The multicore revolution

Giorgio Buttazzo

Retis
Real-Time Systems Laboratory

Scuola Superiore Sant'Anna, Pisa

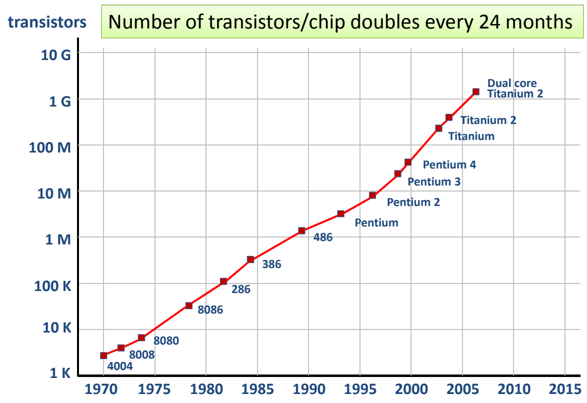
The transition

- On May 17th, 2004, Intel, the world's largest chip maker, canceled the development of the Tejas processor, the successor of the Pentium4-style Prescott processor.
- On July 27th, 2006, Intel announced the official release of the Core Duo processors family.
- Since then, all major chip producers decided to switch from single core to multicore platforms.
- Such a phenomenon is known as the multicore revolution.

The reason why this happened has to do with a market law, predicted by Gordon Moore, Intel's co-founder, in 1965, known as Moore's Law.

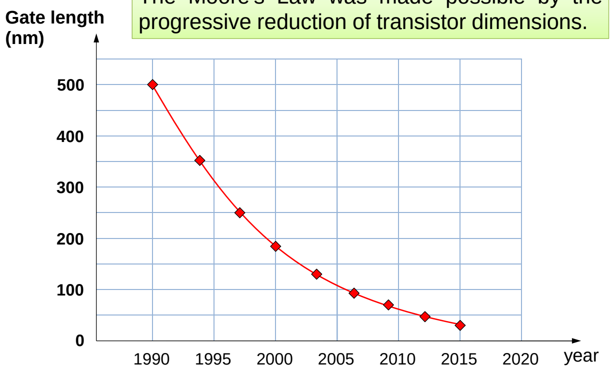
Moore's Law

Number of transistors/chip doubles every 24 months



Gate reduction

The Moore's Law was made possible by the progressive reduction of transistor dimensions.



Benefits of size reduction

There are 2 main benefits of reducing transistor size:

1. a higher number of gates that can fit on a chip;
2. devices can operate at higher frequency.

In fact, if the distance between gates is reduced, signals have to cover a shorter path, and the time for a state transition decreases, allowing a higher clock speed.

However…

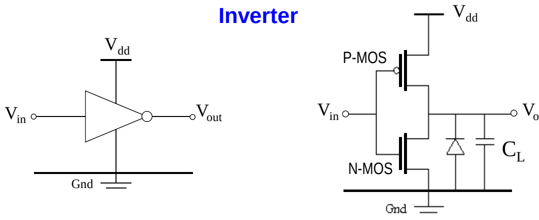
At the launch of Pentium 4, Intel expected single core chips to scale up to 10 GHz using gates below 90 nm. However, the fastest Pentium 4 never exceeded 4 GHz.

Why did that happen?

Power dissipation

The main reason is related to power dissipation in CMOS integrated circuits, which is mainly due to two causes:

- **Dynamic power** (P_d) consumed during operation;
- **Static power** (P_s) consumed when the circuit is off.



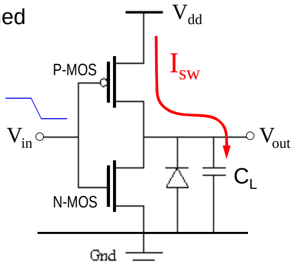
Dynamic power

Dynamic power is mainly consumed during logic state transitions to charge and discharge the load capacitance C_L.

It can be expressed by:

$$P_d \propto C_L \cdot f \cdot V_{dd}^2$$

f = clock frequency

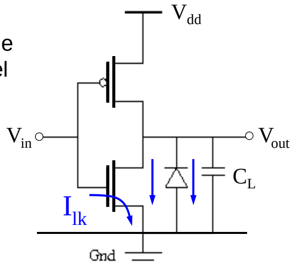


Static power

Static power is due to a quantum phenomenon where mobile charge carriers (electrons or holes) tunnel through an insulating region, creating a leakage current I_{lk}

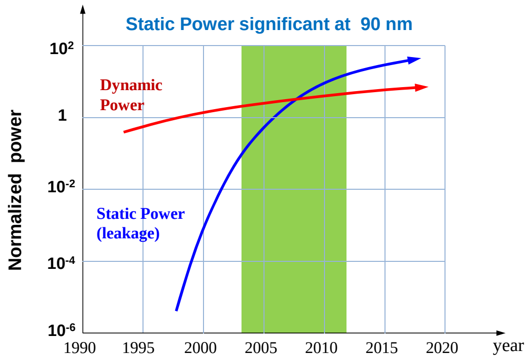
$$P_s \propto V_{dd} I_{lk}$$

It is independent of the switching activity and is always present if the circuit is on.



As devices scale down in size, gate oxide thicknesses decreases, resulting in a larger leakage current.

Dynamic vs. static power



Gate length (nm): 500 350 250 180 130 90 65 45 22

Power and Heat

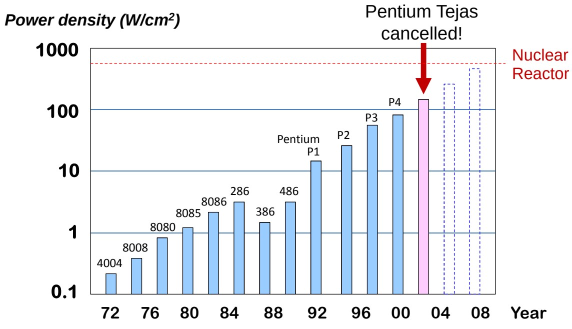
A side effect of power consumption is heat, which, if not properly dissipated, can damage the chip.

$$P \propto C_L \cdot f \cdot V_{dd}^2 + V_{dd} \cdot I_{lk}$$

Scaling down, both f and I_{lk} increased

If processor performance would have improved by increasing the clock frequency, the chip temperature would have reached levels beyond the capability of current cooling systems.

Heating problem



Clock speed limited to less than 4 GHz

Keeping Moore's Law alive

The solution followed by the industry to keep the Moore's law alive was to

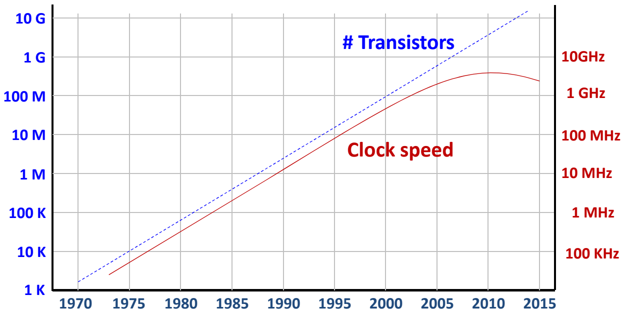
- use a higher number of slower logic gates,
- building parallel devices that work at lower clock frequencies.

In other words…

Switch to Multicore Systems!

Keeping Moore's Law alive

- # of transistors continued to increase according to Moore's Law
- clock speed and performance experienced a saturation effect



How to exploit multiple cores?

The efficient exploitation of multicore platforms poses a number of new problems that are still being addressed by the research community.

When porting a real-time application from a single core to a multicore platform, the following key issues have to be addressed:

- How to split the code into parallel segments that can be executed simultaneously?
- How to allocate such segments to the different cores?

Expressing parallelism

- In a multicore system, sequential languages (as C/C++) are no longer appropriate to specify programs.
- In fact, a sequential language hides the intrinsic concurrency that must be exploited to improve the performance of the system.

To really exploit hardware redundancy, most of the code has to be parallelized.

A big problem for industry

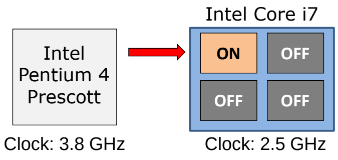
Parallelizing legacy code implies a tremendous cost and effort for industries, mainly due to:

- re-design the application
- re-writing the source code
- updating the operating system
- writing new documentation
- testing the system
- software certification

To avoid such costs, the cheapest solution is to port the software on a multicore platform, but run it on a single core, disabling all the other cores.

A big problem for industry

However, due to the clock speed saturation effect, a core in a multicore chip is slower than a single core:


Intel Core i7

Intel Pentium 4 Prescott

ON OFF
OFF OFF

Clock: 3.8 GHz

Clock: 2.5 GHz

If the application workload was already high, running the application on a single core of a multicore chip creates an overload condition.

To avoid such problems, avionic industries buy in advance enough components for ensuring maintenance for 30 years!

Other problems

In a single core system, concurrent tasks are sequentially executed on the processor, hence the access to physical resources is implicitly serialized (e.g., two tasks can never cause a contention for a simultaneous memory access).

In a multicore platform, different tasks can run simultaneously on different cores, hence several conflicts can arise while accessing physical resources.

Such conflicts not only introduce interference on task execution but also increase the Worst-Case Execution Time (**WCET**) of each tasks.

The WCET issue

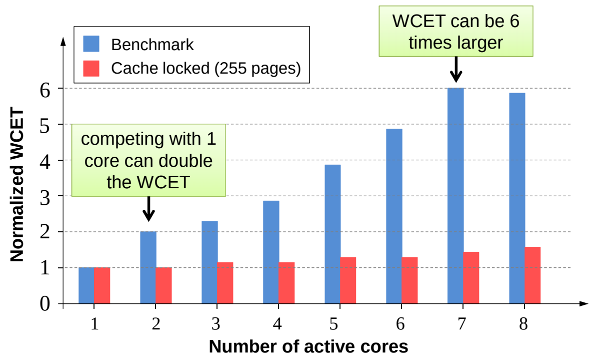
The fundamental assumption

Existing RT analysis assumes that the worst-case execution time (WCET) of a task is constant when it is executed alone or together with other tasks.

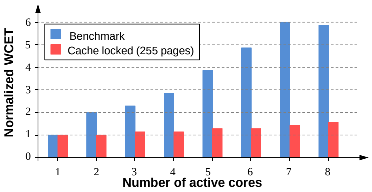
While this assumption is correct for single-core chips, it is NOT true for multicore chips**!**

WCET in multicore

Test by Lockheed Martin Space Systems on 8-core platform



Questions



- Why WCET increases up to 6 times?
- Why WCET on 8 cores is lower than WCET on 7 cores?
- What does this mean for system development, integration and certification?

There are multiple reasons

The WCET increases because of the competition among cores in using shared resources.

- Main memory
- Memory-bus
- Last-level cache
- I/O devices

Competition creates extra delays

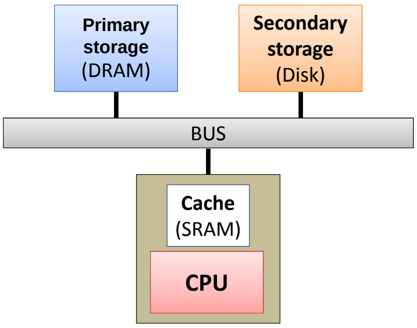
- waiting for other tasks to release the resource
- waiting for accessing the resource

In a single CPU, only one task can run at a time, so applications cannot saturate memory and I/O bandwidth.

To better understand the interference causes, we need to take a quick look at the modern computer architectures.

Types of Memory

There are typically three types of memory used in a computer:



Primary storage (DRAM)

Secondary storage (Disk)

BUS

Cache (SRAM)

CPU

Primary Storage

It is referred to as main memory or internal memory, and is directly accessible to the CPU.

It is volatile, which means that it loses its content if power is removed.

Primary storage includes RAM (based on DRAM technology), Cache and CPU registers (based on SRAM technology):

- **DRAM** (Dynamic random-access memory) requires to be periodically, refreshed (re-read and re-written) otherwise it would vanish.
- **SRAM** (Static random-access memory) never needs to be refreshed as long as power is applied.

Secondary Storage

It is referred to as external memory or auxiliary storage, because it is not directly accessible by the CPU. The access is mediated by I/O channels and data are transferred using intermediate area in primary storage.

It is non volatile, that is, it retains the stored information even if it is not constantly supplied with electric power.

Examples of secondary storage devices are:

- Hard Disk: based on magnetic technology
- CD ROM, DVD: based on optical technology
- Flash memory: can be electrically erased and reprogrammed

Cache Memory

The cache is a local memory used by the CPU to reduce the average time to access data from the main memory.

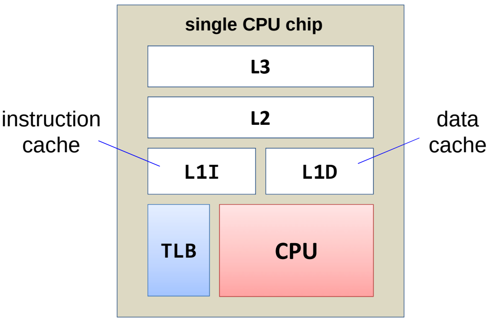
The cache is faster than the RAM, but more expensive, so much smaller in size.

Most CPUs have different types of caches:

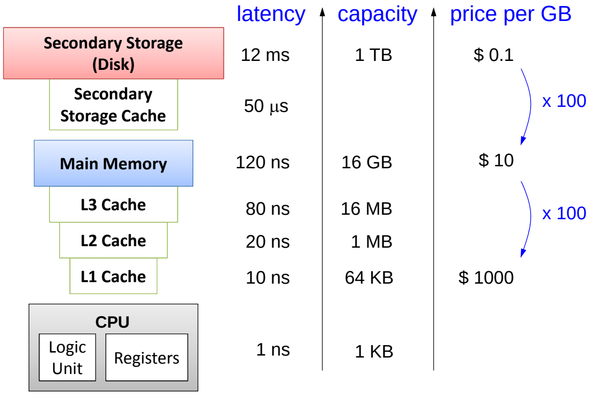
- Instruction Cache, to speed up executable instruction fetch
- Data Cache, to speed up data fetch and store
- Translation Lookaside Buffer (TLB), used to speed up virtual-to-physical address translation for both executable instructions and data.

Cache Levels

The **data cache** is usually hierarchically organized as a set of levels: L1, L2, …



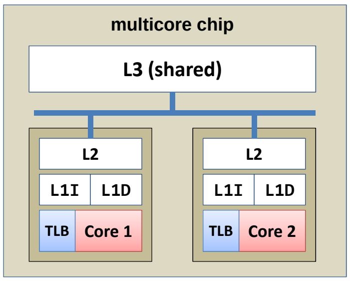
Access times



	latency	capacity	price per GB
Secondary Storage (Disk)	12 ms	1 TB	$ 0.1
Secondary Storage Cache	50 µs		x 100
Main Memory	120 ns	16 GB	$ 10
			x 100
L3 Cache	80 ns	16 MB	
L2 Cache	20 ns	1 MB	
L1 Cache	10 ns	64 KB	$ 1000
CPU: Logic Unit, Registers	1 ns	1 KB	

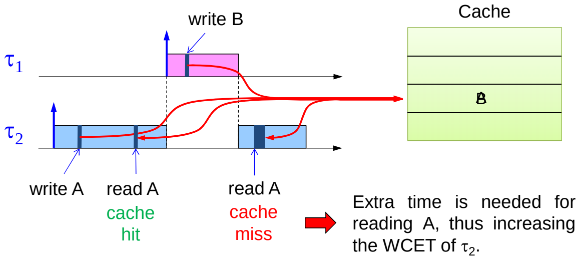
Cache in multicore chips

In multicore architectures, the L3 cache is typically shared among cores:



Cache related preemption delay

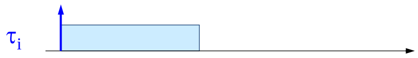
CRPD: delay introduced by high priority tasks that evict cache lines containing data used in the future:



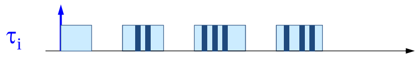
Extra time is needed for reading A, thus increasing the WCET of τ_2.

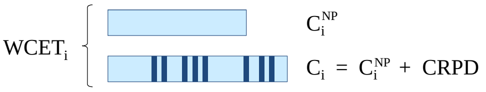
WCET

Task executing alone (or non preemptively) on a single CPU:



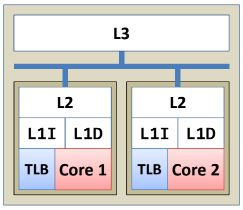
Task experiencing preemptions by higher priority tasks:





CRPD in multicore systems

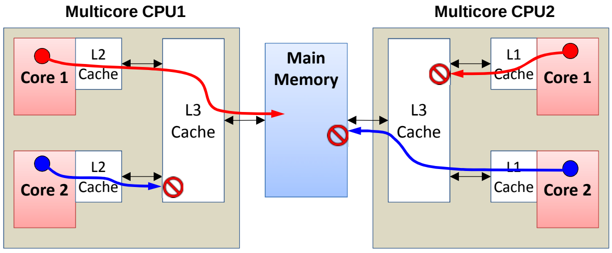
- In multicore systems, L1 and L2 caches have the same problem seen in single-core systems.
- L3 cache lines can also be evicted by applications running on different cores.



- We can partition the last level cache to simulate the cache architecture of a single-core chip, but the size of each partition becomes rather small.

Resource conflicts

When applications in different cores run concurrently and access physical resources, several conflicts may occur:

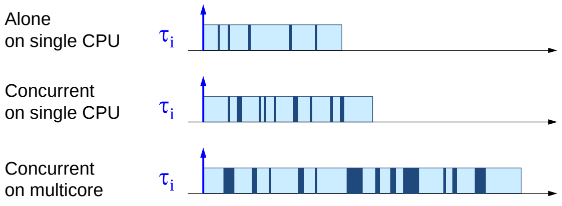


Consequence on WCET

High penalty

In multicore systems task WCETs will be higher due to

- eviction on shared caches
- bus/network arbitration



Randomness of interference

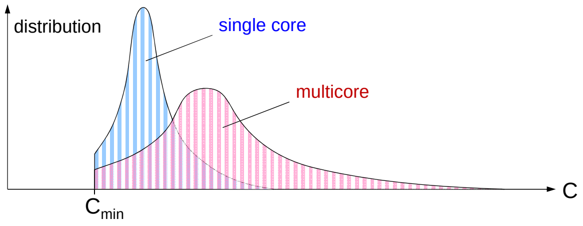
- Interference depends on several factors, (such as allocation, task flow, specific data inputs, task activation times), all summing up and contributing to its randomness.
- When more cores are used, inter-core interferences increase.
- However, the random nature of interference may introduce deviations from the average case, which explain why the WCET on 8 cores is less than WCET on 7 cores.
- The implication of this phenomenon is that worst-case timing analysis, testing, and certification becomes extremely complex!

WCET distribution

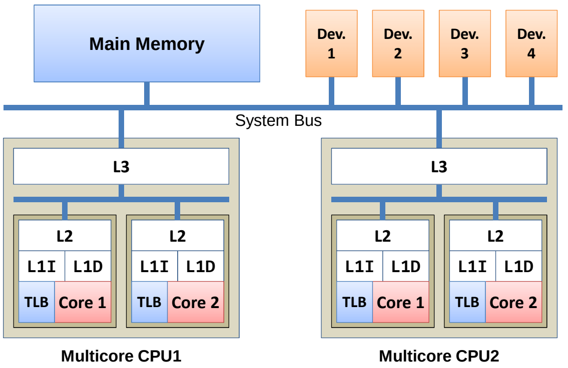
High uncertainty

Execution times vary more, because interference depends on

- phase between cores (synchronization, scheduling)
- access pattern to shared resource (program paths)
- accessed memory locations (program state)

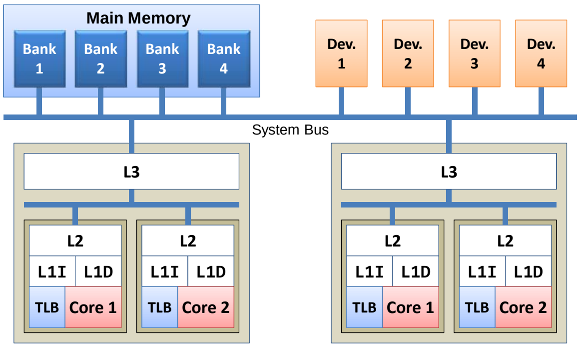


Typical multicore platform



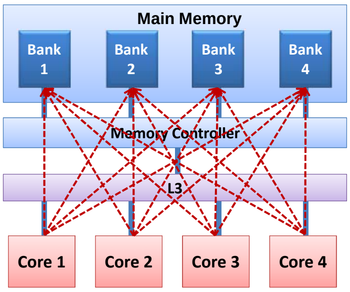
Memory banks

To reduce memory conflicts, the DRAM is divided into **banks**:



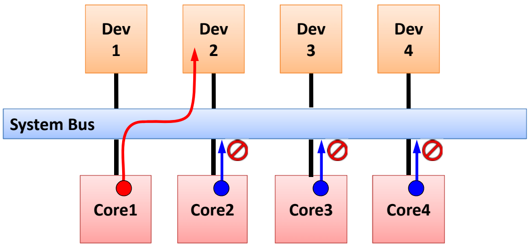
Main memory conflicts

Still, when cores concurrently access the main memory, DRAM accesses have to be queued, causing a significant slowdown:

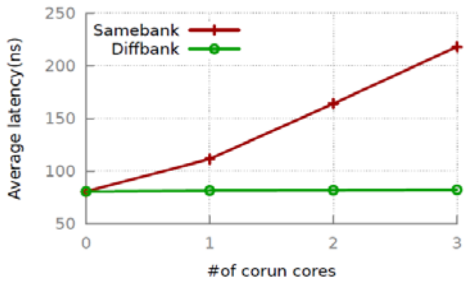


I/O conflicts

A similar problem occurs when tasks running in different cores request to access I/O devices at the same time:



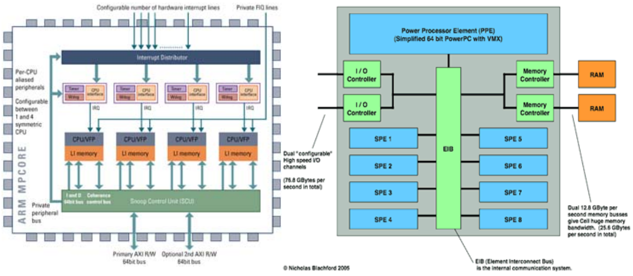
Test on Intel-Xeon



- Diffbank: Core0 → Bank0, Core1-3 → Bank 1-3
- Samebank: All cores → Bank0

Types of multicore systems

ARM's MPCore **STI's Cell Processor**



- 4 identical ARMv6 cores

- One Power Processor Element
- 8 Synergistic Processing Element

Expressing parallelism

Code parallelization can be done at different levels:

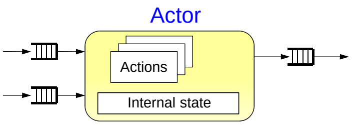
- Parallel programming languages
  (e.g., Ada, Java, CAL).

- Code annotation.
  The information on parallel code segments and their dependencies is inserted in the source code of a sequential language by means of special constructs analyzed by a pre-compiler (e.g., OpenMP).

Expressing parallelism

For instance, CAL [UC@Berkeley, 2003] is a dataflow language.

- Algorithms are described by modular components (actors), communicating through I/O ports:



- Actions read input tokens, modify the internal state, and produce output tokens.

Expressing parallelism

OpenMP specifies parallel code by the pragma directive.

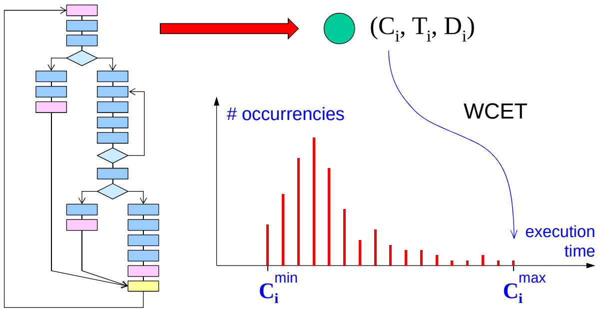
For instance, the following for statement is executed as ***n*** parallel threads:

```
#pragma omp parallel for
for (i=0; i<n; i++)
    b[i] = a[i] / 2.0;
```

In any case, a suitable task model is needed to represent and analyze parallel applications.

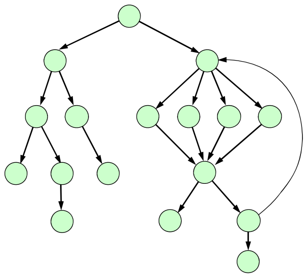
Task model

A sequential task can be efficiently represented by the Liu & Layland model, described by 3 parameters:



Task model

Representing a parallel code requires more complex structures like a graph:

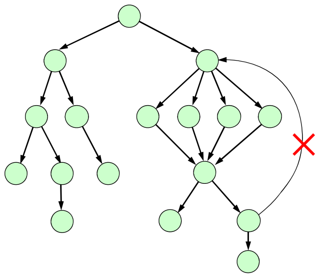


Restrictions are needed to simplify the analysis

Graph models

Directed Acyclic Graphs

A Directed Acyclic Graphs (DAG) is a graph in which links have a direction and there are not cycles:

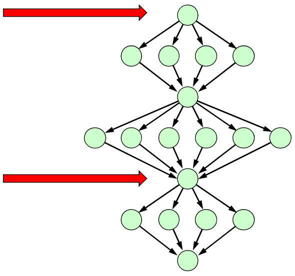


In a DAG this connection is forbidden

Fork-Join graphs

Computation is view as a sequence of parallel phases (fork nodes) followed by synchronization points (join nodes):

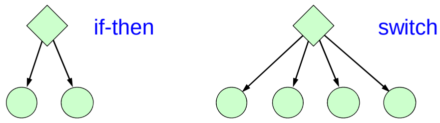
- After a fork node, all immediate successors must be executed (the order does not matter).
- A join node is executed only after all immediate predecessors are completed.



Conditional graphs

They are graphs in which there are nodes that express a conditional statement:

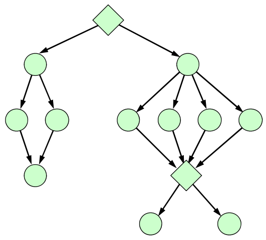
- Only one node among all immediate successors must be executed, depending on the data:



And-Or Graphs

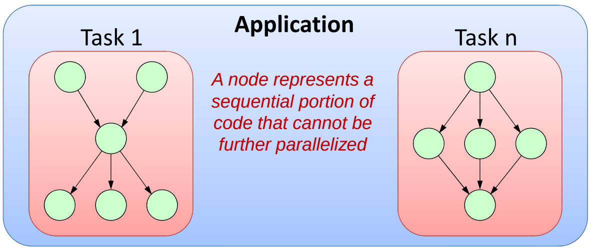
It is the most general graph representation where:

- OR nodes represent conditional statements (◇)
- AND nodes represent parallel computations (○)



Application model

An application can be modeled as a set of tasks, each described by a task graph:



A node represents a sequential portion of code that cannot be further parallelized

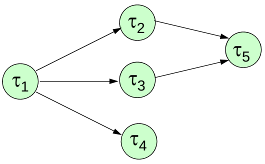
A task graph specifies the maximum level of parallelism

Assumptions and parameters

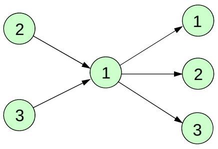
- **Arrival pattern**
  - Periodic (activations exactly separates by a period T)
  - Sporadic (Minimum Interarrival Time T)
  - Aperiodic (no interarrival bound exists)
- Is **preemption** allowed at arbitrary times?
- Is task **migration** allowed?

Task parameters:

$\{C_1, C_2, C_3, C_4, C_5\}$, D, T



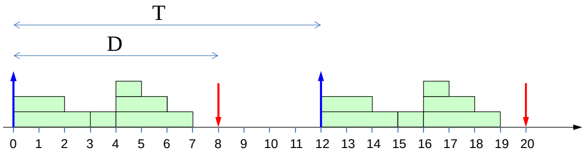
Example



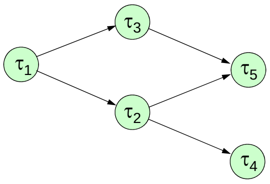
Task parameters:

$\{C_1, C_2, C_3, C_4, C_5, C_6\}$, D, T

Interpretation on an unlimited number of cores



Important factors



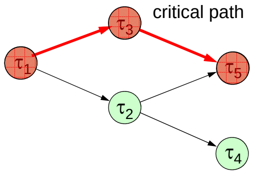
Sequential Computation Time (Volume):

$$C^s = \sum C_i$$

Required CPU bandwidth: $U = \frac{C^s}{T}$

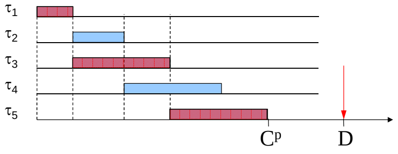
$(C^s \le D) \Rightarrow$ A is feasible on a single core

Important factors



Parallel Computation time

C^p = length of a critical path



$(C^p > D) \Rightarrow$ A is not feasible in any number of cores

Performance issues

Assuming we are able to express the parallel structure of our source code,

- How much performance can we gain by switching from 1 core to *m* cores?
- How can we measure the performance improvement?

Speed-up factor

It measures the relative performance improvement achieved when executing a task on a new computing platform, with respect to an old one.

$$S = \frac{R_{old}}{R_{new}}$$

R_{old} = response time on the old platform

R_{new} = response time on the new platform

Speed-up factor

If the old architecture is a single core platform and the new architecture is a platform with *m* cores (each having the same speed as the single core one), the speedup factor can be expressed as

$$S = \frac{R_1}{R_m}$$

R_1 = response time on 1 processor

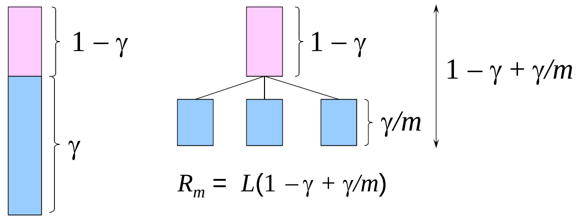
R_m = response time on *m* processors

Speed-up factor

L = length of sequential code

γ = fraction of parallel code

m = number of processors



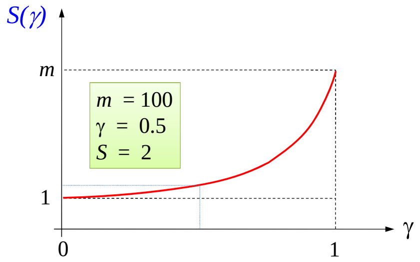
$R_m = L(1 - \gamma + \gamma/m)$

$R_1 = L$

$$S = \frac{R_1}{R_m} = \frac{1}{1 - \gamma + \gamma/m}$$

Speed-up factor

$$S(m,\gamma) = \frac{1}{1-\gamma+\gamma/m}$$ [*Amdahl's law*]

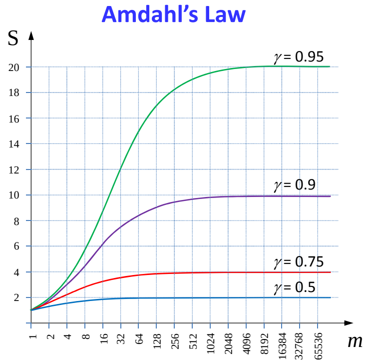


Speed-up factor

$$S = \frac{1}{1-\gamma+\gamma/m}$$

- For large *m*:

$$S(\gamma)_{m\to\infty} = \frac{1}{1-\gamma}$$



Considerations

- Law of diminishing returns:

  Each time a processor is added the gain is lower

- Performance/price rapidly fall down as *m* increases
- Considering communications costs, memory, bus conflicts, and I/O bounds, the situation gets worse
- Parallel computing is only useful for
  - limited numbers of processors, or
  - highly parallel applications (high values of γ)

When MP is not suited

Applications having some of the following features are not suited for running on a multicore platform:

- I/O bound tasks;
- Tasks composed by a series of pipeline dependent calculations;
- Tasks that frequently exchange data;
- Tasks that contend for shared resources.

Other issues

- How to allocate and schedule concurrent tasks on a multicore platform?
- How to analyze real-time applications to guarantee timing constraints, taking into account communication delays and interference?
- How to optimize resources (e.g., minimizing the number of active cores under a set of constraints)?
- How to reduce interference?
- How to simplify software portability?

Multiprocessor models

- **Identical**

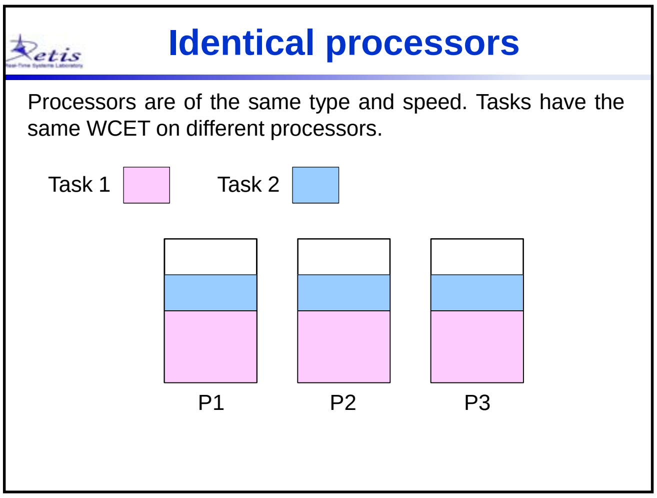
  Processors are of the same type and have the same speed. Each task has the same WCET on each processor.

- **Uniform**

  Processors are of the same type but may have different speeds. Task WCETs are smaller on faster processors.

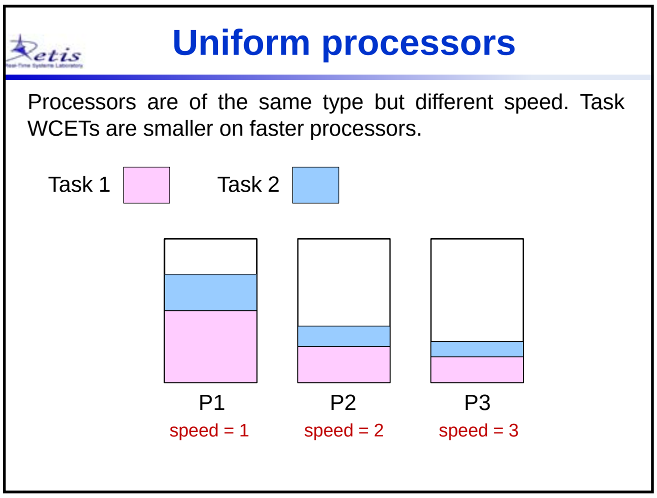
- **Heterogeneous**

  Processors can be of different type. The WCET of a task depends on the processor type and the task itself.



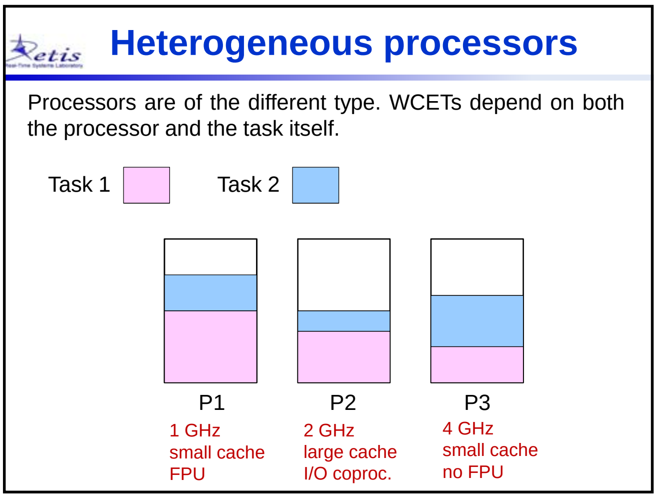
Identical processors

Processors are of the same type and speed. Tasks have the same WCET on different processors.



Uniform processors

Processors are of the same type but different speed. Task WCETs are smaller on faster processors.



Heterogeneous processors

Processors are of the different type. WCETs depend on both the processor and the task itself.