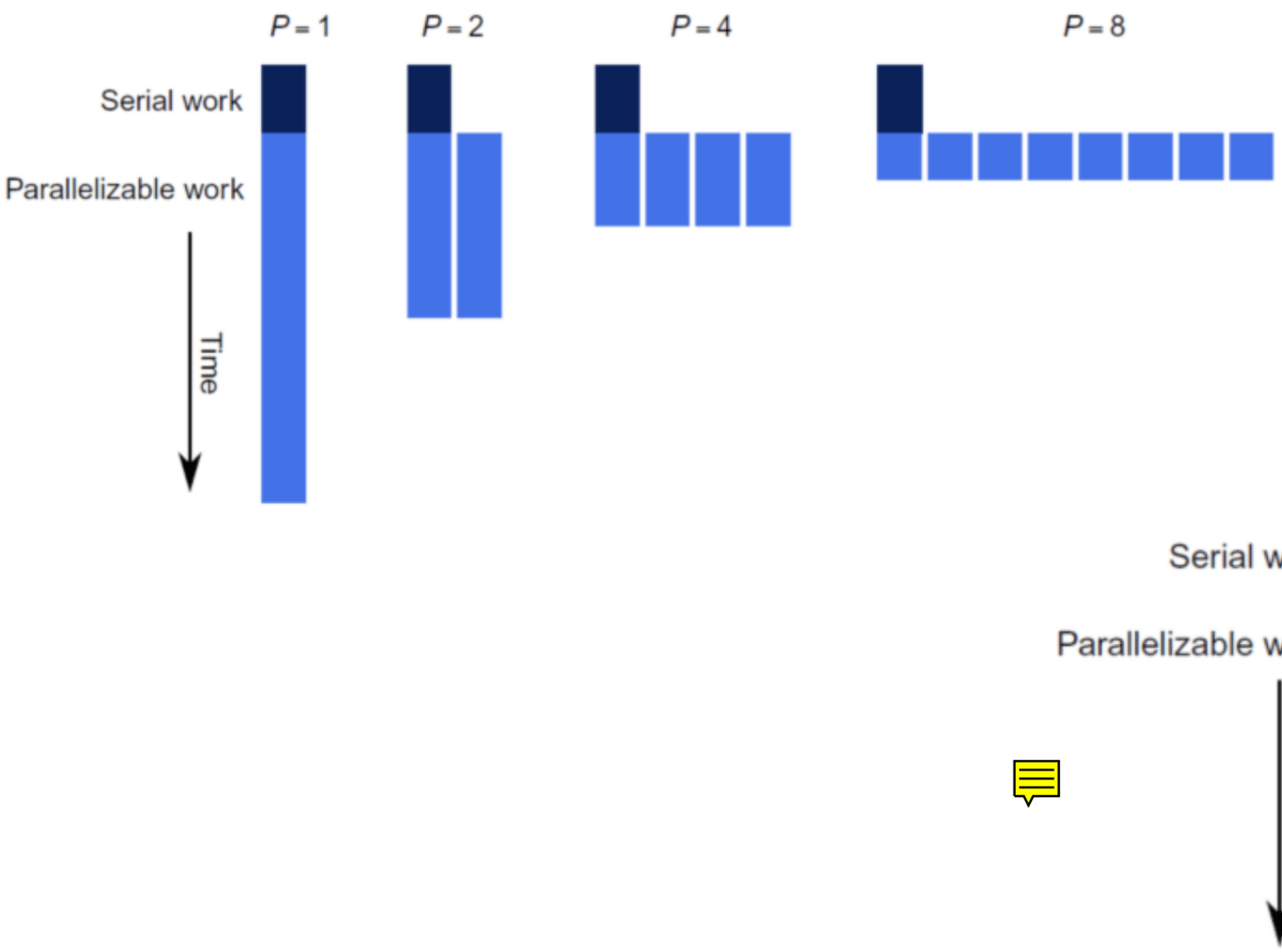
8 197 1.4K

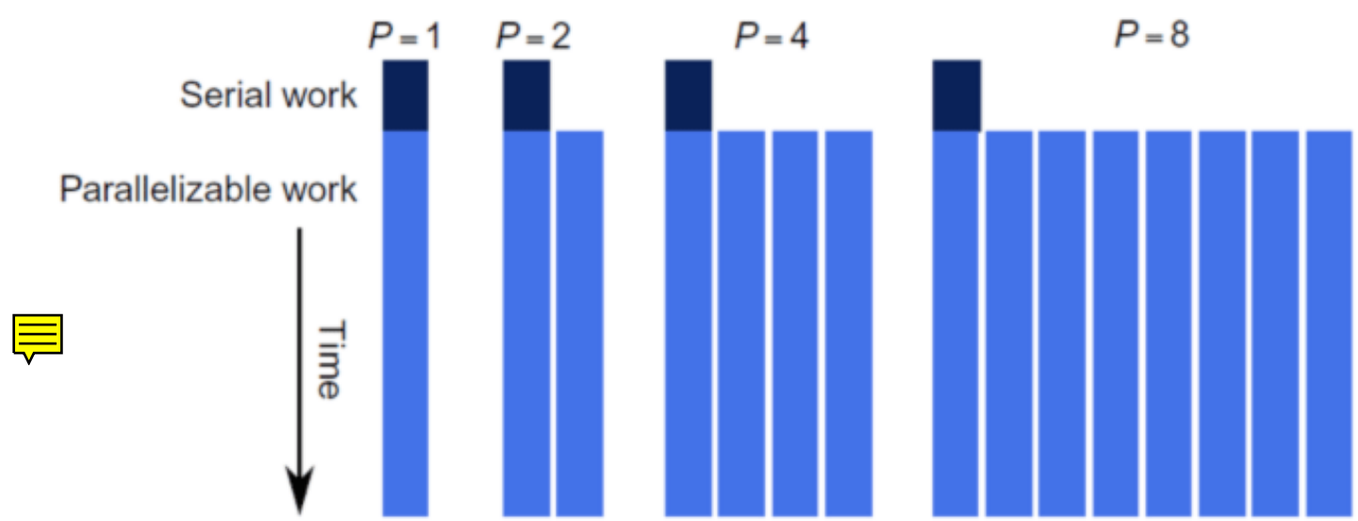
Yann LeCun @ylecun · Jul 13

Note: even in the late 1980s, I was in the "Early 2020s" category.
The time it took to train the "ConvNet du jour" on the "problem du jour" has consistently been around 10 days over the last 33 years.
This is independent of compute power and data.

7 14 223

Amdahl vs. Gustafson





- Nowadays, programs attack and solve larger, more complex problems, so Gustafson's observations fit the historical trend. Nevertheless, Amdahl's Law still haunts you when you need to make an application run faster on the same workload to meet some latency target.

Scaling example

- Number of processors proportional to problem size

- 10 processors, 10 × 10 matrix:
  Time = $20 \times t_{add}$

- 100 processors, 32 × 32 matrix:
  Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$

Memory scalability

- The traditional focus is on the run-time scaling, but memory scaling is often more important.

- Let's consider an application that works on a dataset that can be partly distributed (D) and partly replicated (R) on the computing nodes.
What happens as the problem size grows and we try to tackle this adding more processors ?

  - Soon there is not enough memory on a processor for the job to run.

  - Limited run-time scaling means the job runs slowly; limited memory scaling means the job can't run at all !

Memory scalability

Memory sizes for weak scaling with replicated and distributed arrays

	1 proc	2 proc		4 proc	
	Proc 0	Proc 0	Proc 1	Proc 0	Proc 1
Array R	100 MB	200 MB	200 MB	400 MB	400 MB
Array D	100 MB	100 MB	100 MB	100 MB	100 MB
				Proc 2	Proc 3
				400 MB	400 MB
				100 MB	100 MB

Array R – Array is replicated (copied) to every processor
Array D – Array is distributed across processors

- Soon there is not enough memory on a processor for the job to run.
- Limited run-time scaling means the job runs slowly; limited memory scaling means the job can’t run at all !

Sources of performance loss

- **Overhead**: cost incurred in the parallel solution but not in the sequential solution (e.g. set up processes and threads, tear down)
  - Communication: major component of overhead.
  - Synchronization
  - Computation: extra-work required by parallel computation, like figuring out which part of the data to process
- Memory: Memory hierarchy forms a barrier to performance when locality is poor
  - Temporal locality
    Same memory location accessed frequently and repeatedly
    Poor temporal locality results from frequent access to fresh new memory locations
  - Spatial locality
    Consecutive (or “sufficiently near”) memory locations are accessed
    Poor spatial locality means that memory locations are accessed in a more random pattern

Sources of performance loss

- Non-Parallelizable code
- According to Amdahl's law efficient execution of the non-parallel fraction *f* is extremely important
- We can reduce f by improving the sequential code execution (e.g. algorithm initialization parts), I/O, communication, and synchronization

Sources of performance loss

- Contention: competing for shared resources

- Idle time: often a consequence of synchronization and communication issues

  - bad load balance

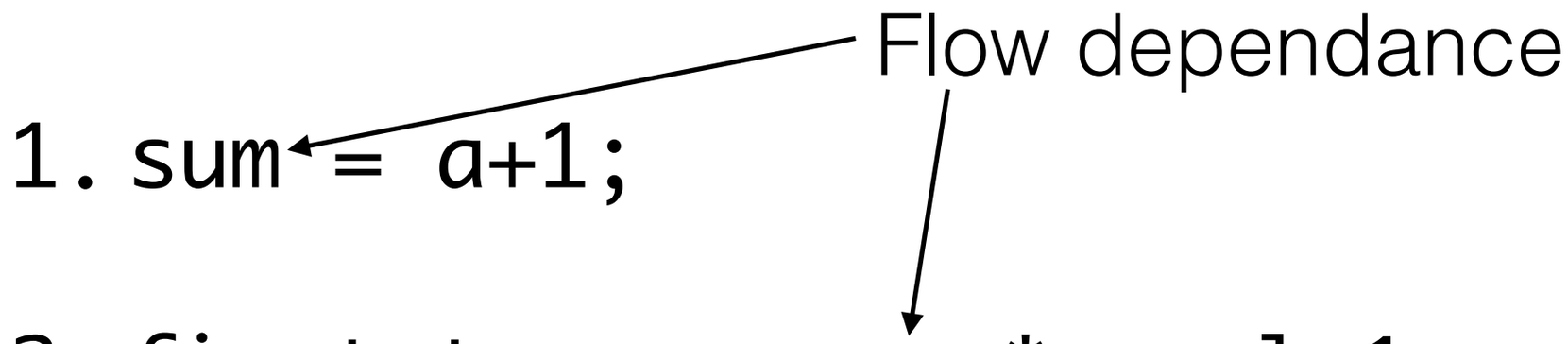
Improving performance

- Address data dependences: i.e. ordering of memory operations that must be preserved to maintain correctness
  - Flow dependance: read after write
  - Anti dependance: write after read
  - Output dependance: write after write
  - Input dependance: read after read

Data dependences

```
1. sum = a+1;

2. first_term = sum * scale1;

3. sum = b+1;

4. second_term = sum * scale2;
```

Data dependences




```
1. sum = a+1;

2. first_term = sum * scale1;

3. sum = b+1;

4. second_term = sum * scale2;
```

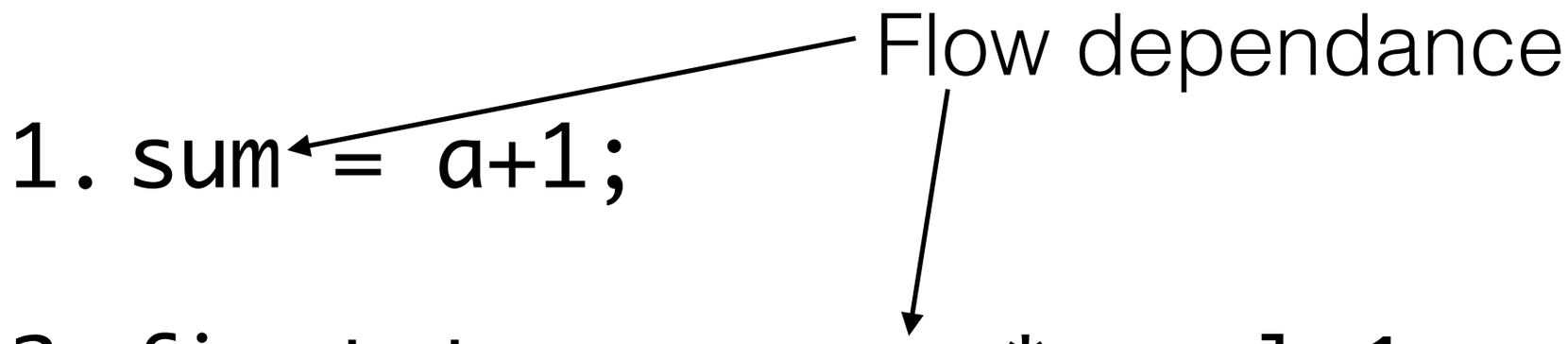
Data dependences

Flow dependance

```
1. sum = a+1;

2. first_term = sum * scale1;

3. sum = b+1;

4. second_term = sum * scale2;
```

Flow dependance

Data dependences




```
1. sum = a+1;

2. first_term = sum * scale1;

3. sum = b+1;

4. second_term = sum * scale2;
```

Data dependences

```
1. sum = a+1;

2. first_term = sum * scale1;

3. sum = b+1;

4. second_term = sum * scale2;
```

Data dependences

```
1. sum = a+1;

2. first_term = sum * scale1;

3. sum = b+1;

4. second_term = sum * scale2;
```

Span

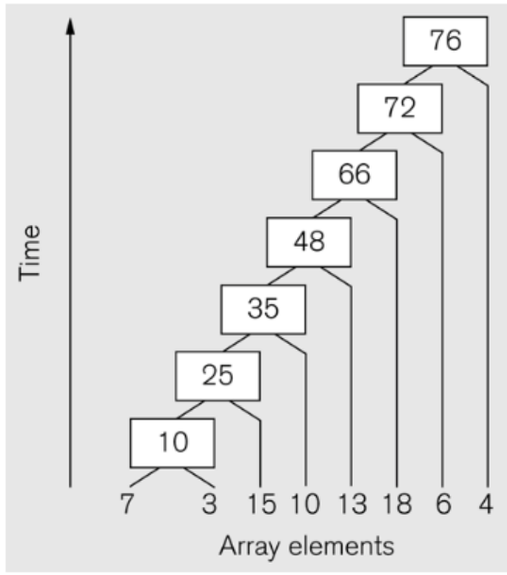
- The *span* is the length of the longest series of operations that have to be performed sequentially due to data dependencies (the critical path).

- The span may also be called the critical path length or the depth of the computation.
  Minimizing the span is important in designing parallel algorithms, because the span determines the shortest possible execution time.

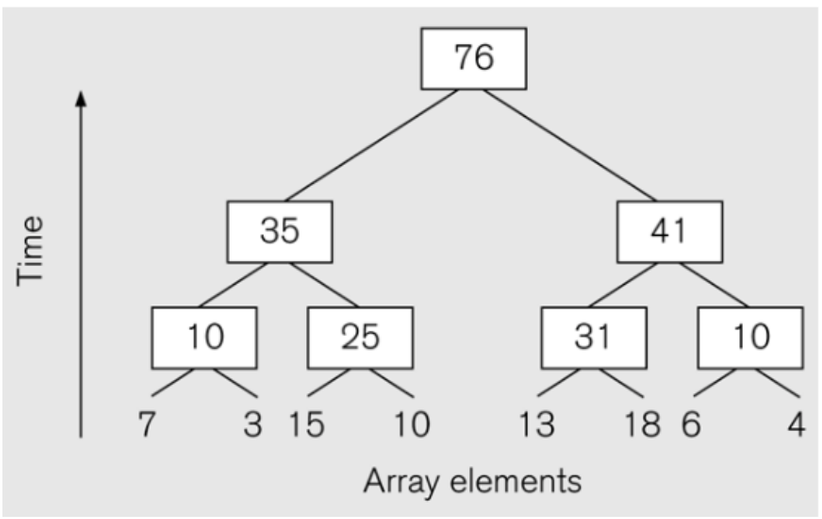
Example: iterative sum

```
sum=0;

for (int i=0;i<n;i++)

  sum+=x[i];
```

- A shorter chain of flow dependences improves parallelization



Serial



Parallelized version

Improving performance

- Implement a correct granularity of parallelism
  - Coarse: threads and processes infrequently depends on data or events of other threads and processes.
  - Fine: frequent interactions.
- Implement locality, either temporal or spatial
  - from memory latency of CTA: **locality rule** - fast programs maximize number of local memory references
  - operate on blocks of data

Improving performance

- Identify the program's **hotspots**:
  - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
  - Profilers and performance analysis tools can help here
  - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

Improving performance

- Identify **bottlenecks** in the program:
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

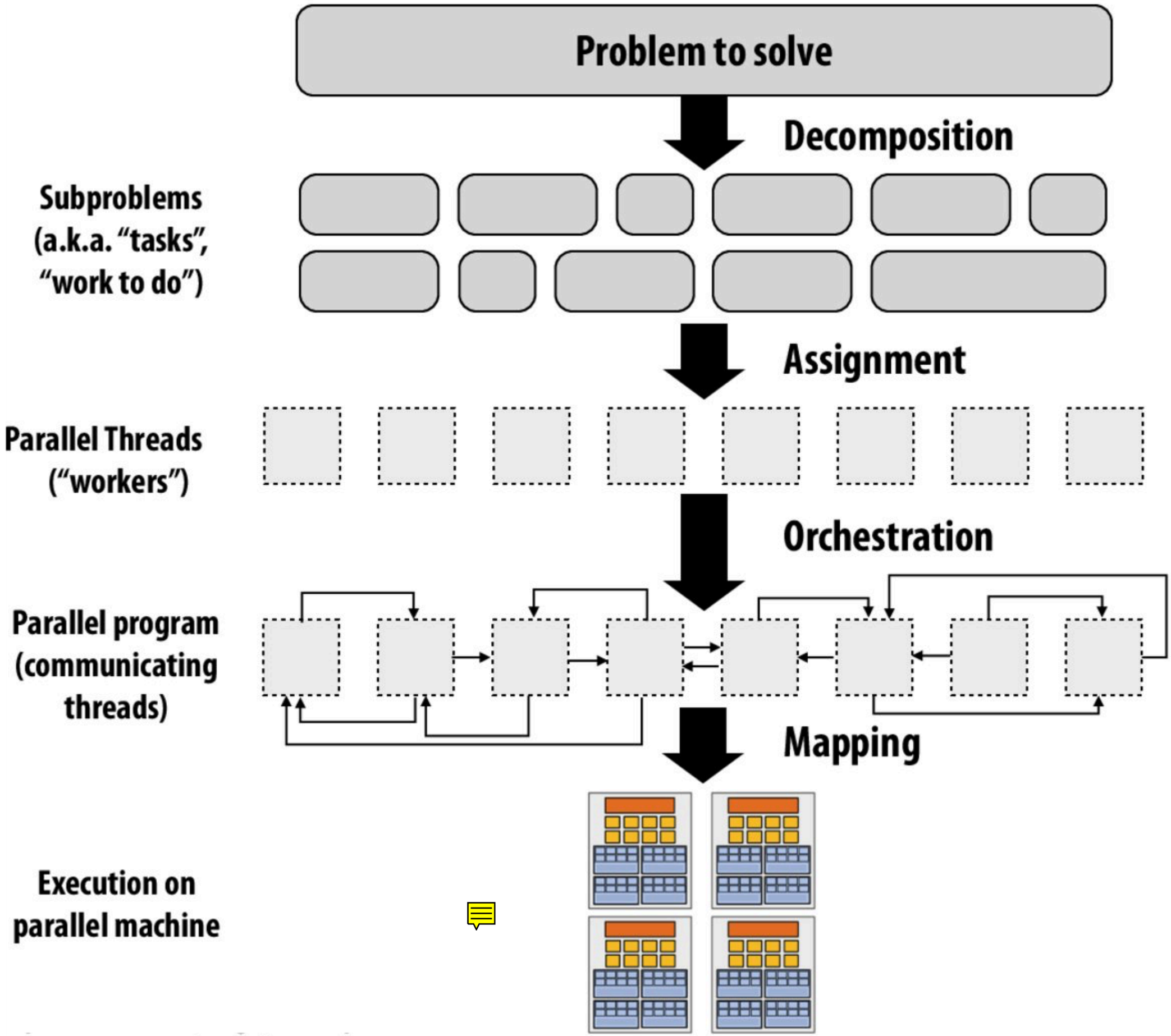
UNIVERSITÀ
DEGLI STUDI
FIRENZE

Creating a parallel program

Parallelizing a program

1. Identify work that can be done in parallel
2. Partition work (and data associated with work)
3. Orchestrate workers, managing data access, communication, synchronization

- Our goal is speedup


Problem to solve
Decomposition
Subproblems
(a.k.a. "tasks",
"work to do")
Assignment
Parallel Threads
("workers")
Orchestration
Parallel program
(communicating
threads)
Mapping
Execution on
parallel machine

Decomposition

- Break the problem in tasks that can be executed in parallel

  - Identify dependences and lack of dependencies

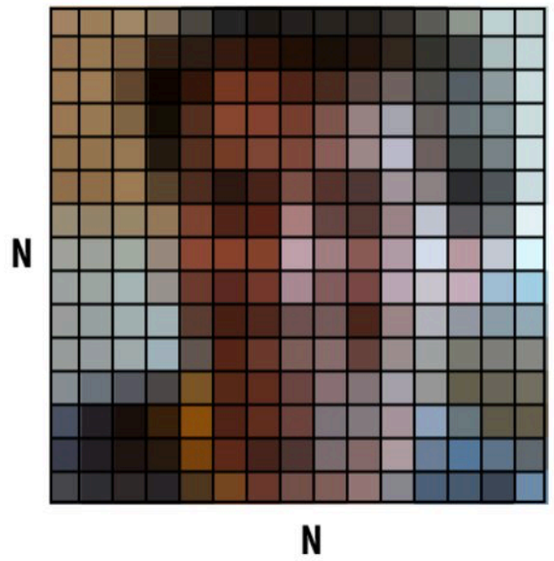
  - Create enough tasks to keep the execution units of the machine busy

  - It is possible to identify tasks at runtime

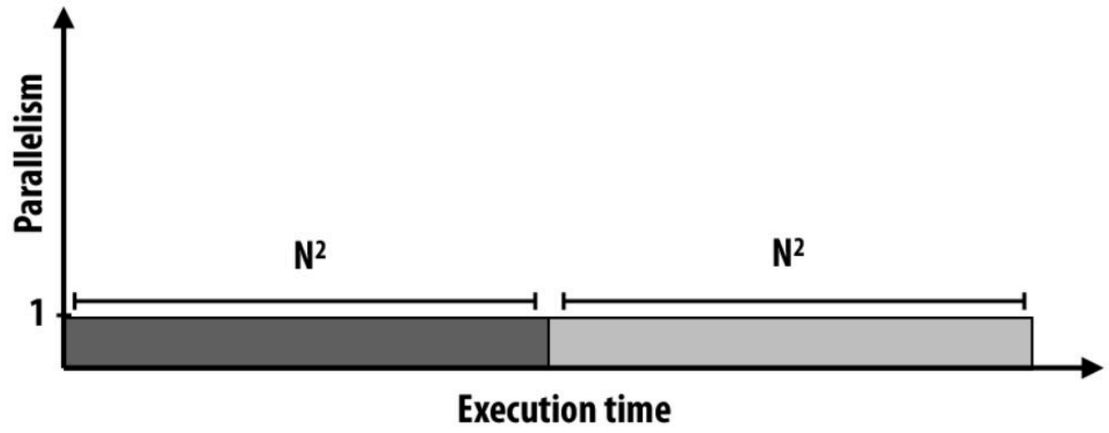
  - Remind Amdahl's law

Parallelization example

- Consider a two step computation on a N×N image:
  - Step 1: double pixel brightness
  - Step 2: compute average pixel value



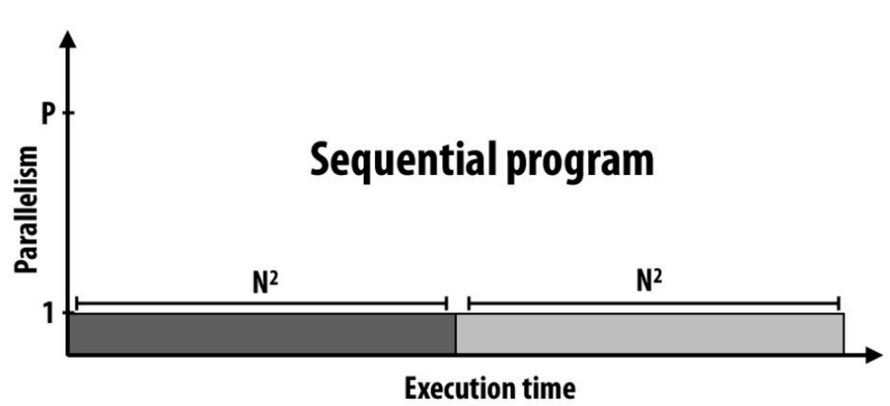
- Sequential program:
  - Both steps take ~N^2 time, so overall time is ~$2N^2$

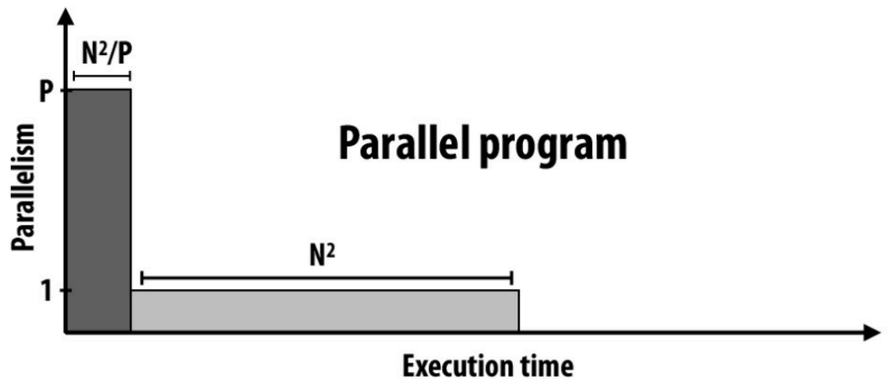


Parallelization example: approach 1

- Let us consider a system with *P* processors
- Parallelize step 1: processing each pixel is independent from each other
  - Time to process step 1: N^2/P
- Sequential processing of step 2
  - Time to process step 2: N^2

$$Speedup \leq \frac{2N^2}{\frac{N^2}{P} + N^2} \approx 2$$



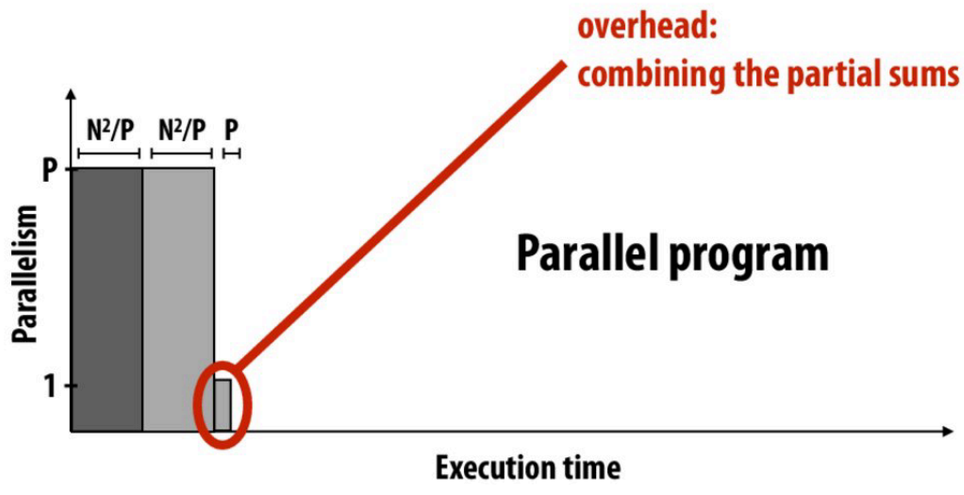


Parallelization example: approach 2

- Parallelize step 1: processing each pixel is independent from each other
  - Time to process step 1: N^2/P
- In step 2: parallelize partial sums, combine results sequentially
  - Time to process step 2: $N^2/P + P$

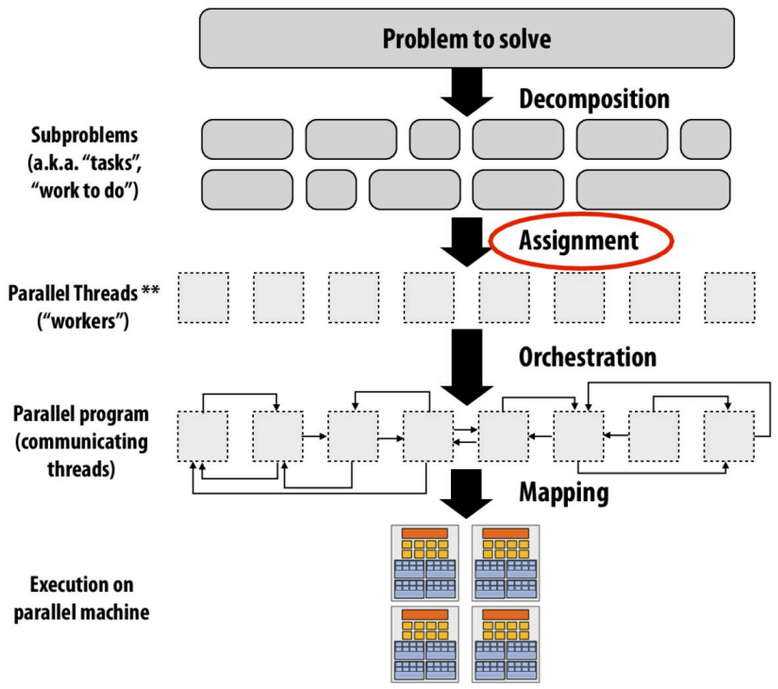
$$Speedup \leq \frac{2N^2}{\frac{N^2}{P} + P}$$

- Speedup → P if N >> P



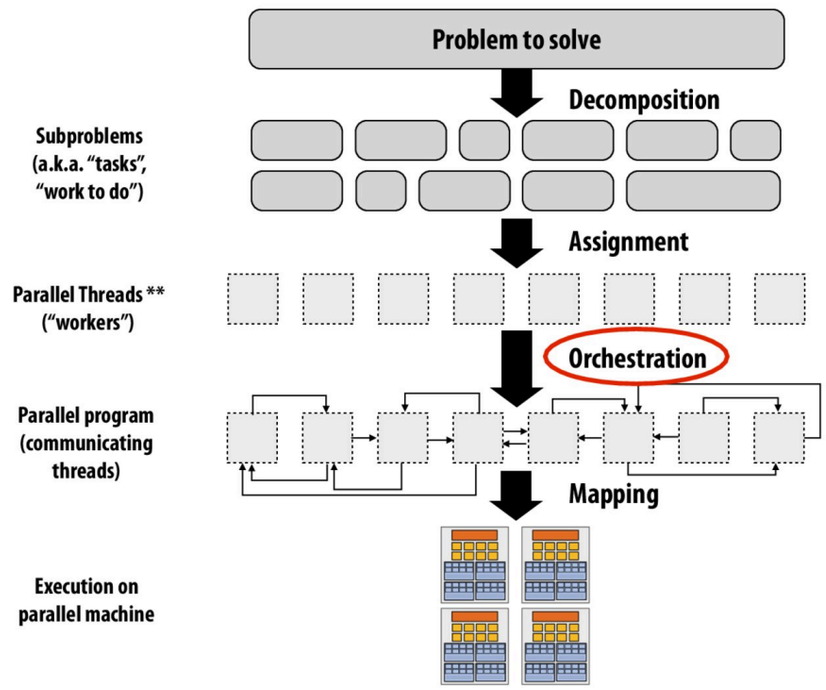
Assignment

- Assign tasks to threads/processes/whatever
  - Tasks are things to be done
  - threads/processes/whatever are workers
- Goals: balance workers, reduce communications
- Can be static or dynamic
- Sometimes it is up to the programmer, sometimes up to the language/framework/library



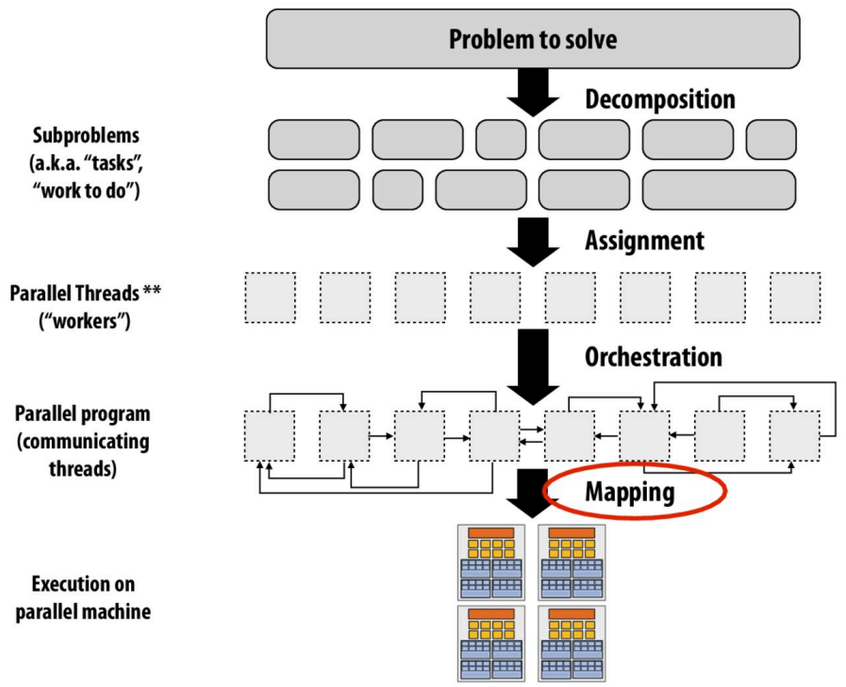
Orchestration

- Involves:
  - Structuring communications
  - Maintaining dependencies (e.g. using synchronization)
  - Scheduling tasks
  - Organizing data structures
- Goals: reduce costs (communication/ sync), overhead, preserve data locality
- Requires knowledge of machine, framework, library details



Mapping to hardware

- Mapping workers to execution units
- Can be done by:
  - Operating system: assign threads to cores
  - Hardware: assign threads to GPU cores
  - Framework: assigning workers to cluster machines
  - Compiler: assigning threads to SIMD lanes (the execution units working on vector elements)



Credits

- These slides report material from:
  - Rob Pike (Google)
  - Guy E. Blelloch and Bruce M. Maggs (CMU)
  - Kayvon Fatahalian and Randal Bryant (CMU)
  - Prof R. Guerraoui (EPFL)
  - Prof. Robert van Engelen (Florida State University)
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)

Books

- Principles of Parallel Programming, Calvin Lyn and Lawrence Snyder, Pearson - Chapt. 1-3

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 1