

# Introduction to parallel and high-performance computing

Simon Scheidegger

[simon.scheidegger@gmail.com](mailto:simon.scheidegger@gmail.com)

July 12<sup>th</sup>, 2018

Open Source Macroeconomics Laboratory – BFI/UChicago

Including adapted teaching material from books, lectures and presentations by  
B. Barney, G. Hager, C. Gheller, P. Koumoutsakos, O. Schenk, G. Wellein

# Day 2, Thursday – July 12<sup>th</sup>

1. Introduction to parallel and high-performance computing (8.00 - 9.30).
2. An ultra-short introduction to C++ (9.45 - 10.15 – hands on).
3. Basics on code optimization & OpenMP session I (10.30 - 11.30 – hands on).
4. Introduction to Projects (11.40 - 12.00 – hands on).
5. Exercise sheet related to the day's topics (12-00-..).

# From making fire to flying rockets in 1 week



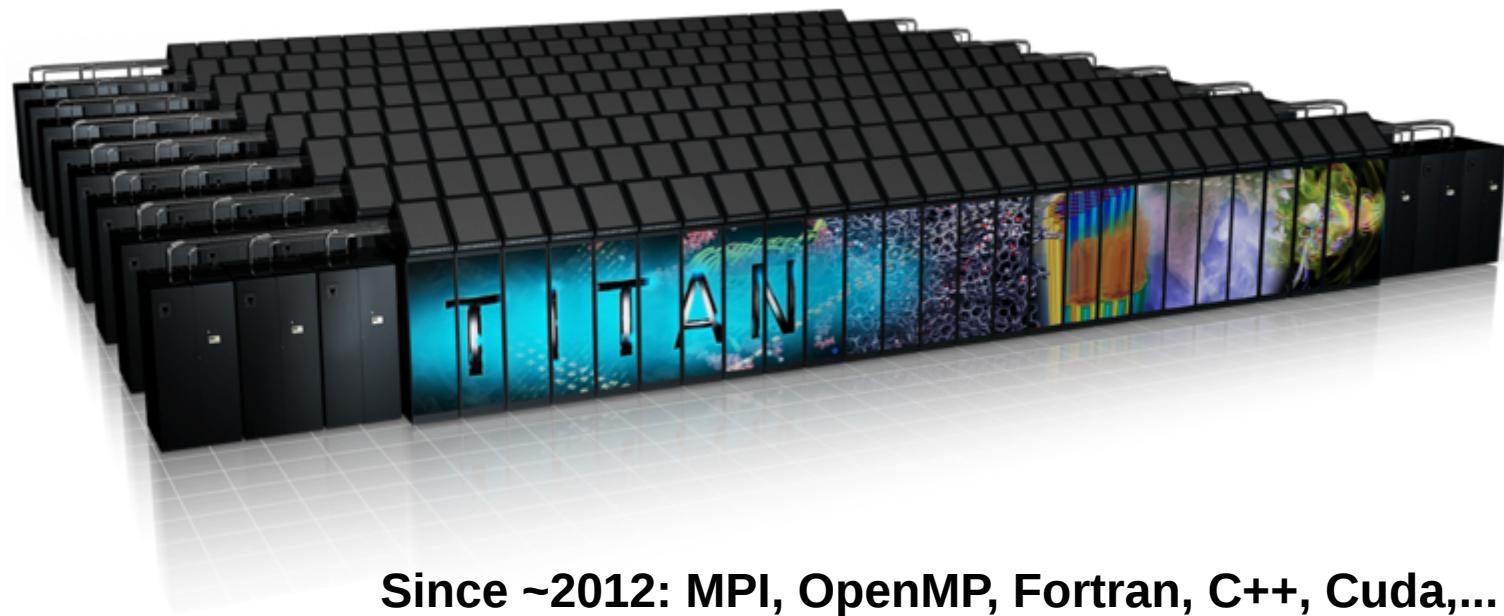
# From making fire to flying rockets in 1 week



For us, we start out here on July 12<sup>th</sup>, 2018

**Fortran: first appeared: 1957**

By July 19<sup>rd</sup>, 2018, we are here...



Since ~2012: MPI, OpenMP, Fortran, C++, Cuda,...

# Scope of today's lecture

The purpose of this lecture is to give you an introduction to the main approaches to exploit modern **parallel** and **High-Performance Computing (HPC)** systems.

- Intended to provide only a very **quick overview** of the extensive and broad topic of **parallel computing**.
- Familiarize the big picture, ideas and concepts.
- Familiarize with the main programming models.

**YOU SHOULD START TO THINK PARALLEL**

# Outline of this introductory lecture

1. Motivation – why should we use parallel programming.
2. Contemporary hardware.
3. Programming Models.

# Some Resources

Full standard/API specification:

- <http://mpi-forum.org>
- <http://openmp.org>

Tutorials:

- <https://computing.llnl.gov/tutorials/mpi/>
- <https://computing.llnl.gov/tutorials/openMP/>
- <http://cse-lab.ethz.ch/index.php/teaching/42-teaching/classes/577-hpcsei>

Books:

- “Introduction to High Performance Computing for Scientists and Engineers”  
Georg Hager, Gerhard Wellein
- “Using Advanced MPI: Modern Features of the Message-Passing Interface”  
(Scientific and Engineering Computation) MIT Press, 2014  
Gropp, W.; Höfler, T.; Thakur, R. & Lusk, E.

## 1. Why parallel computing?

# The world is massively parallel



Galaxy Formation



Planetary Movements



Climate Change



Rush Hour Traffic



Plate Tectonics

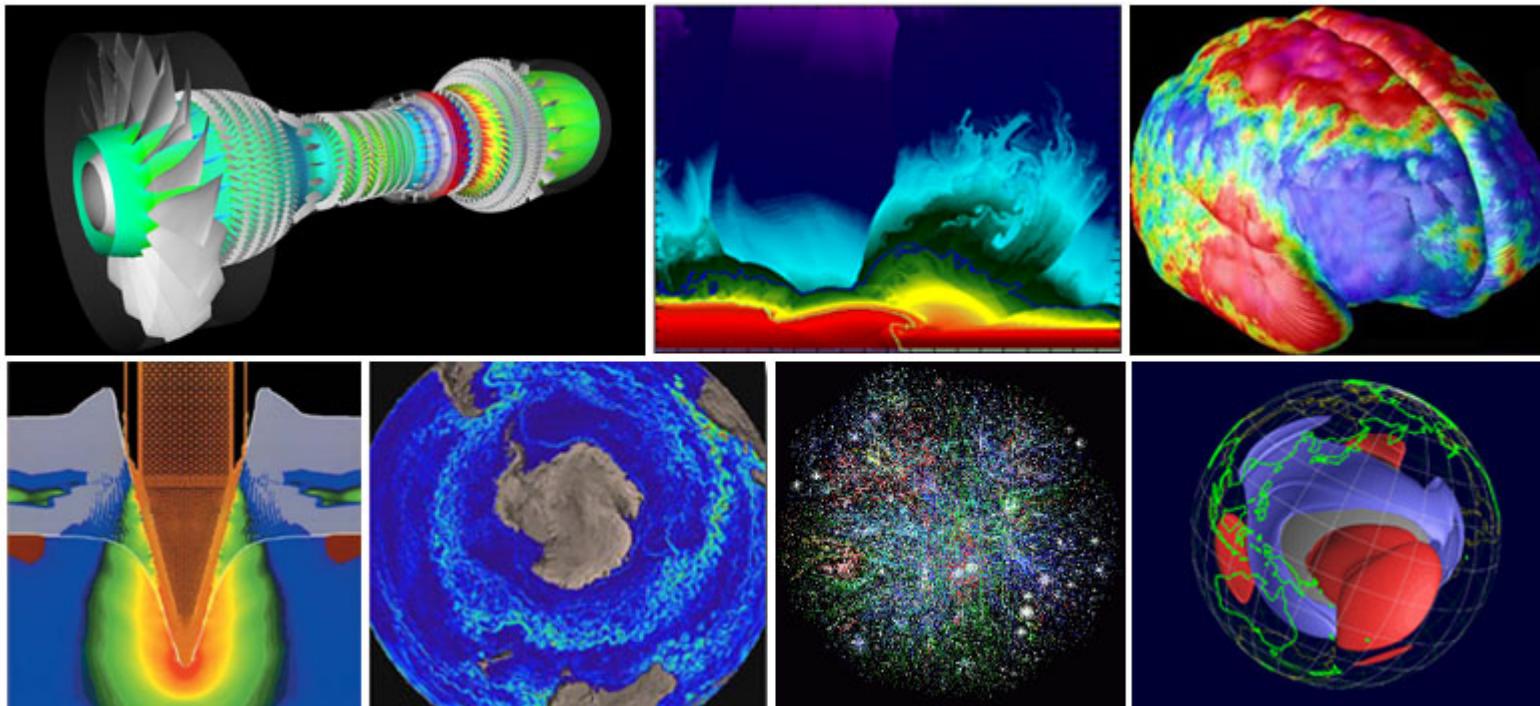


Weather

- In the natural world, many complex, interrelated events are happening at the same time, yet within a temporal sequence.
- Interrelated events happen concurrently.
- Compared to serial computing, parallel computing is much better suited for modeling, simulating and understanding complex, real world phenomena.

# Past: parallel computing = high-end science\*

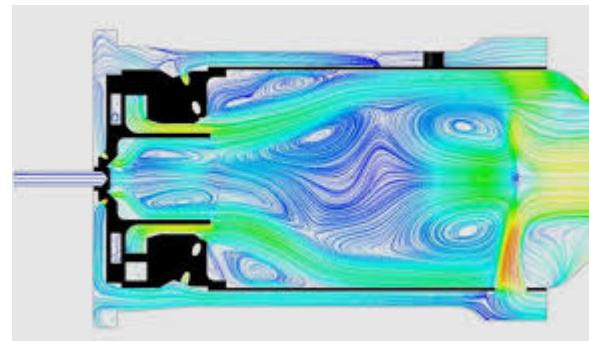
\*with special-purpose hardware



**Parallel computing has been used to model difficult problems in many areas of science and engineering:**

- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics, Chemistry, Molecular Sciences
- Geology, Seismology, Defense, Weapons
- ...

# Today: Parallel computing = every day (industry, academia) computing



Experiment



Macroscopic

Molecular Simulation



Microscopic

- Pharmaceutical design
- Data mining
- Financial modelling
- Advanced graphics, virtual reality
- Artificial intelligence/Machine learning
- ...

# Why parallel computing?

## **HPC helps with:**

### **-Huge data**

- Data management in memory
- Data management on disk

### **-Complex problems**

- Time consuming algorithms
- Data mining
- Visualization

## **Requires special purpose solutions in terms of:**

- Processors
- Networks
- Storage
- Software
- Applications

# Why parallel computing\* (II)

## **Transmission speeds**

Speed of a serial computer is directly dependent on how fast data can be moved through hardware. Absolute limits are the speed of light (30cm/ns) and the transition limit of copper wire (9cm/ns). Increased speeds necessitate increasing proximity of processing elements.

## **Limits of miniaturization**

Processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with atomic-level components, a limit will be reached on how small components can be.

## **Economic limits**

It is increasingly expensive to make a single processor faster. Using a larger number of commodity processors to achieve same (or better) performance is less expensive.

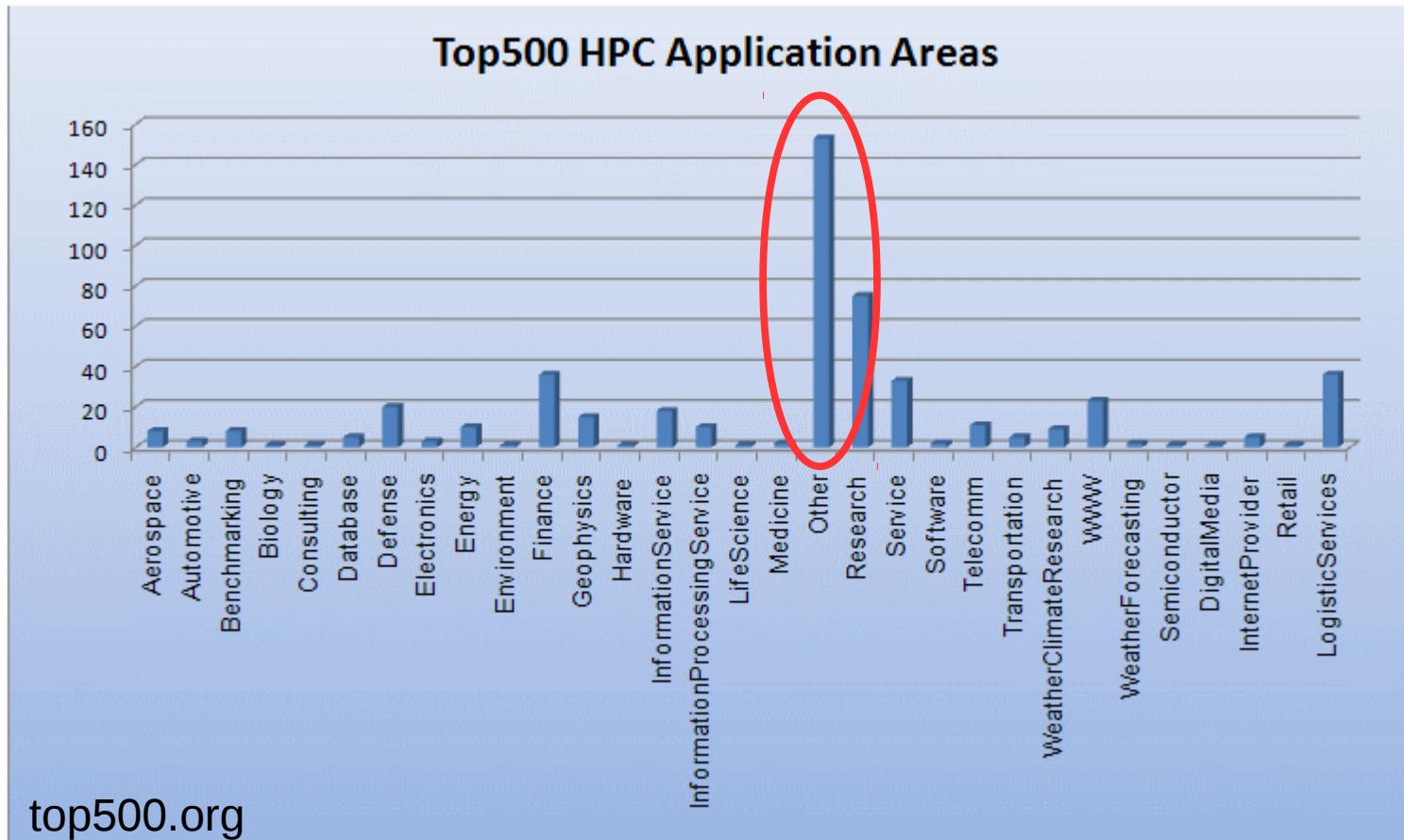
## **Energy limits**

Limits imposed by cooling needs for chips and supercomputers.

## A way out

- Current computer architectures are increasingly relying upon hardware level parallelism.
- Multiple execution units.
- Multi-core.

# Who and what applies parallel computing?



# Why should **YOU** parallelize your code?

- A single core may be **too slow** to perform the required task(s) in a “**tolerable**” amount of time. The definition of “tolerable” certainly varies, but “**overnight**” is often a reasonable estimate.
- The **memory** requirements **cannot be met** by the amount of memory which is available on a single “Desktop”.



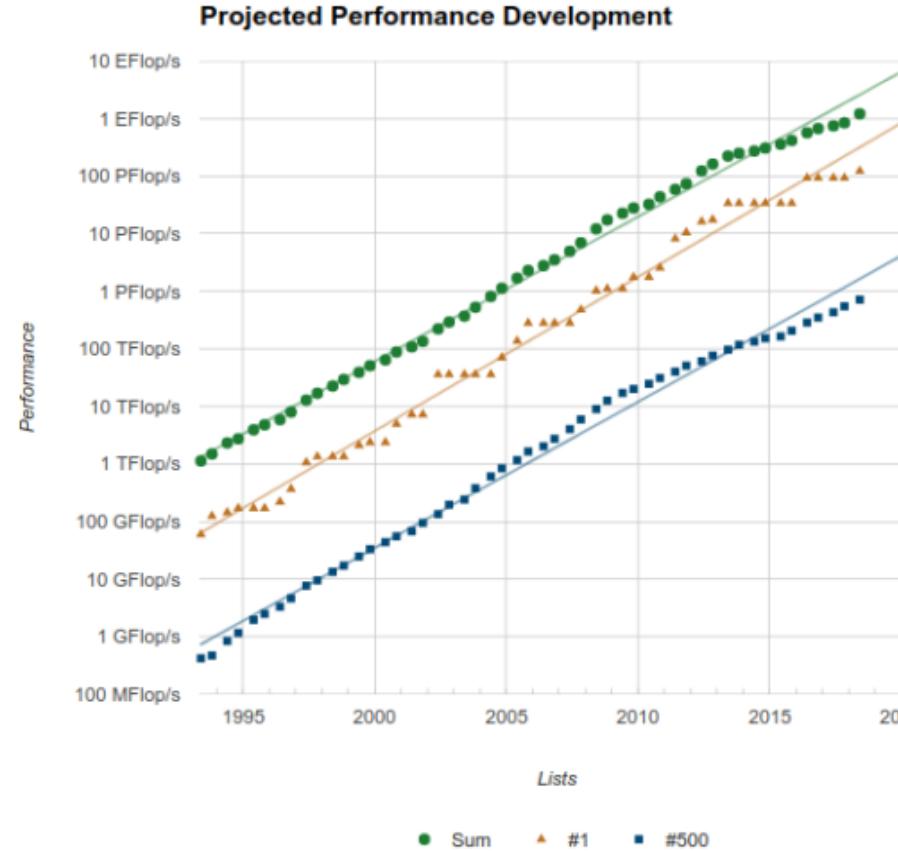
# Heterogeneity in Nature – Stylized modelling not always possible

- Supernova explosions – observable.
- Modelling in 1D:
  - incl. fancy (GR/nuclear) physics.
  - no explosions.
  - only 1D models.  
due to lack of compute power  
(early 2000's).
- Only recently (last ~5y):  
'realistic' 3D models possible to run  
on supercomputers
  - resources now available.
  - **stylized models not sufficient.**



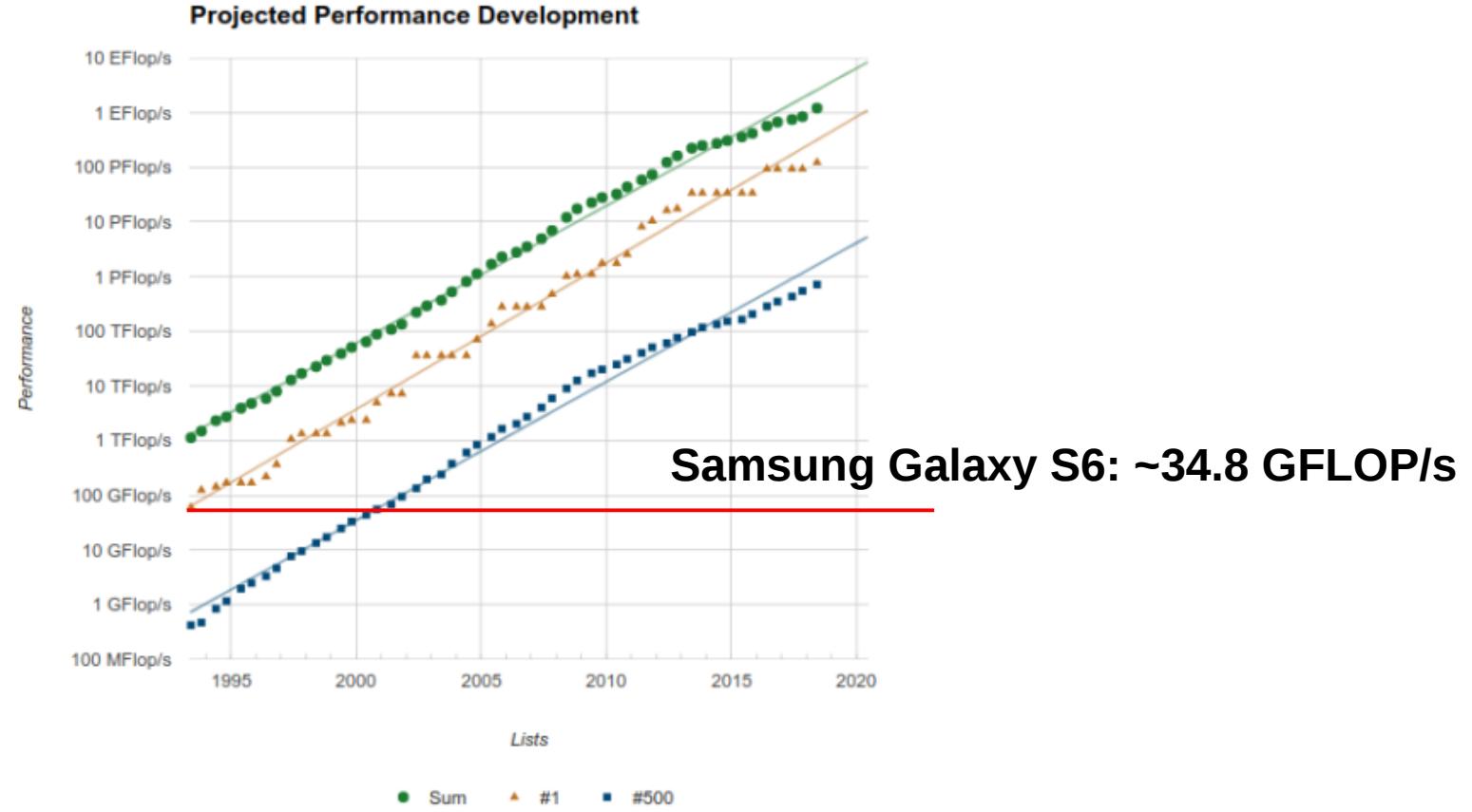
**Fig.:** February 24, 1987; **SN 1987A** in the large Magellanic cloud.  
Closest SN next to earth since 1604.  
Distance from earth: ~50 kpc; ~ 150'000 LY.  
1 kpc =  $3.08 \times 10^{19}$  m, 1 LY =  $9.46 \times 10^{15}$  m  
Progenitor:  $20 M_{\text{sun}}$

# Supercomputers



- HPC “third” pillar of science – nowadays is the dawn of an era (in physics, chemistry, biology,...,next to experiment and theory).
- ‘In Silico’ experiments.
- Modern Supercomputers (e.g. **Summit** at Oak Ridge National Lab)  
200 x  $10^{15}$  Flops/Sec. → **1 day vs. Laptop: ~30,000y.**

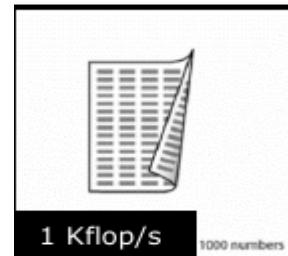
# Supercomputers



- Move away from stylized models towards “realistically-sized” problems.
- **Creating ‘good’ HPC software can be very difficult...**

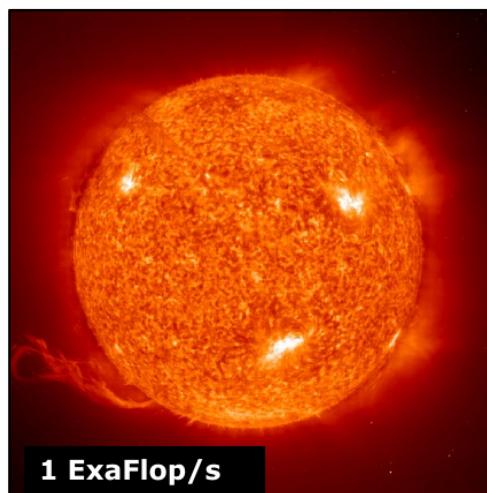
# Petascale Computers – How can we tap them?

- Modern Supercomputers (Summit, World's fastest)  
 $200 \times 10^{15}$  Flops/Sec. → **1 day vs. Laptop: ~30,000y**



**Let us say you can print:**

$10^{18}$  numbers (Exaflop) = 100,000,000 km  
(distance to the sun) stack printed per second  
**(Exaflop/s)**



# 1. Why parallel computing?

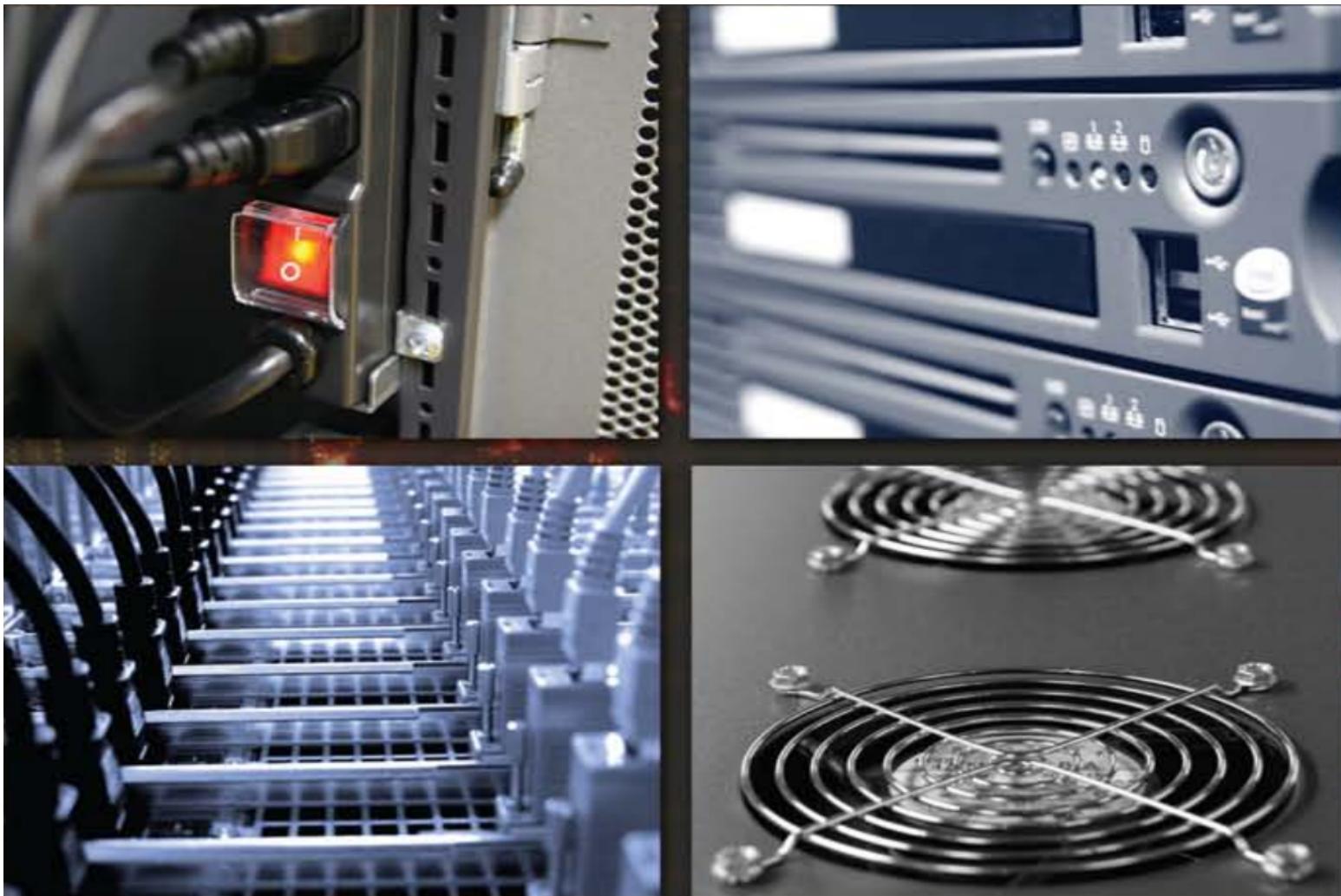
[www.top500.org](http://www.top500.org) (June 2018)



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,282,544	122,300.0	187,659.3	8,806
2	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
3	Sierra - IBM Power System 5922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/NNSA/LLNL United States	1,572,480	71,610.0	119,193.6	
4	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	AI Bridging Cloud Infrastructure (ABCi) - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
6	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
7	Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209
8	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL United States	1,572,864	17,173.2	20,132.7	7,890
9	Trinity - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States	979,968	14,137.3	43,902.6	3,844
10	Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/SC/LBNL/NERSC United States	622,336	14,014.7	27,880.7	3,939



## II. Contemporary hardware



# Hardware and how to use it

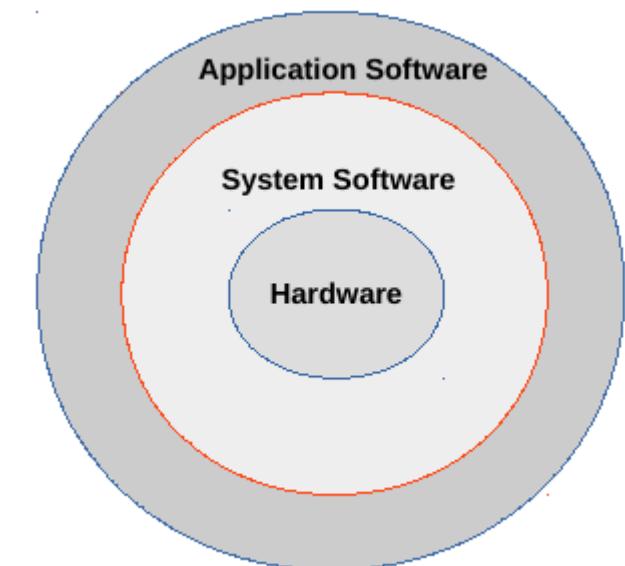
- The **hardware** can execute only simple **low-level instructions** (load,...)
- Complex applications (abstract models and their representation as high-level code) are **separated** from simple instructions by multiple layers.
- **System Software**

## **Operating System:**

- I/O operation.
- Storage & memory.
- Protection among concurrent applications.

## **Compilers:**

- Translate from high level language such as C++ to hardware instructions.



# From a programming language to hardware

<http://cse-lab.ethz.ch/index.php/teaching/42-teaching/classes/577-hpcsei>

A computer is a “stupid” device, only understands “on” and “off”.

The symbols for these states are 0 and 1 (binary).

First programmers communicated in 0 and 1.

Later programs were developed to translate from symbolic notation to binary. The first was called “**assembly**”.

```
> add A, B (programmer writes in assembly language)  
> 1000110010100000 (assembly translates to machine language)
```

Advanced programming languages are better than Assembly for

- programmer thinks in a more natural language
- productivity of software development
- portability

# Basics: von Neumann Architecture

[https://computing.llnl.gov/tutorials/parallel\\_comp](https://computing.llnl.gov/tutorials/parallel_comp)

Virtually all computers have followed this basic design.

Comprised of **four main components**:

- **Memory, Control Unit, Arithmetic Logic Unit, Input/Output.**

**Read/write, random access memory** is used to store both program instructions and data:

- Program instructions are coded data which tell the computer to do something.
- Data is simply information to be used by the program.

**Control unit:**

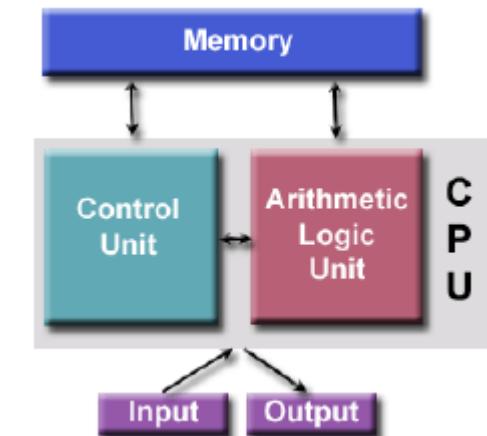
- fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.

**Arithmetic Unit:**

- performs basic arithmetic operations.

**Input/Output**

- interface to the human operator.



→ Many parallel computers follow this basic design, just multiplied in units.  
The basic, fundamental architecture remains the same.

# Performance measures

- **Execution time:** time between start and completion of a task.
- **Throughput/Bandwidth:** total amount of work per elapsed time.
- Performance and execution time:

$$Perf_x = \frac{1}{Exectime_x}$$

$$Perf_x > Perf_y \implies Exectime_y > Exectime_x$$

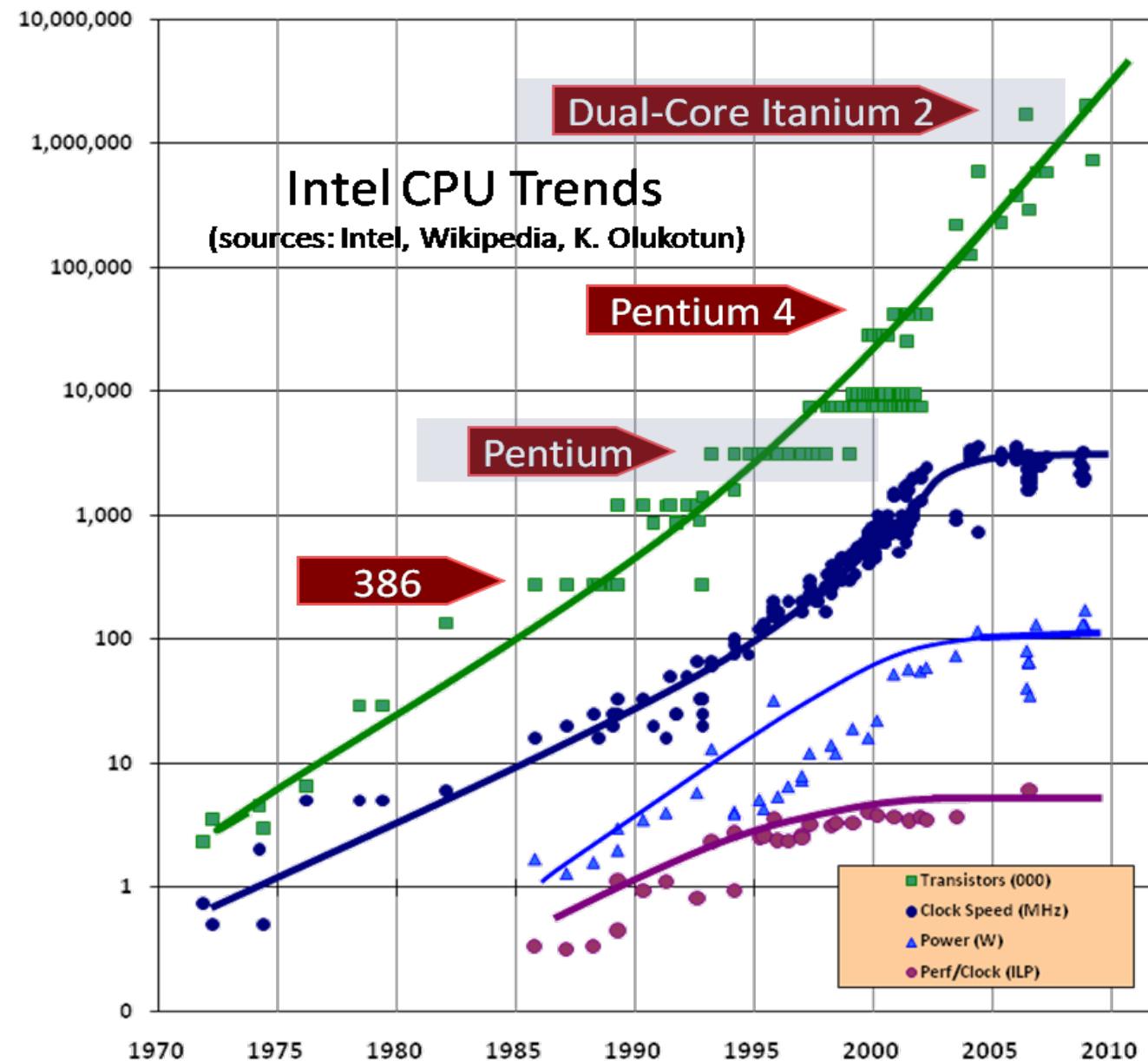
**x** is n times faster than **y** means:

$$\frac{Perf_x}{Perf_y} = n = \frac{Exectime_y}{Exectime_x}$$

# The free lunch

- For a long time speeding up computations was a “free lunch”:
    - the density of the transistors in chips increased, decreasing the size of integrated circuits.
    - the clock speeds steadily rose, increasing the number of operations per second (MHz to GHz).
  - But the free lunch has been over for a few years now:
    - We are reaching the limitations of transistor density.
    - Increasing clock frequency requires too much power.
- We used to focus on floating point operations per second. Now we also think about floating point operations per Watt.

# The free lunch is over



Clock speed cannot increase  
More:

1. heat (too much of it and too hard to dissipate).
2. current leakage.
3. power consumption (too high – also memory must be considered).

Processor performance doubles every ~18 month:

- Still true for the number of transistors.
- Not true for clock speed ( $W \sim f^3$  + hardware failures).
- Not true for memory/storage.

# The free lunch is over II\*

\*<http://www.gotw.ca/publications/concurrency-ddj.htm> (Herb Sutter – tech. evangelist)

“Clock speed isn’t the only measure of performance, or even necessarily a good one, but it’s an instructive one: We’re used to seeing 500MHz CPUs give way to 1GHz CPUs give way to 2GHz CPUs, and so on. Today we’re in the 3GHz range on mainstream computers.”

“If you’re a software developer, chances are that you have already been riding the “free lunch” wave of desktop computer performance. Is your application’s performance borderline for some local operations? “Not to worry,” the conventional (if suspect) wisdom goes; “tomorrow’s processors will have even more throughput, and anyway today’s applications are increasingly throttled by factors other than CPU throughput and memory speed (e.g., they’re often I/O-bound, network-bound, database-bound).” Right?

“Right enough, in the past. But dead wrong for the foreseeable future.

**The good news is that processors are going to continue to become more powerful.**

**The bad news is that, at least in the short term, the growth will come mostly in directions that do not take most current applications along for their customary free ride.”**

“But if you want your application to benefit from the continued exponential throughput advances in new processors, it will need to be a well-written concurrent (**usually multi-threaded**) application.

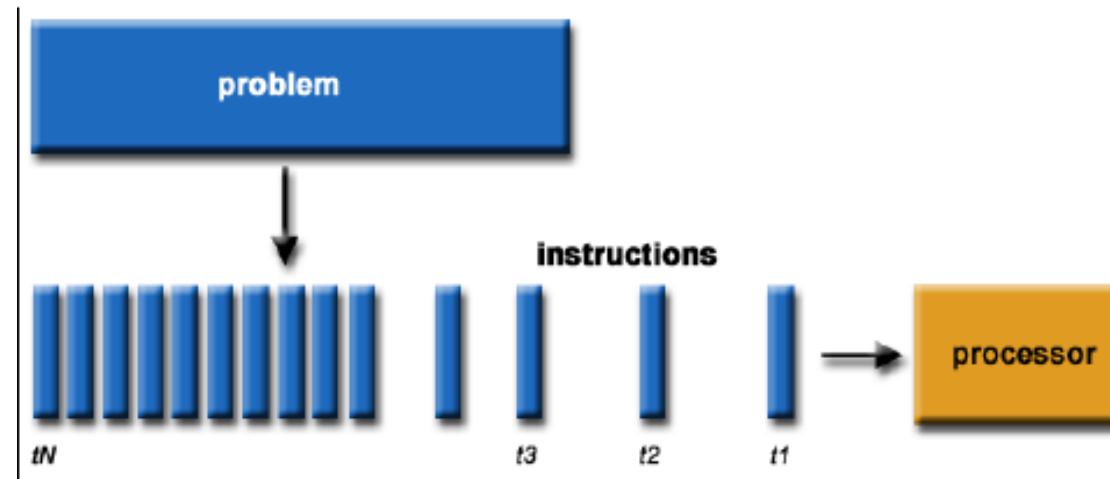
**And that’s easier said than done,...”**

# What is parallel computing?

## **Serial Computing:**

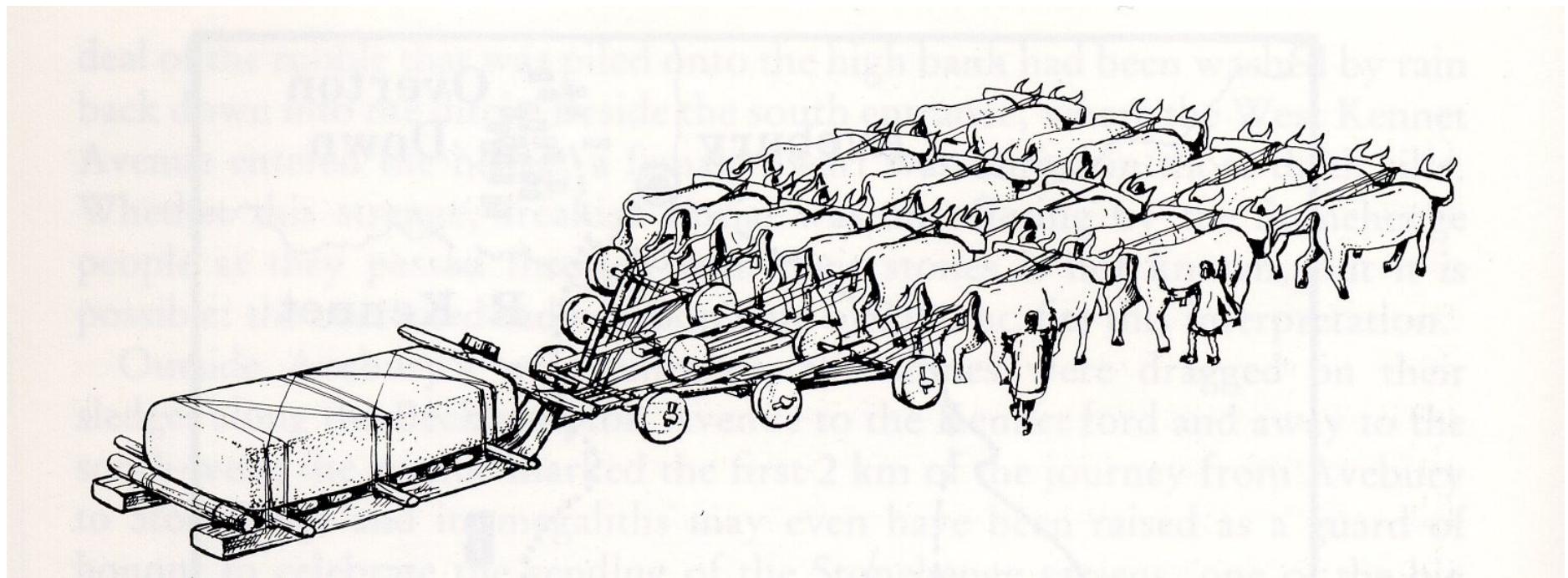
**Traditionally, software has been written for serial computation.**

- A problem is broken into a discrete series of instructions.
- Instructions are executed serially one after another.
- Executed on a single processor.
- Only one instruction may execute at any moment in time.



“To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox”

(Skjellum et al. 1999)

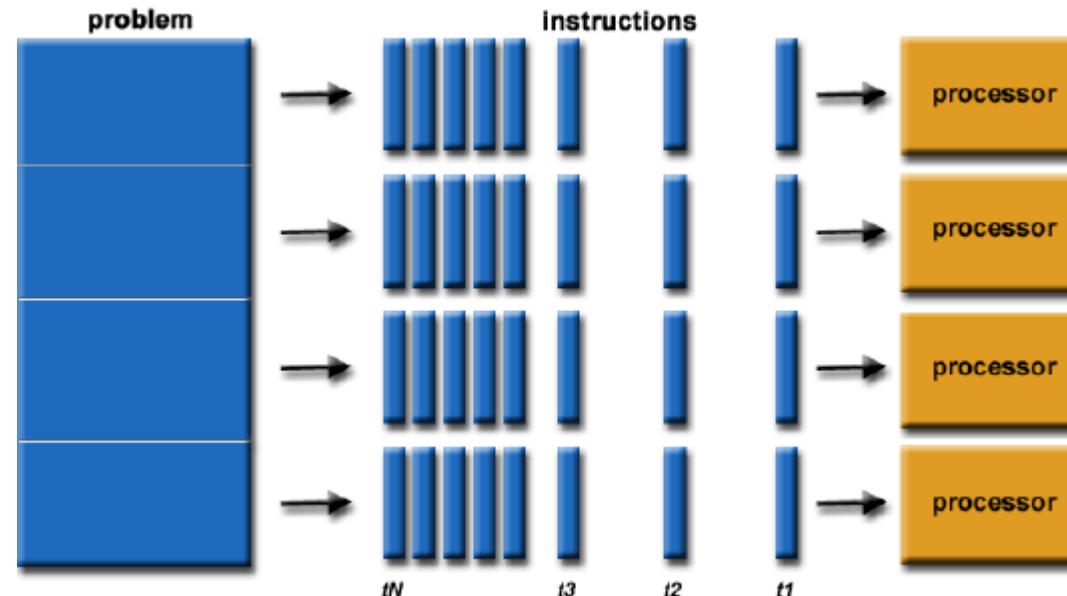


# What is parallel computing II?

## Parallel Computing:

**In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.**

- A problem is broken into a discrete series of instructions that can be solved concurrently.
- Each part is further broken down to a series of instructions.
- Instructions from each part execute simultaneously on different processors.
- An overall control/coordination mechanism is employed.



# Flynn's Taxonomy

[https://computing.llnl.gov/tutorials/parallel\\_comp](https://computing.llnl.gov/tutorials/parallel_comp)

There are different ways to classify parallel computers.

One of the most commonly used (since 1966) is **Flynn's Taxonomy**.

It distinguishes multiprocessor computer architectures according to how they can be classified along the two independent dimensions of **Instruction Stream** and **Data Stream**.

Each of these dimensions can have only one of two possible states: **Single** or **Multiple**.

<b>S I S D</b> Single Instruction stream Single Data stream	<b>S I M D</b> Single Instruction stream Multiple Data stream
<b>M I S D</b> Multiple Instruction stream Single Data stream	<b>M I M D</b> Multiple Instruction stream Multiple Data stream

# Single Instruction, Single Data (SISD)

A serial (non-parallel) computer.

## **Single Instruction:**

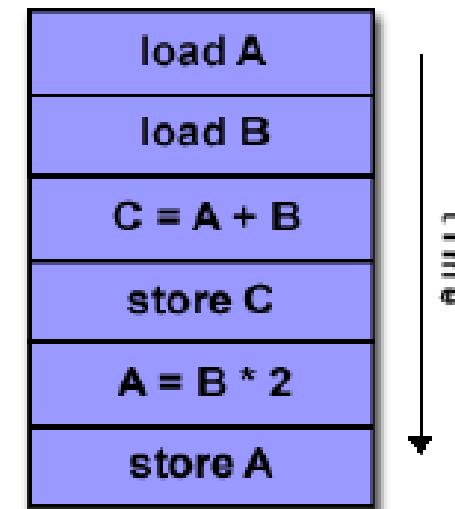
- Only one instruction stream is being acted on by the CPU during any one clock cycle.

## **Single Data:**

- Only one data stream is being used as input during any one clock cycle.

This is the oldest type of computer.

Examples: single processor/core PCs.



# Single Instruction, Multiple Data (SIMD)

A type of **parallel** computer.

## **Single Instruction:**

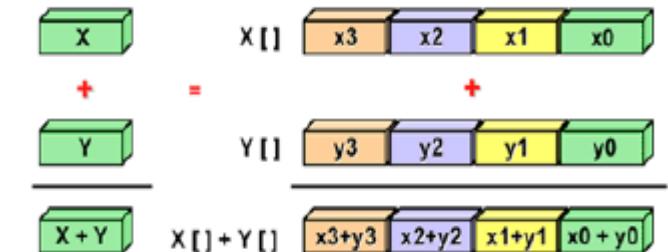
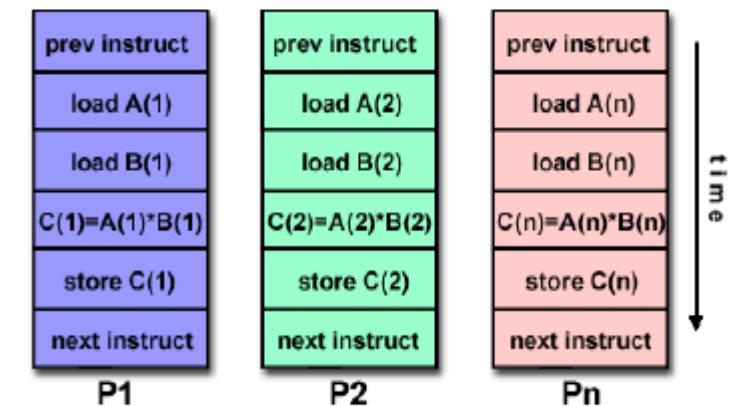
- All processing units execute the same instruction at any given clock cycle.

## **Multiple Data:**

- Each processing unit can operate on a different data element.

Best suited for specialized problems characterized by a high degree of regularity.

Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.



# Multiple Instruction, Multiple Data (MIMD)

A type of parallel computer.

## Multiple Instruction:

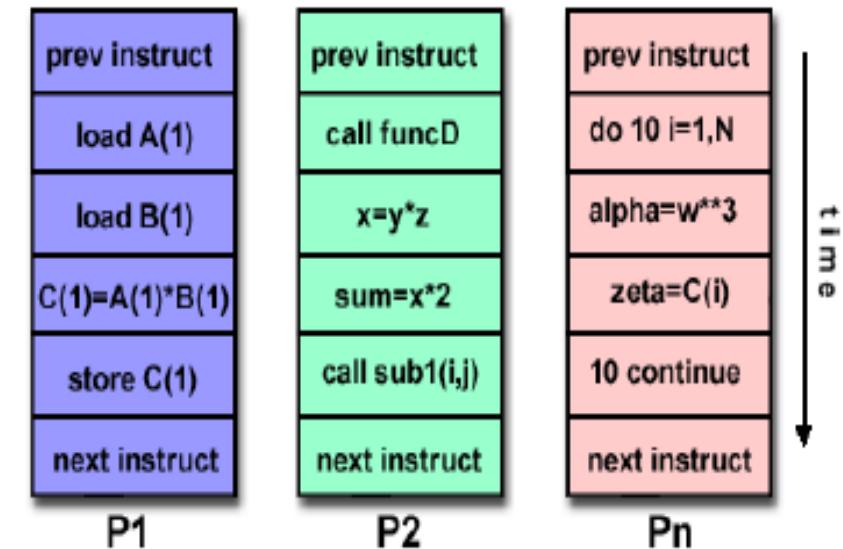
- Every processor may be executing a different instruction stream.

## Multiple Data:

- Every processor may be working with a different data stream.

**Currently, the most common type of parallel computer most modern supercomputers fall into this category.**

- Note: many MIMD architectures also include SIMD execution subcomponents.



# Speedup, Efficiency & Amdahl's Law

$T(p,N)$  := time to solve problem of total size  $N$  on  $p$  processors.

**Parallel speedup:**  $S(p,N) = T(1,N)/T(p,N)$

→ Compute same problem with more processors in **shorter time**.

**Parallel Efficiency:**  $E(p,N) = S(p,N)/p$

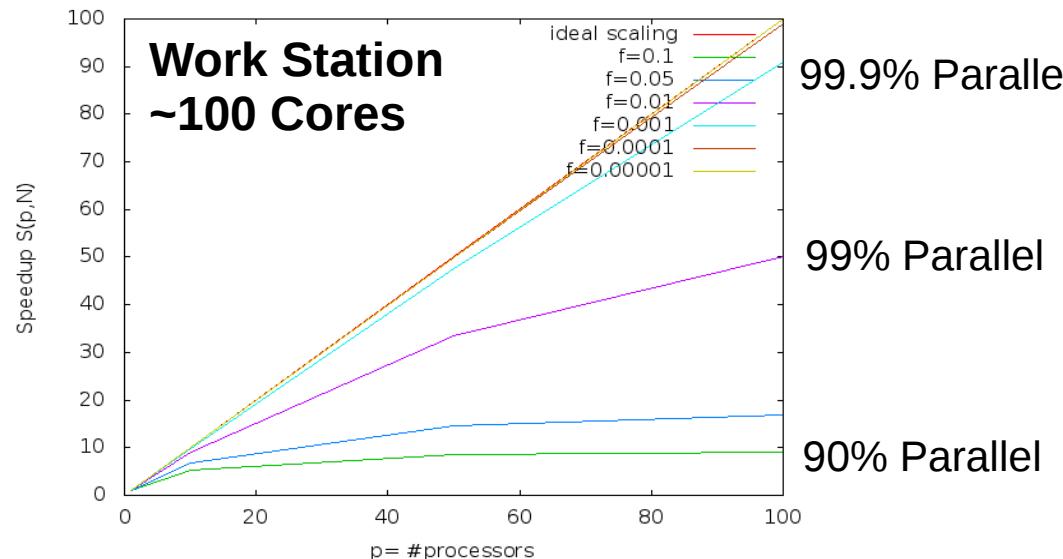
**Amdahl's Law:**  $T(p,N) = f * T(1,N) + (1-f) T(1,N)/p$

f...sequential part of the code that can not be done in parallel.

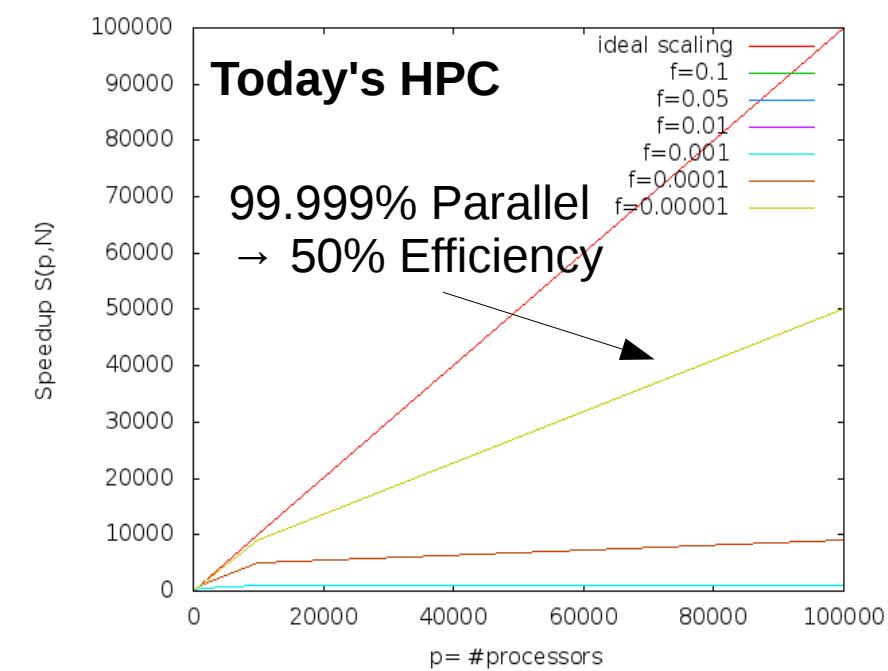
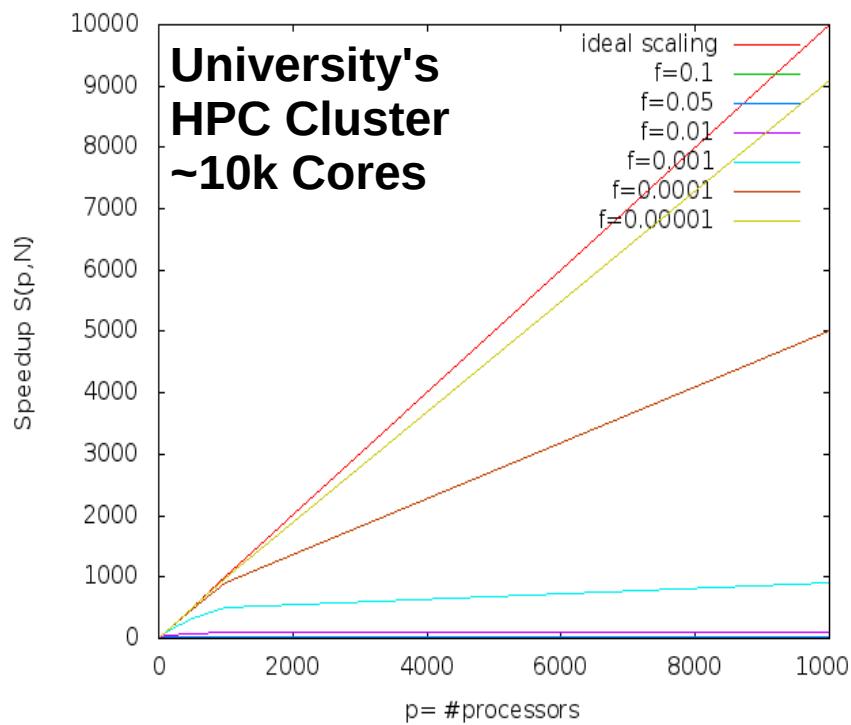
$$S(p,N) = T(1,N)/T(p,N) = 1 / (f + (1-f)/p)$$

For  $p \rightarrow \infty$ , speedup is limited by  $S(p,N) < 1/f$

# Amdahl's Law: Scaling is tough



For  $p \rightarrow \infty$ :  
 Speedup is limited by  $S(p,N) < 1/f$



# Weak versus strong scaling

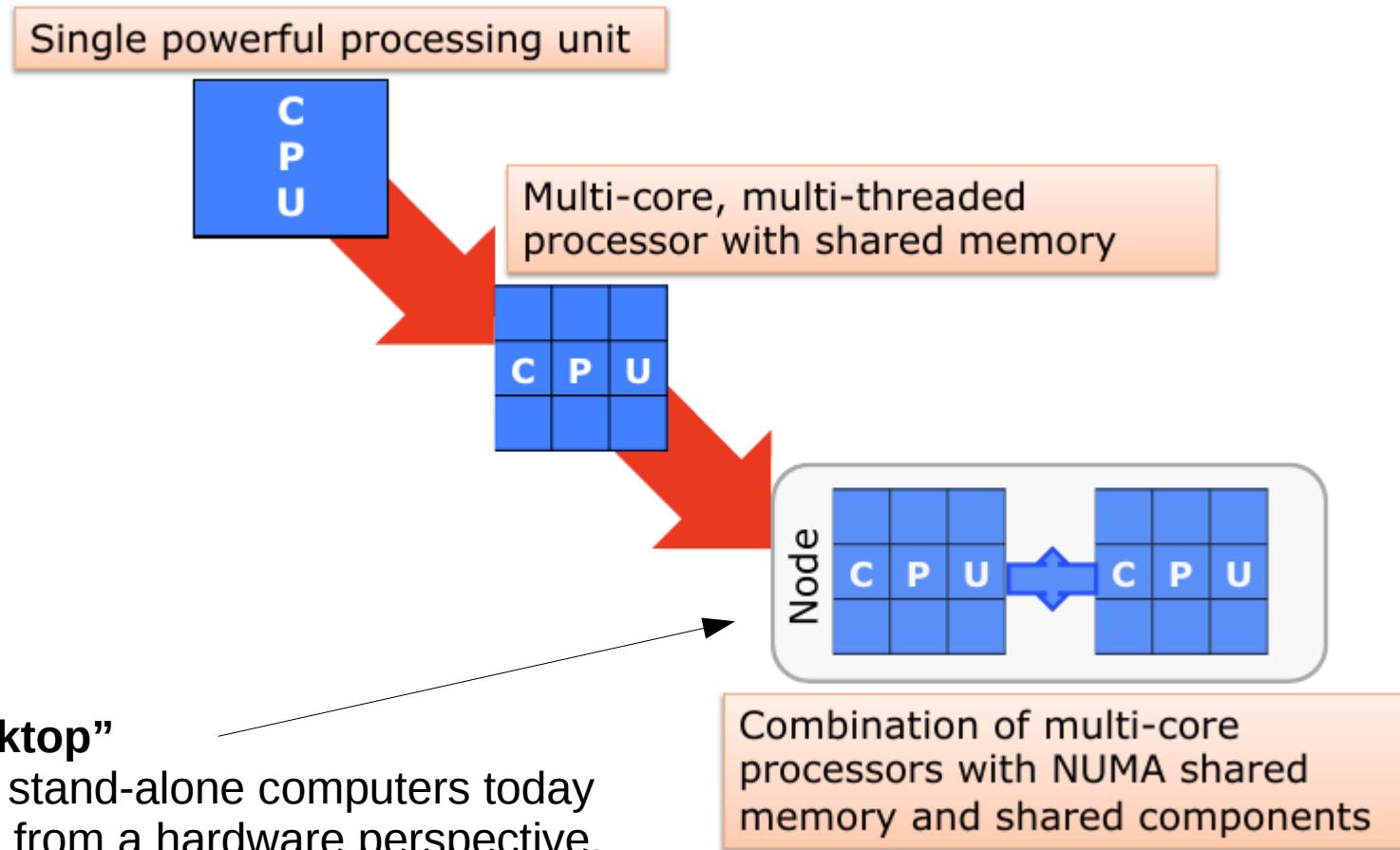
## Strong scaling:

Is defined as how the solution time varies with the number of processors for a fixed total problem size.

## Weak scaling:

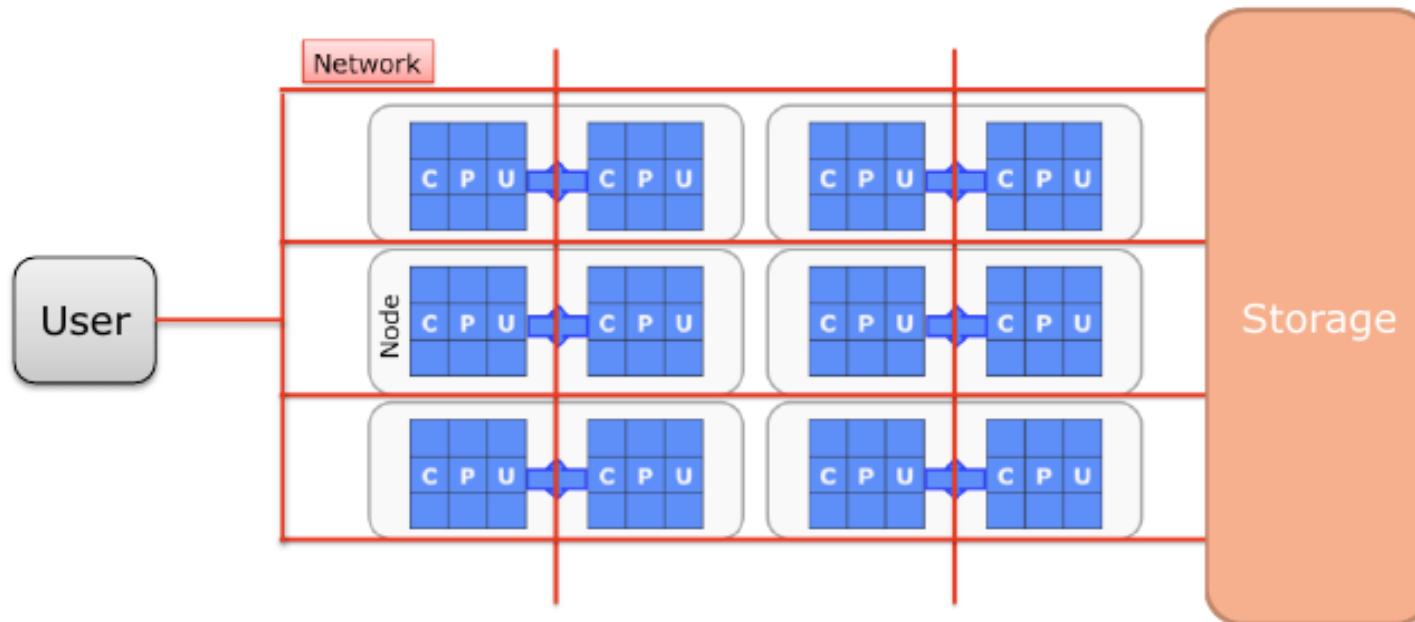
Is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

# HPC systems: off-shelf components



# HPC systems

- Each compute node is a multi-processor parallel computer itself.
- Multiple compute nodes are networked together.
- Building blocks need to communicate (via the network) and give feedback (User/Storage).



# Strengths of Commodity Clusters

- **Excellent performance to cost.**
- **Exploits economy of scale.**
  - Through mass-production of components
  - Many competing cluster vendors
- **Flexible just-in-place configuration.**
  - Scalable up and down
- **Rapid tracking of technology.**
  - First to exploit newest components
- **Programmable.**
  - Uses industry standard programming languages and tools
- **User empowerment.**
  - Low cost, ubiquitous systems
  - Programming systems make it relatively easy to program for expert users
- **nearly all of TOP-500 deployed systems commodity clusters.**

# Two problems with this model

1. pure CPU-systems are becoming too expensive:

- Power consumption.
- Cooling and infrastructural costs (can be  $O(\text{mio \$/year})$  ).  
→ e.g. one Olympic swimming pool per hour cooling water needed.

2. It is hard to increase the performance:

- Speed-up the processor.
- Increase network bandwidth.
- Make I/O faster.
- ...

# HPC system's power consumption

**...how to get 1000x more performance without building a nuclear power plant next to your HPC?**  
(J. Shalf, September 09, Lausanne/Switzerland)

**Coming HPC systems are anticipated to draw enormous amounts of electrical power...**

**Example: N.1 Top 500**

Year (Nov.)	System	Performance (Tflop/s)	Electrical power (KW)
2002	Hearth Simulator	41	3200
2005	Blue Gene/L	367	1433
2008	Roadrunner	1105	2483
2009	Jaguar	2331	6950
2011	K computer	11280	12659
2012	Titan	27112	8209
2014	Tianhe-2	54902	17808
→2020	?????????	1000000	>100000 (100 MW!!!)

→ Not sustainable

# Improving performance: multi & many-cores

Adding more and more “classical” cores can increase the computing power without having to increase the clock speed. However:

- Hardware strongly limits the scalability.

Progressively, a **new generation of processors with less general, more specialized architecture**, capable superior performance on a narrower range of problems, is growing:

## **Many-core accelerators:**

- Nvidia Graphics Processor Units (**GPUs**).
- Intel Xeon Phi Many Integrated Cores (**MIC**) architecture.
- ...

They all support a massively parallel computing paradigm.

## 2 types of “Accelerators”



**GPU:** NVIDIA Tesla K20c

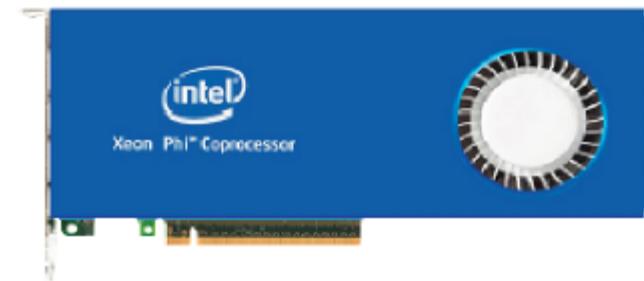
Kepler GK110, 28 nm

13 mp × 192 cores @ 0.71 GHz

5 GB GDDR5 @ 2.6 GHz

225W

→ Devices can have  $O(\text{Teraflops})$



**MIC:** Intel Xeon Phi 3120A

Knights Corner (KNC), 22 nm

57 cores @ 1.1 GHz

6GB GDDR5 @ 1.1 GHz

300W

up to 4 threads per core

512-bit vectorization (AVX-512)

# Why accelerators are more efficient than CPUs?

A few reasons:

- **Lower frequencies** of GPU clocks.
- More of the transistors in a GPU are working on the computation. **CPUs are more general purpose. They require energy to support such flexibility.**
- **Single Instruction Multiple Data** (SIMD) approach, which leads to a simpler architecture and instruction set.

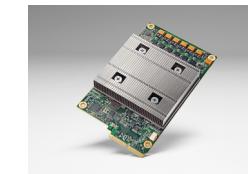
# More hardware accelerators for ML\*



FPGA – field programmable gate arrays



TPU – Tensor Processing Unit



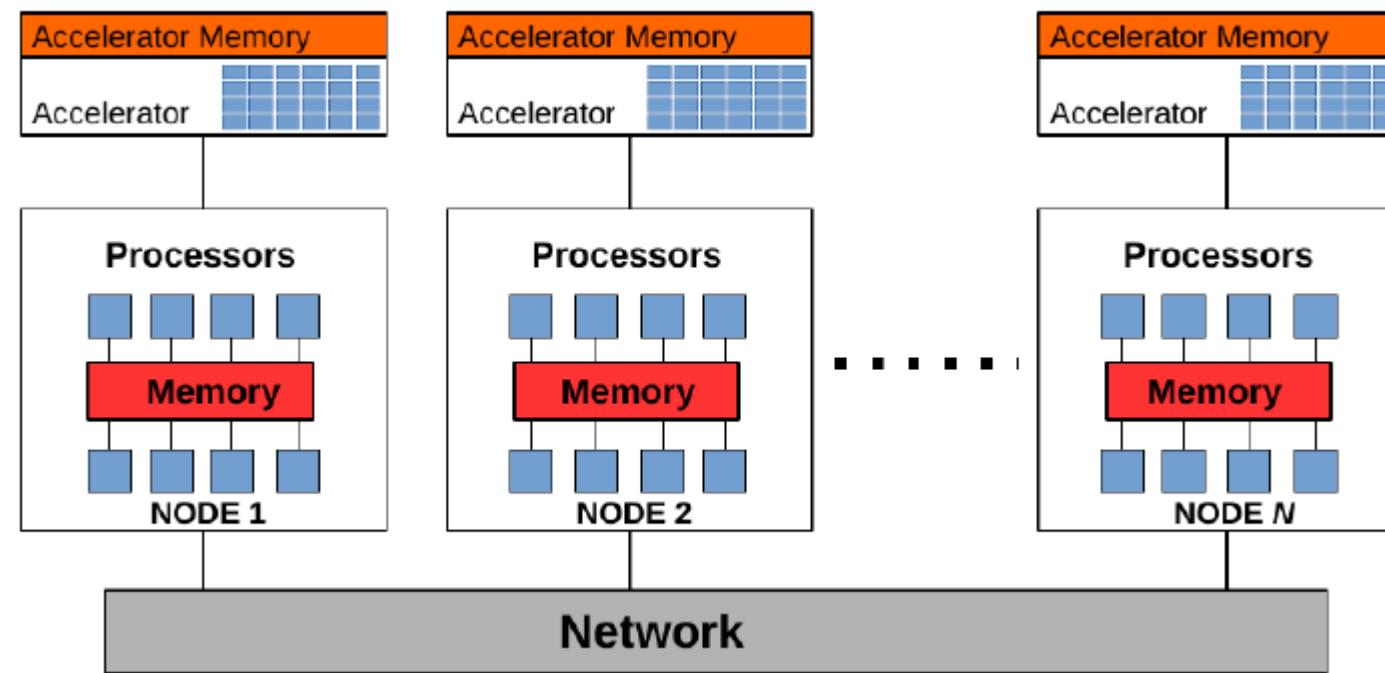
GPU



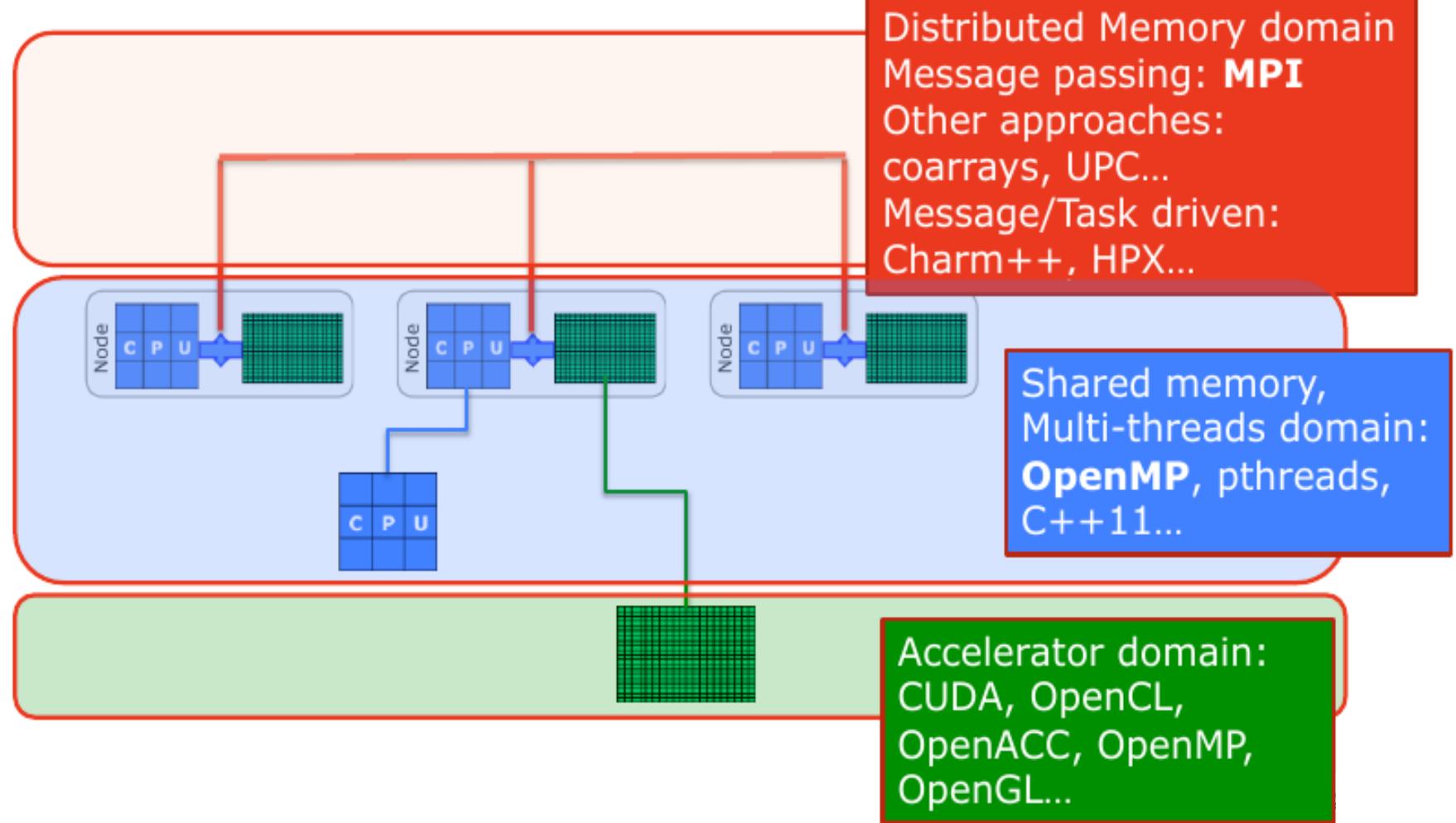
Lake Crest



# Today's HPC systems

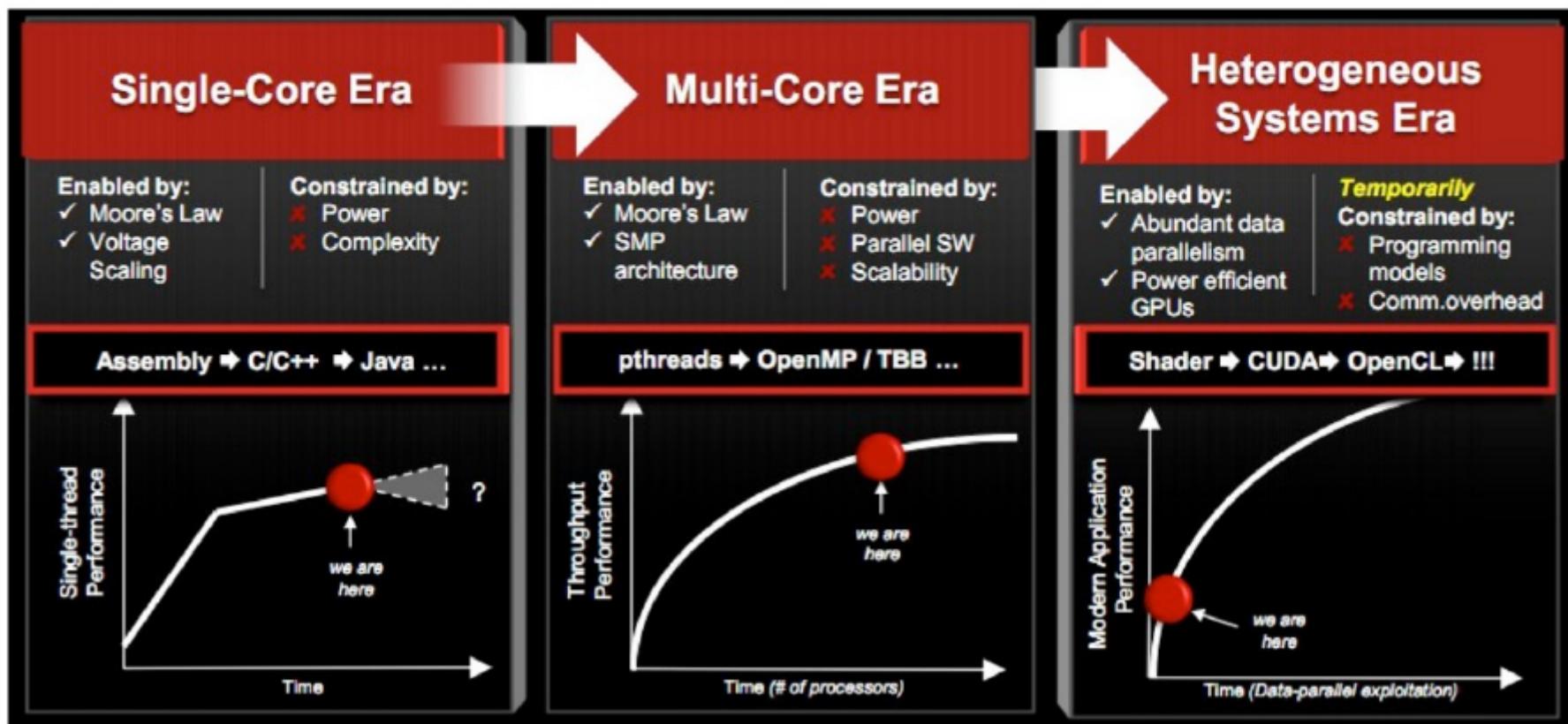


# Overall picture of programming models



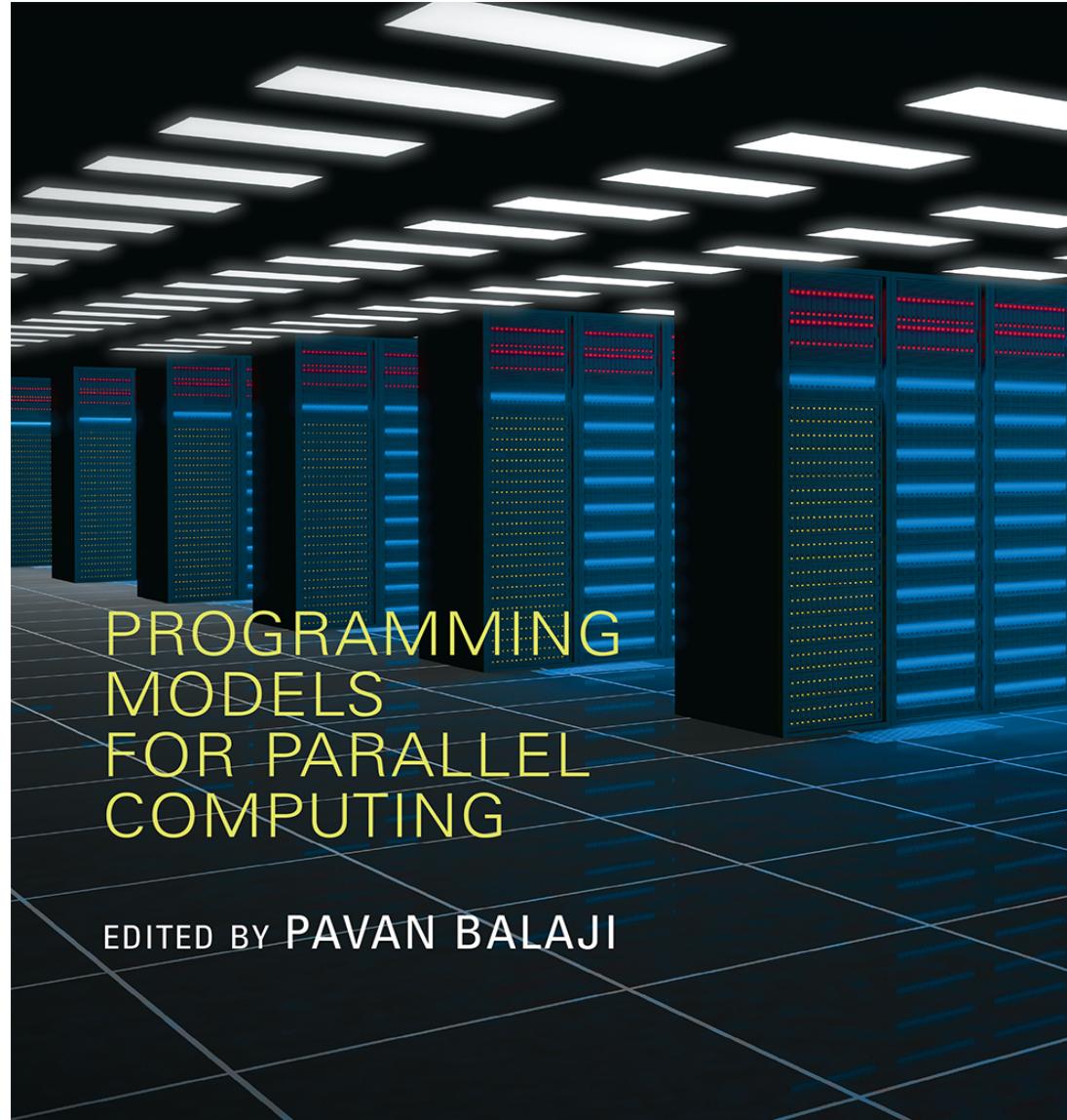
# Where are we going?

**Heterogeneous systems are the next future. Right now, they represent the only viable solution for efficient high performance computing**



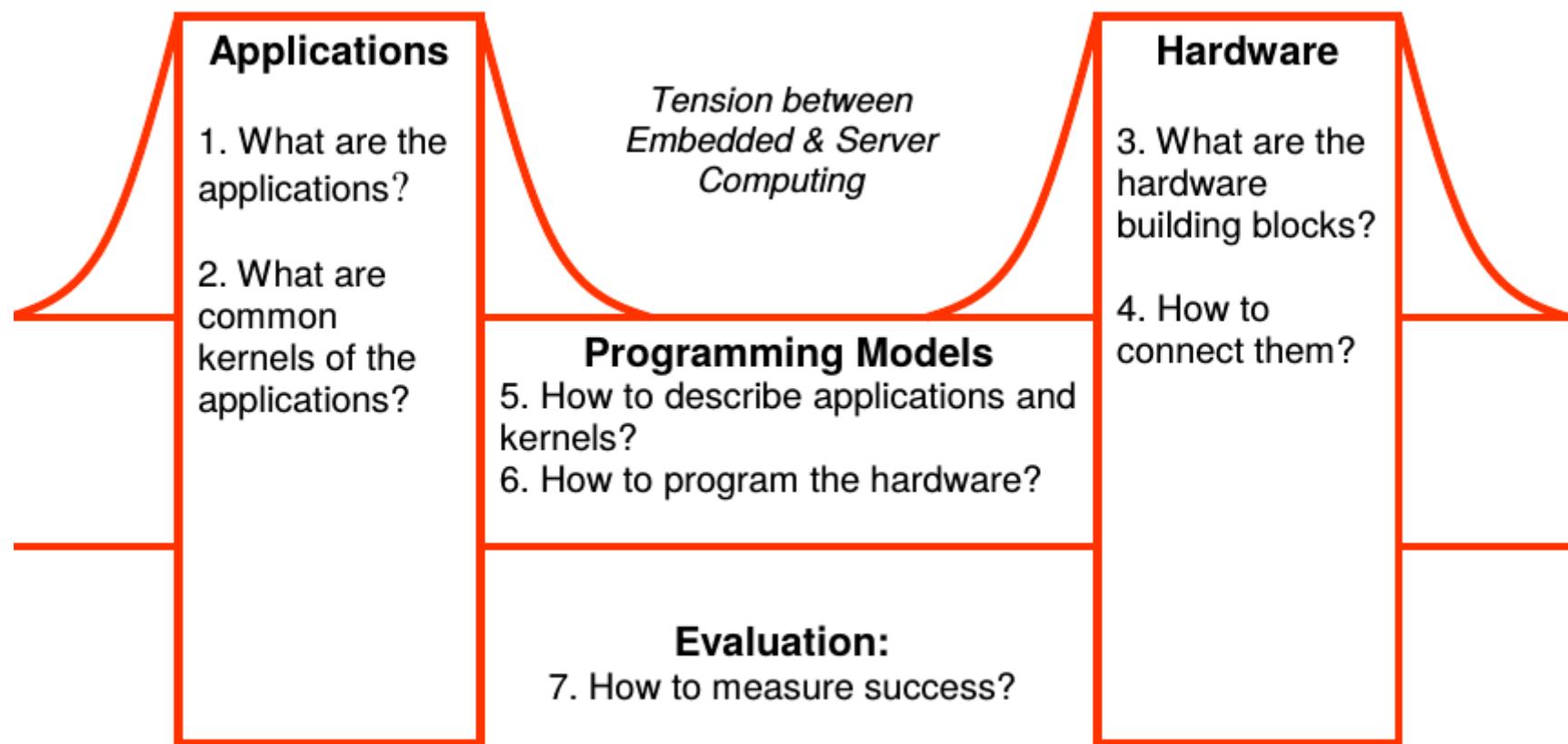
(Slide from C. Gheller)

# 3. Programming Models



# Questions for parallel programming

Asanovic et. al. (2006) – The landscape of Parallel Computing Research: A View from Berkeley



**Figure 1.** A view from Berkeley: seven critical questions for 21<sup>st</sup> Century parallel computing.  
(This figure is inspired by a view of the Golden Gate Bridge from Berkeley.)

# Programming models: Overview

**There are several parallel programming models in common use:**

1. Shared Memory (without threads)
2. Threads
3. Distributed Memory / Message Passing
4. Data Parallel
5. Hybrid
6. Single Program Multiple Data (SPMD)
7. Multiple Program Multiple Data (MPMD)

**Parallel programming models exist as an abstraction above hardware and memory architectures.**

Although it might not seem apparent, these models are NOT specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware.

**Which model to use?**

This is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.

# Parallel program design: Understand the problem

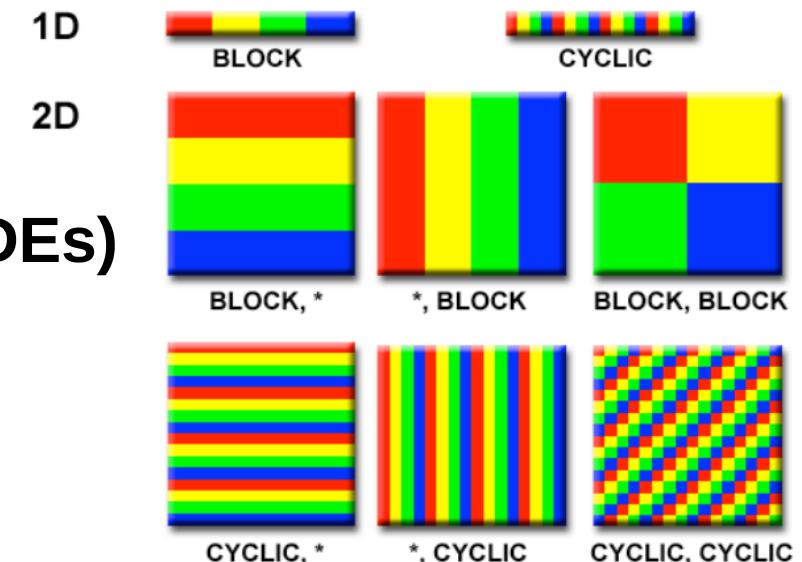
- The first step in developing parallel software is to first understand the problem that you wish to solve in parallel.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.
  - Example of Parallelizable Problem: Monte Carlo integration
  - Example of a Non-parallelizable Problem: Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:  $F(n) = F(n-1) + F(n-2)$ 

This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the  $F(n)$  value uses those of both  $F(n-1)$  and  $F(n-2)$ . These three terms cannot be calculated independently and therefore, not in parallel.
- **Identify hotspots** (where is the real work being done?)
- **Identify bottlenecks** in the program
  - are there disproportionately slow regions in code, e.g. I/O; can restructuring of program help?

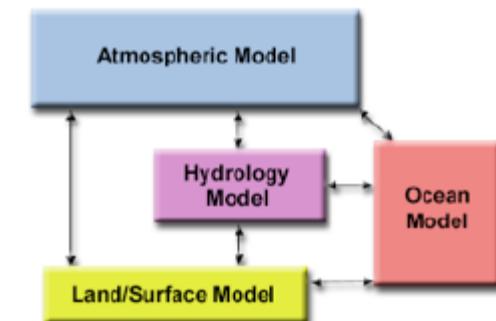
# Decomposition of Problem

[https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)

e.g. domain decomposition (e.g. for PDEs)



**Functional decomposition**  
(e.g. for climate modelling)



# Communication

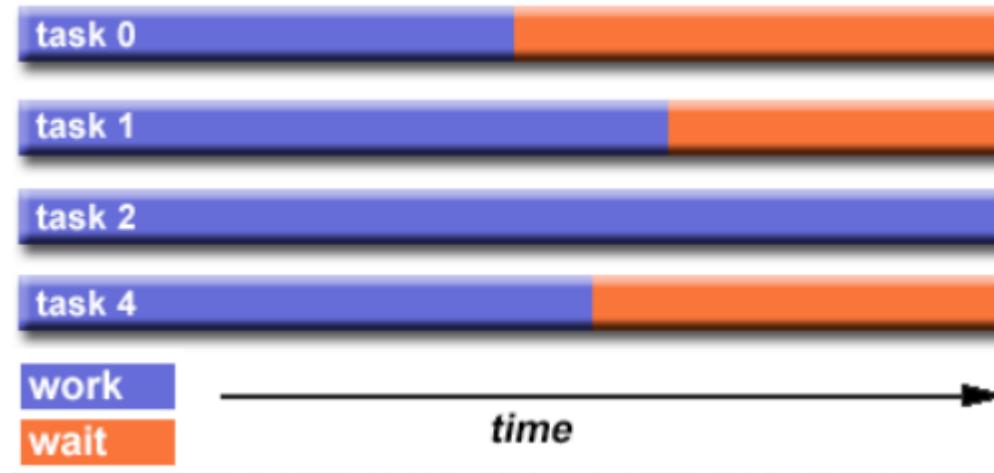
## **Is communication needed?**

The need for communication between tasks depend upon your problem

- you don't need communications: embarrassingly parallel (e.g. Monte Carlo)
- you need communications
  - e.g. stencil computation in a finite difference scheme (data dependency)
- Cost of communication (e.g. transmit data implies overhead)

# Load balancing

[https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)

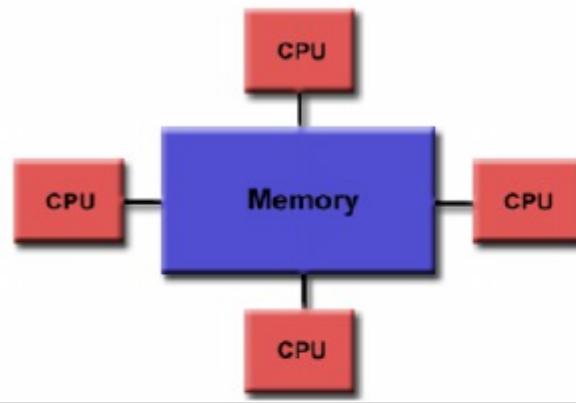


- Load balancing refers to the practice of distributing work among tasks so that all tasks are kept busy all of the time.
- **It can be considered a minimization of task idle time.**
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

# We will consider mainly OpenMP\* & MPI\*\*

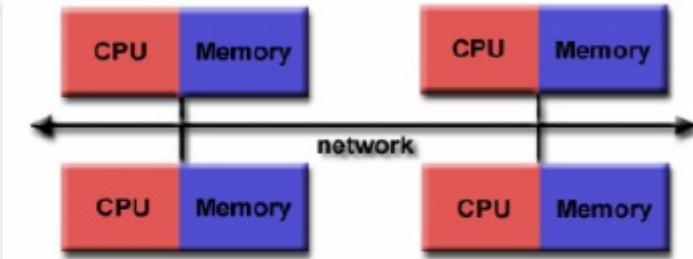
\*<https://computing.llnl.gov/tutorials/openMP/> (**Open Multi Processing**)

\*\*<https://computing.llnl.gov/tutorials/mpi/> (**Message Passing Interface**)



Shared Memory

OpenMP



Distributed Memory

MPI

# Using SIMD instruction sets → Vectorization

**Vectorization** performs multiple operations in **parallel** on a core with a single instruction (SIMD – single instruction multiple data).

Data is loaded into vector registers that are described by their width in bits:

- 256 bit registers: 8 x float, or 4x double
- 512 bit registers: 16x float, or 8x double

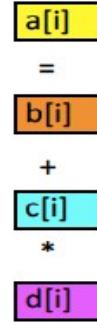
**Vector units** perform arithmetic operations on vector registers simultaneously.

Vectorization is key to maximising computational performance.

# Vectorization illustrated

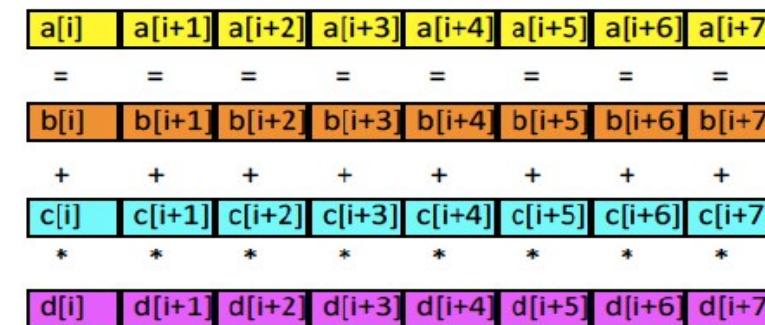
sequential

```
a [ i ] = b[ i ] + c[ i ] * d[ i ];
```



8× vectorization

```
a [ i:8 ] = b[ i:8 ] + c[ i:8 ] * d[ i:8 ];
```



In an optimal situation all this is carried out by the compiler automatically. Compiler directives can be used to give hints as to where vectorization is safe and/or beneficial.




---

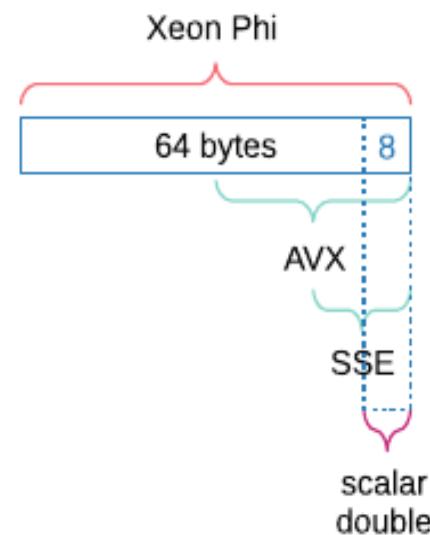
```

1 ! vectorized part
2 rest = mod(N,4)
3 do i=1,N-rest,4
4   load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
5   load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
6   ! "packed" addition (4 SP flops)
7   R3 = ADD(R1,R2)
8   store [r(i),r(i+1),r(i+2),r(i+3)] = R3
9 enddo
10 ! remainder loop
11 do i=N-rest+1,N
12   r(i) = x(i) + y(i)
13 enddo

```

---

# Advanced Vector Extensions (AVX)



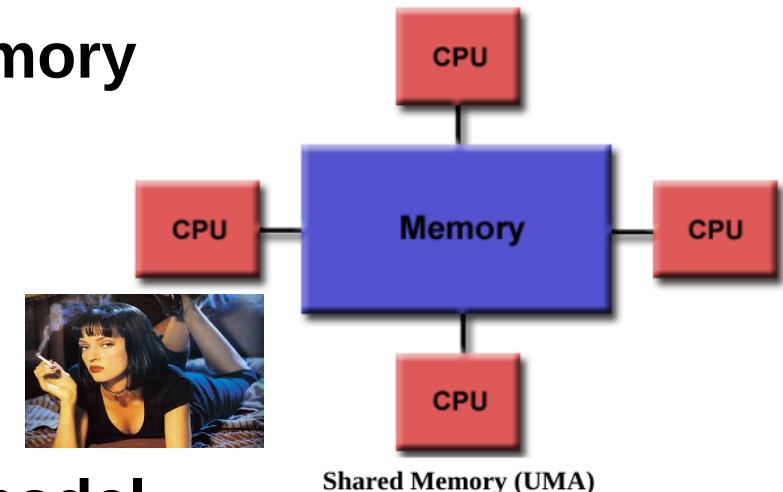
**Fig. 7.** Vector registers on modern CPUs: a scalar program can utilize only 1/4 of computational parallelism on AVX-enabled CPUs, e.g. the SandyBridge.

# Shared memory systems

- Process can access same **GLOBAL** memory

- **Uniform Memory Access (UMA)** model

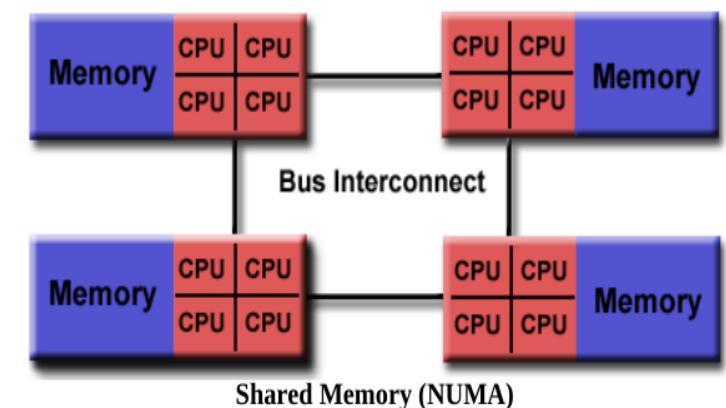
- Access time to memory is uniform.
- Local cache, all other peripherals are shared.



- **Non-Uniform Memory Access (NUMA) model**

- Memory is physically distributed among processors.
- Global virtual address spaces accessible from all processors.
- **Access time to local and remote data is different.**

→ **OpenMP**, but other solutions available (e.g. Intel's TBB).



N-

# Shared Memory – Pro's & Con's

## Pro's

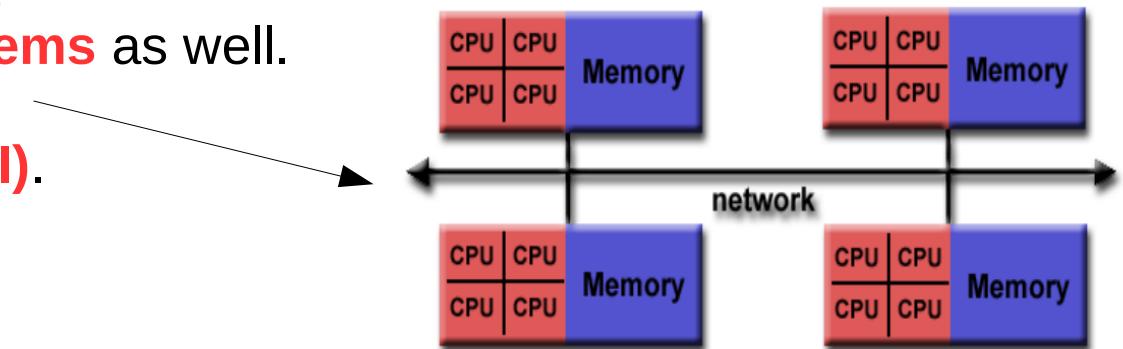
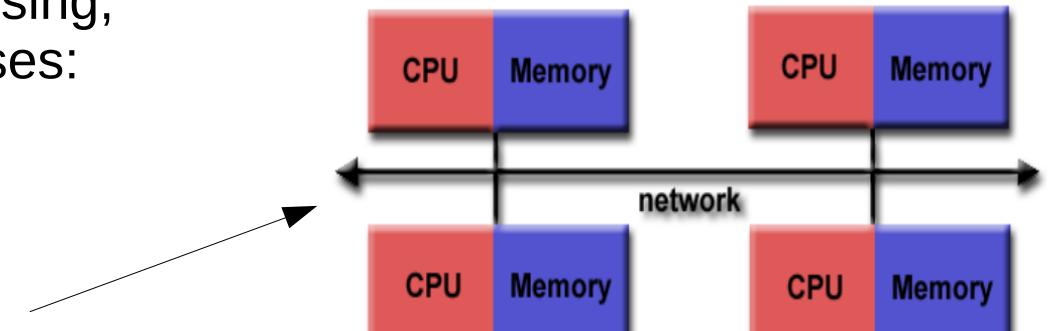
- Global address space provides a **userfriendly programming perspective** to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

## Con's

- **Lack of scalability** between memory and CPUs. Adding more CPUs geometrically increases traffic on the shared memory-CPU path...
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

# Distributed-memory parallel programming (with MPI)

- We need to use explicit message passing, i.e., communication between processes:
  - Most tedious and complicated but also the most flexible parallelization method.
- Message passing is required if a parallel computer is of **distributed-memory** type, i.e., if **there is no way for one processor to directly access the address space of another**.
- However, it can also be regarded as a programming model and used on shared-memory or **hybrid systems** as well.
- **Message Passing Interface (MPI)**.



# Parallel programming with MPI

The Message Passing Interface Standard (**MPI**) is a **message passing library standard** based on the consensus of the **MPI Forum**, which has over 40 participating organizations, including vendors, researchers, software library developers, and users.

The goal of the MPI is to establish a **portable**, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library.

**Several MPI implementations exist** (Mpich, OpenMPI, IntelMPI, CrayMPI, ...).

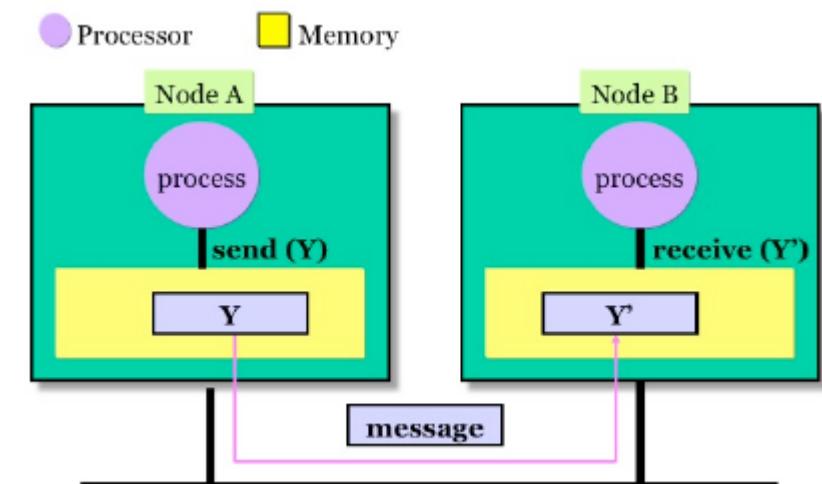
The current MPI standard currently defines over 500 functions, and it is beyond the scope of this introduction even try to cover them all. Here, I **will concentrate on the important concepts of message passing and MPI in particular, and provide some knowledge that will enable you to consult more advanced textbooks and tutorials.**

# Message Passing Interface (MPI)

- Resources are LOCAL (different from shared memory).
- Each process runs in an “isolated” environment. Interactions requires **Messages** to be exchanged
- Messages can be: **instructions, data, synchronization.**
- MPI works also on Shared Memory systems.
- Time to exchange messages is much larger than accessing local memory.

→ **Massage Passing is a COOPERATIVE Approach, based on 3 operations:**

- **SEND** (a message)
- **RECEIVE** (a message)
- **SYNCHRONIZE**



# MPI availability

- MPI is standard defined in a set of documents compiled by a consortium of organizations : <http://www.mpi-forum.org/>
- In particular the MPI documents define the APIs for C, C++, FORTRAN77 and FORTRAN 90.
- Bindings available for Perl, Python, Java...
- In all systems MPI is implemented as a **library of subroutines/functions** over the network drivers and primitives.

# GPU Programming (no hands on this time)

## **API currently available**

- **CUDA**
  - NVIDIA product, best performance, low portability

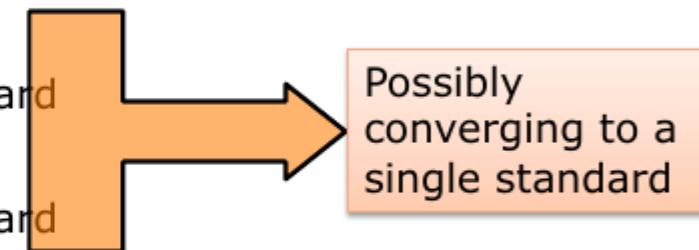
- **OpenCL**
  - Open standard, API similar to CUDA, high portability

- **OpenACC**
  - Directive based open standard
- **OpenMP**
  - Directive based open standard

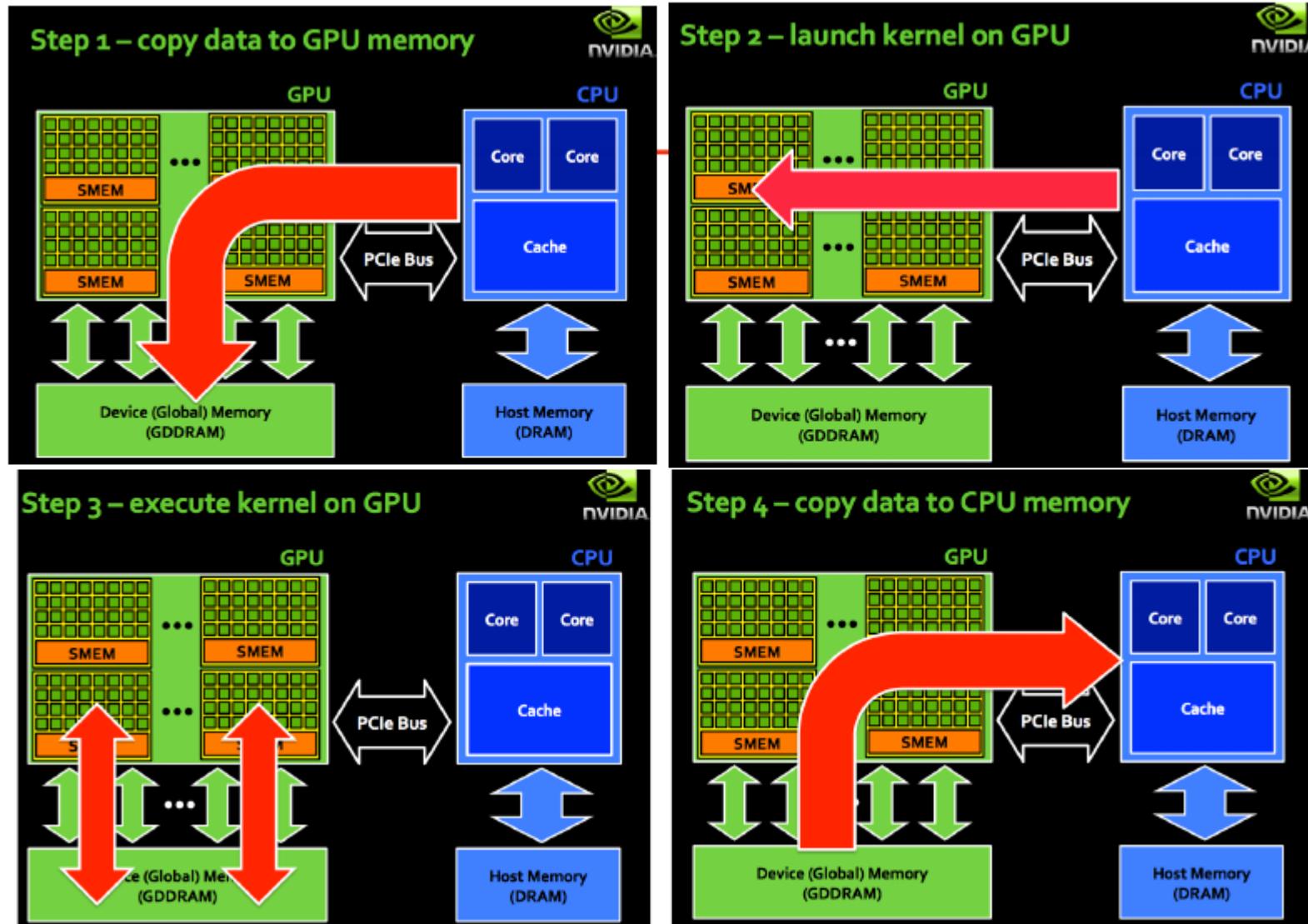
- **Libraries**
  - MAGMA, cuFFT, Thrust, CULA, cuBLAS, cuSOLVER...

- **C++11**
  - High level abstraction

**Supported languages: C, C++, Fortran but also Java, Python  
(not from all the APIs)**



# GPU Programming II



# Degrees of Parallelism

- Current supercomputers are hybrid.
- There are 4 main degrees of parallelism on multi-core (HPC) systems.
- Parallelism among nodes/sockets (e.g. MPI).
- Parallelism among cores in a socket (e.g. OpenMP).
- Vector units in each core (e.g. AVX).
- Accelerators...

→ All should be exploited at once (→ lecture next week)

# Your competitors in the US

<http://cowles.yale.edu/news/cowles-funds-special-access-high-performance-computing>

Yale University

## Cowles Foundation for Research in Economics

---

[HOME](#)   [PEOPLE](#)   [ABOUT US](#)   [RESEARCH PROGRAMS](#)   [PUBLICATIONS](#)

[Home](#) » [News](#) » Cowles Funds Special Access to High Performance Computing

## Cowles Funds Special Access to High Performance Computing



*Wed, 09/30/2015*

Yale University operates several high performance computing clusters. Members of the Department of Economics use primarily the clusters called Omega and Grace. The Cowles Foundation has funded the purchase of 10 nodes on the newest of the HPC clusters, Grace. Each of these nodes has 20 CPU cores. Faculty and graduate students in the Department of Economics have access to a special queue that give them priority over other members of the Yale research community for the use of the Cowles funded processors.

You can find more information about the HPC clusters here:

<http://research.computing.yale.edu/services/high-performance-computing>. Members of the Department of Economics at Yale can request access here: <http://research.computing.yale.edu/hpc-support/account-request>. The request should be for an account on Grace and you should identify yourself a member of the economics department and request access to the queue for the Cowles nodes. Graduate students should include the name of the faculty sponsor of their research.



< 0

# Questions?

