



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

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 \subseteq 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:
advancing
delivering
question

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

- I'm doing

A **parallel** program potentially runs more quickly than a sequential program by executing different parts of the computation simultaneously (in parallel).

- If I'm here
projector
the processor

It may or may not have more than one logical thread of control.

- If the as
independent
talking a
parallel.

I'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.

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”)

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.

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.

Grid computing is a collection of geographically distributed 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.

Grid computing is a collection of highly distributed 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.

Grid computing is a collection of highly distributed 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.

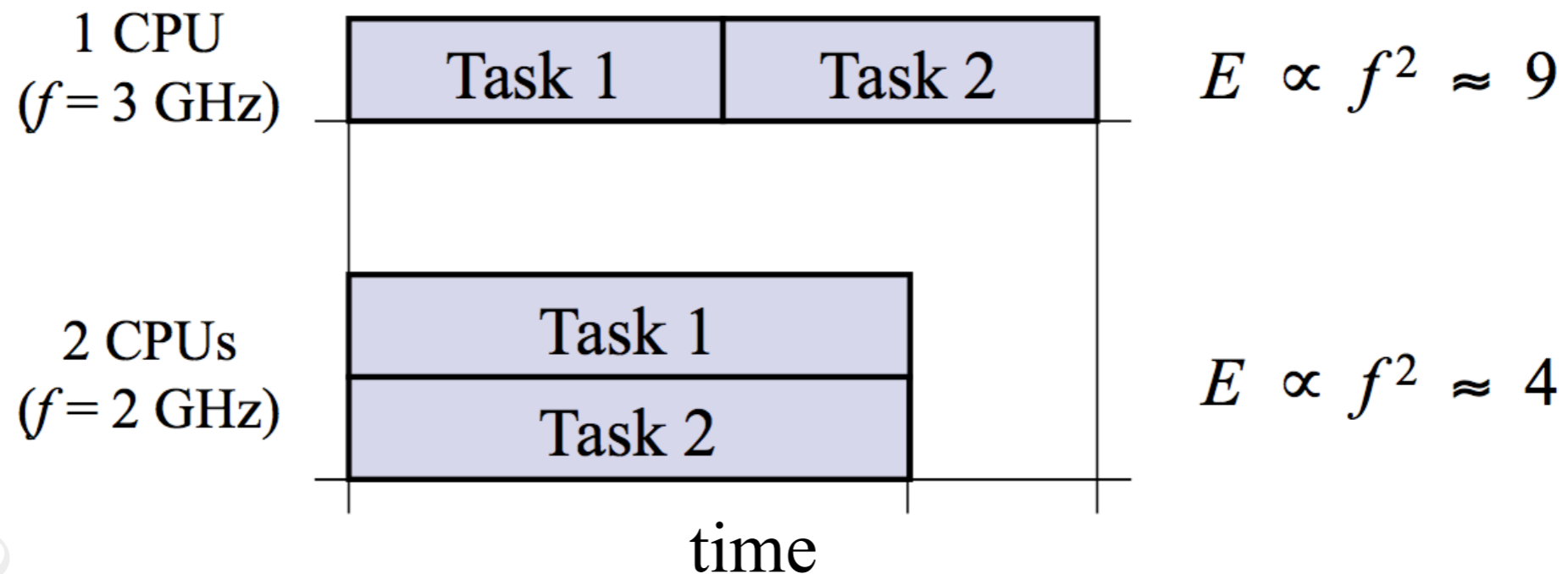
- Typically: interactions infrequent, with heavier weight and assumed to be unreliable, coarse grained, much overhead and long uptime.

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 : $P \propto f V^2$
- The voltage required to operate the CPU at given frequency is, roughly proportional to the frequency ($P \propto f^3$).
- By increasing the frequency, same amount of work can be finished in shorter time (roughly speaking, inversely to the frequency): $E \propto f^2$

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



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).
- 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).
- Signalling rates are limited by the parasitic 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.

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

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 3×10^{12} memory moves per second
- If data travels at the speed of light (3×10^8 m/s) between the CPU and memory and r is the average distance between the CPU and memory, then r must satisfy
 - $3 \times 10^{12} r = 3 \times 10^8$ m/s \times 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 / ($\sqrt{3 \times 10^6}$) = 10^{-10} m which is the size of a relatively small atom!

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



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						
	IF	ID	EX	MEM	WB		
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						
	IF	ID	EX	MEM	WB		
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						
	IF	ID	EX	MEM	WB		
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*.

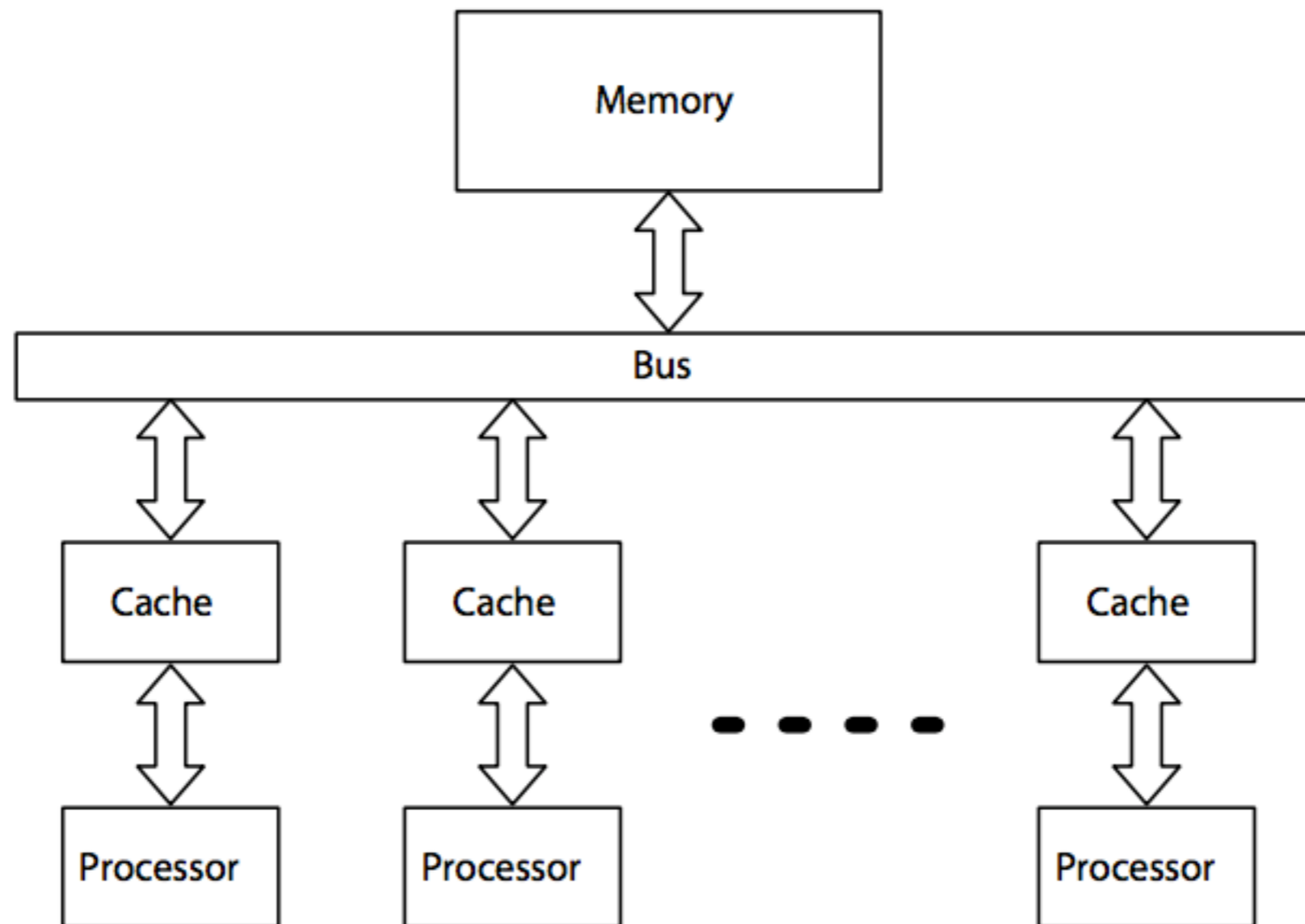
Data Parallelism

- 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

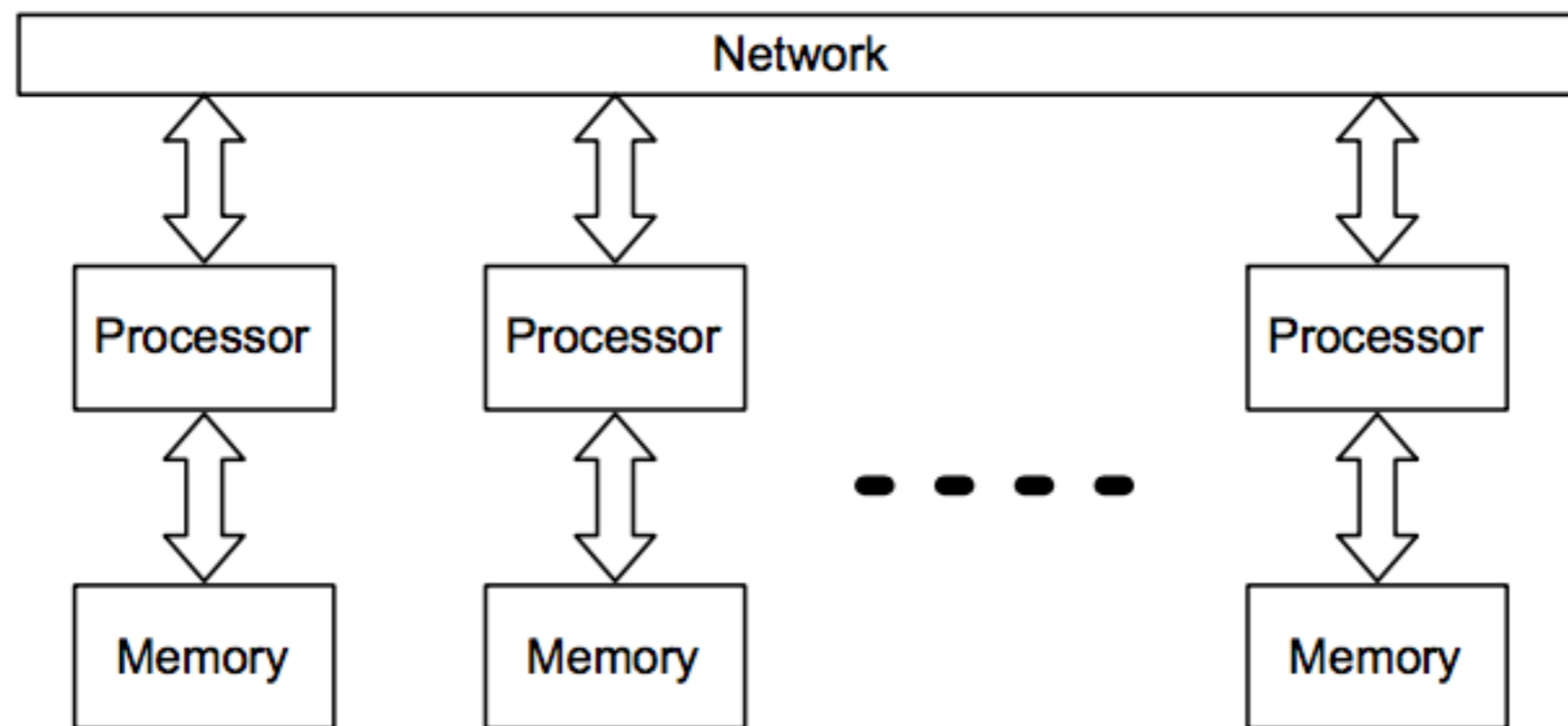




- Natural extension of sequential computer: all memory can be referenced (single address space). Hardware ensures memory coherence.
- Good:
 - Easier to use
Through multi-threading
- Bad:
 - Easier to create faulty programs
Race conditions
 - More difficult to debug
Intertwining of threads is implicit



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.

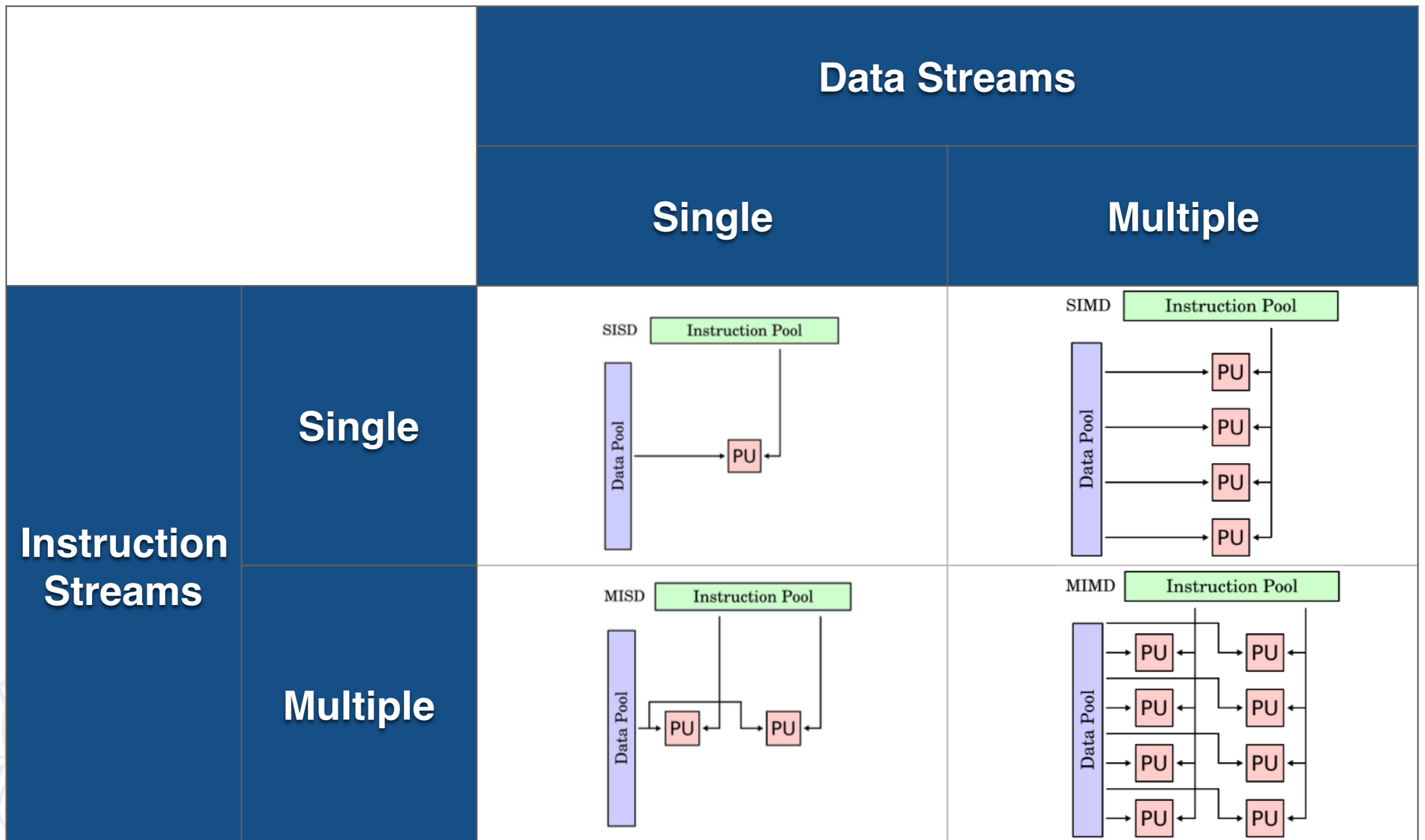
Flynn's taxonomy

		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





UNIVERSITÀ
DEGLI STUDI
FIRENZE



Parallel architectures

RAM model

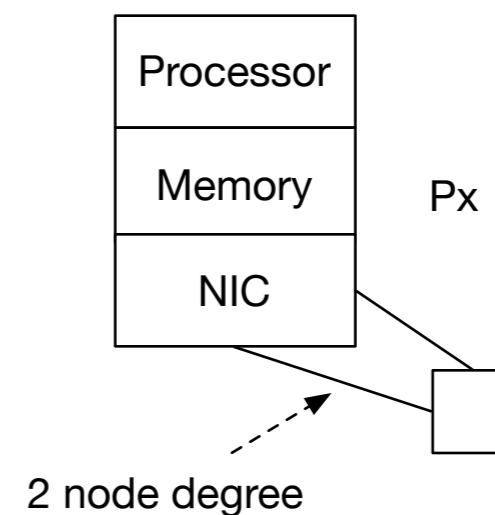
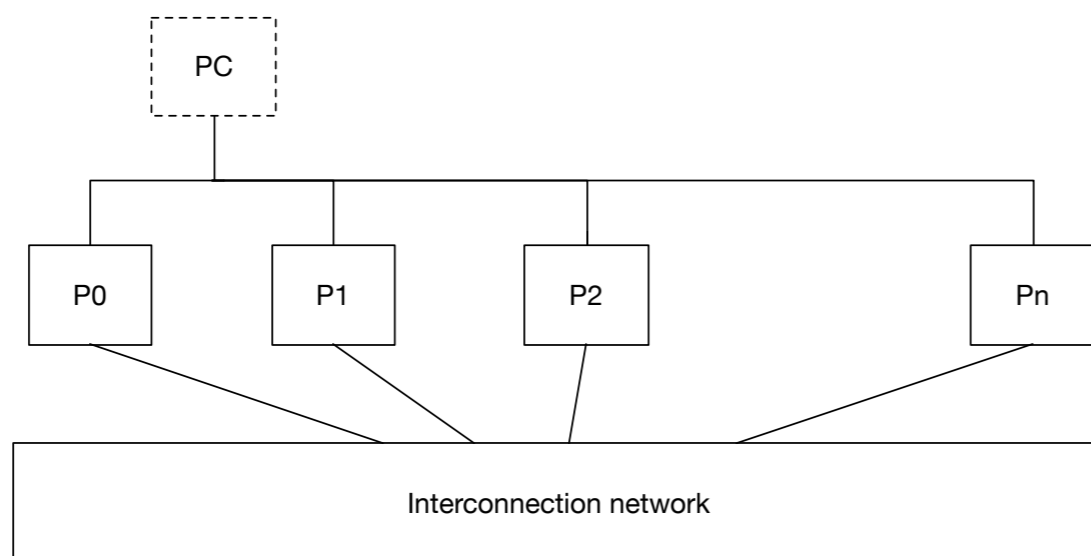
- Random Access Machine: is an abstract model for a sequential computer
- It models a device with an instruction execution unit and unbounded memory.
 - Memory stores program instructions and data.
 - Any memory location can be referenced in 'unit' time
 - The instruction unit fetches and executes an 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

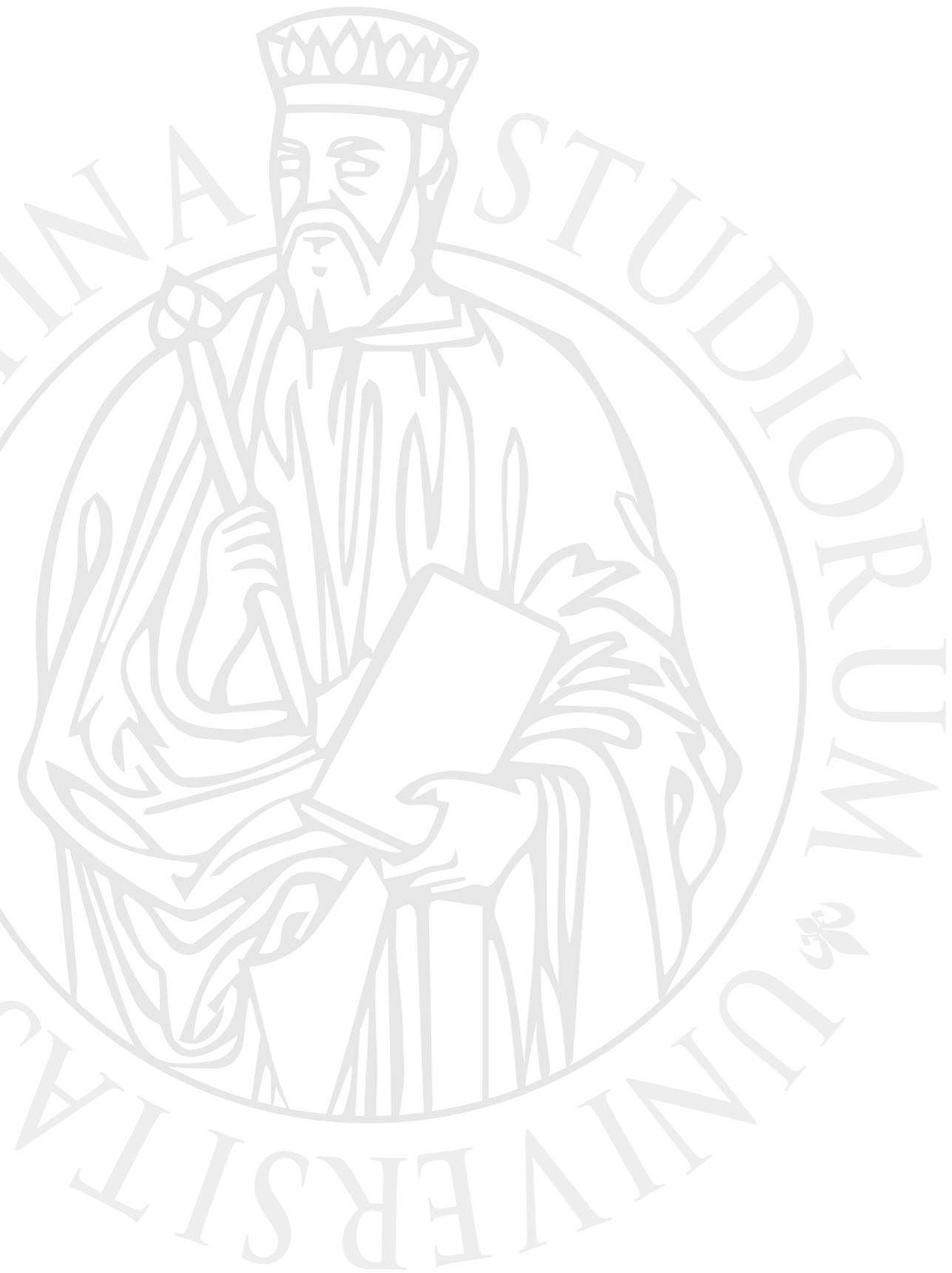


Memory Latency λ

- Memory Latency = delay required to make a memory reference. Non local memory latency is indicated with λ .
- Relative to processor's local memory latency, \approx unit time \approx 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)



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 inverse of **processing bandwidth**, or average throughput of the stream.
- **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 $\mathbf{t} = \sum_{i=1,k} p_i t_i$
 - **completion time** (mean) if the stream has length m is $\mathbf{t}_c = m \mathbf{t}$
 - **processing bandwidth** (throughput) is the average number of operations executed by Σ in the time unit: $\mathbf{B} = 1 / \mathbf{t}$
 - **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 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

$$B^n = 1 / t^n$$
- **completion time t_c^n** is the average time to complete the execution of all stream elements. For $m \gg n$ the relation with service time is $t_c^n \cong m t^n$

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 why scalability is harder to evaluate than speedup

Speedup limitations

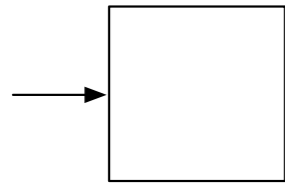
- Several factors can limit the speedup
 - Processors may be idle
 - Extra computations are performed in the parallel version
 - Communication and synchronization overhead



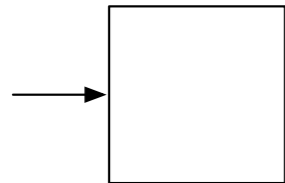
Relative speedup

- 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

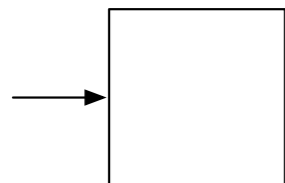
Speedup: example



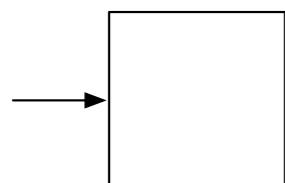
- Search in parallel by partitioning the search space into P chunks



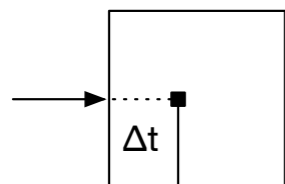
- $S_P = ((x \times t_s/P) + \Delta t) / \Delta t$



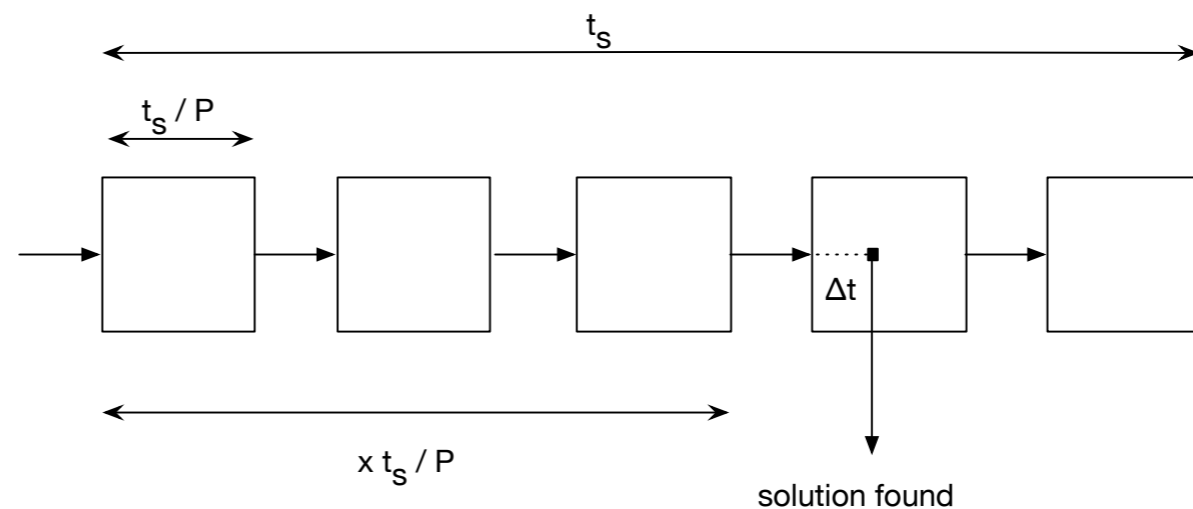
- Worst case for sequential search (item in last chunk): $S_P \rightarrow \infty$ as Δt tends to zero



- Best case for sequential search (item in first chunk): $S_P = 1$



solution found

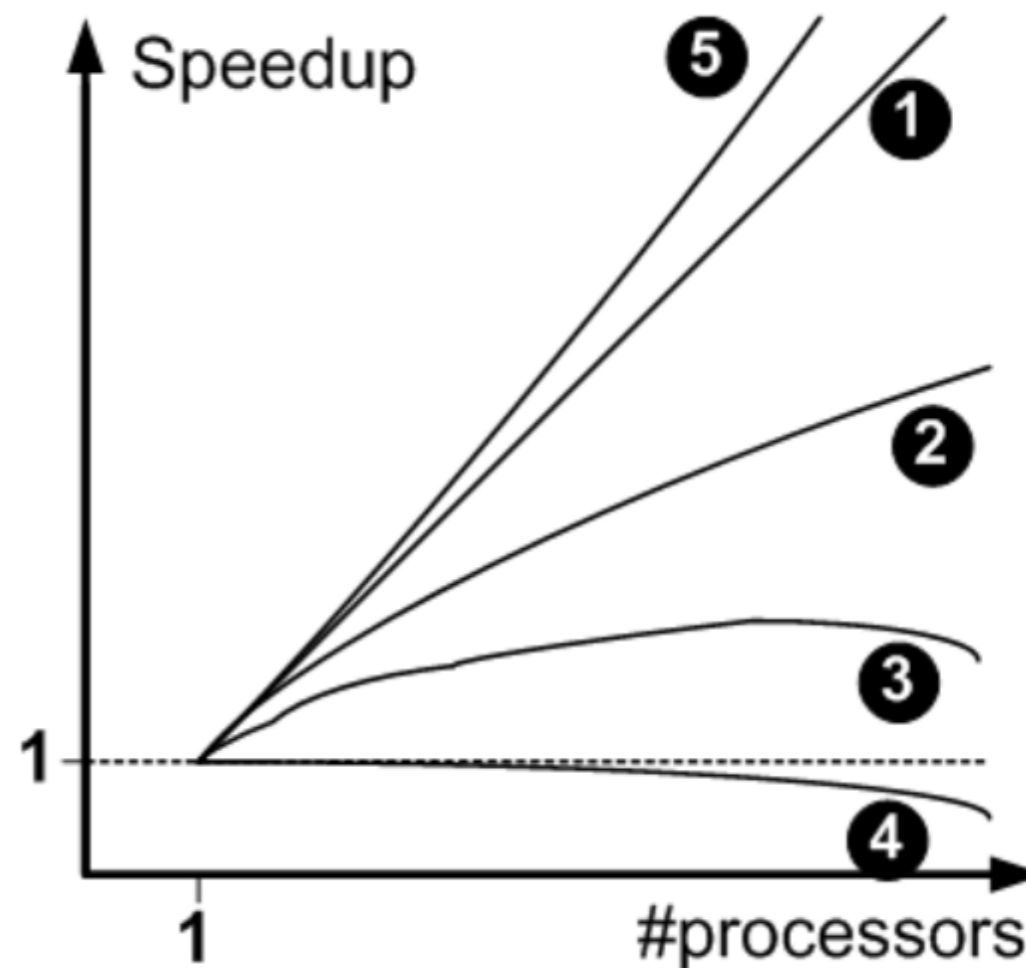


Sequential search

Parallel 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

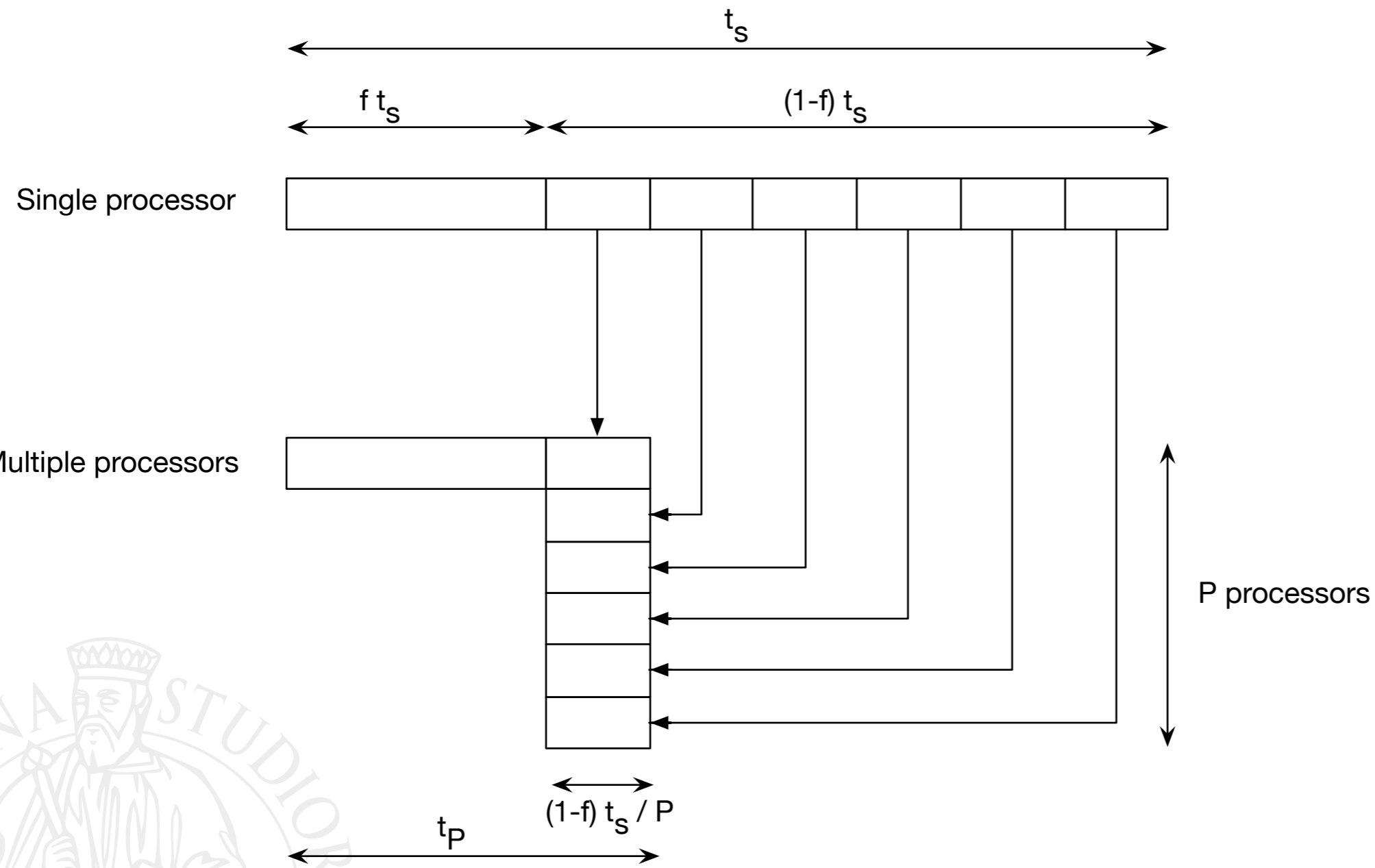
Amdahl's Law

- 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.



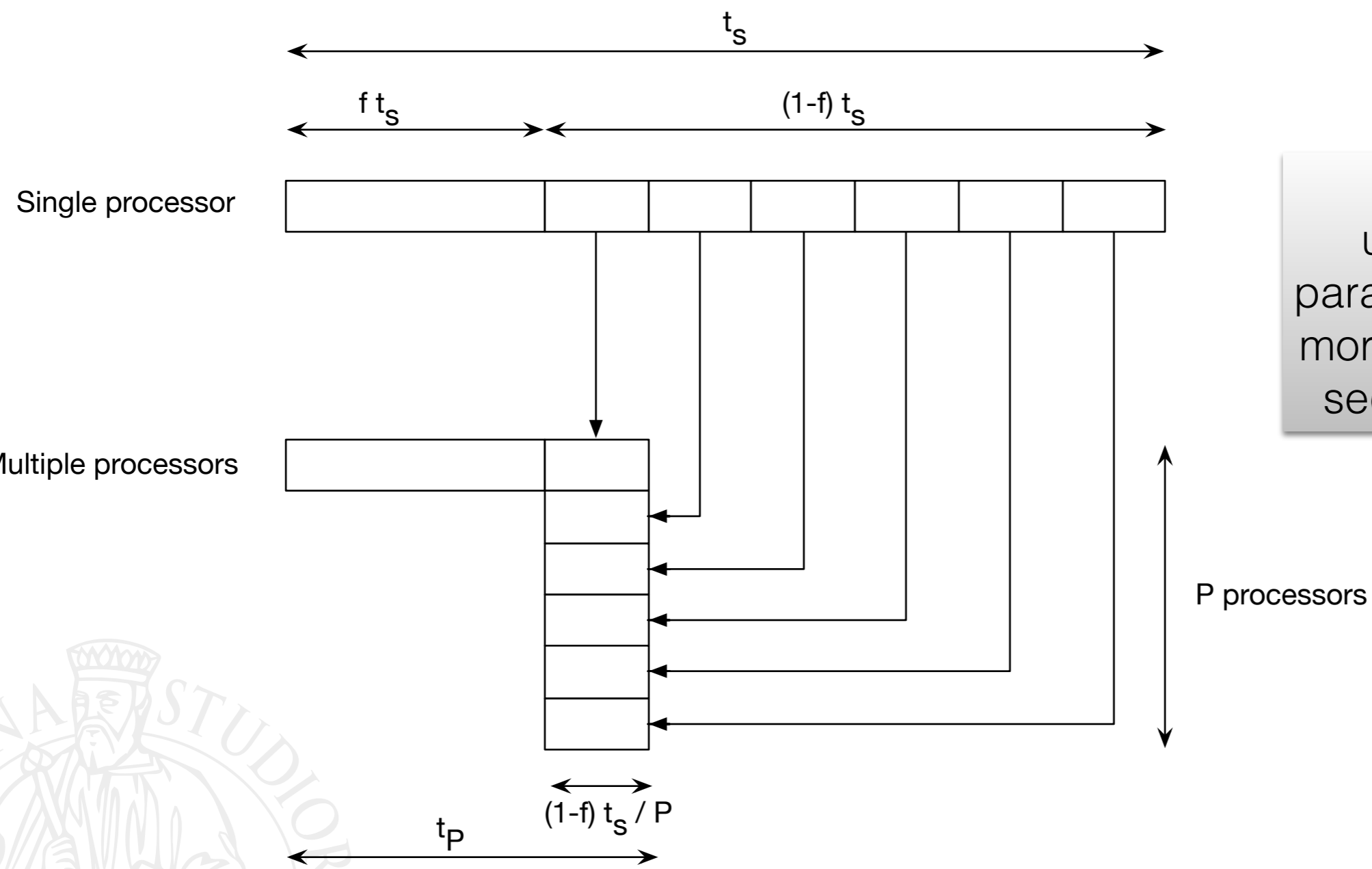
Amdahl's Law

- 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

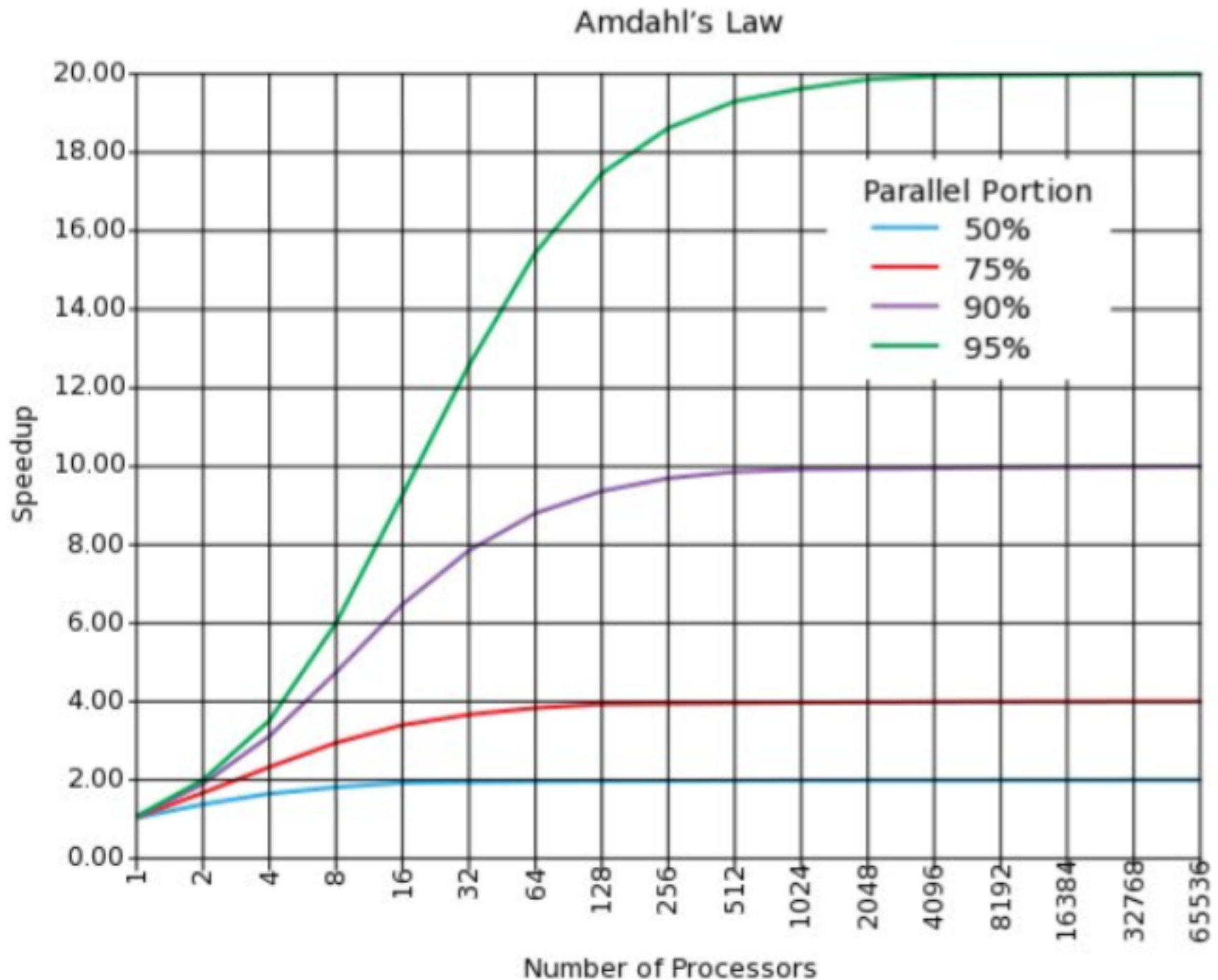
Amdahl's Law

- 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

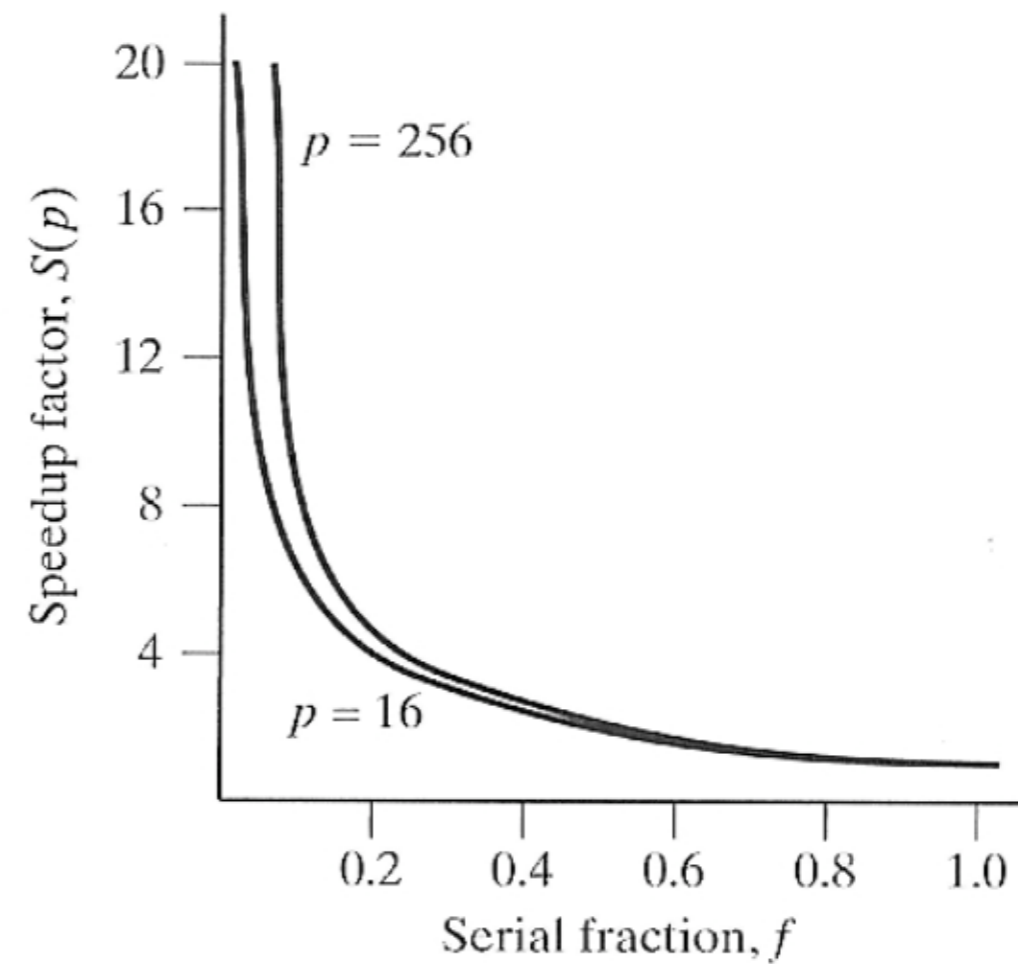
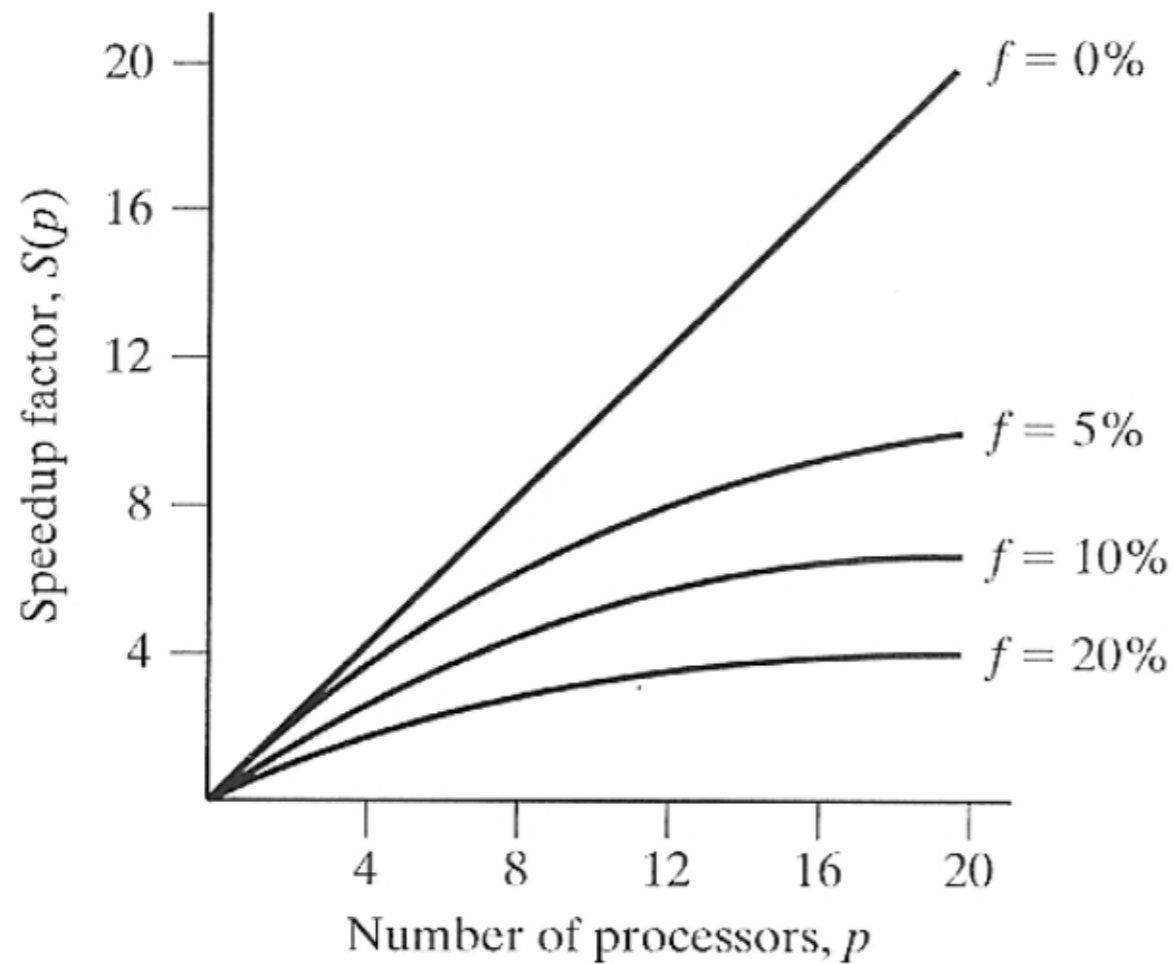
- Amc
proc
 $S_P =$
- As a
limit
- Eve
neve



u will

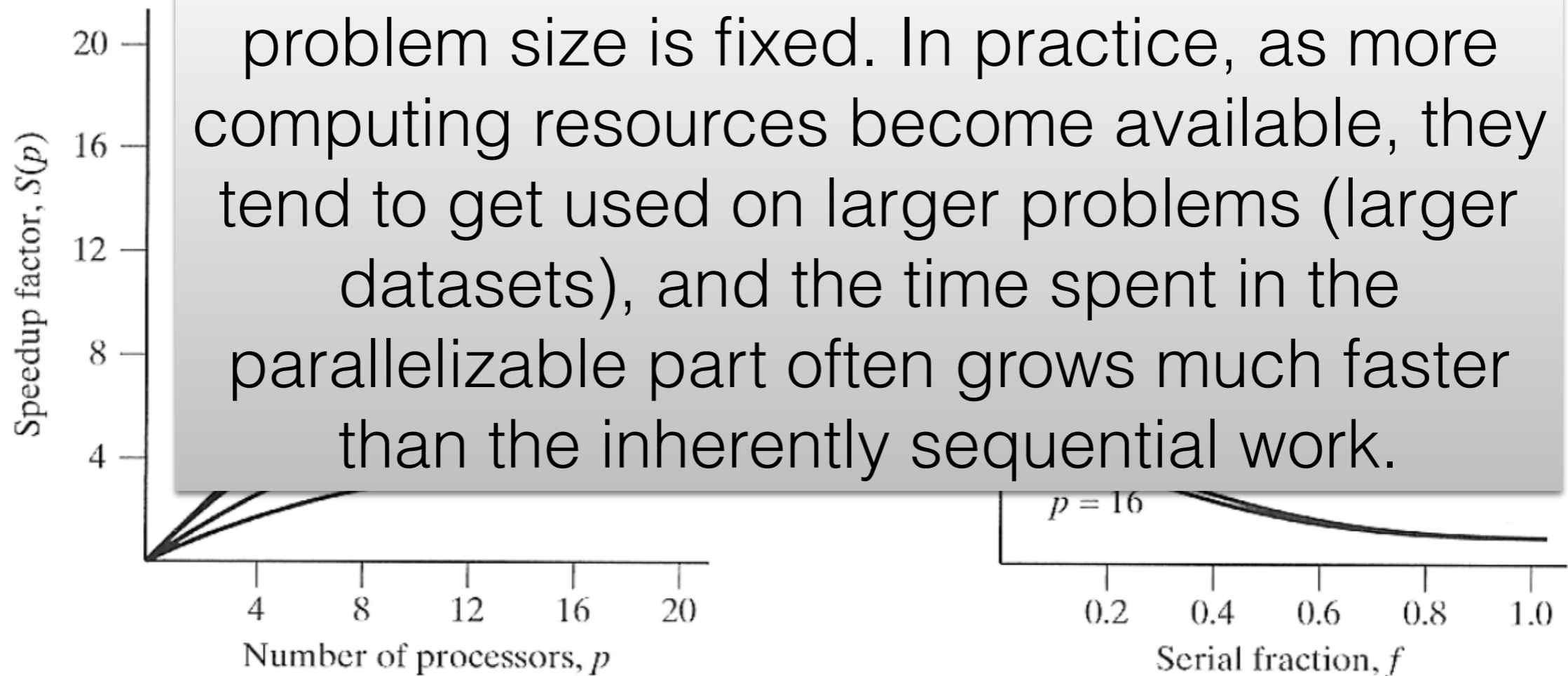


Amdahl's Law



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

Scaling example - cont

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

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.

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 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 !

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.

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

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

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 $\beta(N) = 1 - \alpha(N)$ be the parallelizable fraction (overhead is ignored)

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

then, the scaled sequential time is

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

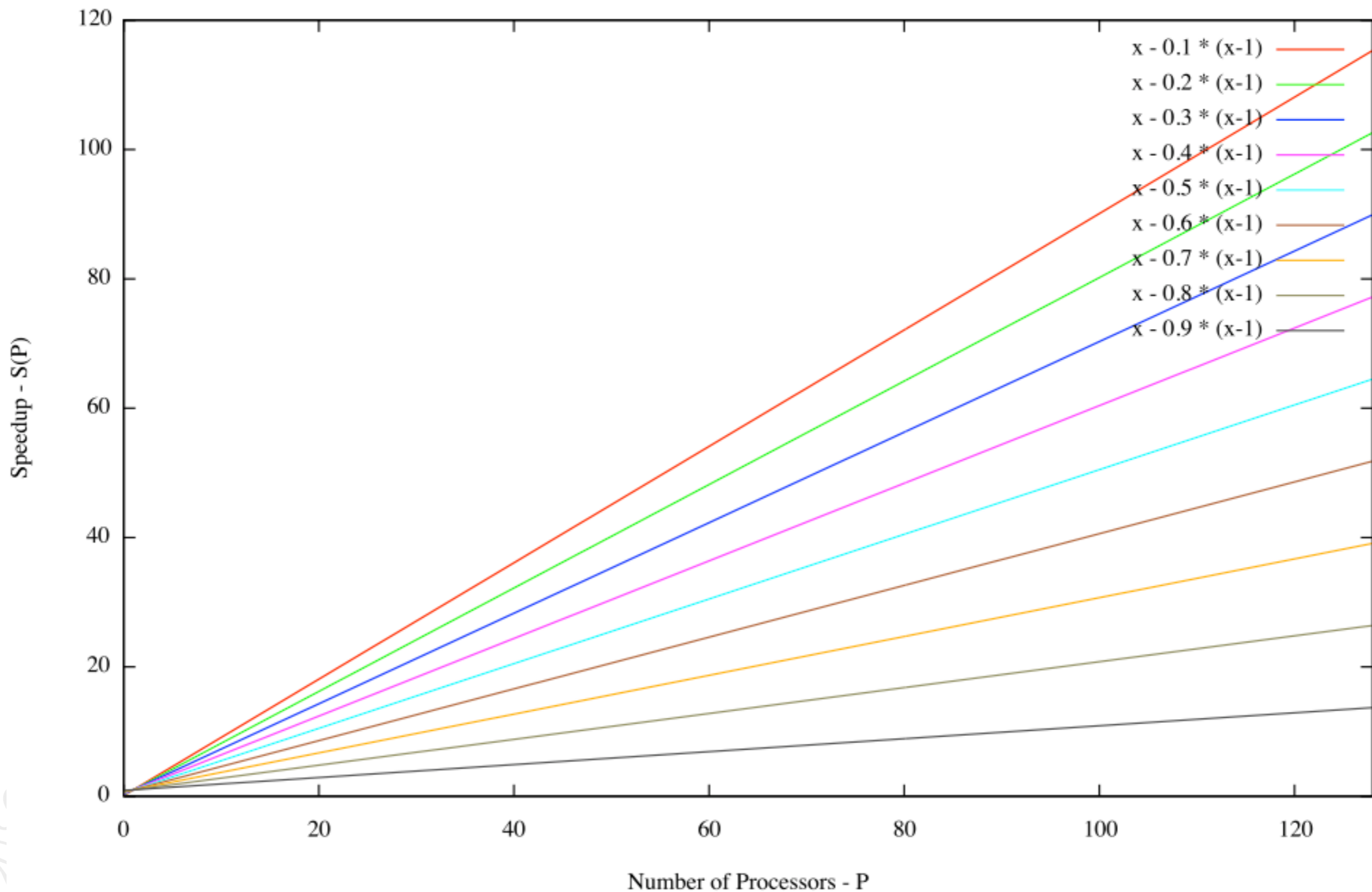
If $\alpha(N)$ is small then the speedup is almost P

giving

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

Gustafson's Law

Gustafson's Law: $S(P) = P - a * (P - 1)$



Scaling example

- Number of processors proportional to problem size
- 10 processors, 10×10 matrix:
Time = $20 \times t_{\text{add}}$
- 100 processors, 32×32 matrix:
Time = $10 \times t_{\text{add}} + 1000/100 \times t_{\text{add}} = 20 \times t_{\text{add}}$



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

Flow dependance

1. $sum \leftarrow a+1;$

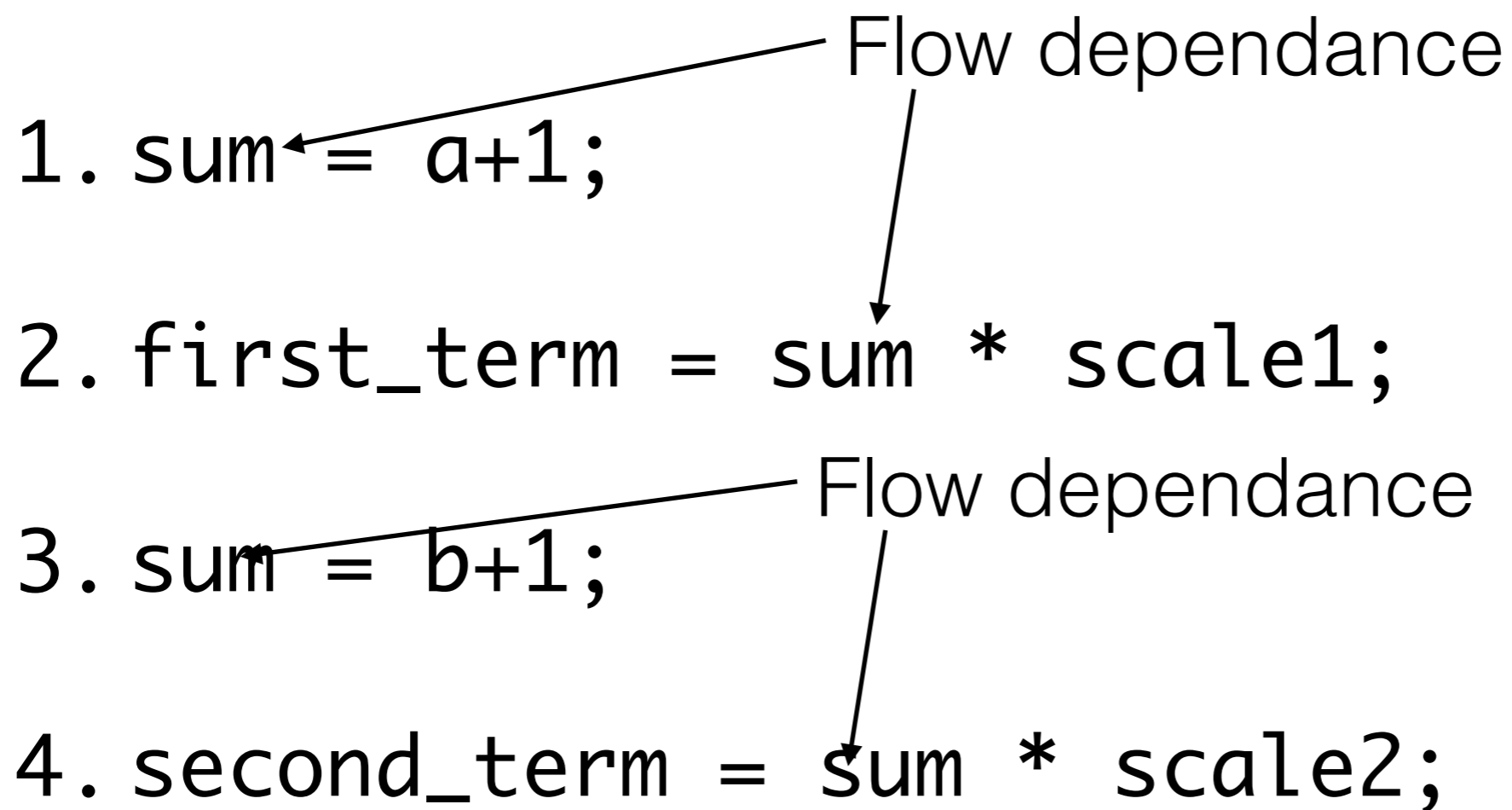
2. $first_term = sum * scale1;$

3. $sum = b+1;$

4. $second_term = sum * scale2;$



Data dependences



Data dependences

Flow dependance

1. $sum \leftarrow a+1;$

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. $sum = b+1;$

Anti dependance

4. $second_term = sum * scale2;$



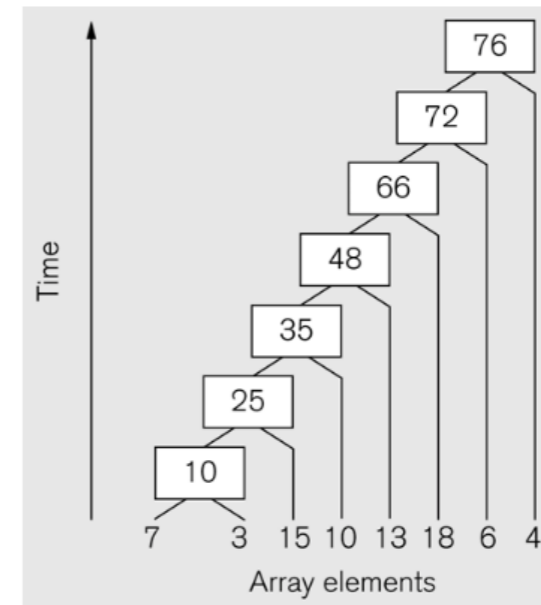
Example: iterative sum

```
sum=0;
```

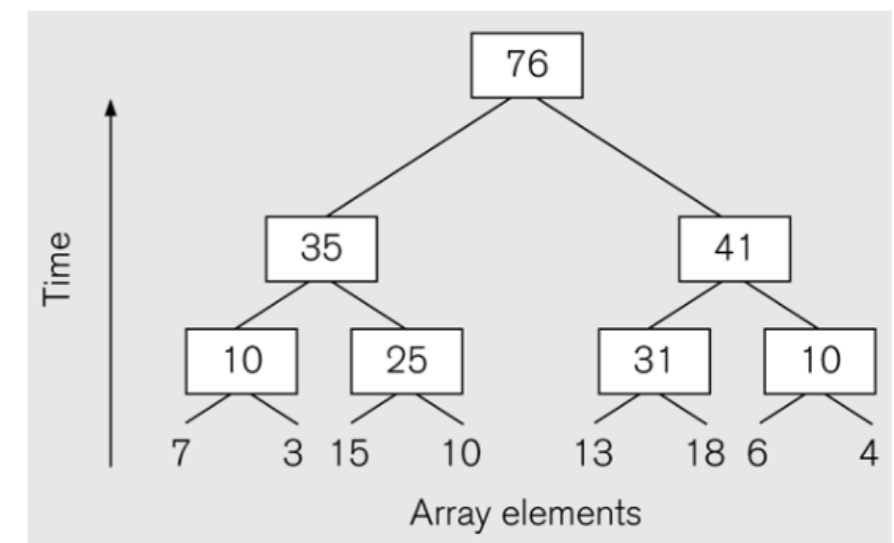
```
for (int i=0;i<n;i++)
```

```
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

Credits

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



Books

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

