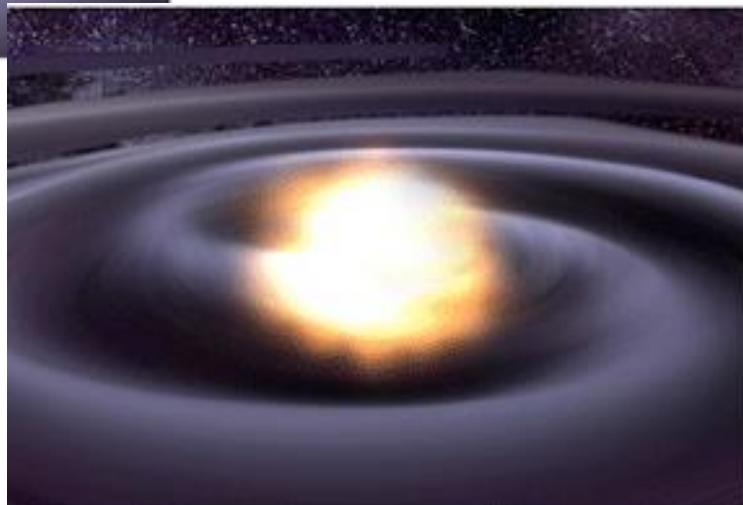
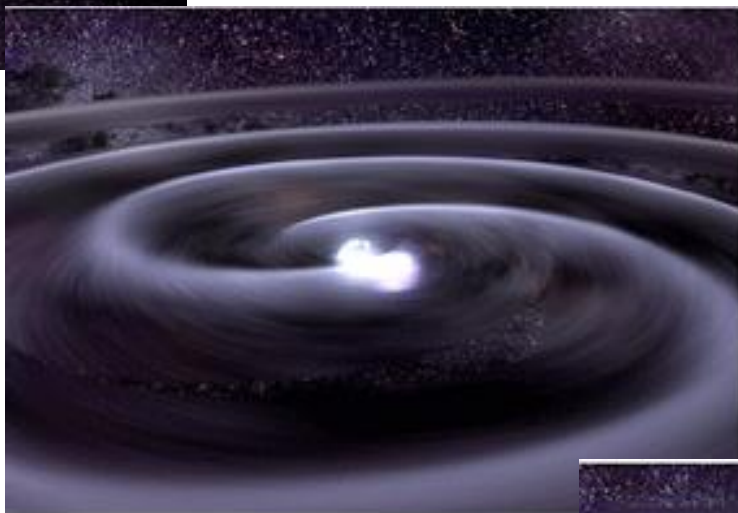
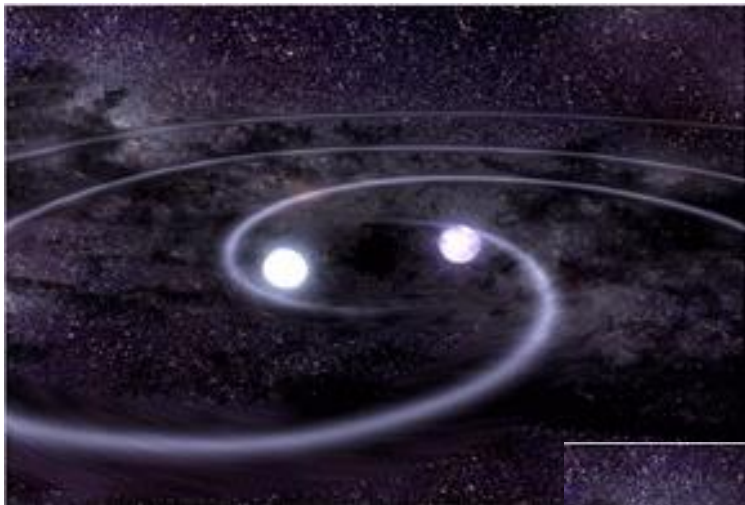
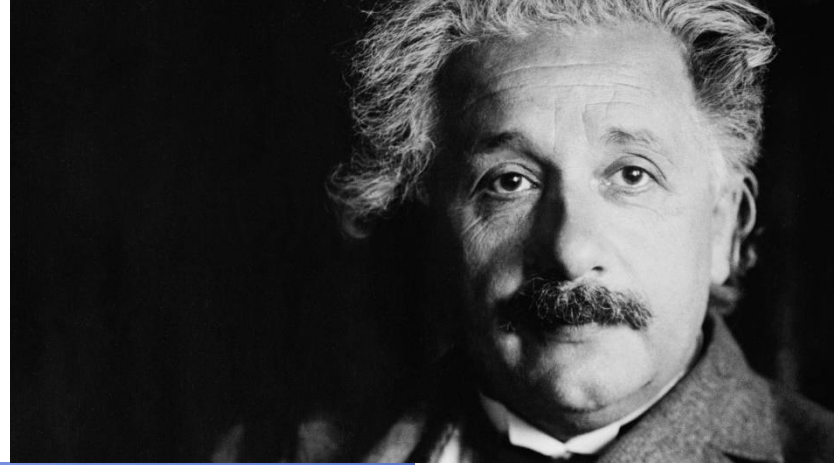


# Practical Parallel Programming I

## »» Introduction

Jan Lemeire  
2021–2022

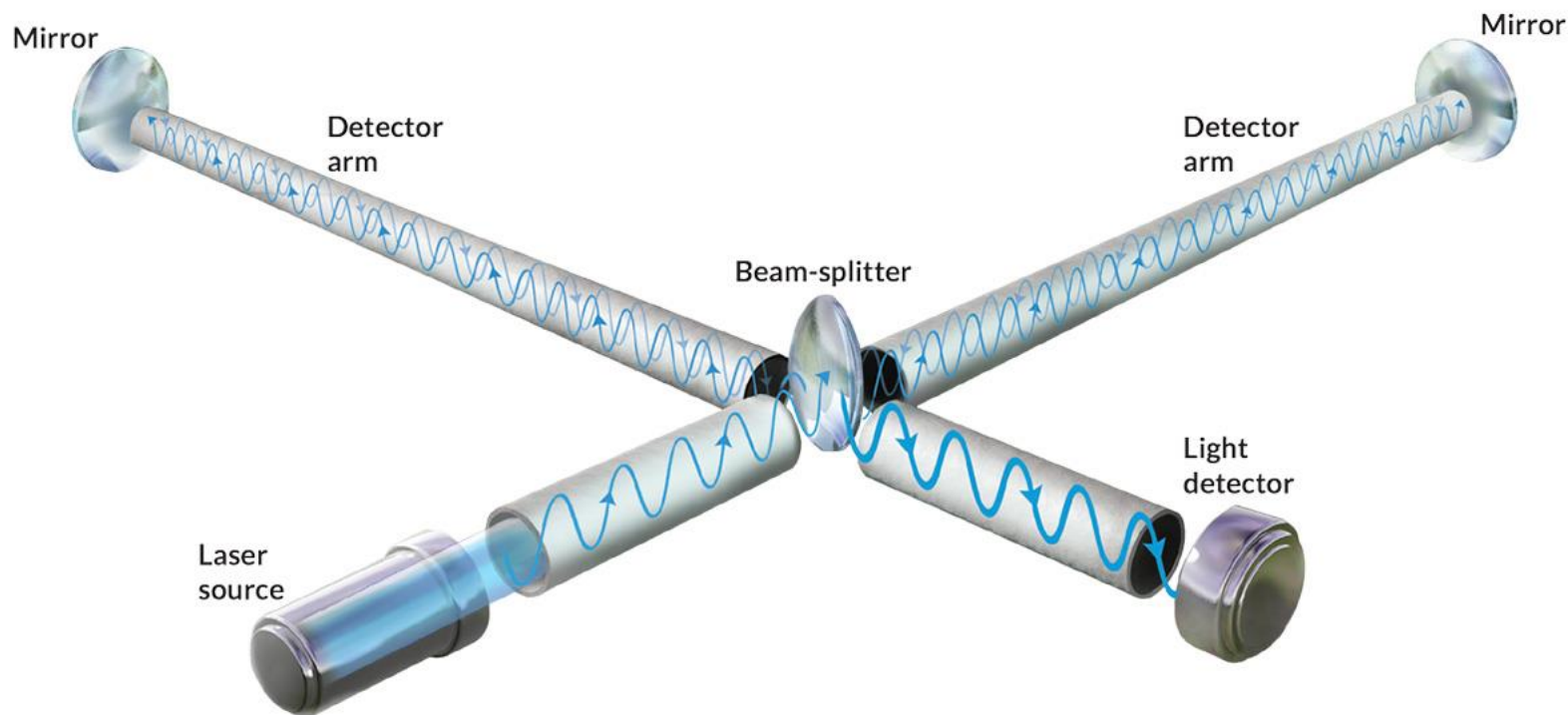
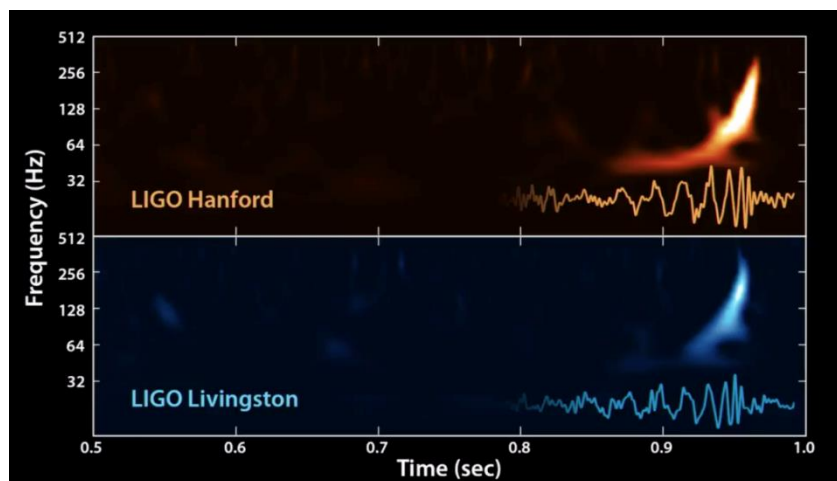




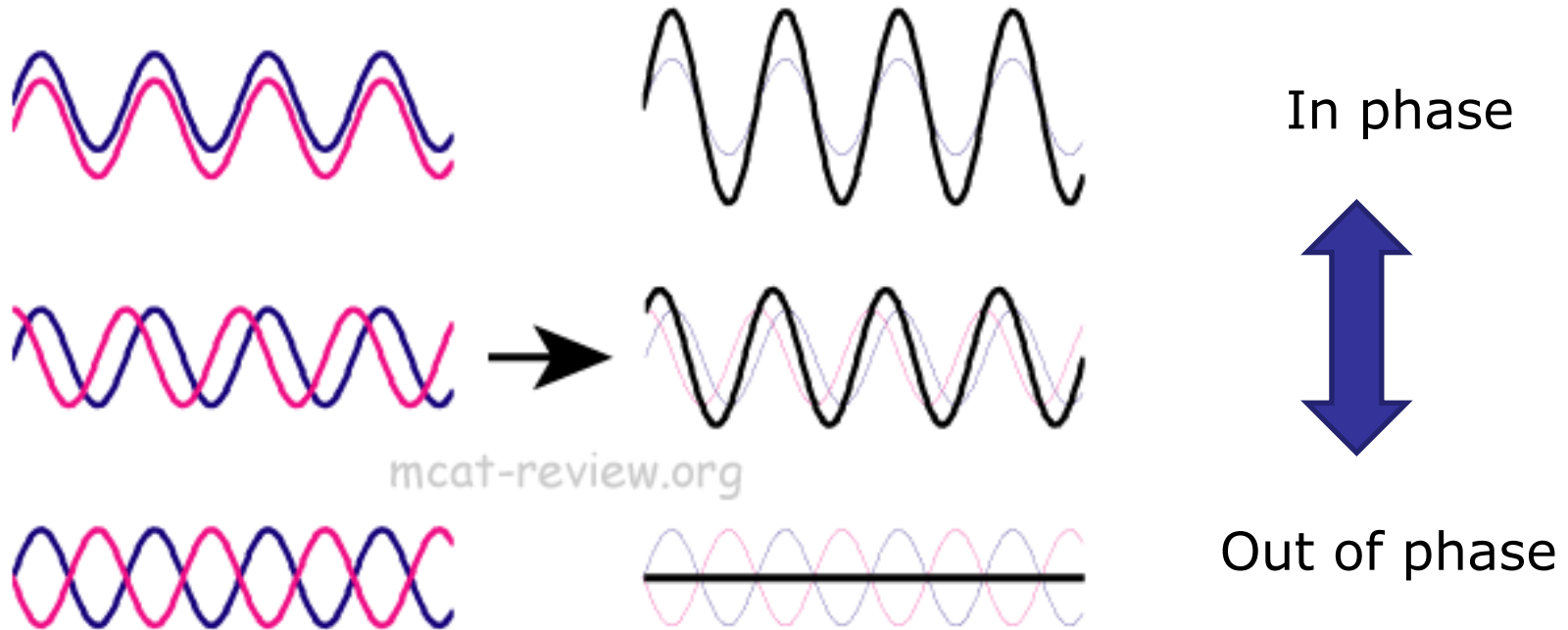
# LIGO

The First Observation  
of Gravitational Waves

September 14, 2015



## Superposition of signals



**UPDATE: August 14, 2017**

**Detection of a gravitational wave by 3 detectors,  
also by the European Virgo.**

**1.8 billion years ago, collision of black holes  
of 25 and 31 times the mass of the sun**

14 augustus 2017 om 12.30.43 uur  
Belgische tijd.

Het verschijnsel is voor het eerst tegelijk waargenomen door drie detectoren, en twee soorten van detectoren: de twee LIGO's in de VS en de Europese Virgo. Voor de Europese Virgo was de waarneming een primeur. Virgo, die een uitgebreide update heeft gekregen en nu de Advanced Virgo heet, was nog maar twee weken opnieuw bezig met het verzamelen van wetenschappelijke gegevens.

De botsing vond plaats op zowat 1,8 miljard lichtjaar afstand.

GW170814, zoals de nieuw ontdekte zwaartekrachtgolf is gedoopt (Gravitational Wave en de datum), is veroorzaakt in de laatste ogenblikken van de versmelting van twee zwarte gaten die 31 en 25 keer zo zwaar waren als de zon

Albert Einstein had in 1915 geopperd dat het heelal bestaat uit ruimte en tijd, en dat die één geheel vormen. Na een heftige gebeurtenis kan die "ruimtetijd" trillen. De schokgolven, zwaartekrachtgolven, gaan dan door het heelal als rimpelingen na een steen in een vijver. Bij zo'n rimpeling rekt de ruimte iets uit of krimpt zij iets. Hoe groter de massa en hoe sneller de beweging van een massarijk object, des te sterker de zwaartekrachtgolf.





06 wetenschap

DE STANDAARD  
VRIJDAG 19 JULI 2019

ok Vlaanderen zet zijn schouders onder de Einstein Telescope

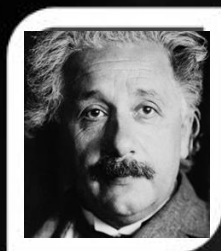
# Drielandenregio wil 'kosmisch afluistercentrum'

De Einstein Telescope komt 200 meter diep  
10 kilometer. © rr

zuiden van

gaten en gravitatiegolven voortvloei-  
gen) te huisvesten. 'Telescope' is overi-

1



THEORY



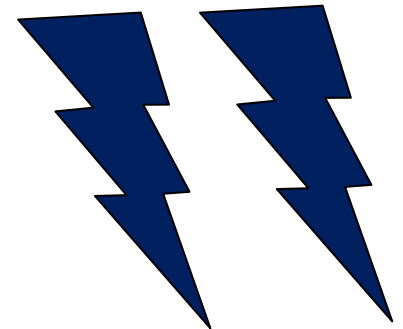
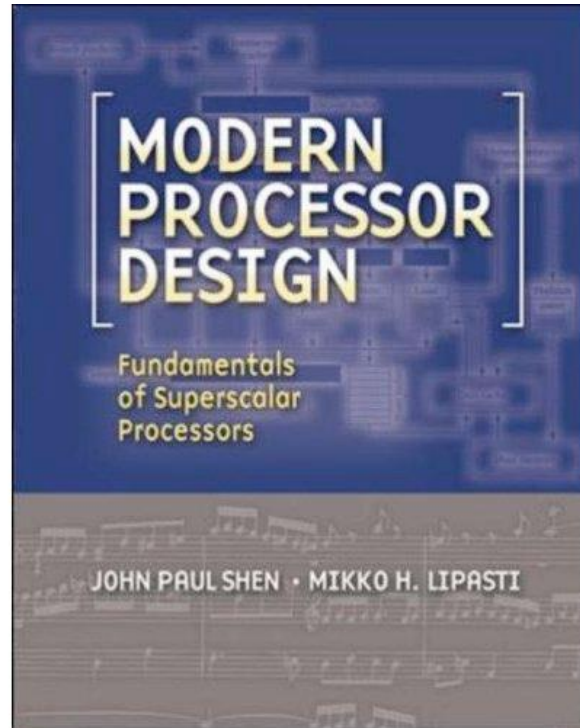
2



EXPERIMENTATION

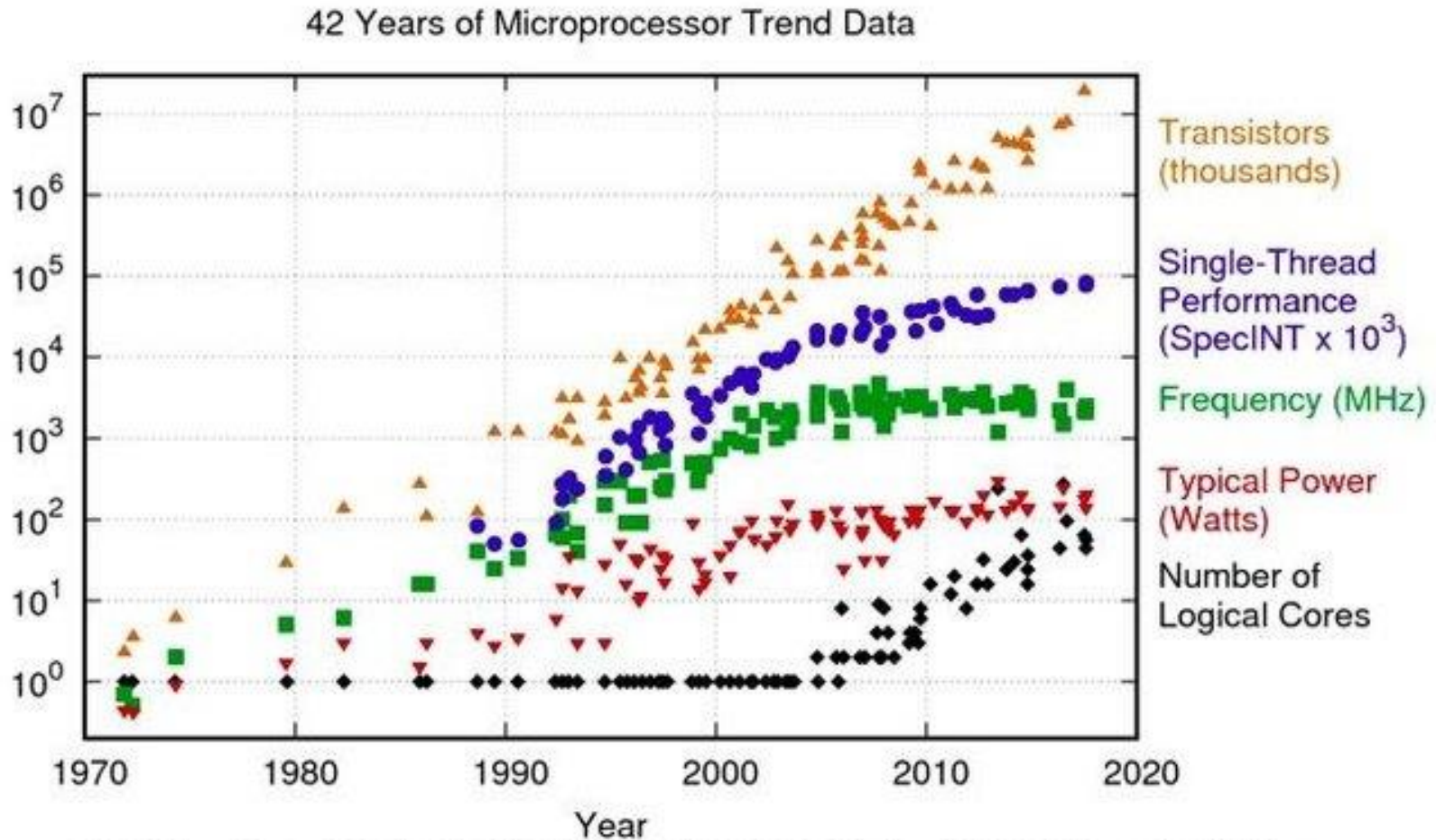


# 2005: “10GHz in 2010”



<https://www.quora.com/Why-havent-CPU-clock-speeds-increased-in-the-last-5-years>

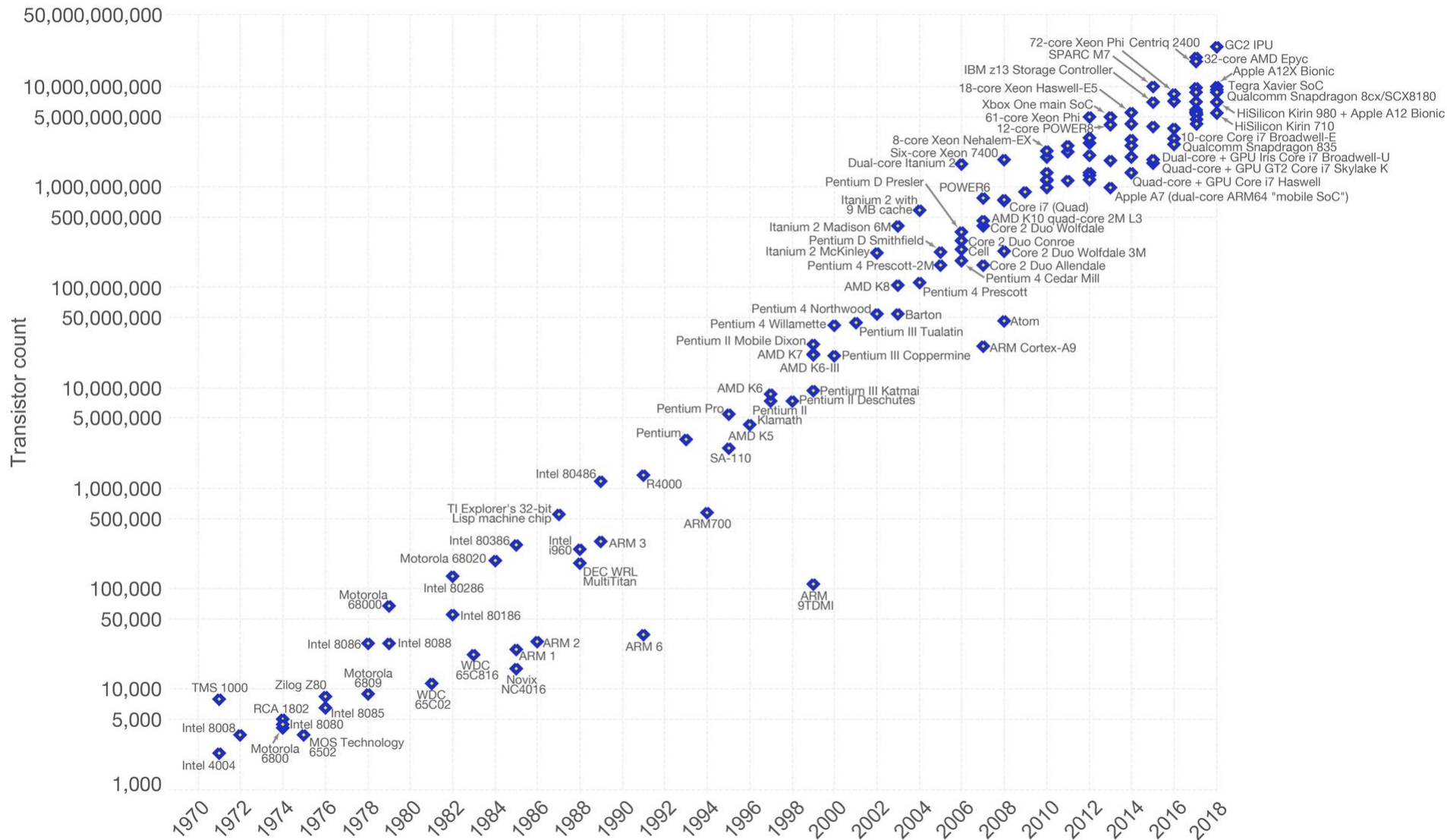
# The Free Lunch is Over?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

# Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



# The Free Lunch is Over

## Why You Don't Have 10GHz Today?

- heat/surface is the problem (power wall)
- 12 nm would mean electric paths of 10 atoms wide!

## *Moreover:*

- memory bottleneck
- instruction level parallelism (ILP) wall

## What about Moore's Law?

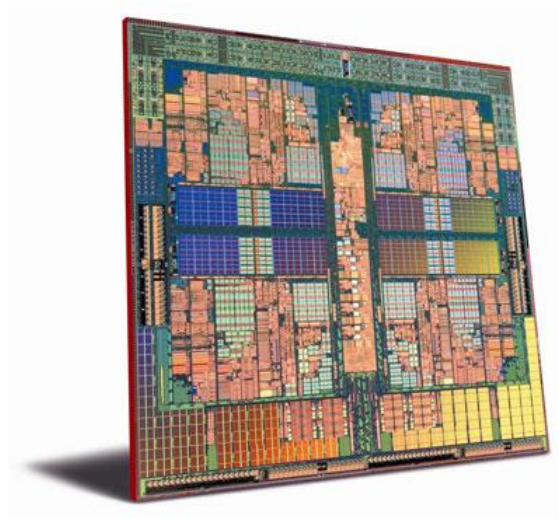
- increase of Clock speed: stopped
- increase of Transistors: ongoing

*It's now about the number of cores!*

<http://www.gotw.ca/publications/concurrency-ddj.htm>



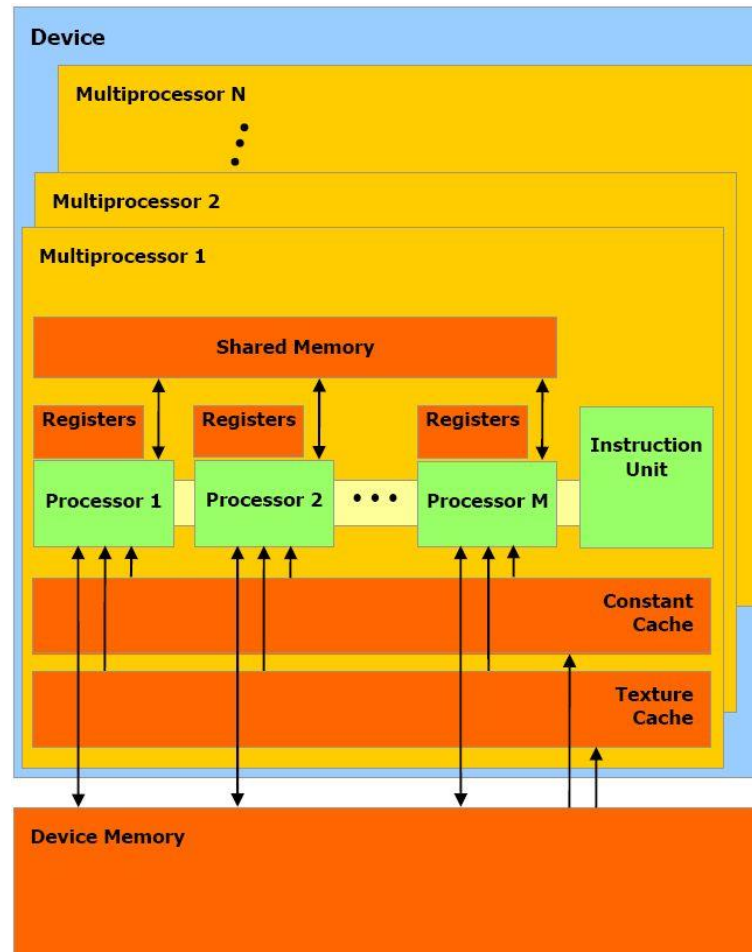
# Multi- & manycores



# Graphics Processors (GPUs)



*Graphics card*

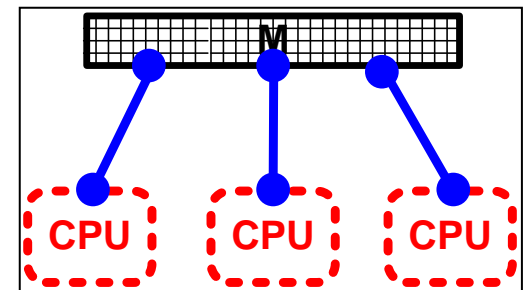
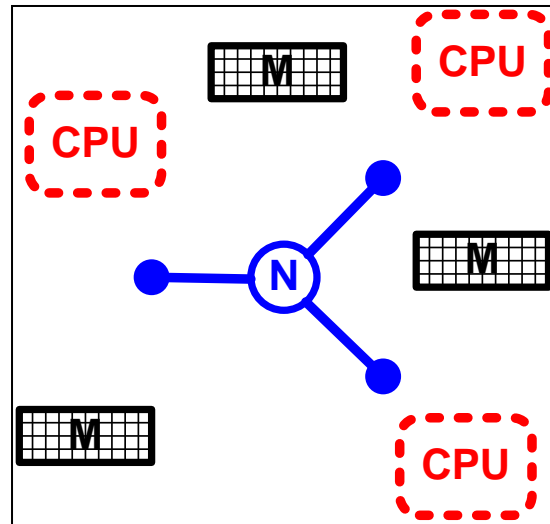
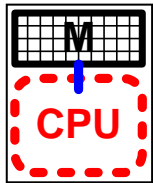


# Overview

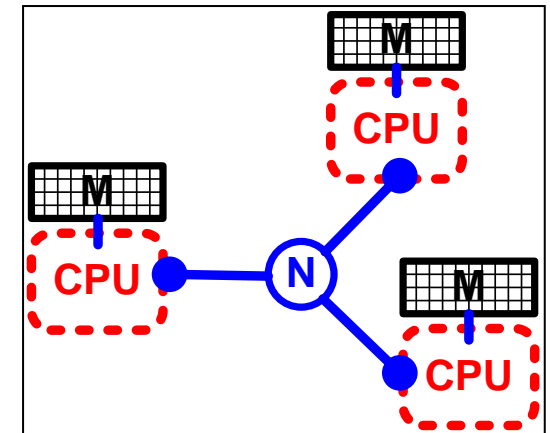


1. Definition
2. Why?
3. Parallel compiler?
4. Parallel architectures
5. Parallel Processing Paradigms
  - Multi-threading.
  - Message-passing.
6. End notes

# What is a Parallel System?



- Memory
- Processors
- Interconnect

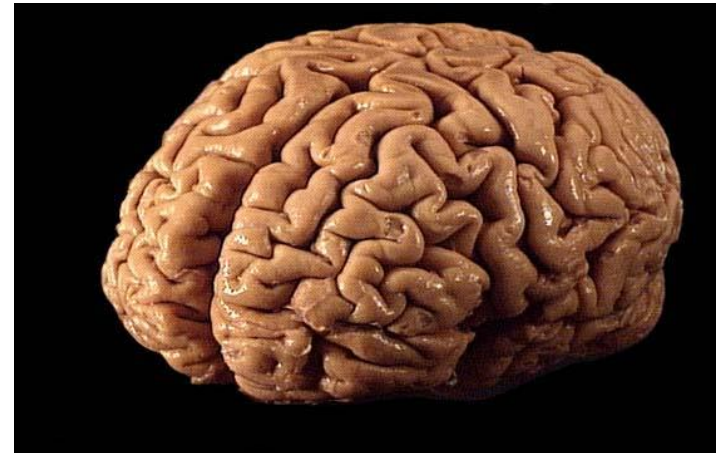




# Biggest Parallel System?



***Internet***



***Brain***

Frequency of brain waves: 10Hz  
Number of neurons: 100 billion =  $10^{11}$

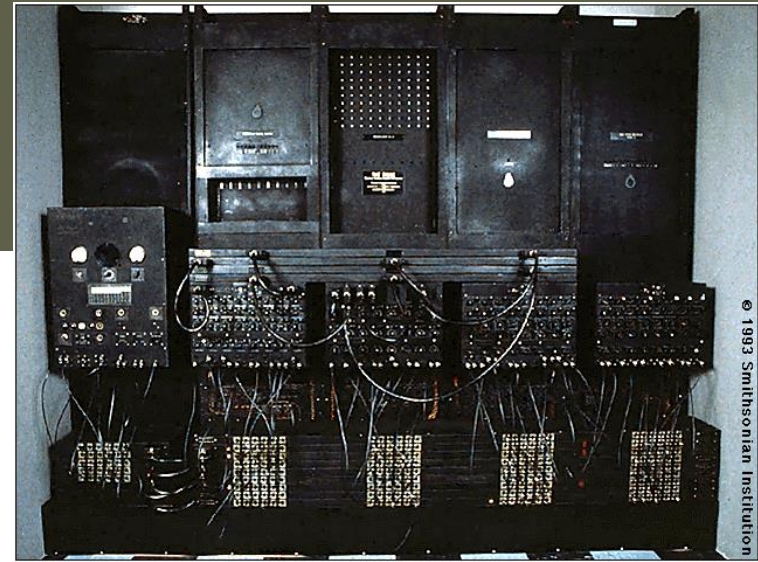
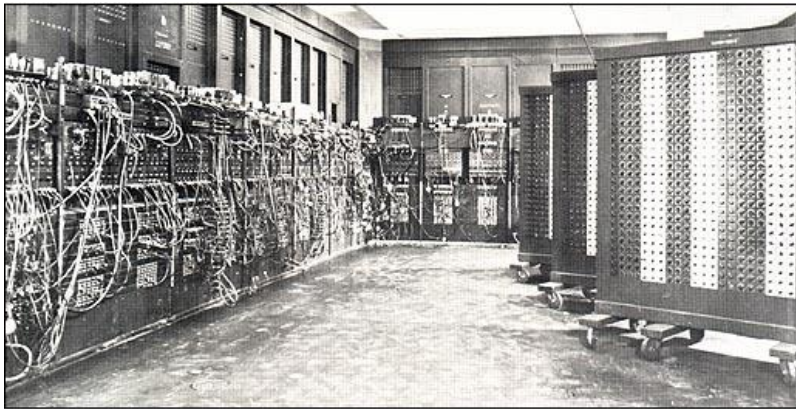
# Overview



1. Definition
2. History
3. Why?
3. Parallel compiler?
4. Parallel architectures
5. Parallel Processing Paradigms
  - Multi-threading.
  - Message-passing.
6. End notes

# ENIAC

The first computer @ WWII



ENIAC's vacuumtubes



By Mauchly and Eckert, 1945

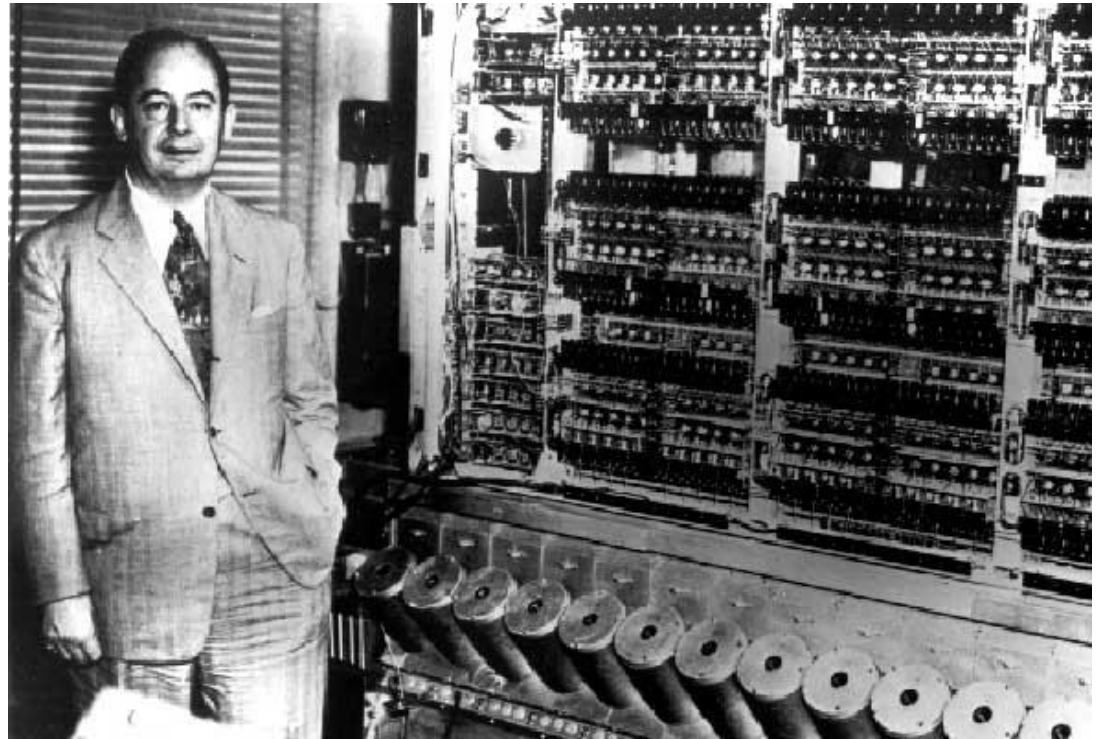


# Von Neumann wants to improve the ENIAC. He builds the EDVAC

And lays the foundations of the modern processor...



**John Von Neumann**





# Processors: the next step

## ◆ ENIAC

- ✦ Parallel processing of data!
- ✦ 'Programming' is done by rewiring the hardware...

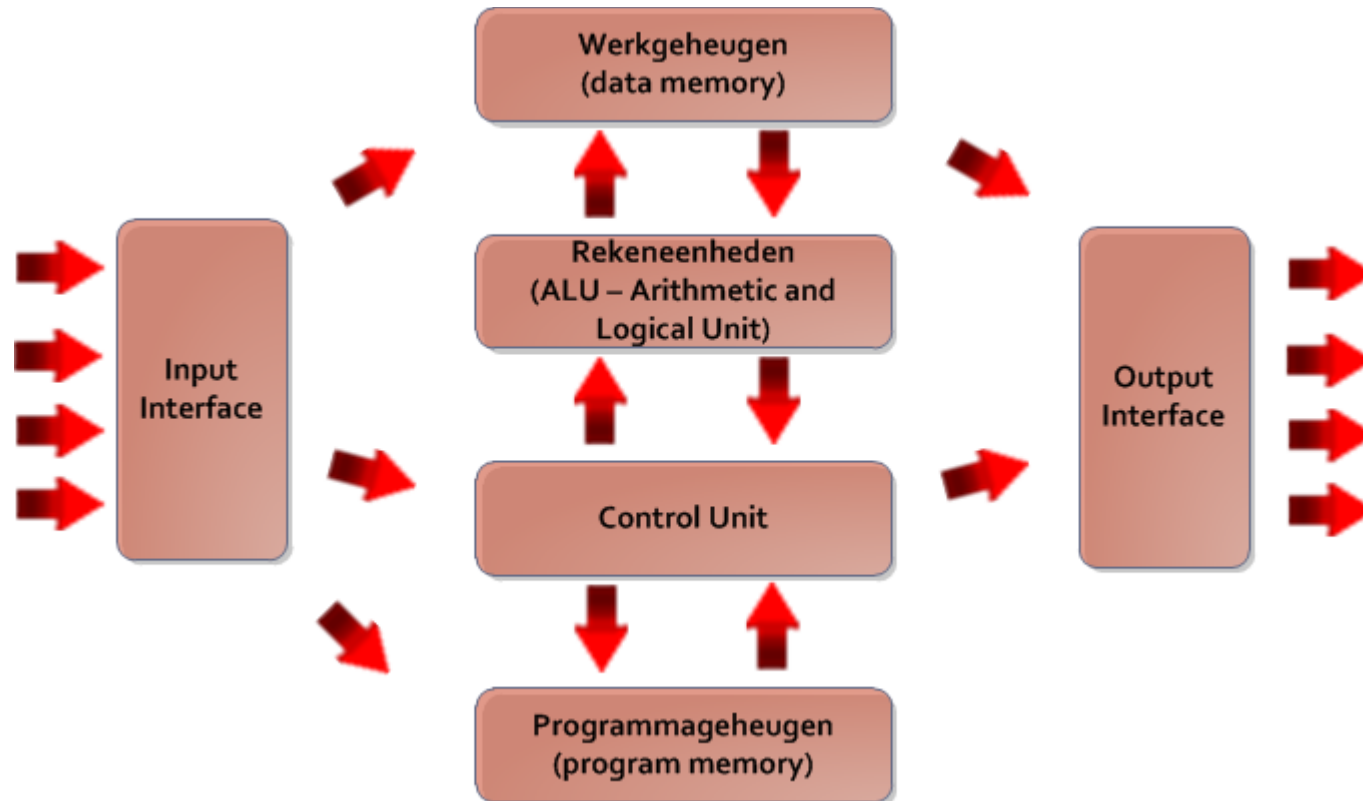
## ◆ EDVAC: Stored-program computer

- ✦ The program is stored in memory
- ✦ The program is also 'input' like the 'data'

## ◆ EDVAC: serial/sequential processing

- ✦ The instructions of the program are executed one-by-one, in order, by the same system which is organized by the Von Neumann architecture

# The Von Neumann architecture



INHERENTLY SEQUENTIAL PROCESSOR

# First parallel computers

## ◆ 1980s, early `90s: a golden age for parallel computing

- ✦ special parallel computers: **Connection Machine, MasPar, Cray (VUB also!)**
- ✦ True supercomputers: incredibly exotic, powerful, expensive
- ✦ Based on vectorization (see further)

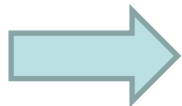
## ◆ But...impact of data-parallel computing limited

- ✦ Thinking Machines sold 100s of systems in total
- ✦ MasPar sold ~200 systems

# Parallel computing now

❖ Parallel computing steamrolled from behind by the inexorable advance of commodity technology

- ✦ Economy of scale rules!
- ✦ Commodity technology outperforms special machines
- ✦ Massively-parallel machines replaced by *clusters* of ever-more powerful *commodity microprocessors*
- ✦ Clusters: federates of standard pcs (MPI & OpenMP)



*In this course we focus on widespread commodity parallel technology*

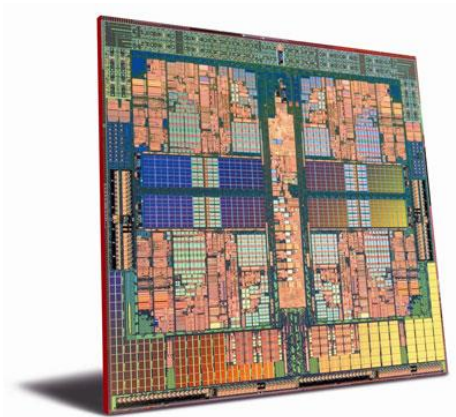
# More...



***Supercomputer***



***Cluster***

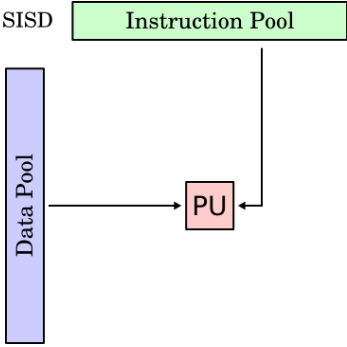
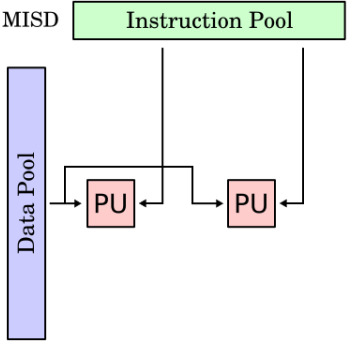
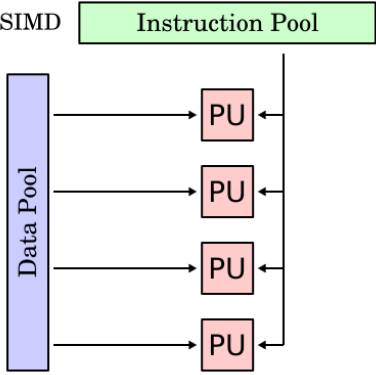
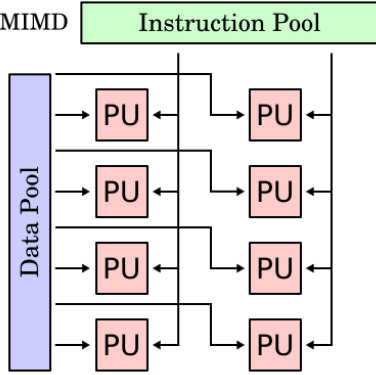


***Multicore***

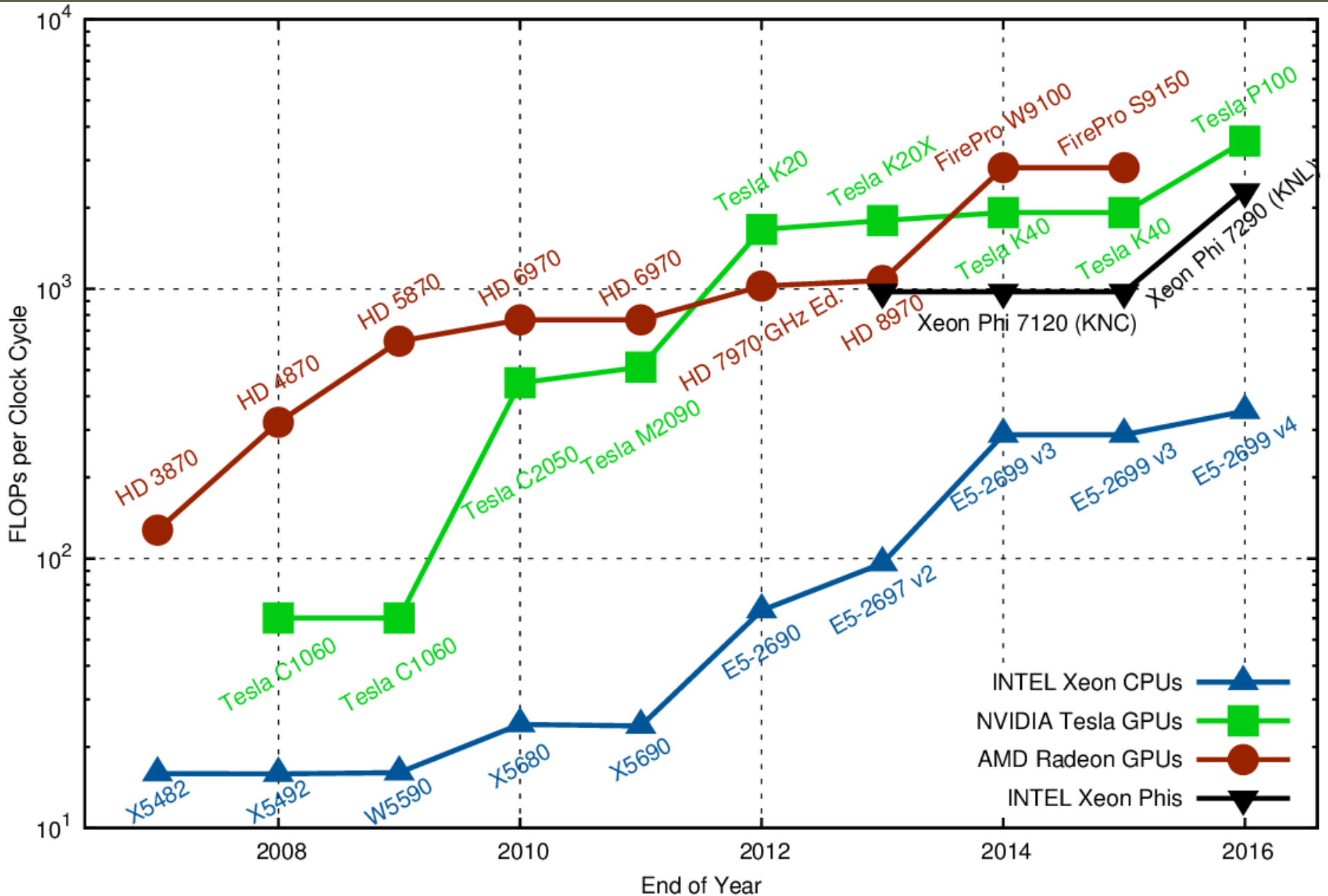
***But also a single core...***



# Flynn's taxonomy of architectures

	Single Instruction	Multiple Instructions
Single Data	<p>SISD</p> 	<p>MISD</p> 
Multiple Data	<p>SIMD</p> 	<p>MIMD</p> 

# CPUs vs GPUs



# Overview



1. Definition

**2. Why?**

3. Parallel compiler?

4. Parallel architectures

5. Parallel Processing Paradigms

- Multi-threading.
- Message-passing.

6. End notes

# Why use parallel systems

- ◆ Complete computation faster
- ◆ More (local) memory available



But... not simple!

Why?? Since a parallelizing compiler does not exist



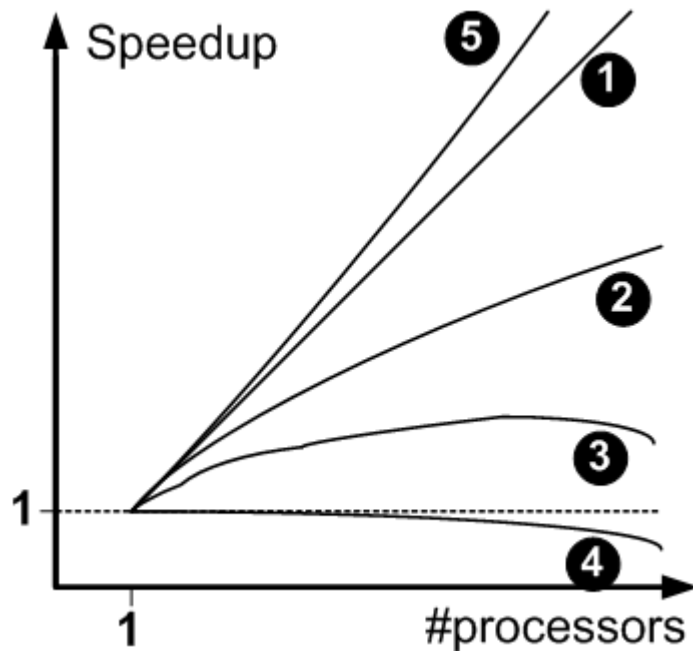


# Speedup

$$Speedup = \frac{T_{seq}}{T_{par}}$$

- ◆ Ideally: speedup = number of processors

# Speedup i.f.o. processors



- 1) Ideal, linear speedup
- 2) Increasing, sub-linear speedup
- 3) Speedup with an optimal number of processors
- 4) No speedup
- 5) Super-linear speedup

# Parallel vs Distributed

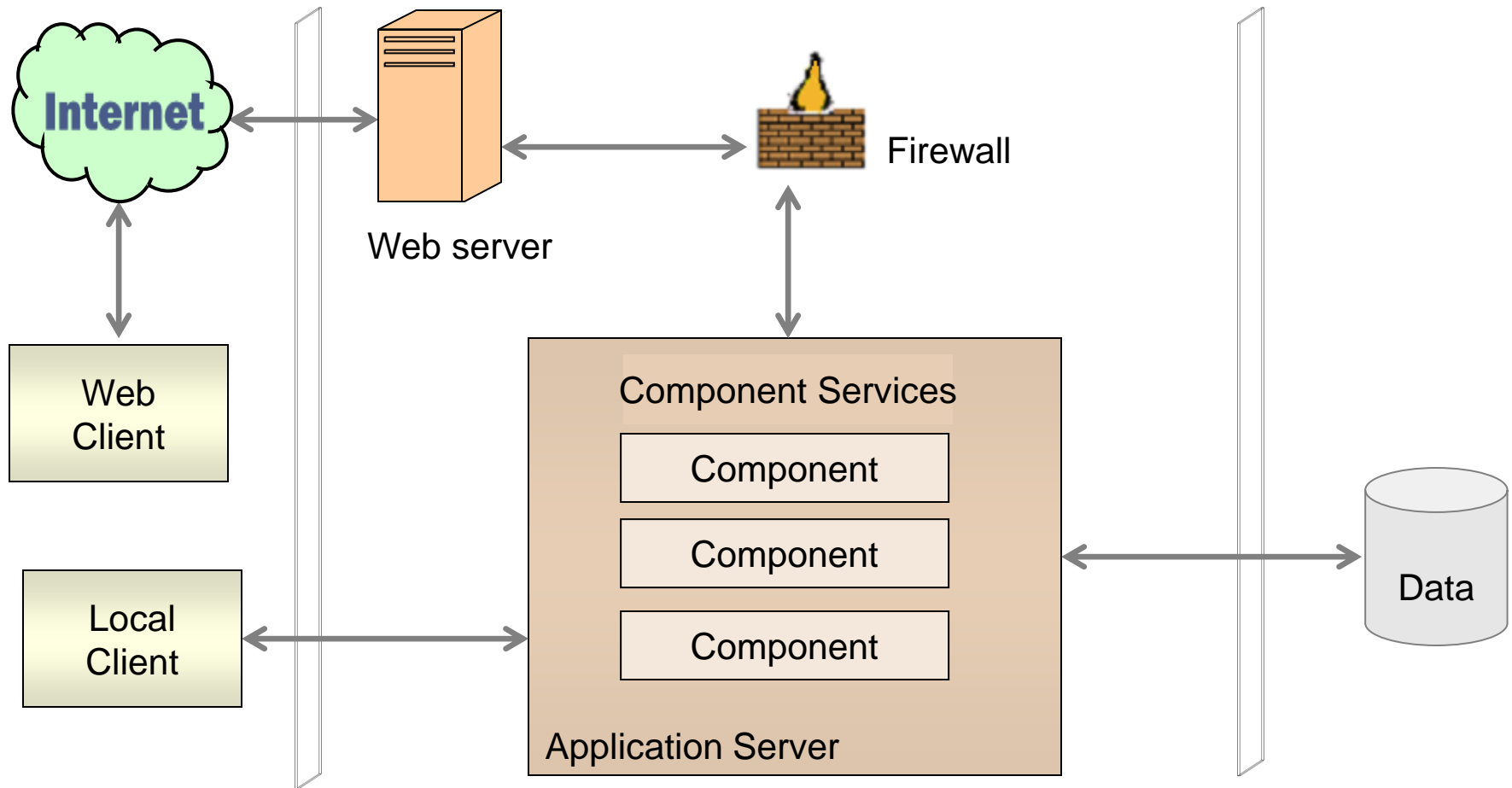
PPP 20-21

## OUR FOCUS



- ◆ 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.

# Example of distributed computing: 3rd generation web application



# Overview



1. Definition

2. Why?

**3. Parallel compiler?**

4. Parallel architectures

5. Parallel Processing Paradigms

- Multi-threading.
- Message-passing.

6. End notes



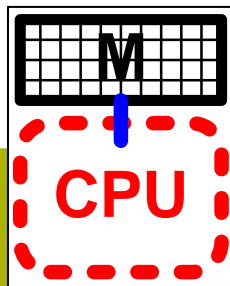
# Sequential programming world

- ◆ Understand this to port it to the parallel world
- ◆ Write Once, Compile Everywhere
  - ✦ C, C++, Pascal, Modula-2, ...
- ◆ Compile Once, Run Everywhere
  - ✦ Java, C#
- ◆ Sequential programming is close to our algorithmic thinking ( $> 2$  GL).
- ◆ Von Neumann architecture provides useful abstraction

# The Random Access Machine

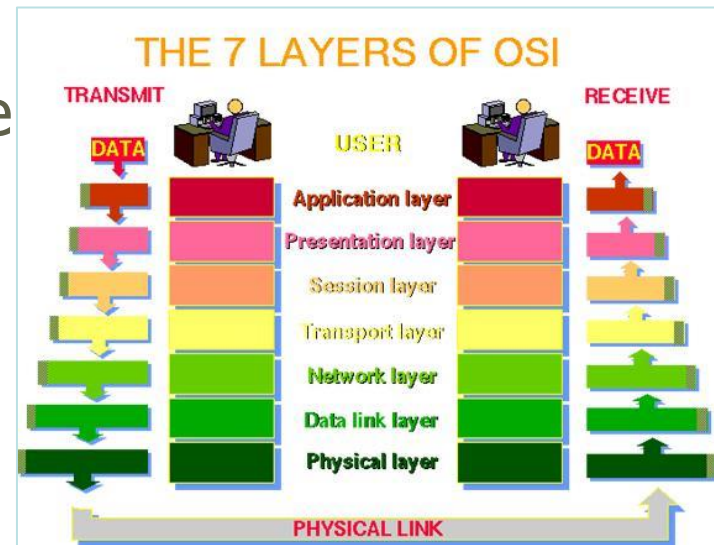
PPP 58-60

- ◆ Sequential computer = *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.
- ◆ Model guides algorithm design.
  - ✦ Programs do not perform well on e.g. vector machines.



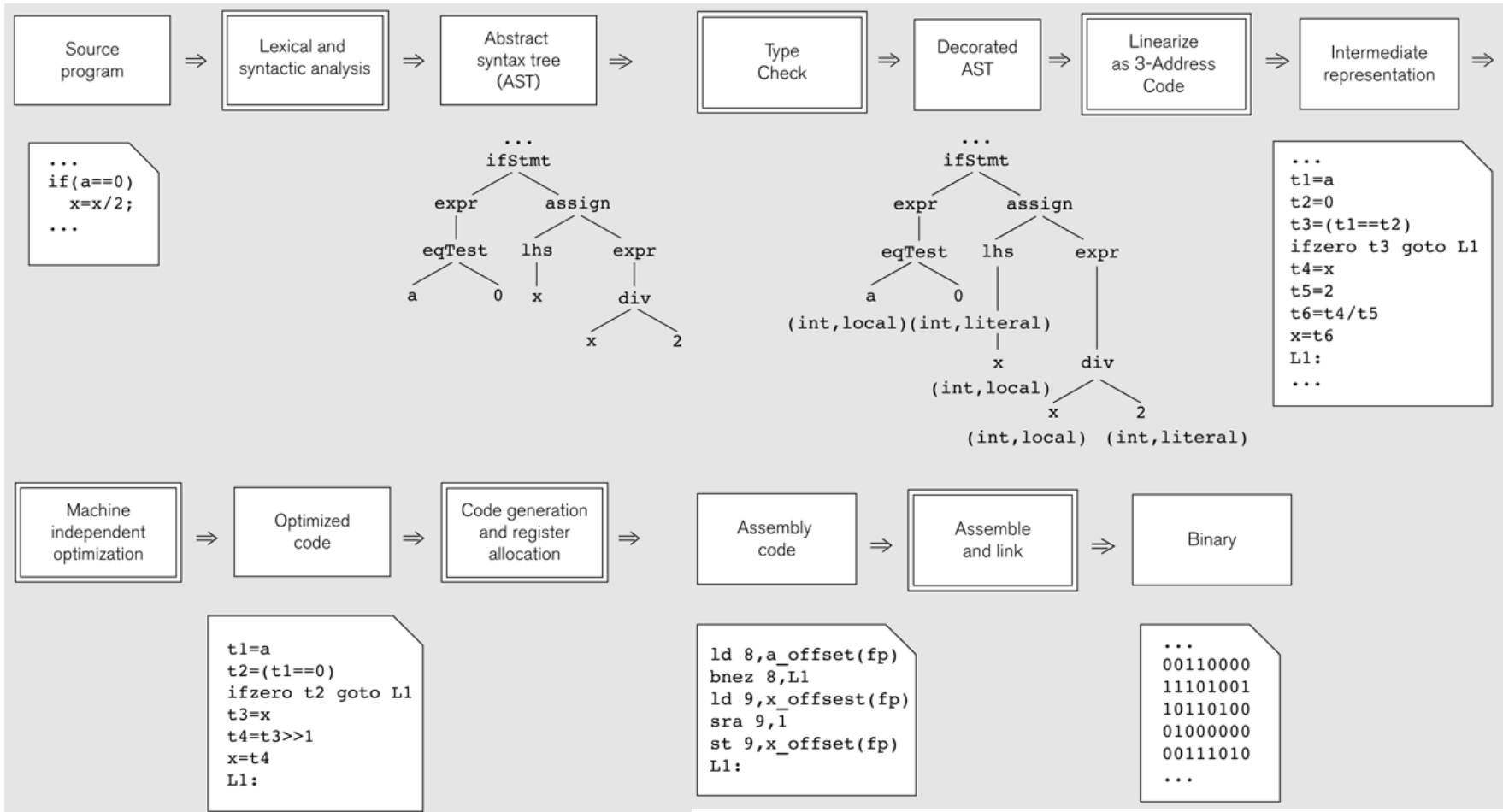
# Software success relies on *abstraction & user transparency*

- ◆ A library offers a service and hides implementation details for you.
- ◆ Layered approaches such as the OSI model in telecommunication
- ◆ 3<sup>rd</sup> generation language => assembler => machine code => machine
  - ✦ Language hides hardware details
- ◆ Software engineering concepts



# Generic Compilation Process

PPP 22-25



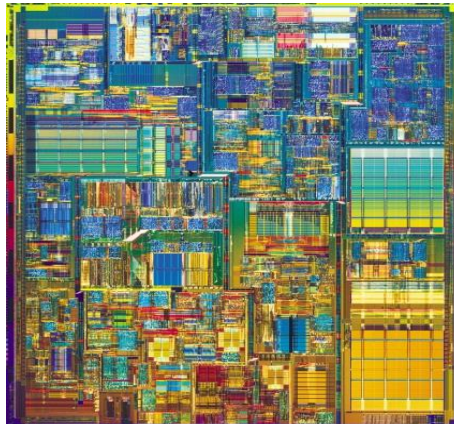
**Algorithm**



**Implementation**



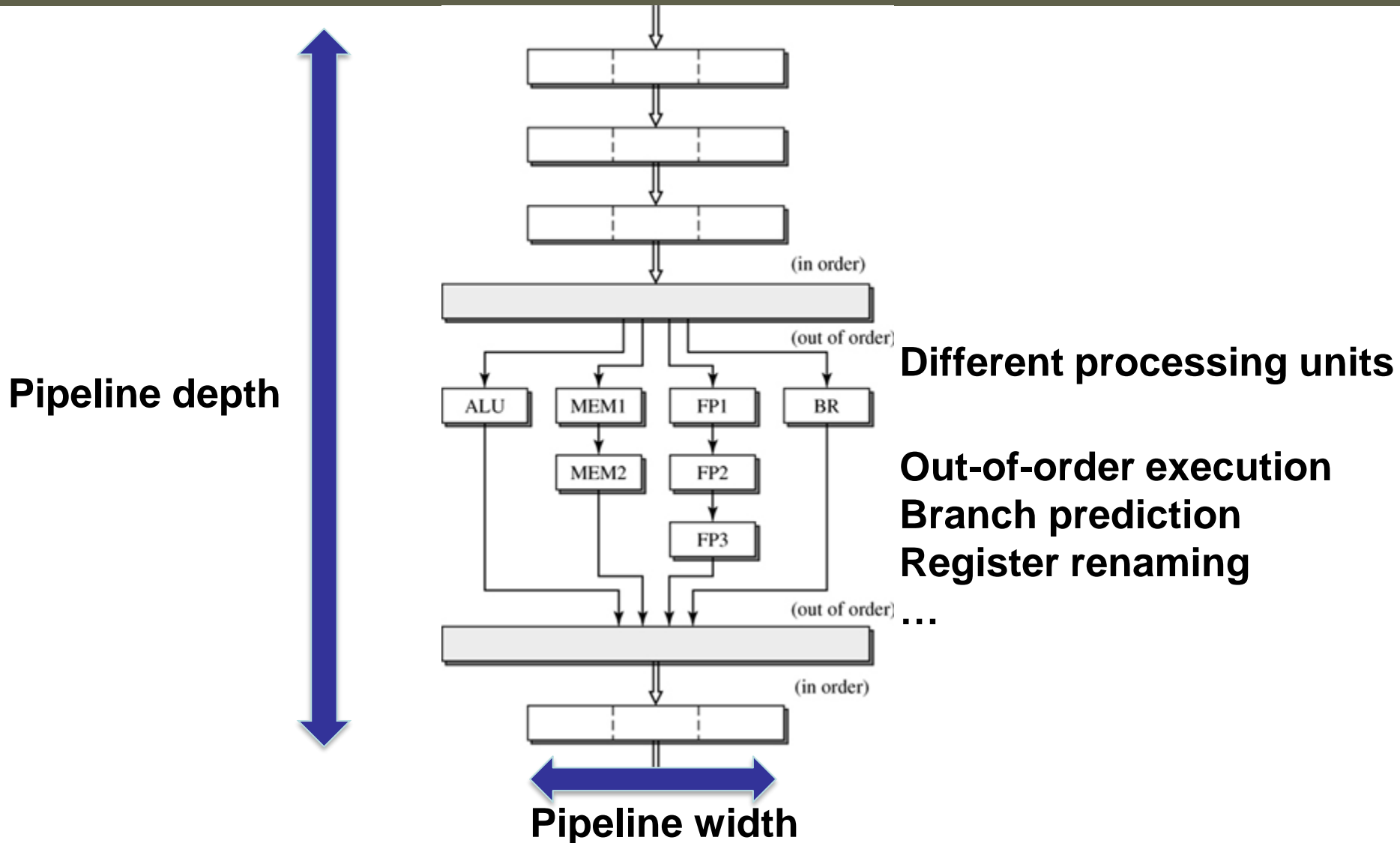
**Compiler**



**Automatic  
optimization**



# 'Sequential' processor: super-scalar out-of-order pipeline



# Speeding up by Instruction-level Parallelism (ILP)

## **Exploit parallelism at the level of instructions:**

1. Pipelining
  - maximal attainable speedup = pipeline depth
2. Start multiple instructions simultaneously
  - maximal attainable speedup = pipeline width
3. Apply instructions on vectors (multiple data elements)
  - maximal attainable speedup = vector width

*Is handled in the next chapter.*

# ILP is not for free

- ◆ There should be enough independent instructions available
  - ✦ Conditional control instructions (branches) limit ILP: the processor doesn't know what the next instruction will be
  - ✦ Data dependencies: instructions need the outcome of previous instructions
  - ✦ ...
- ◆ Programmer, compiler and processor have work to maximally exploit ILP

***In practice, attaining an ILP of 4 seems the maximum***

# Parallel compilers

PPP 22-23

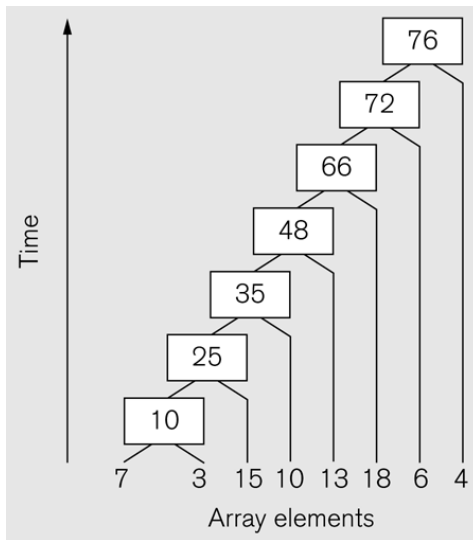
- ◆ *Goal:* automatically compile sequential program into an efficient parallel program that does the same thing.
  - ➔ *Programmers would not have to learn special parallel constructs*
- ◆ *Is a dream that seems beyond reach...*
  - ✦ Many user-defined algorithms contain data dependencies that prevent efficient parallelization.
  - ✦ Automatic dependency analysis and algorithm transformation: still in their infancy, far from optimal. Real breakthrough not expected in the near future.
  - ✦ For efficient parallel programs, a simple hardware model such as the RAM model does not work.

# Example: Iterative Sum

PPP 23

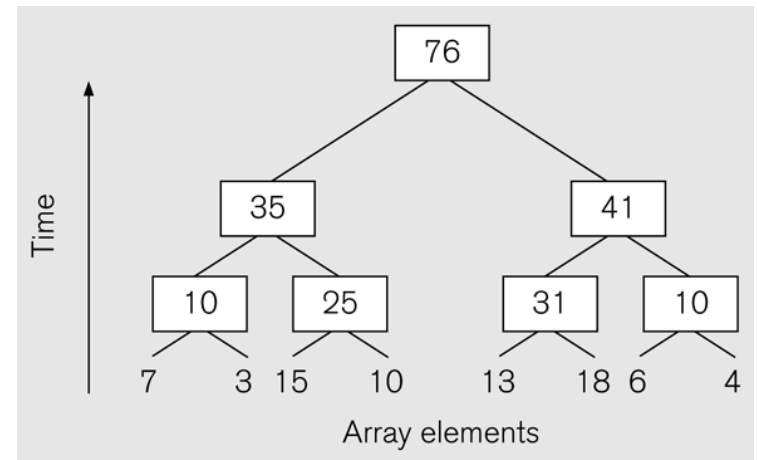
```
n data values  $x_0, \dots, x_n$  in array  $x$   
sum=0;  
for (int i=0;i<n;i++)  
    sum+=x[i];
```

◆ Parallelism? Independent computations needed.



*By associativity  
of sum*

*Can this be done  
by a compiler??*





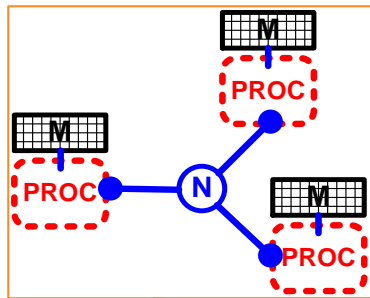
# Overview



1. Definition
2. Why?
3. Parallel compiler?
- 4. Parallel architectures**
5. Parallel Processing Paradigms
  - Multi-threading.
  - Message-passing.
6. End notes

# Parallel Systems

## Distributed memory



Message-  
passing  
MPI

*coarse-grain  
parallelism*

## Shared memory

*coarse-grain  
parallelism*

Explicit  
multi-  
threading

OpenMP

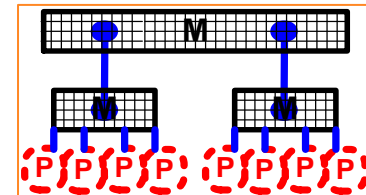
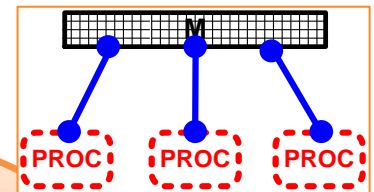
OpenCL/  
CUDA

*SIMT*

Vector  
instructions

*SIMD*





*fine-grain  
parallelism*



# 1. Shared Memory

PPCP 10-12

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

-  Easier to use
  - Through multi-threading
-  Easier to create faulty programs
  - Race conditions
-  More difficult to debug
  - Intertwining of threads is implicit
-  Easier to create inefficient programs
  - Easy to make non-local references

## 2. Distributed Memory

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.

# 3. Fine-grain parallelism

Needs many small pieces that can be processed in parallel.

- 👍 Enormous processing power: vector processors, GPUs
- 👎 No single programming model
  - OpenCL versus vectorization
- 👎 Harder to program
- 👎 Independence & locality & high computational intensity needed to reach peak performance.

# Overview

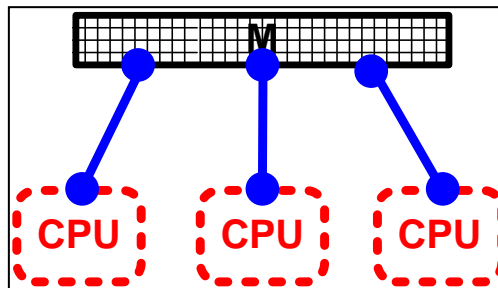


1. Definition
2. Why?
3. Parallel compiler?
4. Parallel architectures
- 5. Parallel Processing Paradigms**
  - **Multi-threading.**
  - **Message-passing.**
6. End notes



# 1. Multithreading

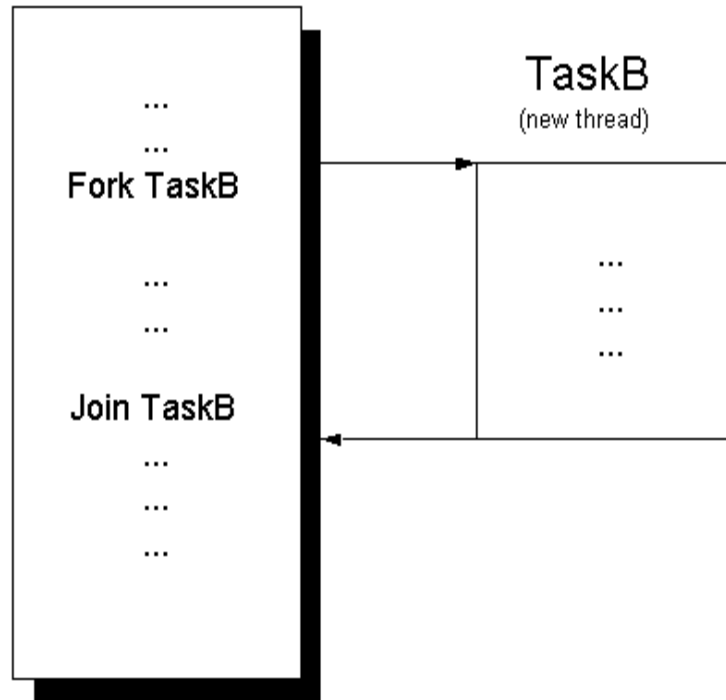
- ◆ One process is split into separate threads,
  - ✦ executing a different sequence of instructions
  - ✦ having access to the same memory
- ◆ = ***Shared address space approach***



# Multi-threading primitives

## ◆ Fork & join

Master process



Parallelism: 2 programs running at the same time

# Example: Counting 3s

PPP 29

n data values  $x_0, \dots, x_n$   
in array *array*

```
count=0;
for (int i=0;i<array.length;i++)
    if (array[i] == 3)
        count++;
```

- ◆ Parallelism? Yes.
- ◆ Multithreaded solution: divide counting

# Multithreaded Counting 3s (C++)

*parameters: array arr of size n, NBR\_THREADS*

```
void count_function(int threadID, int n, int* arr, int* count) {  
    for (int i = 0; i < n; ++i)  
        if (arr[i] == 3)  
            (*count)++;  
}
```

**Note: this program is faulty  
Will be discussed later**

```
vector<thread*> threads; // vector of pointers to threads  
const int ELEMENTS_PER_THREAD = n / NBR_THREADS, count = 0;  
  
// *** STARTING THE THREADS  
for (int t = 0; t < NBR_THREADS; t++)  
    // pass the function to be executed and all the necessary  
    parameters  
    threads.push_back(new thread(count_function, t,  
ELEMENTS_PER_THREAD, arr + t * ELEMENTS_PER_THREAD, &count));  
  
// *** waiting for all threads to finish  
for (int t = 0; t < threads.size(); t++) {  
    threads[t]->join();  
    delete threads[t];  
}
```

# Counting 3s: experiments

## On a dual core processor

Counting 3s in an array of 1000 elements and 4 threads:

- \* Seq : counted 100 3s in 234us
- \* Par 1: counted 100 3s in 3ms 615us

Counting 3s in an array of 40000000 elements and 4 threads:

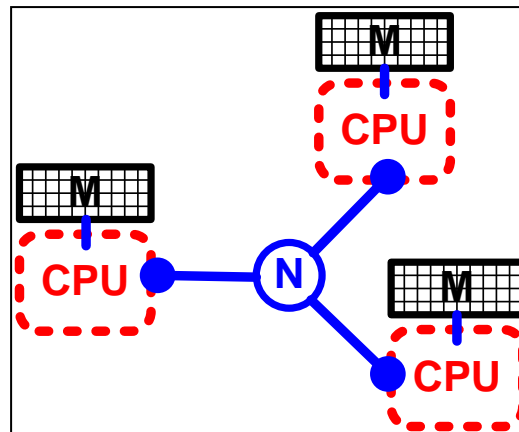
- \* Seq : counted 4000894 3s in 147ms
- \* Par 1: counted 3371515 3s in 109ms

# 2. Message-passing

## ◆ Different processes

- ✦ Communicate through messages
- ✦ Got their own dedicated memory (and got full control over it)

## ◆ = ***Message-passing approach***





# Messages...

- ◆ The ability to send and receive messages is all we need

- ◆ void Send(message, destination)
- ◆ char\* Receive(source)
- ◆ boolean IsMessage(source)

- ◆ But... we also want performance!
  - ➔ More functions will be provided

# Message-passing Counting 3s

```
int count3s_master(int* array){
    int length_per_slave=array.length/nbr_slaves;
    for (slave: slaves)
        send integer subarray of length length_per_slave to slave;

    int sum=0;
    for (slave: slaves)
        sum+= receive integer from slave;
    return sum;
}

int count3s_slave(){
    int* array = receive array from master;
    count=0;
    for (int i=0;i<array.length;i++)
        if (array[i] == 3)
            count++;
    send count to master;
}
```

**pseudo code**

**= sequential program!**

# Focus on low-level approaches

- ◆ MPI, multi-threading & vectorization: low-level primitives
- ◆ Higher-level alternatives exist, but have not proven to be successful for a wide variety of parallelization problems
  - ✦ Fail to hide low-level aspects
- ➔ We'll focus on the 3 main low-level approaches
- ➔ You will be able to learn/use the other approaches yourself

# Overview



1. Definition
2. Why?
3. Parallel compiler?
4. Parallel architectures
5. Parallel Processing Paradigms
  - Multi-threading.
  - Message-passing.
- 6. End notes**

# The goals of this course

PPP 39-41

Learn to write good parallel programs, which

- ◆ Are **correct**
- ◆ Achieve **good performance**
- ◆ Are **scalable** to large numbers of processors
- ◆ Are **portable** across a wide variety of parallel platforms.
- ◆ Are **generic** for a broad class of problems.

# To attain goals...

- ◆ Master low-level and high-level IT skills
  - ✦ Low-level: hardware and system
  - ✦ High-level: Software engineering
- ◆ Combine knowledge and inventivity
- ◆ Approach: look at it as a user who wants to know as little as possible