UE Parallel Algorithms and Programming

Frédéric Desprez

INRIA
LIG
Corse team

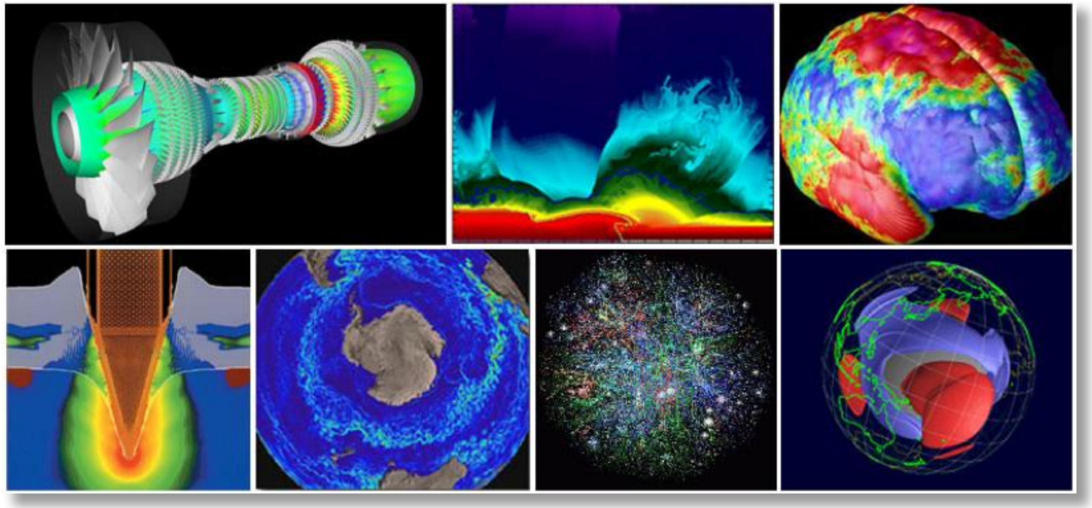
Contact

INRIA

Corse Team

LIG Laboratory

Minatec Campus



- **Email**

  Frederic.Desprez@inria.fr

- **Web page**

  fdesprez.github.io

- **Team web page**

  https://team.inria.fr/corse/

- **Web page of the UE**

  fdesprez.github.io/teaching/par-comput/

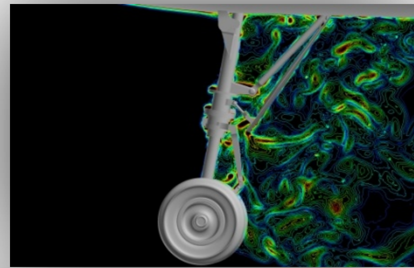
General Organization

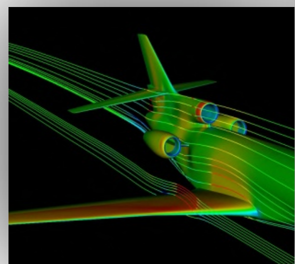
Schedule

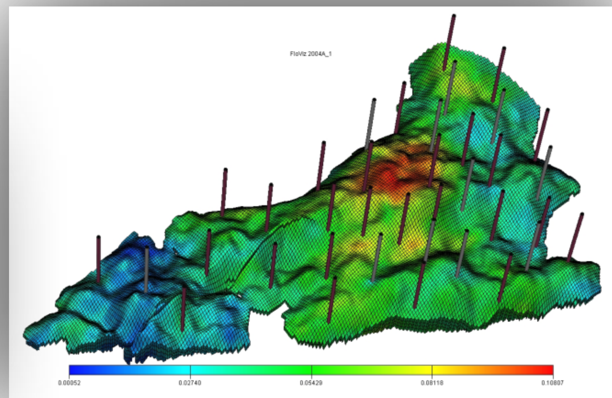
- 30 hours
  - 15 hours of lectures (F. Desprez, T. Ropars)
  - 6h of tutorials (T. Ropars, J.-F. Méhaut, B. Videau)
  - 12h of lab work (T. Ropars, J.-F. Méhaut, B. Videau)

Evaluation

- Short reports and a codes for lab work
- A final exam







Schedule

- **23/01/18** (3h lecture)
  - Introduction to parallelism and code optimization
  - Decomposition
- **31/01/18** (1h30 lecture, 1h30 tutorial)
  - Parallel architectures,
  - Classification
- **06/02/18** (1h30 lecture, 1h30 tutorial)
  - Shared memory
  - OpenMP
- **13/02/18** (1h30 lecture, 1h30 lab work)
  - Collective communications,
  - Algorithms
- /// ***Vacations*** ///
- **27/02/18** (3h lab work)
  - OpenMP
- **06/03/18** (1h30 lecture, 1h30 tutorial)
  - Parallel linear algebra,
  - vector matrix product,
  - Matrix matrix product on a ring
- **13/03/18** (1h30 lecture, 1h30 lab work)
  - Message passing,
  - MPI
- **20/03/18** (1h30 lecture, 1h30 lab work)
  - Algorithms on ring, contd.
  - Matrix matrix product on a grid of processors
- **27/03/17** (3h lab work)
  - Message passing (MPI)
- **03/04/17** (1h30 lecture, 1h30 tutorial)
  - Map Reduce
- **10/04/17** (1h30 lecture, 1h30 lab work)
  - Performance evaluation

- **Final exam (week of April 30th)**

Some References

Parallel Programming – For Multicore and Cluster System

T. Rauber, G. Rünger

Parallel Algorithms

H. Casanova, A. Legrand, Y. Robert

Sourcebook of Parallel Computing

J.J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy,L. Torczon, A. White

Parallel Computer Architecture

D.E. Culler, J. Pal Singh

Advanced Parallel Architecture - Parallelism, Scalability, Programmability

K. Hwang

Some References, contd.

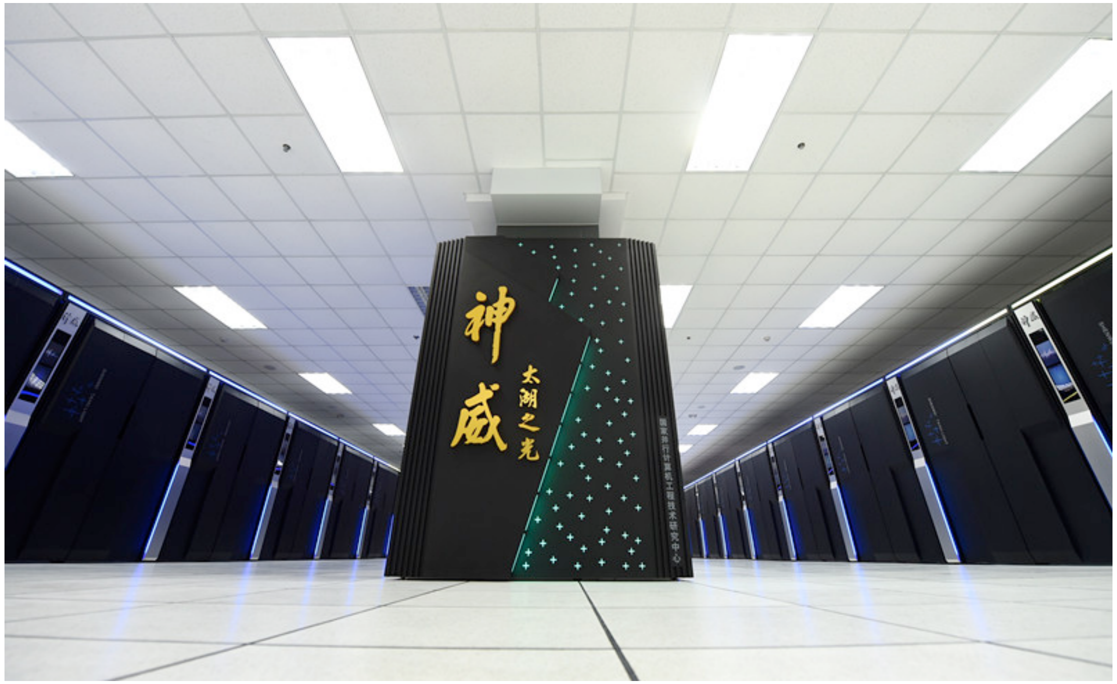
Online courses

- **Why parallel, why now,** Dr Clay Breshears, Intel
- **Applications of Parallel Computers,** J. Demmel, U.C. Berkeley CS267
- **Architecture et Système des Calculateurs Parallèles**, F. Pellegrini, LaBRI

INTRODUCTION

Why a Lecture on Parallel Computing ?

- Because it's everywhere !



Sunway Taihu light supercomputer

iPad

Because We Need It!

- **To**
  - **Solve problems more rapidly**
    - Execute more requests per second (example Google)
    - Enhance the response time of interactive applications (online games)
  - **Obtain better results for the same execution time**
    - Model refinement (example Météo France)
    - Use more complex models (multi-physics)
  - **Work on problems of larger scales**
    - Simulations, web page searches
- **Parallelism**
  - **To be able to accelerate an application by**
    - Dividing this application in subtasks, and
    - Execute these subtasks on different compute units
  - **To succeed, we need to be able to**
    - Find parallelism in the application
    - Find the best computation/communication ratio
    - To understand the behavior of the target platform

What is it all about?

- High Performance Computing (HPC)

  "How do we make computers compute bigger problems faster?"
- This field is both old and new, very diverse, complicated, interesting

- **Two main issues**
  - How do we build faster/bigger computers?
  - How do we write faster software for those computers?

- **Several different perspectives, from practical to theoretical**
  - Computer Architecture
  - Operating Systems
  - Networks
  - Programming Languages and Models
  - Algorithms

Performance measure units

- HPC units
  - Flop: floating point operation, generally in double precision
  - Flop/s: floating point operations per second
  - Bytes: data size (8 for a double precision number)

- Typical sizes millions, billions, trillions…

Mega	Mflop/s = 10^6 flop/sec	MByte = 2^{20} = 1048576 ~ 10^6 Bytes
Giga	Gflop/s = 10^9 flop/sec	GByte = 2^{30} ~ 10^9 Bytes
Tera	Tflop/s = 10^{12} flop/sec	TByte = 2^{40} ~ 10^{12} Bytes
Peta	Pflop/s = 10^{15} flop/sec	PByte = 2^{50} ~ 10^{15} Bytes
Exa	Eflop/s = 10^{18} flop/sec	EByte = 2^{60} ~ 10^{18} Bytes
Zetta	Zflop/s = 10^{21} flop/sec	ZByte = 2^{70} ~ 10^{21} Bytes
Yotta	Yflop/s = 10^{24} flop/sec	YByte = 2^{80} ~ 10^{24} Bytes

- Today's most powerful supercomputer ~ 93 Pflop/s (34 in 2014, 17 in 2013, 8.7 in 2012)
  - Updated list twice a year: www.top500.org

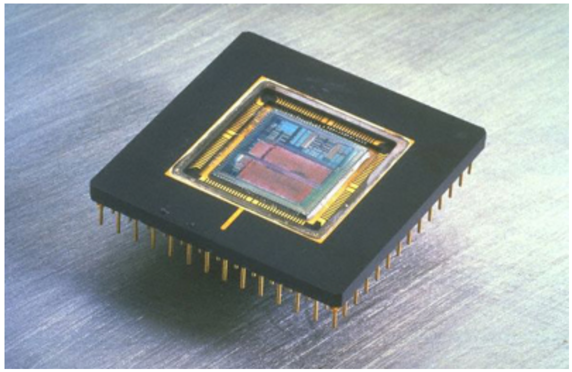
Why Not Accelerate Sequential Processors?

- **If we want to get a sequential machine at 1 Tflop/s/1 Tbyte**
  - Data should travel from memory to the CPU (distance r)
  - To get a data item per cycle (10^{12} times per second) at the speed of light (c = 299 792 458 m/s ≈ 3e8 m/s)
  - Thus $r < c/10^{12} = .3$mm

- **We need to put 1 Tera-Byte of data in 0.3 mm²**
  - Each word is located in ≈ 3 Angstroms², the size of a small atom

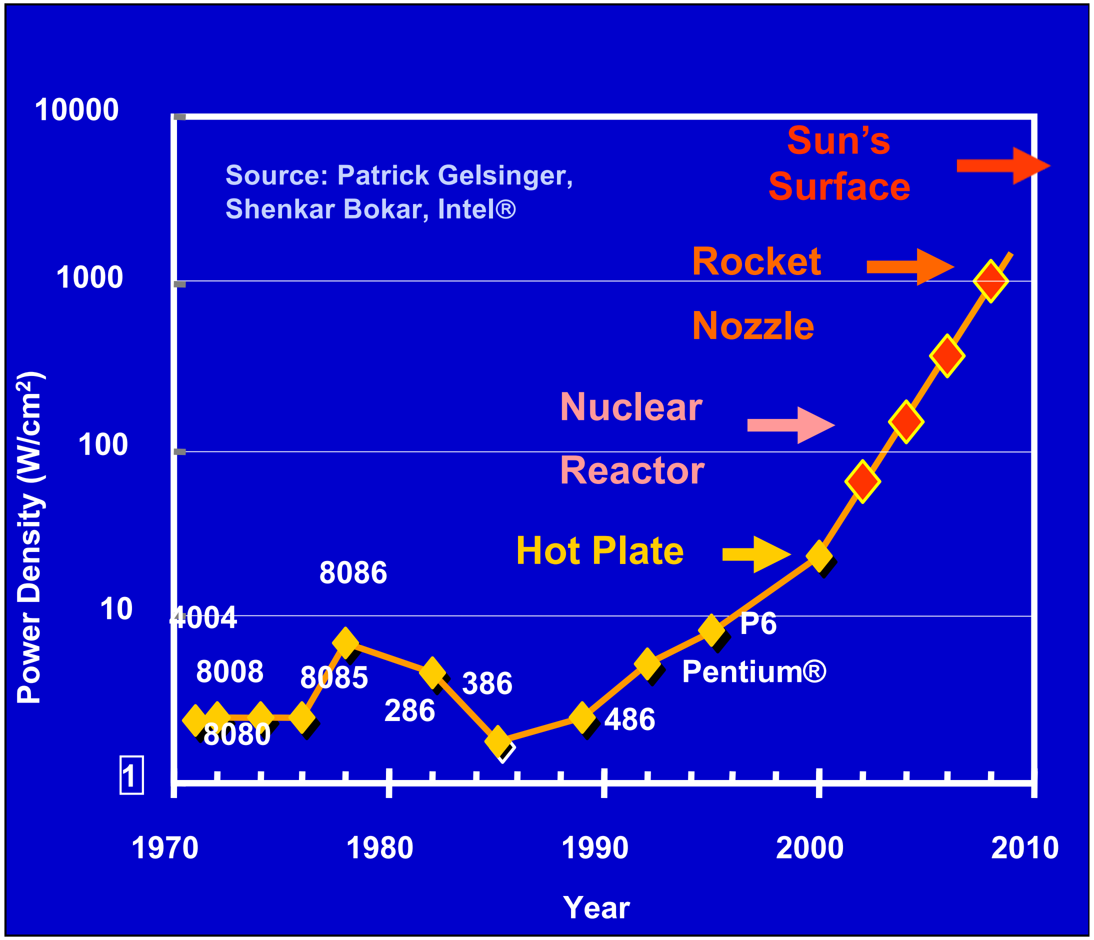
- Impossible to get this using today's technology

- Beware of the heat of such a processor!



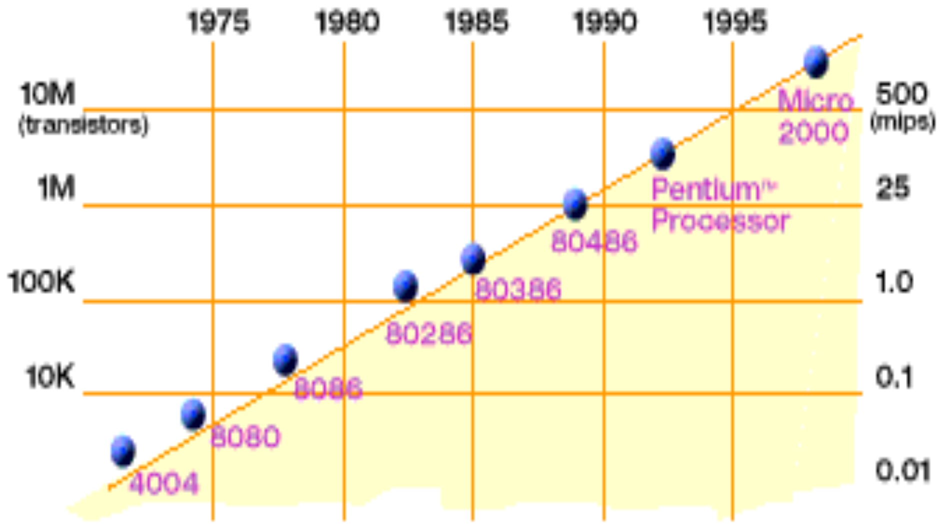
Density and Power Problems

- **Concurrent programs are more efficient from the energy point of view**
  - Dynamic power is proportional to V^2fC
  - Increasing frequency (f) increases also voltage (V)
  - Increasing the number cores increases the capacity (C) but linearly
  - Saving power by lowering the frequency



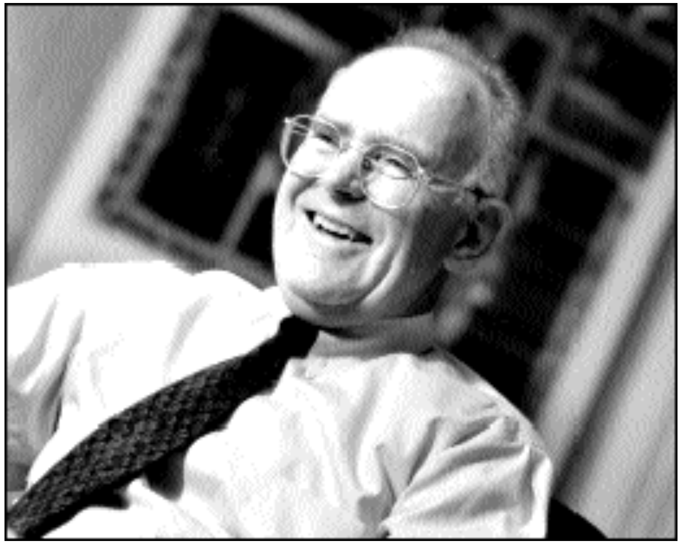
- **Sequential processors waste electrical power**
  - Speculation, dynamic verification of dependences
  - Finding parallelism

Moore’s Law



2X transistors/chip every 18 months
(60% increase per year)

Microprocessors have become smaller, denser, and more powerful



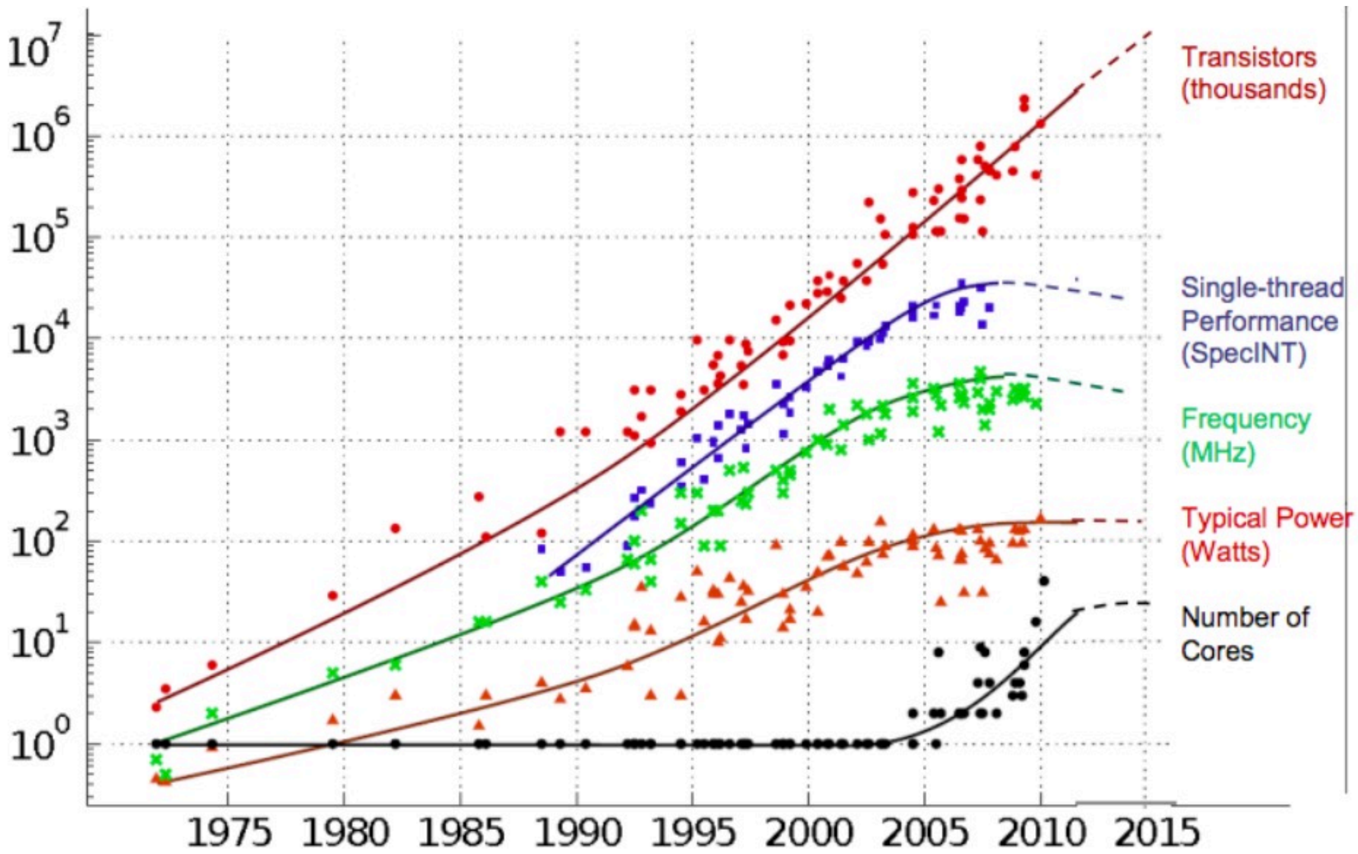
Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density would double roughly every 18 months

Source: Jack Dongarra

Moore's Law, contd.

- In 1965, empirical reasoning based on a relation between circuits complexity and time
- Law that was verified since then
- **Increase due to several factors**
  - Processors' complexity increase
    - transistor density, size's increase
  - Adding functionalities
    - internal caches,
    - longer instructions buffers,
    - several instructions per cycle,
    - multithreading,
    - pipelines depth,
    - re-arrangement of instructions

Processors' Revolution



- Processors' density still increases ~x^2/2 years
- Clock speed remains roughly the same
- Number of cores increases
- Electrical power is stable

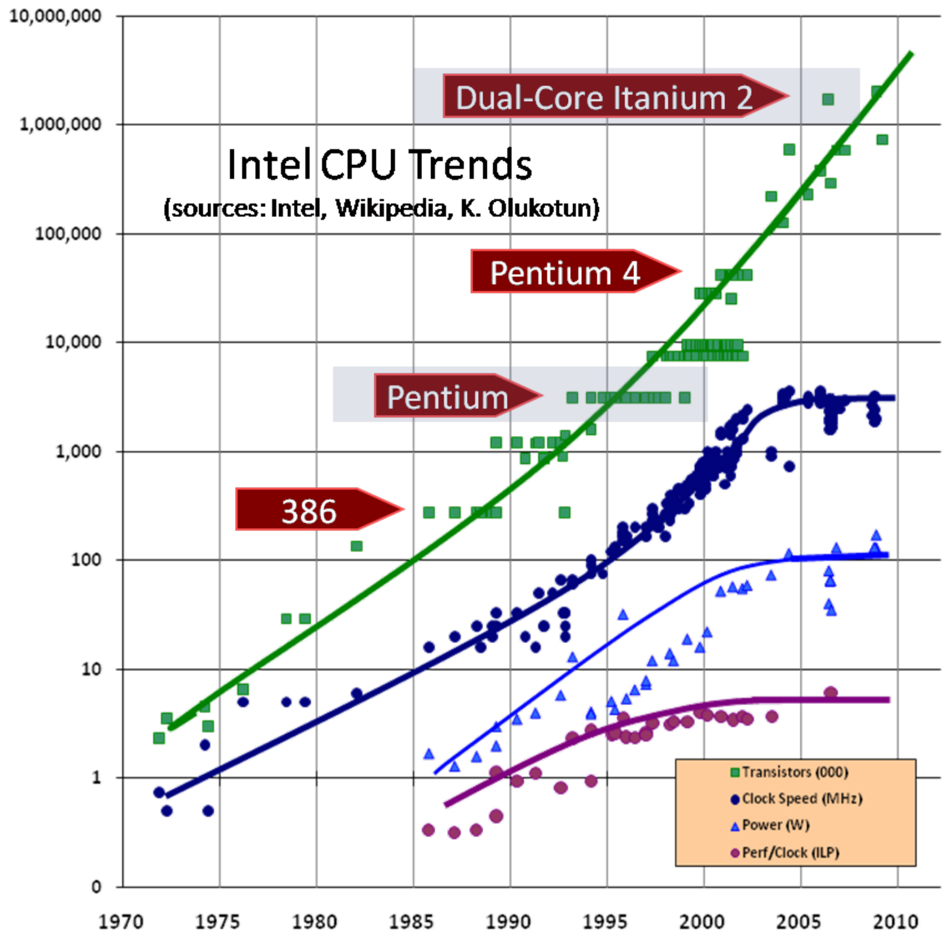
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

“Free lunch is over”, Herb Sutter

- Clock speed will not double anymore …
- … but performance need to increase anyway because of applications’ needs!

- Some issues related to the increase of clock’s speed
    - Power consumption
    - Heat Dissipation
    - Leaks

- But also
    - Physic limitation due to the speed of light (signal propagation)



The Free Lunch Is Over, A Fundamental Turn Toward Concurrency in Software, Herb Sutter, Dr. Dobb's Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm

Stabilization to Come

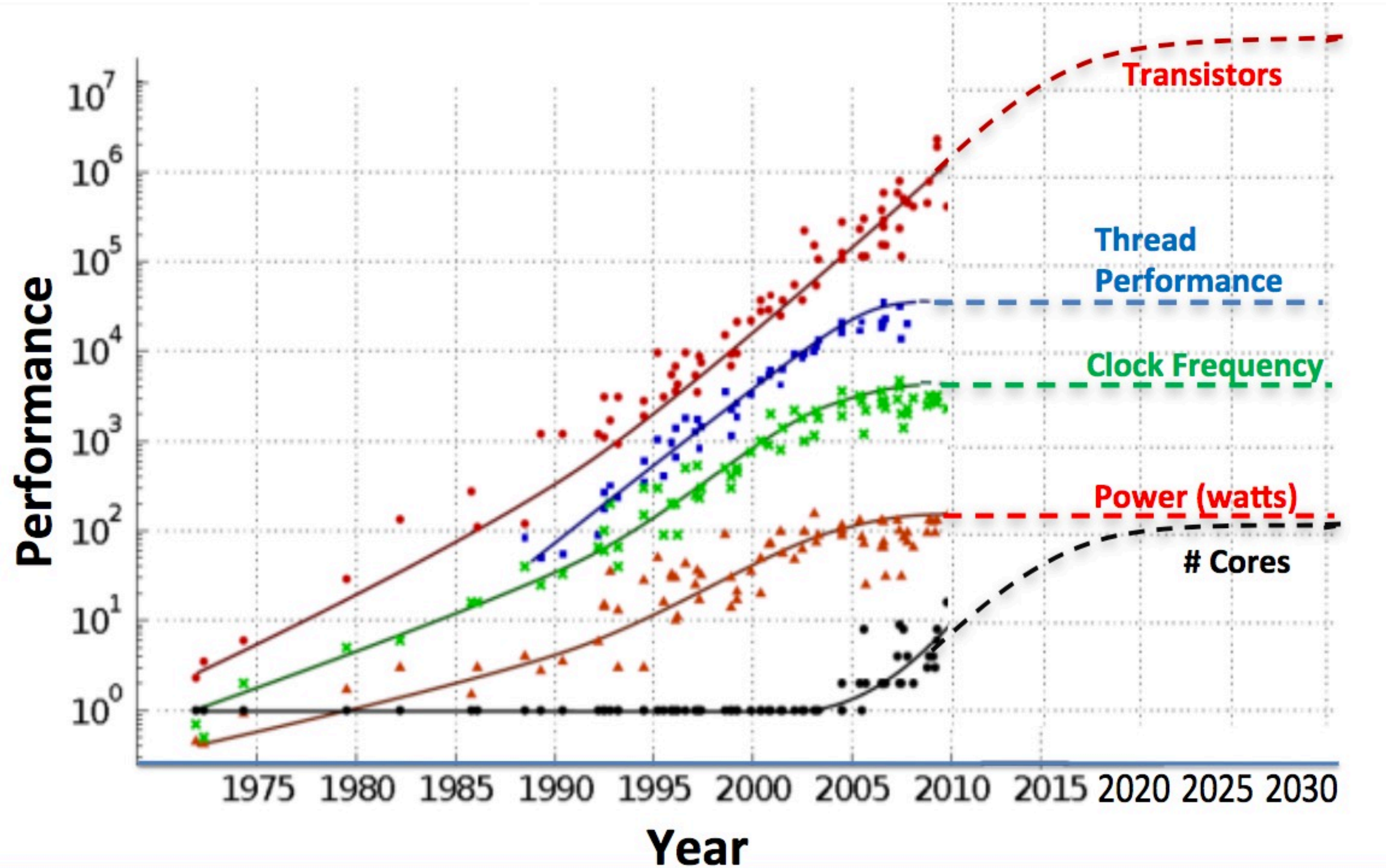
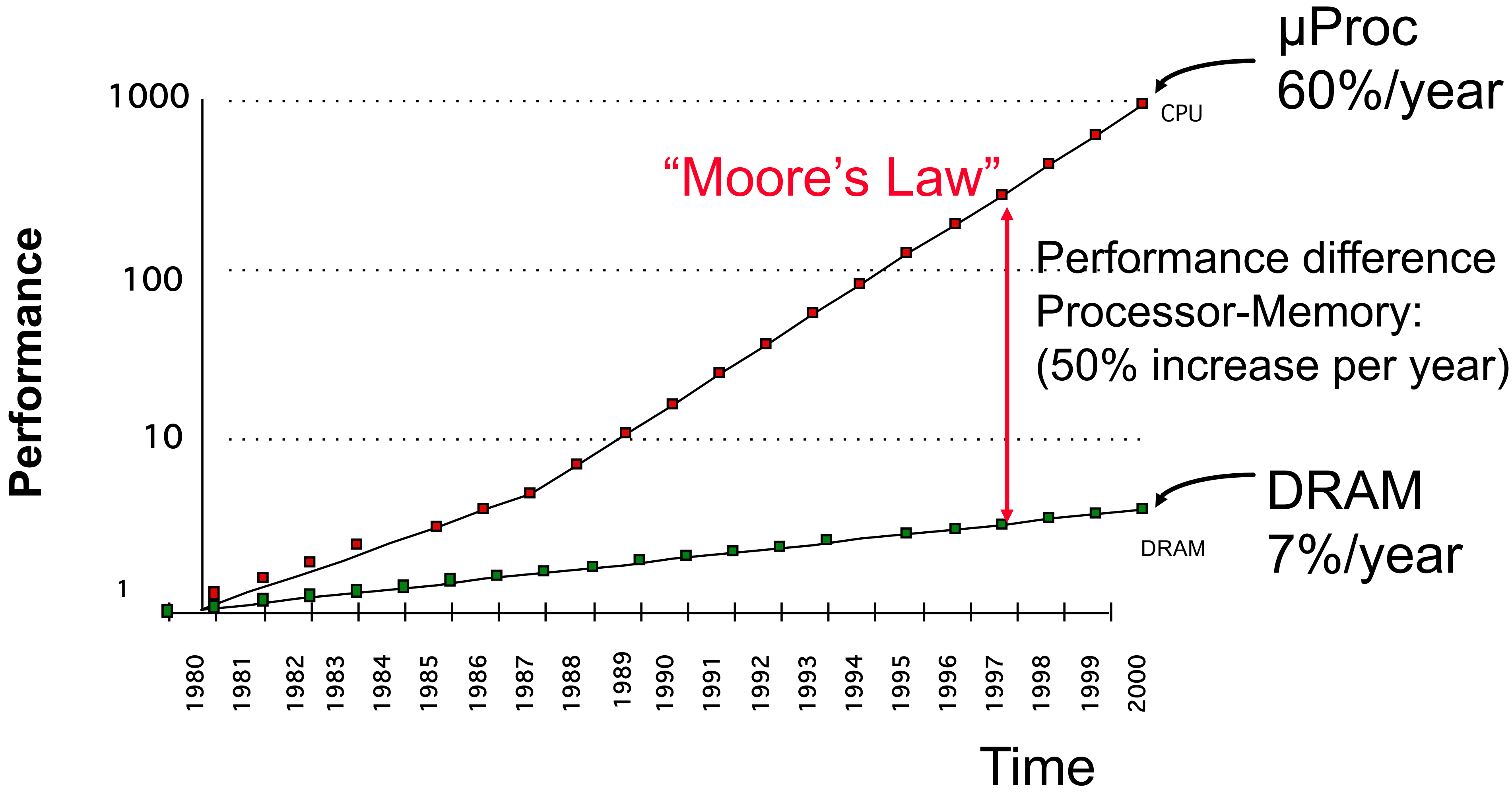


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

Processor-DRAM Difference (Latency)

Goal: finding algorithms that minimize data transfers, not only computations



Data Load Problem

- Computation execution time does not depend only from processor’s speed
- We have to take into account the data movement speed from memory to the processor
- Memory access speed has only increased of ~10% per year
  - Bottleneck that tends to increase

- System performance depend of the fraction of total memory that can be stored in a cache

- Better performance for parallel units because
  - Bigger aggregated caches
  - Larger aggregated bandwidth
  - Be careful with data locality!

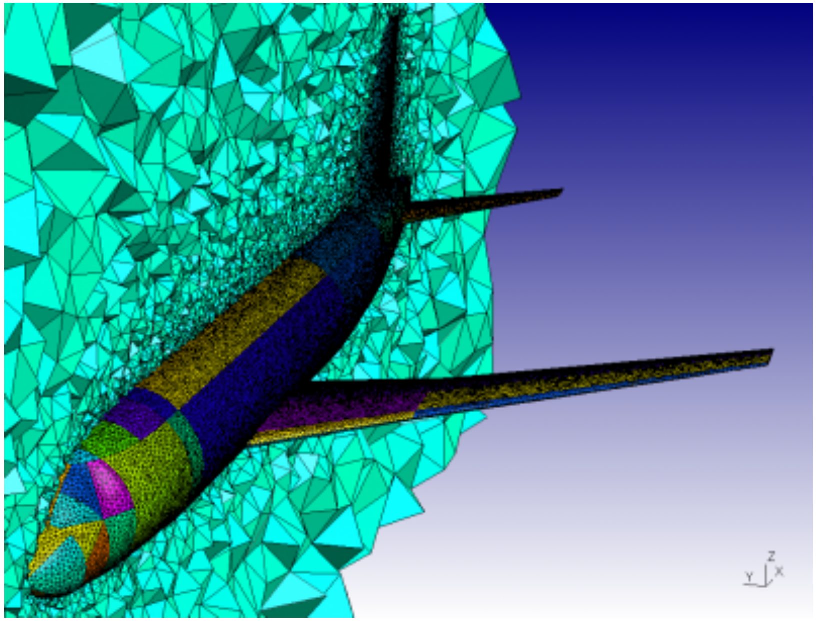
Data Load Problem, Contd.

- This problem exists also for large distributed architectures (computational grids, clouds)
- Different scale but similar problem
- Communication: Internet
  - Examples:
    - Web pages stored in Google datacenters for indexing and search
    - Genomic databases for bioinformatics applications

Consequence

- The only way to **increase performance** is to **increase the number of computing elements working in parallel**

- **The data transfer cost argument still holds !**
  - You have to cope with the computation grain and communication grain ratio
  - Do what it needs to have the computation volume be (really) larger than the communication volume

PARALLEL APPLICATIONS

Some Examples of Parallel Applications

- **Without computers, either**
  - We study problems on paper (theory)
  - We build an instrument and we perform experimentations
- **Limitations**
  - Too complicated
    - Ex: modeling a Tsunami
  - Too costly
    - Ex: crash test of an airplane
  - Too slow
    - Ex: climate evolution, planet evolution
  - Too dangerous
    - Ex: nuclear experiments, drugs
- **Using computer to simulate a phenomenon**
  - Based on physic rules and using numerical methods

Some Complex Problems

- **Science**
  - understanding matter from elementary particles to cosmology
  - storm forecasting and climate prediction
  - understanding biochemical processes of living organisms
- **Engineering**
  - combustion and engine design
  - computational fluid dynamics and airplane design
  - earthquake and structural modeling
  - pollution modeling and remediation planning
  - molecular nanotechnology
- **Business**
  - computational finance
  - information retrieval
  - data mining
- **Defense**
  - nuclear weapons simulation — cryptology

Growing Needs

Exponential growth of computation power needs

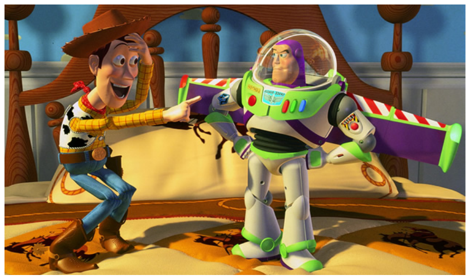
- Simulation is now the third pillar of science (with theory and experimentation)

Exponential growth of data volumes

- Data acquisition, analysis, and visualization improvement
- Storage and transport issues
- Collaborative tools
- Big data

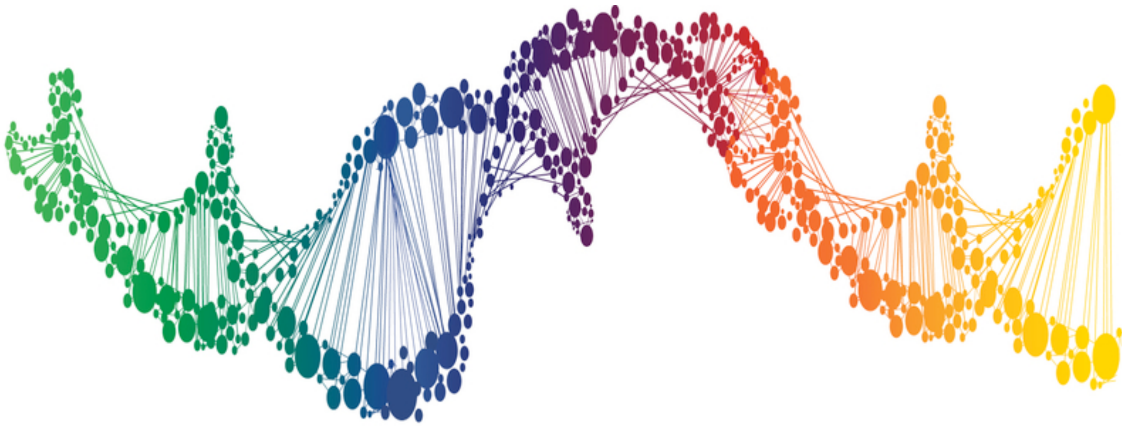
Animation

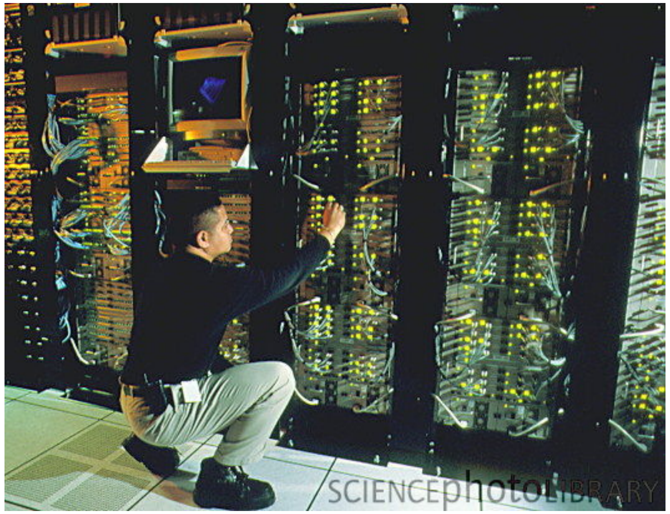
- Rendering is used to apply lights, textures, and shades over 3D models to generate 2D images for a movie
- Massive use of parallel computing to generate the huge number of images for a complete movie (24 images per second)
- **Some examples**
  - 1995, Pixar, Toy Story: first movie created using computers (“renderfarm” 100 machines dualprocs computers)
  - 1999, Pixar, Toy Story 2: using a system with 1400 processors for a better image quality
  - 2001, Pixar, Monster Inc.: 250 servers with 14 processors (3500 processors)
  - 2009, Industrial Light and Magic, Transformers 2: render farm with 5700 cores




Bioinformatics

- Large growth of computations thanks to the arrival of fast sequencing instruments for DNA (including for humans)
- Celera corp.: *whole genome shotgun algorithm*
    - Dividing the gene in small segments
    - Finding the DNA sequences experimentaly
    - Using a computer to construct the whole sequence by finding overlaps
    - Huge number of comparisons

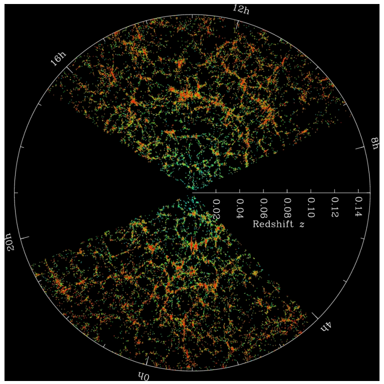




Astrophysics

Exploring the evolution of galaxies, thermonuclear processes, working on data coming telescopes

- Analysis of large volumes of data
  - Data coming from “Sky surveys”
    - Sloan Digital Sky Surveys, http://www.sdss.org/
  - Analysis of these data to find new planets, understand the evolution of galaxies

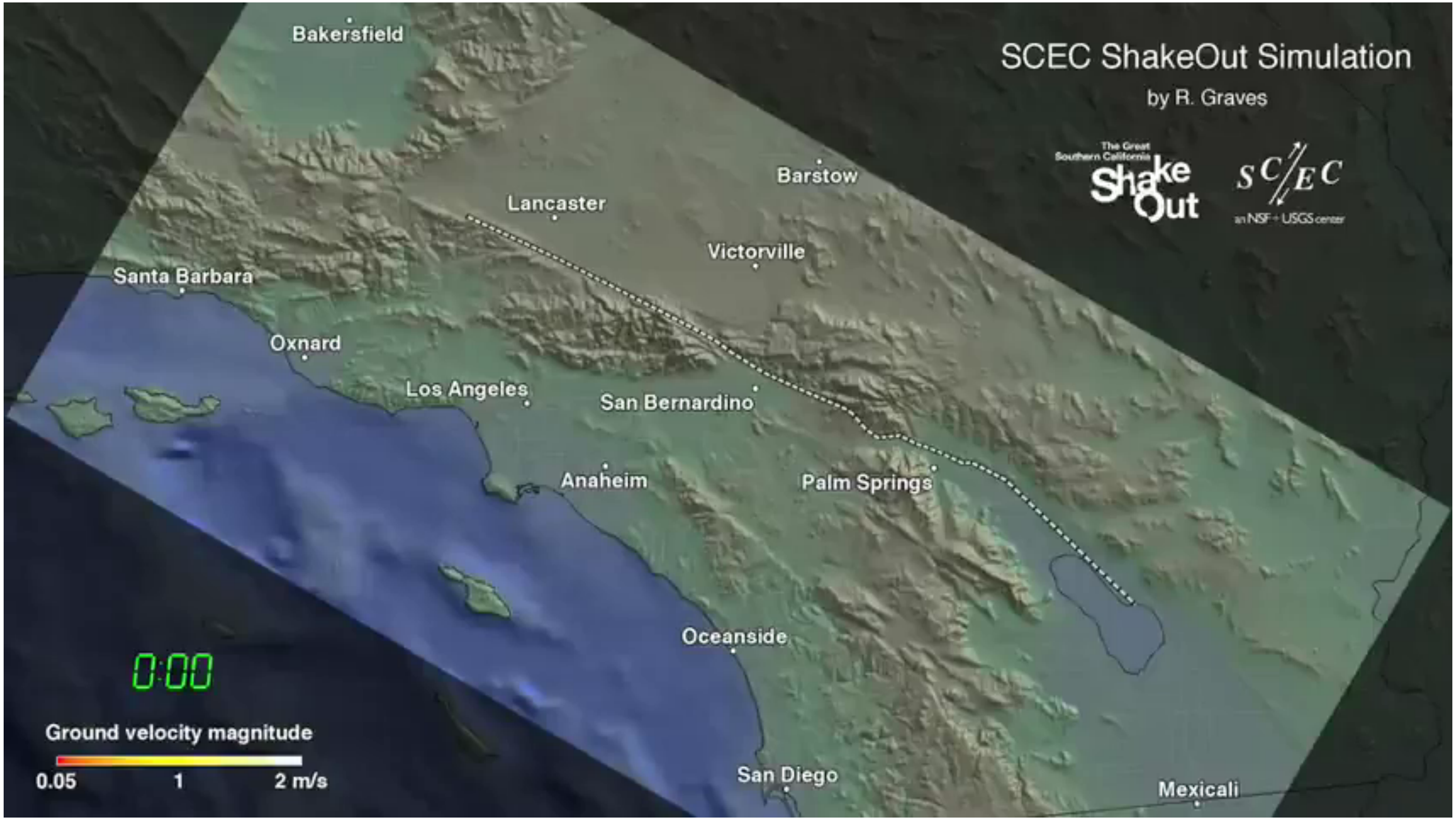


Credit: The Sloan Digital Sky Survey.

Earthquake Simulation

Southern California Earthquake Center ShakeOut Simulation workgroup.

Simulation by Rob Graves, URS/SCEC.

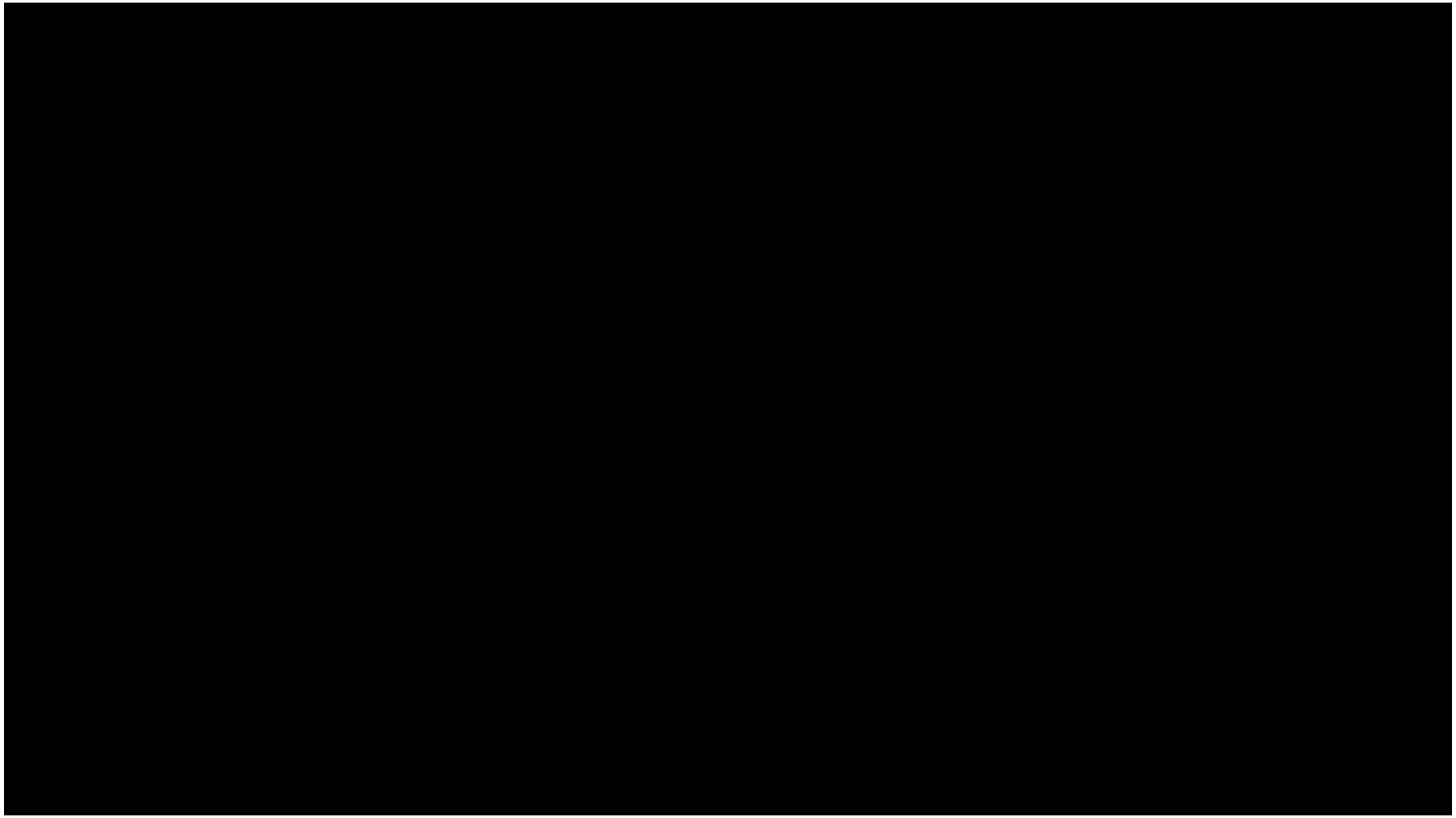


Ocean Circulation Simulation

Ocean Global Circulation Model for the Earth Simulator

Seasonal Variation of Ocean Temperature

http://www.vets.ucar.edu/vg/POP/index.shtml



Ocean Circulation Simulation, Contd.

Development of a mathematical model for the circulation for the oceans of the southern hemisphere

- Ocean divided into 4096 East-West regions, 1024 North-South regions, 12 layers in height ($50*10^6$ 3D cells)
- One iteration of the model simulates the circulation of the ocean for 10 minutes:
  - $30*10^9$ floating point operations
- For one year of simulation:
  - 52560 iterations
- Six years of simulation leads to 10^{16} operations !

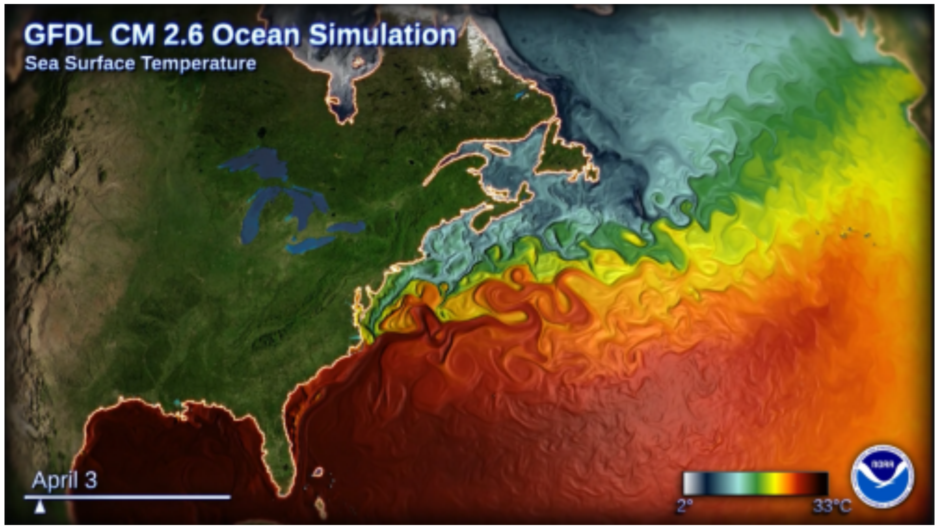
Climate Modeling

- Compute

  (temperature, pressure, humidity, wind speed) = f(latitude, longitude, height, time)

- Approach
  - Domain discretization (one measure point every 10 km)
  - Execute one algorithm that predicts the time at t+1 as a function of the one at t

- Utilization
  - Weather forecast
  - Natural disasters prediction
  - Evaluate climate changes
  - Sports events



Climate Modeling, Contd.

ESE Computational Technology Requirements Workshop

Computing Requirements for Weather

	2002 System	2010+ System	

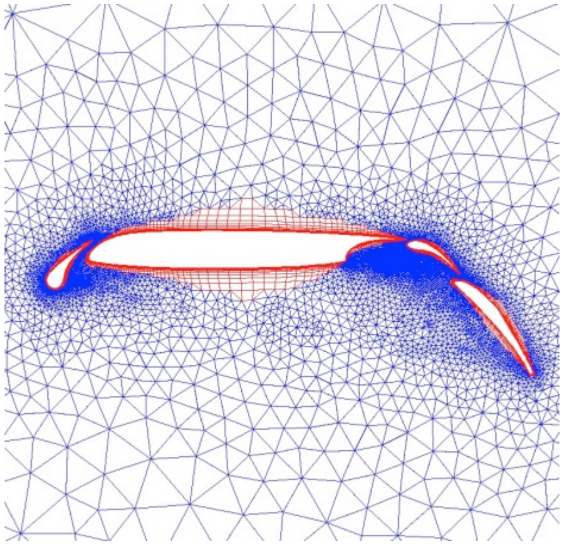
Resolution
• Horizontal
• Vertical levels
• Time step
• Observations
  ○ Ingested
  ○ Assimilated
	
	100 km
	55
	30 minutes
	
	10^7 / day
	10^5 / day
		
		10 km
		100
		6 minutes
		
		10^{11} / day
		10^8/ day	

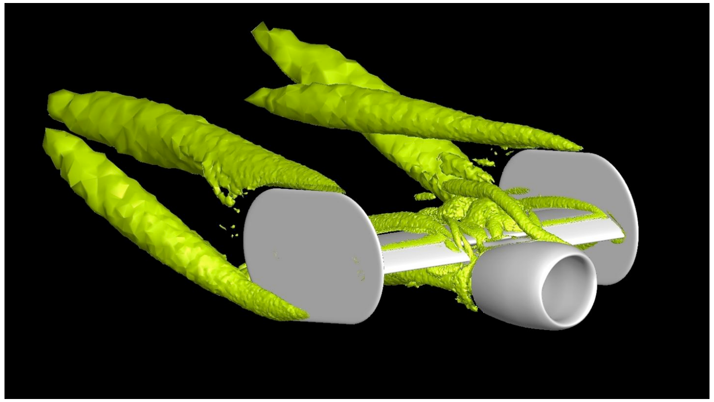
System Components:	Atmosphere Land-surface Data assimilation	Atmosphere, Land-surface, Ocean, Sea-ice, Next-generation data assimilation Chemical constituents (100)	
Computing: • Capability (single image system) • Capacity (includes test, validation, reanalyzes, development)	 10 GFlops 100 GFlops	Must Have 20 TFlops (2000x) 400 TFlop (4000x)	Important 50 TFlops 1 PFlops
Data Volume: • Input (observations) • Output (gridded)	 400 MB / day 2 TB / day	 1 PB / day 10 PB / day	

Networking/Storage
• Data movement
  ○ Internal
  ○ External
• Archival
	
	4 TB / day
	5 GB / day
	1 TB / day
		
		20 PB / day
		10 TB / day
		10 PB / day	

Air Flow Computing

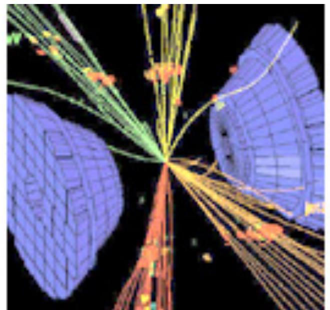
- Computing the air flows around a airplane wing
- Grid divided into small triangles: finite elements
- Refinement around some specific parts to be more accurate
- Domain decomposition

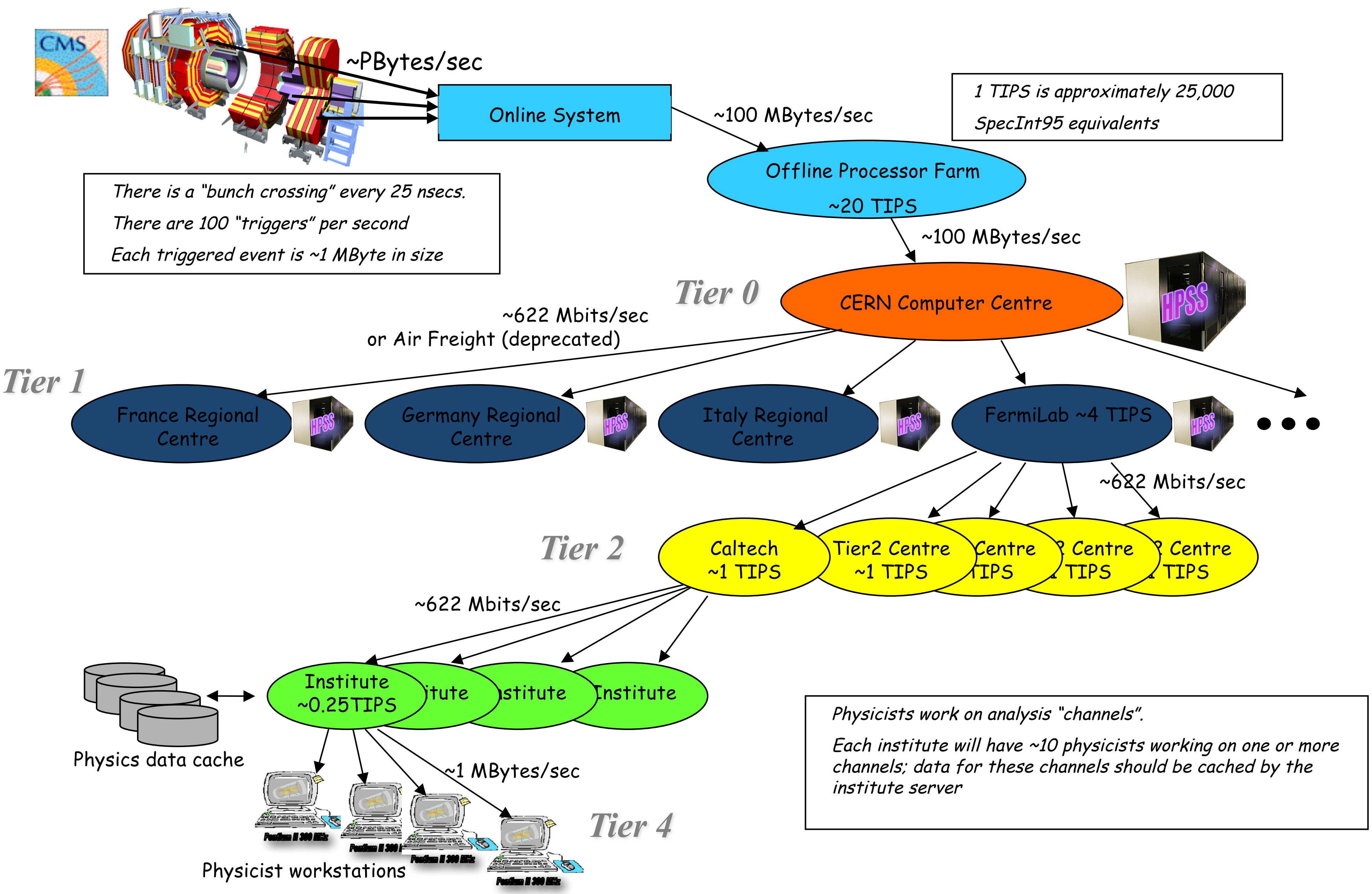




Large Hadron Collider (LHC)

- Higher energy collisions are the key to further discoveries of more massive particles ($E=mc^2$)
- One particle predicted by theorists remains elusive: the Higgs boson
- The LHC is the most powerful instrument ever built to investigate elementary particles
  - beams of protons collision at an energy of 14 TeV with a 27 km circumference instrument.
- Data
  - 40 million collisions per second
  - After filtering, 100 collisions of interest per second
  - A Megabyte of data digitised for each collision = recording rate of 0.1 Gigabytes/s
  - 10^{10} collisions recorded each year
  - = 10 Petabytes/year of data





Credits: Harvey Newman, Caltech

WHAT IS PARALLELISM?

What is Parallel Computing?

Being able to accelerate an application by

1. Dividing this application in sub-tasks
2. Execute these sub-tasks in parallel over different computation units

To succeed, we have to be able to

1. find parallelism in the application
2. find the appropriate computation/data exchange grain
3. get some knowledge of the target architecture to obtain an efficient solution

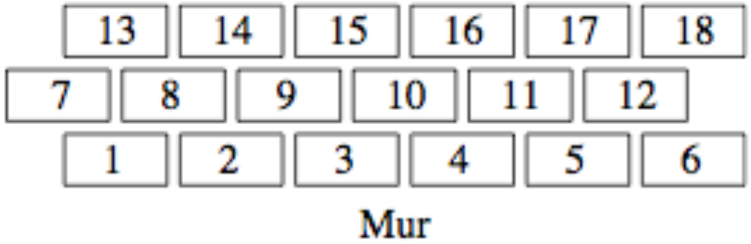
Parallelism

- **Parallelism is used everywhere on a computer**
  - Input/output operations overlap
  - Loading and preparing the next instructions while executing others
  - Using different units at the same time (integer and floating point units, multiple floating point units, graphical processing units)
  - Multitasking, data-prefetching and computing,
  - Horizontal multi-programming, VLIW (Very Long Instruction Word) processors

- In this lecture, we are going to study parallelism in the broad sense (models, architectures, algorithmic) with a special focus on the use of different units (processors, cores) for the computation

One Glance of Parallelism

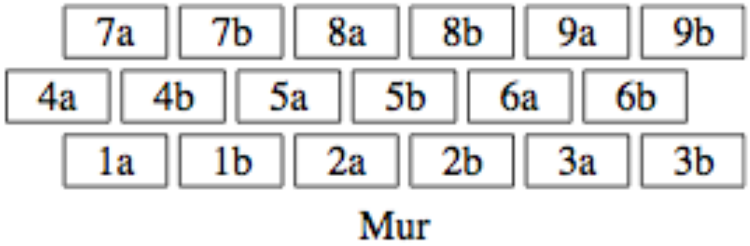
- Work of a construction worker building a wall



- Alone, he builds it row after row
  - Slow!

One Glance of Parallelism, Contd.

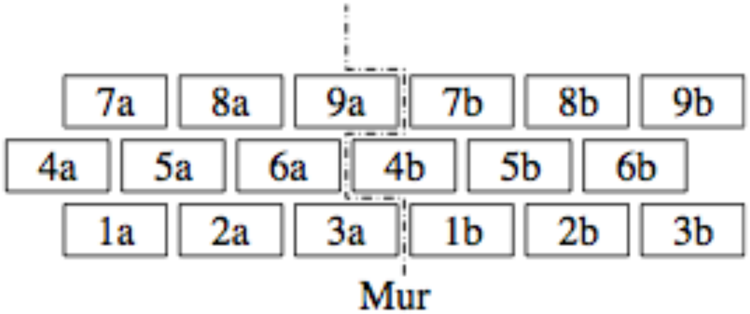
- Work of two construction workers (a et b) building a wall



- One brick after an other
    - They interfere between each other to pick the bricks up and put them in place

One Glance of Parallelism, Contd.

- Work of two construction workers (a et b) building a wall



- Each one gets one part of the wall to work on
- More efficient but
  - b has a longer way to walk to fetch the bricks
  - They still interfere between each other to pick the bricks up
- Other ways
  - a works from right to left, b starts earlier but synchronization problems
  - a throws the bricks to b when he fetches one for him

One Glance of Parallelism, Contd.

- **Some thoughts about this simple example**
  - Two construction workers are more efficient than a single one, but
  - More work because of the interactions between the two construction workers

- **In general**
  - To get a parallel application, it should be able to be decomposed into independent sub-parts
  - We should be able to organize the repartition of work
  - Overhead due to the work repartition (fetching the bricks)
  - Find the best parallel algorithm ...
  - Maybe not the most efficient sequential one !

What do we expect?

To get a good **speed-up** !

- Ideally, we expect to get a speedup of *p* over *p* processors!

Unfortunately, this is not often the case

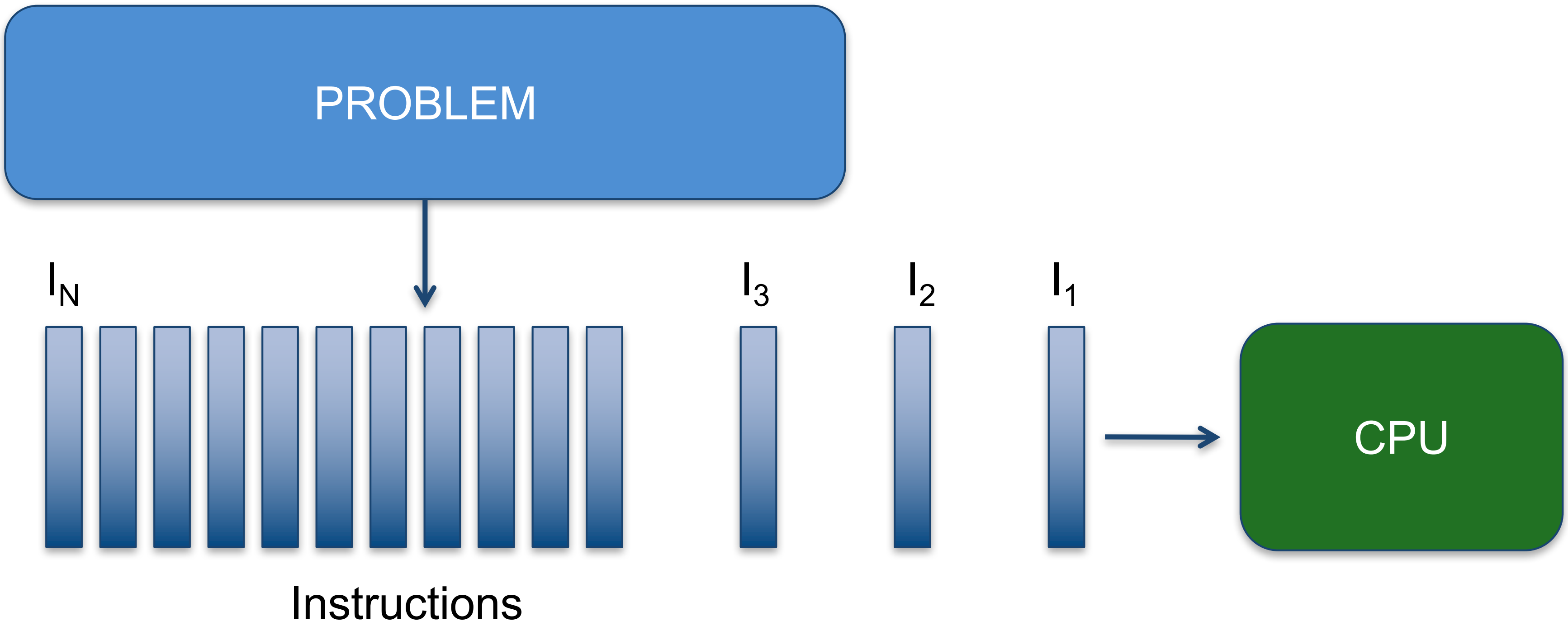
- Sequential parts of the algorithm
- Overhead problems due to redundant computations, data transfers (memory, disks, network)

Sometimes, the gain can be higher than *p!*

- This is called superlinear speed-up
- Thanks to different memory speeds (main memory vs caches), less computations thanks to parallelism (searches in trees)
- Applications for which the sequential execution is impossible (infinite execution time)

Back to Computers

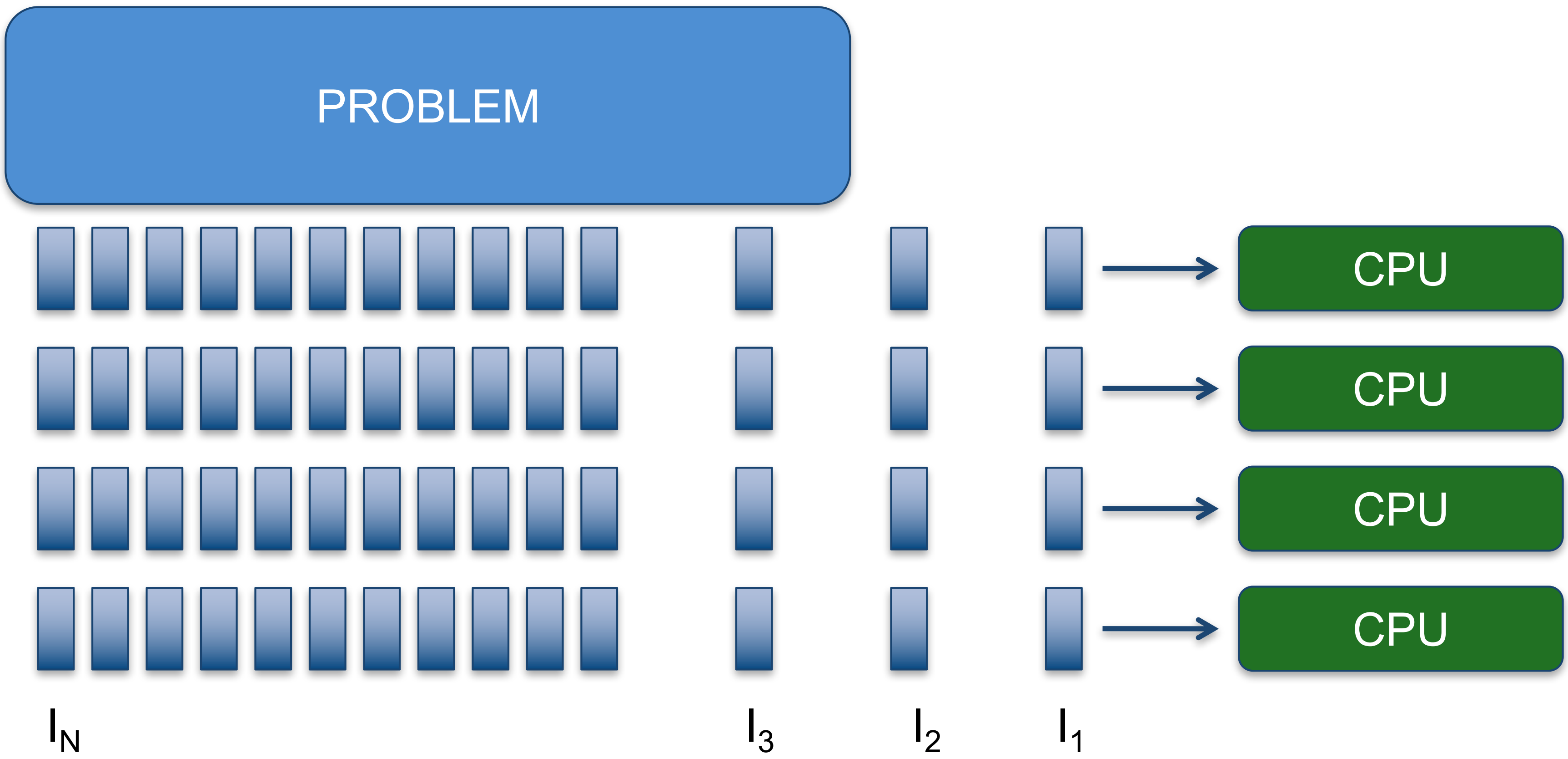
- Programs are usually designed to execute on sequential processors
  - Unique Central Processing Unit (CPU)
  - Application based on a sequence of instructions executed one after an other
  - Only one instructions is executed at a given time



Parallelism

Under its simplest form, we use several resources to solve a problem

- Problem divided in several (possibly) independent parts
- Using several CPU



What is a Parallel Machine ?

- A collection of computing elements able to communicate and cooperate in order to solve large size problem more efficiently (i.e. in a shorter time)

- **A collection of processing elements**
  - How many of them?
  - How much computing power?
  - What can they perform?
  - What is the size of their memory?
  - What is their organization?
  - How are performed the input/output?

- **… able to communicate …**
  - How are they connected?
  - What can they exchange?
  - What is their data exchange protocol?

What is a Parallel Machine ? Contd.

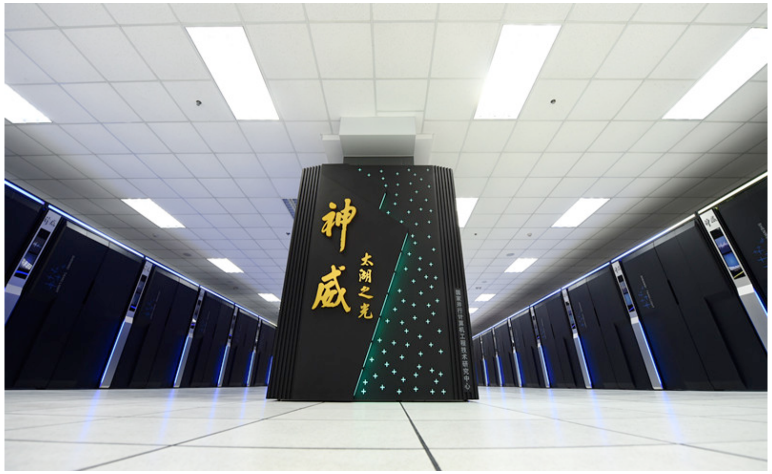
- **… and to cooperate …**
  - How do these computing elements synchronize themselves?
  - What is their degree of autonomy?
  - How are they seen by the operating system?
- **… in order to solve large size problem more efficiently (i.e. in a shorter time)**
  - What are the problems with lot of internal parallelism?
  - What is the computation model used?
  - What is the degree of specialization of the machines to a given problem?
  - How should we choose the algorithms?
  - What efficiency can be expected?
  - How are these machines programmed?
  - What languages are needed?
  - How should parallelism be expressed?
  - Is parallelism implicit or explicit?
  - Is parallelism extraction automatic or manual?

TOP 500 (November 2017)

#	Site	Vendor	Computer	Country	Cores	Rmax [Pflops]	Rpeak (Pflops)	Power [MW]
1	National Supercomputing Center in Wuxi	NRCPC	Sunway TaihuLight NRPC Sunway SW26010, 260C 1.45GHz	China	10,649,600	93.0	125.4	15.4
2	National University of Defense Technology	NUDT	Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi	China	3,120,000	33.9	54.9	17.8
3	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC50, Xeon E5 12C 2.6GHz, Aries, NVIDIA Tesla P100	Switzerland	361,760	19.6	25.3	2.27
4	Japan Agency for Marine-Earth Science and Technology	Exa-Scaler	Gyoukou ZettaScaler-2.2 HPC System, Xeon 16C 1.3GHz, IB-EDR, PEZY-SC2 700Mhz	Japan	19,860,000	19.1	28.2	1.35
5	Oak Ridge National Laboratory	Cray	Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x	USA	560,640	17.6	27.1	8.21
6	Lawrence Livermore National Laboratory	IBM	Sequoia BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	1,572,864	17.2	20.1	7.89
7	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	979,968	14.1	23.9	3.84
8	Lawrence Berkeley National Laboratory / NERSC	Cray	Cori Cray XC40, Intel Xeons Phi 7250 68C 1.4 GHz, Aries	USA	622,336	14.0	27.9	3.94
9	JCAHPC Joint Center for Advanced HPC	Fujitsu	Oakforest-PACS PRIMERGY CX1640 M1, Intel Xeons Phi 7250 68C 1.4 GHz, OmniPath	Japan	556,104	13.6	24.9	2.72

Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway (1st)

- 10,649,600 cores
- 125,435.9 TFlop/s peak performance
- 93,014.6 TFlop/s obtained
- 1,310,720 GB memory
- 15,371.00 kW energy consumption
- Processor: Sunway SW26010 260C 1.45GHz (3.06 TFlop/s peak !)
- Interconnect: Sunway
- Operating system: Sunway RaiseOS 2.0.5



Piz Daint (2nd in nov. 2017, 3rd in June)

- 321,760 cores
- 25,326 TFlop/s peak performance
- 19,600 TFlop/s obtained
- 2,270.00 kW energy consumption
- Processor: Intel Xeon E5-26xx (various), Nvidia Tesla P100
- Operating system: Linux

Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x (5)

- 560640 cores
- Theoretical Peak (Rpeak): 27112.5 TFlop/s
- Linpack Performance (Rmax): 17590.0 TFlop/s
- http://www.olcf.ornl.gov/titan/

Before using parallelism, optimize the codes!

- **Some classical optimizations**
  - Instruction prefetching
  - Instruction re-ordering
  - Pipelined functional units
  - Branch prediction
  - Functional units allocation
  - Hyperthreading

- On the other hand, this requires a complexification of
  - the hardware (parallel functional units) and
  - the software (compilers, operating systems, runtime systems) to support them

FINDING PARALLELISM

How to find parallelism?

- **Starting from a sequential language?**

- The application has an intrinsic parallelism
- The chosen programming language does not have a "parallel" extension

➔ The compiler, the operating system and / or the hardware must find a way to discover the hidden parallelism!

- Correct behavior for some trivially parallel applications (parallelization of simple nested loops for example)
- But in general, disappointing results and problems related to the dynamicity (pointers in C for example)

“Automatic” parallelism in today’s processors

- **Parallelism a the bit level (BLP, *Bit Level Parallelism*)**
  - In floating point operations
- **Instruction parallelism (ILP, *Instruction Level Parallelism*)**
  - Executing several instruction per clock cycle
  - Super-scalar, VLIW, EPIC, ThLP (Thread level parallelism: multithreading)
- **Parallelism of memory managers**
  - Overlapping memory accesses with computations (prefetch)
  - Vector operations in parallel (A[*] ← 3×A[*])
- **Parallelism at the system level**
  - Executing different tasks on different processors (or cores)
  - <u>fork</u> [func1(), func2()], <u>join</u> [*]

- **Limits to this “implicit” parallelism**
  - Intelligence level of processors and compilers
  - Complexity of applications
  - Number of elements in parallel

Other approach: cooperation

The programmer and the compiler work together

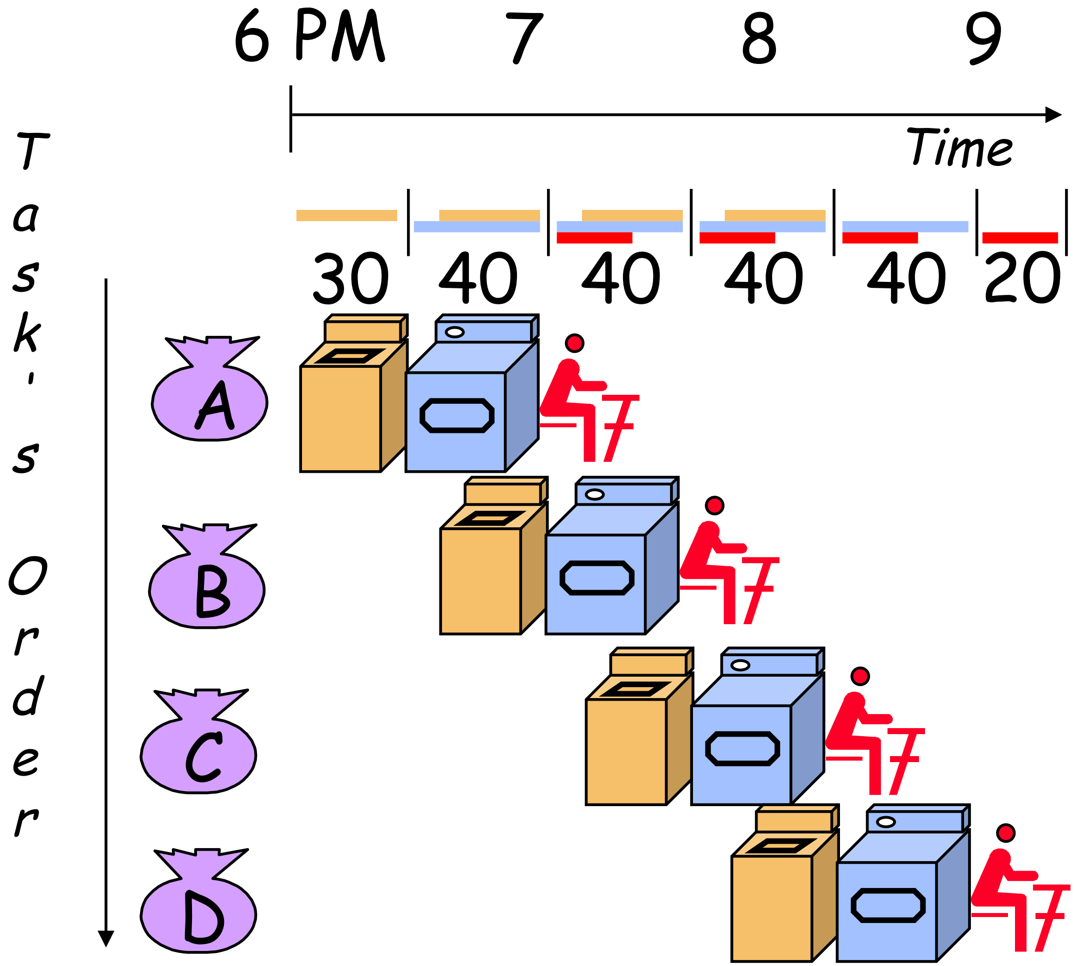
- The application has an intrinsic parallelism
- The language has extensions to express parallelism
- The compiler will translate the program for multiple units

- The programmer gives advice to the compiler on the areas to be optimized, what are the parallel loops, ...
- The compiler, starting from the information it possesses on the hardware (size of the caches, number of parallel units, information about the performances), will be able to generate an efficient code

Pipelines

Dave Patterson's example: 4 people who wash their laundry

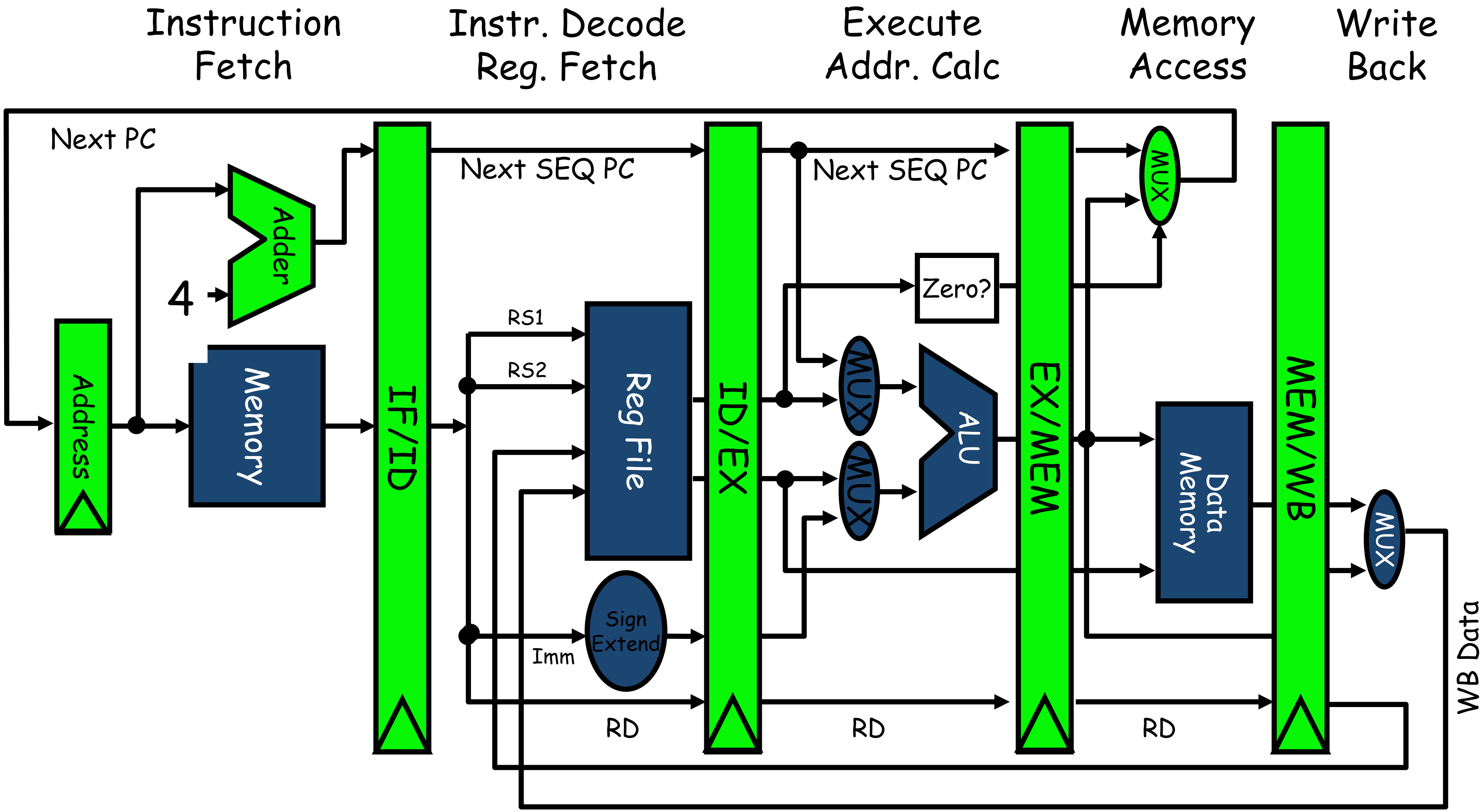
washing (30 min) + drying (40 min) + folding (20 min) = 90 min



- In this example
  - The sequential execution takes 4 * 90 min = 6 h
  - The pipelined execution takes 30+4*40+20 = 3.5 h
- Bandwidth = loads/h
- BW = 4/6 l/h without pipeline
- BW = 4/3.5 l/h with pipeline
- BW <= 1.5 l/h with pipeline
- The pipeline improves the **bandwidth** but not the **latency** (90 min)
- The bandwidth is limited by the **slower** stage
- Potential acceleration = **number of pipeline stages**

Stages of the MIPS processor

Figure 3.4, Page 134 , CA:AQA 2e from Patterson & Hennessy

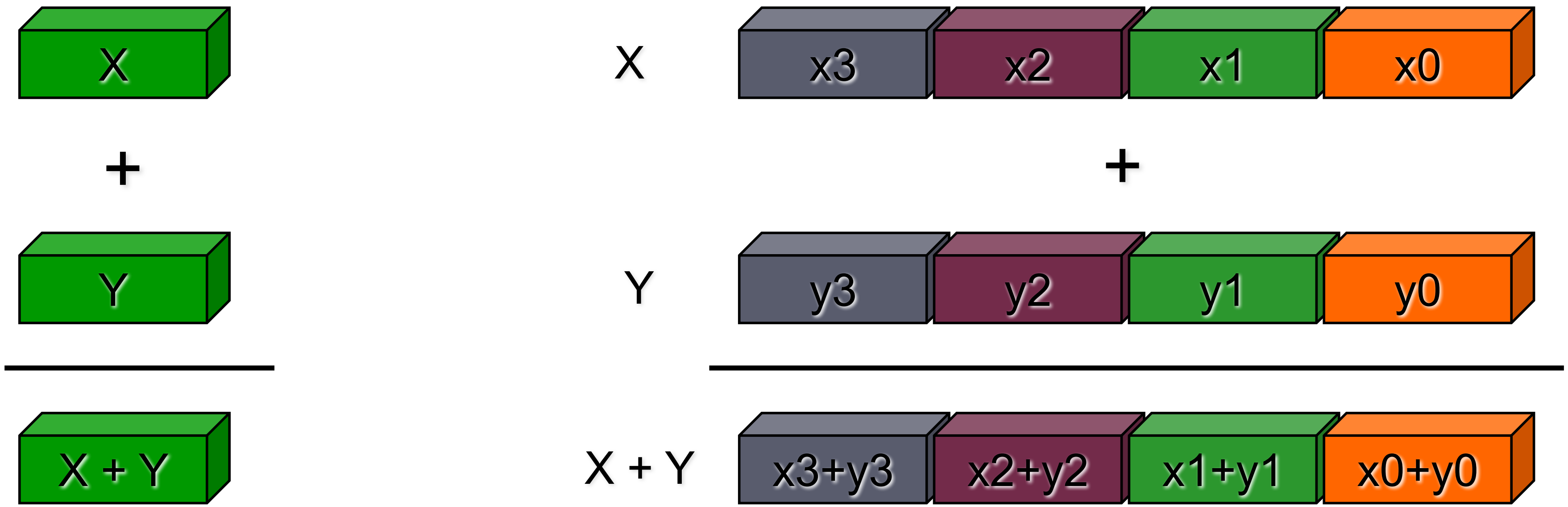


Pipeline used by arithmetic units

– A floating point unit can have a latency of 10 cycles and a bandwidth of 1 cycle

SIMD: Single Instruction, Multiple Data

- Scalar computation
  - "classic" mode
  - **One operation** produces **one result**
- SIMD computation
  - with SSE / SSE2
  - SSE = streaming SIMD extensions
  - **one operation** produces **multiple results**



Credits: Alex Klimovitski & Dean Macri, Intel Corporation

SSE/SSE2 on Intel processor

- SSE2 data types: everything that can fit in 16 bytes, thus

- Instructions perform additions, multiplications, etc. in parallel over all the data stored in these 16 bits registers

- **Challenges**
  - Should be contiguous and aligned in memory
  - Some instructions to move data from one part of a register to an other

- Similar to GPU, vector processors (but more simultaneous operations)

Special instructions and compilers

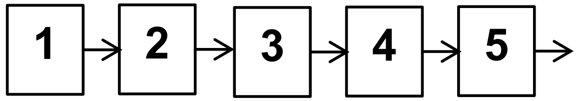
- In addition to the SIMD instructions, the processor may also have other instructions
  - multiply-add instruction (*Fused Multiply-Add, FMA)*

$$x = y + x*z$$

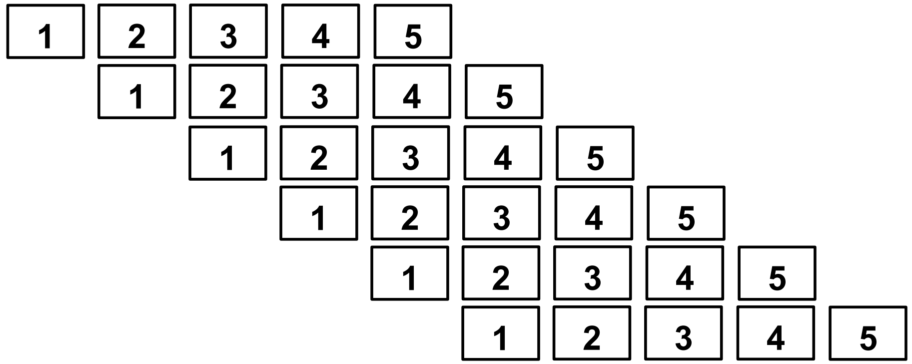
  - The processor executes these instructions at the same frequency as a * or a +
- In theory compilers know these instructions
  - When compiling, the compiler will re-arrange the instructions to get a good scheduling instructions that will maximize the pipeline (FMA and SIMD)
  - It uses mixtures of such instructions in internal loops
- In practice, the compiler needs help
  - Taking compilation flags into account
  - Re-arrange the code so that it finds easier the good "pipelines”
  - Use special functions
  - Writing in assembly code!

ILP: superscalar approach

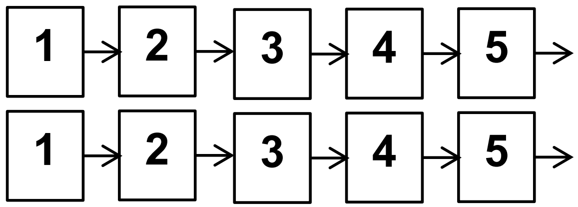
Scalar pipeline:



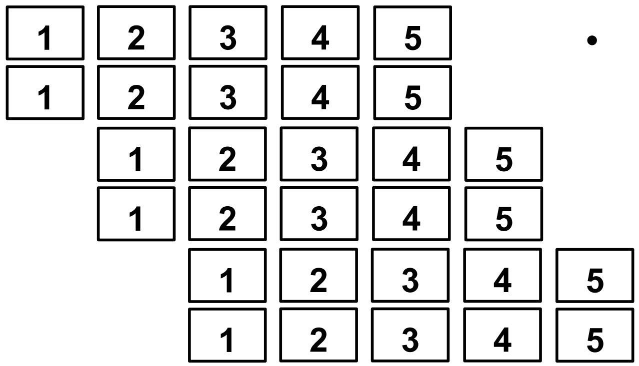
Scalar execution:



Superscalar pipeline:



Superscalar execution:



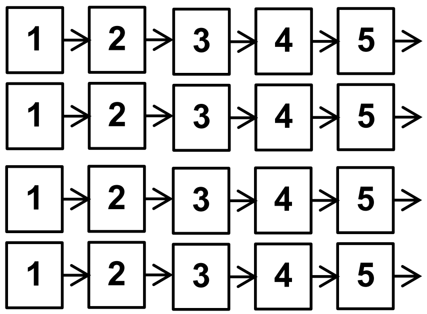
- Hardware
  - detects dependences
  - Manages resources

<u>Scheduling at runtime multiple instructions from a sequential code</u>

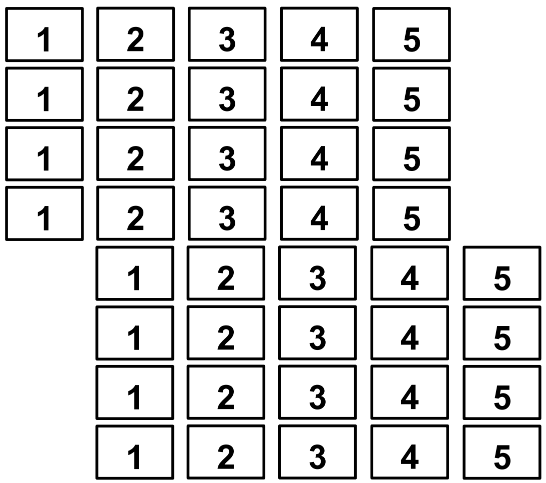
ILP: VLIW (*Very Long Instruction Width*)

- Parallelization of instructions at compile time
- **Problems**
  - ISA VLIW between the different processors (with different sizes)
  - The speed of cache/DRAM load main vary: scheduling problem

4x VLIW pipeline:



VLIW execution:



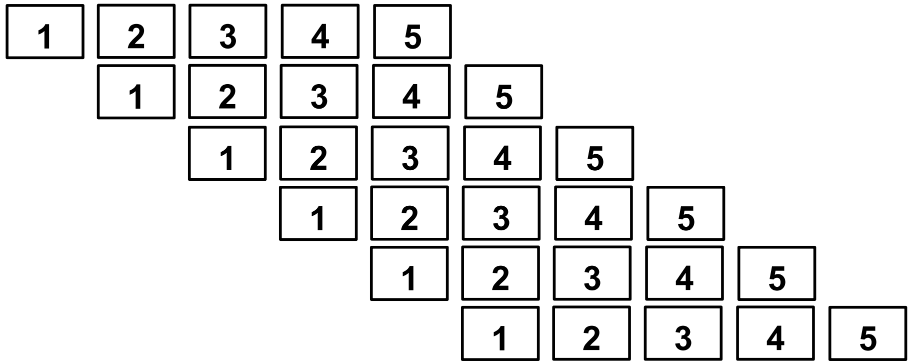
Numerous cores use 2-3x VLIW

Vector machines

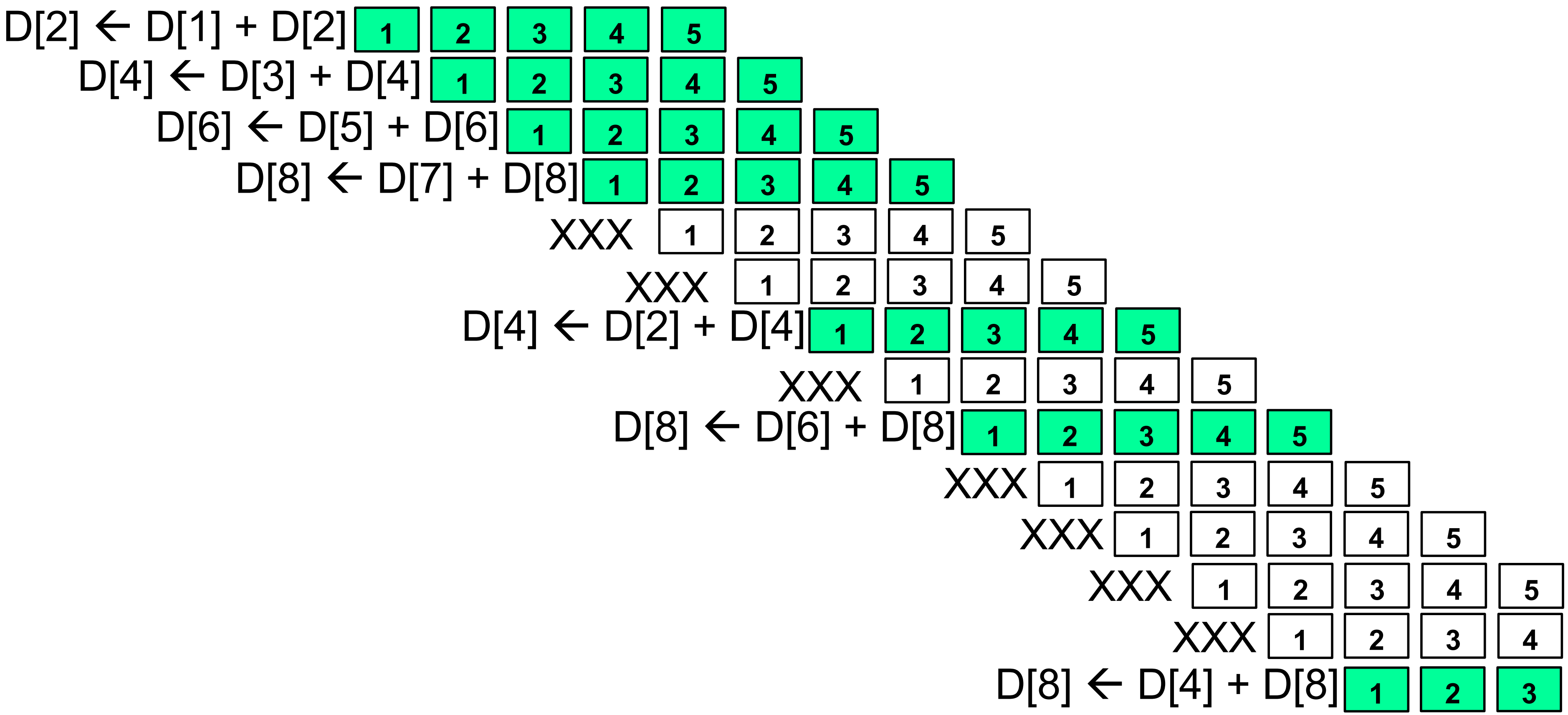
- **Goal :** vector operations (data parallelism) without the need for parallel programming
  - Simple: A[*] ← B[*] x C[*]
  - Scatter / gather: e.g. S ← sum (D[*])
- Two types of vector machines
  - Pipelined vector
  - Array vector
- In the 1970's and 1980's, all the supercomputers were vector machines CRAY, CDC, …

Simple vector operation

A[1] ← B[1]*C[1]
A[2] ← B[2]*C[2]
A[3] ← B[3]*C[3]
A[4] ← B[4]*C[4]
A[5] ← B[5]*C[5]
A[6] ← B[6]*C[6]



Pipelined vector operation: gather (sum)



ONE EXAMPLE: MATRIX PRODUCT OPTIMIZED IN SEQUENTIAL

Credits: J. Demmel and K. Yelick, Berkeley

Motivation

- Most applications run at less than 10% of a processor's peak performance
- Most losses are on a processor only
  - Execution of the code with performances of 10 to 20% of peak performance
  - Other losses are due to communications
- The loss of performance is due to the management of the data in memory
  - Moving data is more expensive than arithmetic operations and connections

- Purpose of this section
  - Understanding how algorithms behave

Idealized sequential machine model

- Integers, floats, pointers, arrays stored as bytes
- Operations
  - Reading and writing to fast memories (registers)
  - Arithmetic and logical operations on these registers
- Order specified by the program
  - Read the most recent written data
  - Compilers and the architecture translate expressions into lower-level instructions

A = B + C ⇒

Read address(B) to R1
Read address(C) to R2
R3 = R1 + R2
Write R3 to address(A)

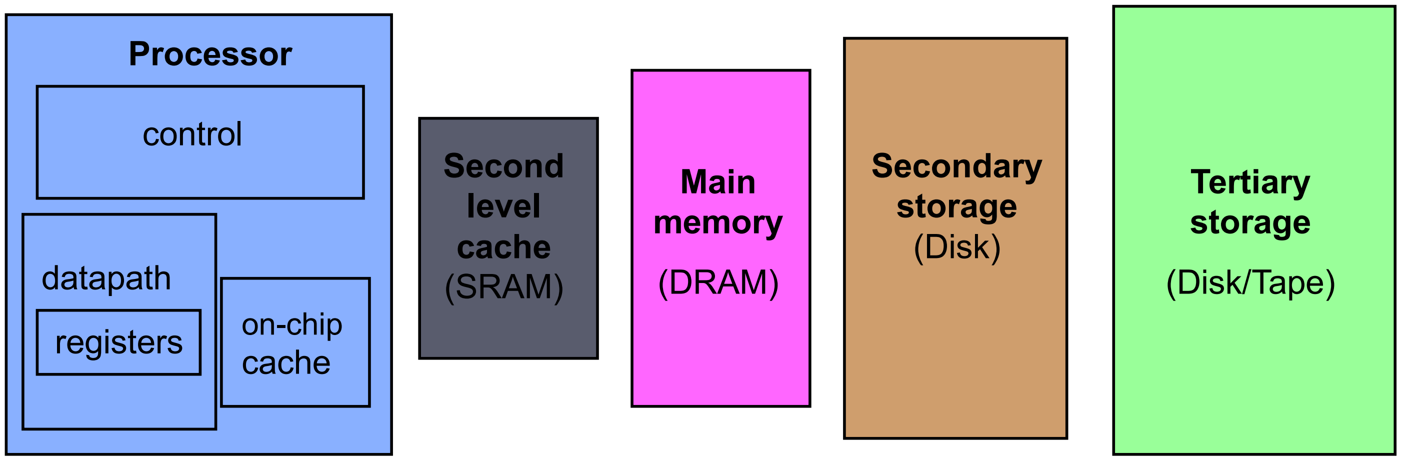
  - The hardware executes the instructions in the order specified by the compiler
- Idealized cost
  - Each operation has the same cost (reading, writing, adding, multiplying, etc.)

Real sequential processors

- **Real processors have**
  - Registers and caches
    - Quick and small data areas
    - Store recently used data or close
    - Different memory operations can have very different costs
  - Parallelism
    - Several functional units that run in parallel
    - Different orders, mixes of instructions with different costs
  - Pipelines
- **Why is this our problem?**
  - In theory, compilers and hardware understand the architecture and how it works to optimize the program
  - In practice, this is false
  - They do not know the algorithm that will derive greater benefit from the processor

Memory hierarchy

- Most programs have a strong locality in their access to data
  - **Spatial locality:** access data close to previous accesses
  - **Time locality:** re-use data that was previously accessed
- Memory hierarchies try to exploit locality to improve performance



Speed	1ns	10ns	100ns	10ms	10sec
Size	Ko	Mo	Go	To	Po

Approaches to support memory latency

- Bandwidth increased faster than latency decreased
  - 23% per year vs. 7% per year
- Some techniques
  - Eliminate operations by saving data in small (but fast, caches) memories and re-using them
    - Adding some **time locality** in the programs
  - Use bandwidth better by retrieving a piece of memory and saving it to a cache and using the entire block
    - Adding some **spatial locality** in programs
  - Use bandwidth better by allowing the processor to perform multiple reads at the same time
    - Competition in instruction flow (loading an entire array in vector processors, prefetching)
  - Computation/memory operation overlap
    - Prefetching

Caches bases

- **Cache**: fast (and expensive) memory that keeps copies of data in main memory (hidden management to the user)
  - Simple example: a data item at address xxxxx1101 is stored in the cache at address 1101
- **Cache hit:** access to data in cache, inexpensive
- **Cache miss:** access to a data item not in cache, expensive
  - Need to access the upper level (slower)
- Length of a cache line: number of bytes loaded at the same time
- **Associativity**
  - **Direct-mapping:** only one address (line)
    - The data stored at the address xxxxx1101 is stored at address 1101 of the cache, in a 16-word cache
  - **N-way:** $n \geq 2$ lines with different addresses can be stored
    - Up to $n \leq 16$ words with addresses xxxxx1101 can be stored at address 1101 of the cache (cache can store 16n words)
- Hierarchical caches with decreasing speeds and increasing sizes
  - In processors and outside

Why having multiple cache levels?

- **On-chip vs off-chip**
  - Internal caches are faster but limited in size

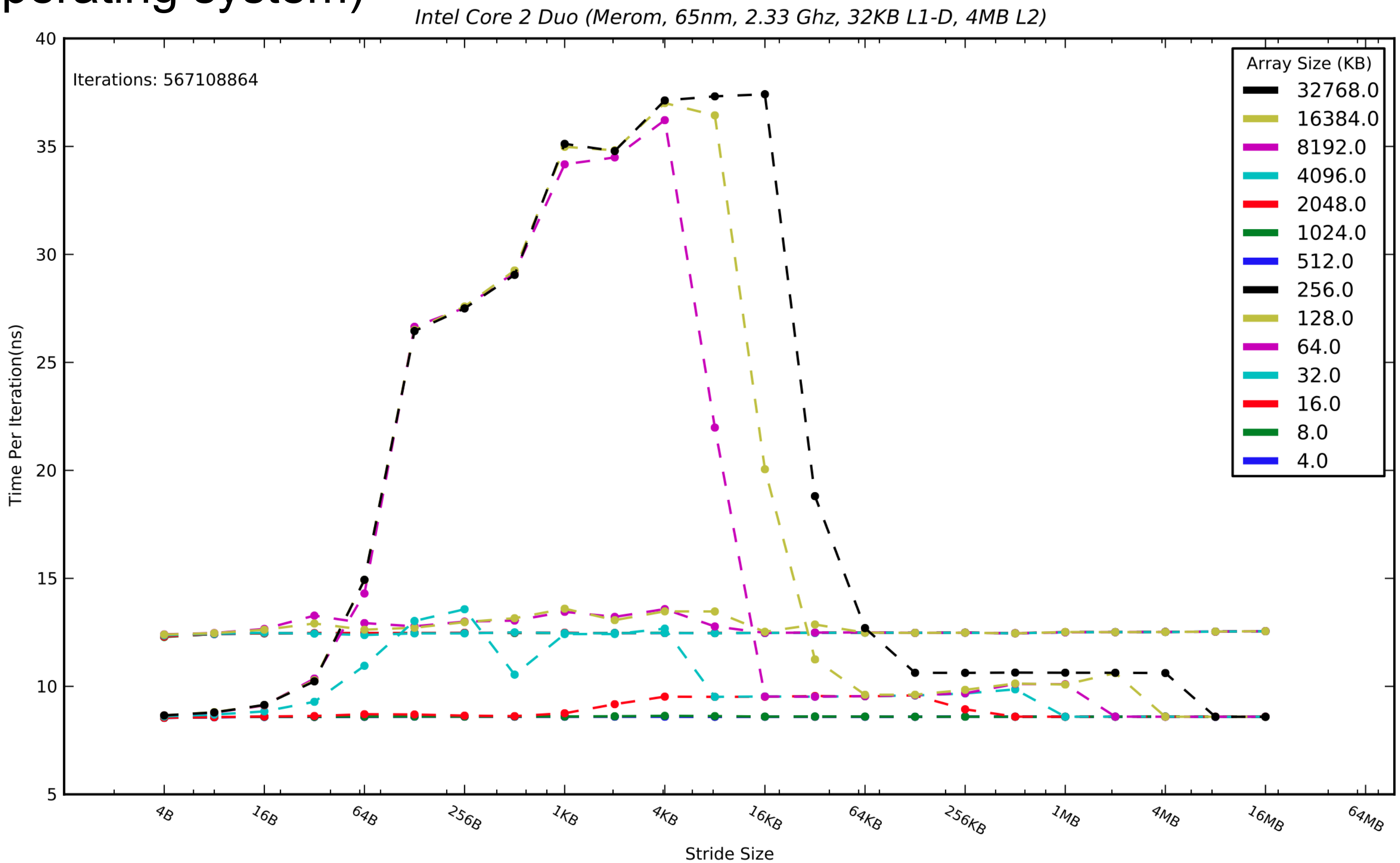
- **Large size caches are slower**
  - Hardware takes longer to check for longer addresses
  - Associativity, which allows for larger sets of data, has a cost

- Some examples
  - Cray deleted a cache to speed up certain accesses on the T3E
  - IBM (Power 5 and 6) uses a cache called "victim cache" that is less expensive

- There are other levels of memory hierarchy
  - Registers, pages (TLB, virtual memory), …
  - And not always hierarchical

Cache modeling

- **Microbenchmarks available (membench, stanza triad)**
  - Works well enough for simple and non-hierarchical caches
  - Complicated for new generations of processors (not to mention the influence of the operating system)



What lessons can be learned?

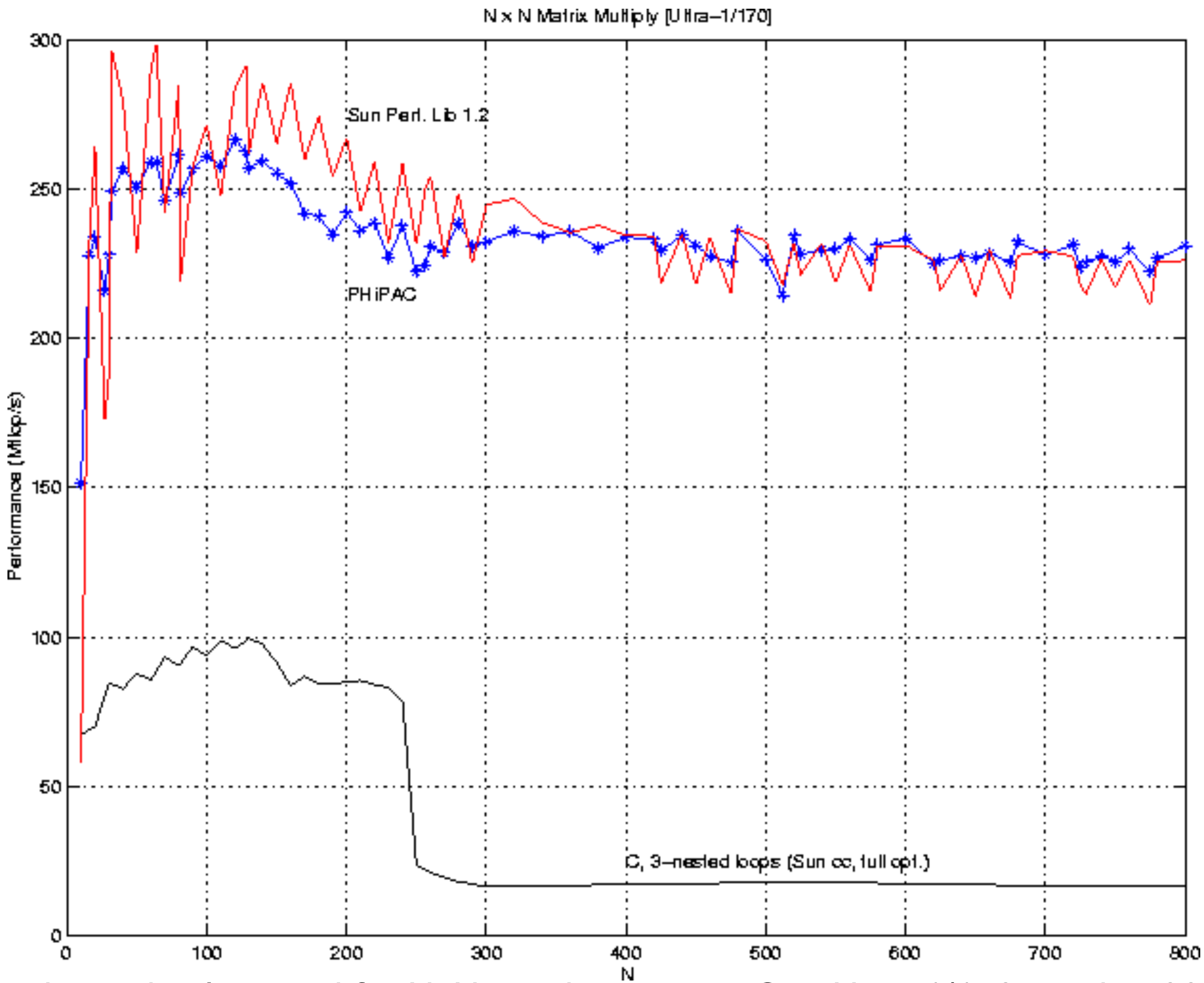
- **The actual performance of a program may be difficult to understand depending on the architecture**
  - The slightest modification of the architecture and the program has a big influence on the performance
  - To write efficient programs, one must take into account the architecture
    - True for sequential and parallel processors
  - One would like simple models to design efficient algorithms

- **Consider caches in the program**
  - Use a divide-and-conquer algorithm so that the data is ideally placed in the L1 and L2 caches

Matrix product

- **An important kernel of several numerical applications**
  - Appears in most linear algebra algorithms
    - Bottleneck of many applications
  - Other applications: graph algorithms, neural networks, image processing, ...
- Optimizations can be used for other applications
- Quite easy to optimize
- The most studied algorithm in the world of HPC

Impact of these optimizations



Matrix product's speed for NxN matrices over a Sun Ultra-1/170, peak = 330 MFlops

Data storage

- A matrix is a 2D array but the memory is rather 1D
- **Storage conventions**
  - By columns (by default in Fortran): A(i,j) à A+i+j*n
  - By rows (by default in C): A(i,j) at A+i*n+j
  - Recursive

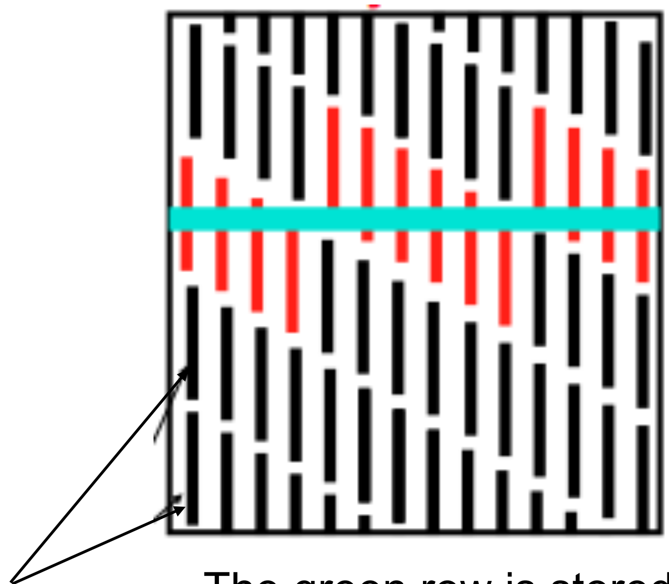
By columns

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

By rows

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Memory storage by column



Cache lines

The green row is stored in red cache lines

Simple memory model to optimize

- We assume **two levels of memory** in the hierarchy, fast and slow
- Initially all data is in the slow memory (main memory)
  - m : number of memory (words) exchanged between fast memory and slow memory
  - t_m: time per operations in slow memory
  - f : number of arithmetic operations
  - t_f : time per arithmetic operation << t_m
  - q : $\frac{f}{m}$ average number of flops per access to slow memory

Computational intensity: very important for algorithm efficiency

- Minimal execution time when all data are in fast memory:

$$f * t_f$$

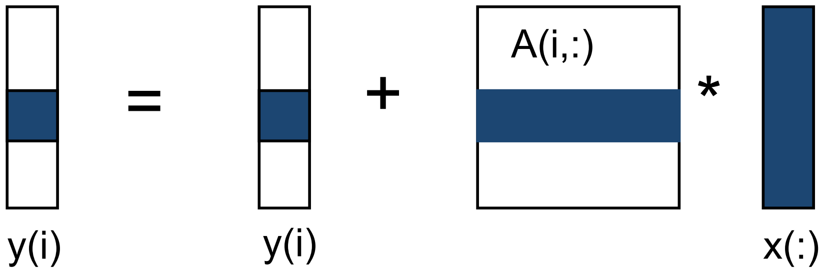
- Effective Time
  - $f * t_f + m * t_m = f * t_f * (1 + \boxed{t_m/t_f} * 1/q)$

Balance of the machine: key for the efficiency of the computer

- A larger q leads to a time closer to the minimum $f * t_f$
  - $q \geq t_m/t_f$ mandatory to obtain at least one half of the peak performance

Matrix-vector product

```
// y = y + A*x
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



Matrix-vector product

```
read x(1:n) in fast memory
read y(1:n) in fast memory
for i = 1:n
      read line i of A in fast memory
      for j = 1:n
            y(i) = y(i) + A(i,j)*x(j)
write y(1:n) in main memory
```

- m = number of references to slow memory = $3n + n^2$
- f = number of arithmetic operations = $2n^2$
- $q = f / m \approx 2$

Matrix-vector multiplication limited by the speed of the main memory

Modeling the matrix-vector product

- Execution time for a NxN = 1000x1000 matrix
- Time

$$f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$$
$$= 2*n^2 * t_f * (1 + t_m/t_f * 1/2)$$

- For t_f and t_m, data from the R. Vuduc PhD thesis (pp 351-3)
  - For t_m we use the min memory latency min / word per cache line

	Clock	Peak	Mem Lat (Min,Max)		Linesize	t_m/t_f
	MHz	Mflop/s	cycles		Bytes	
Ultra 2i	333	667	38	66	16	24.8
Ultra 3	900	1800	28	200	32	14.0
Pentium 3	500	500	25	60	32	6.3
Pentium3N	800	800	40	60	32	10.0
Power3	375	1500	35	139	128	8.8
Power4	1300	5200	60	10000	128	15.0
Itanium1	800	3200	36	85	32	36.0
Itanium2	900	3600	11	60	64	5.5

Machine balance (q has to be at least equal to this value to obtain ½ of peak performance)

http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

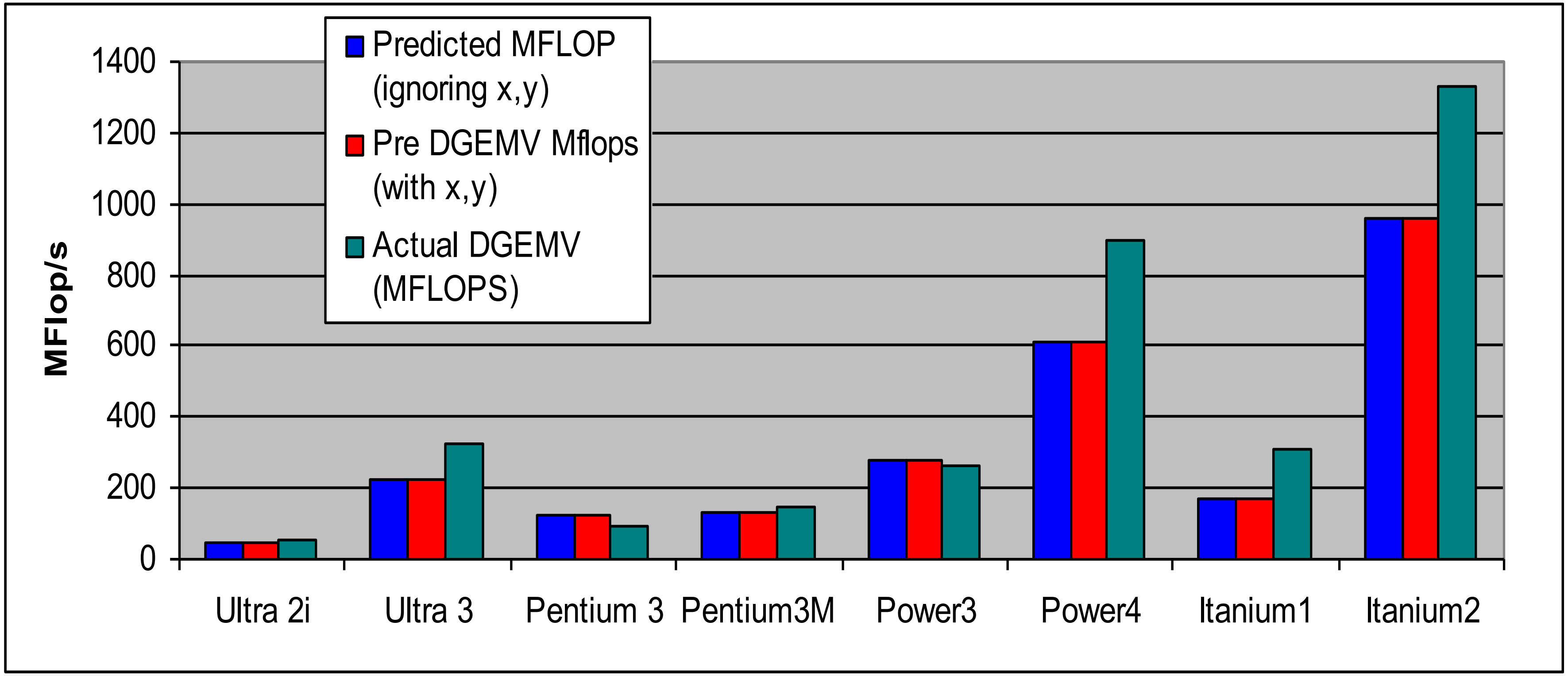
Simplifying assumptions

- **What simplifications have been made?**
  - We have ignored the parallelism in the processor between memory access and arithmetic operations
    - Sometimes we forget the arithmetic terms in this type of analysis
  - It was assumed that the fast memory could store 3 vectors
    - Reasonable if we talk about memory caches
    - False if one has registers (~ 32 words)
  - It was assumed that the cost of a fast memory access was 0
    - Reasonable when using registers
    - Not really just if you use a memory cache (1-2 cycles for L1)
  - Memory latency was assumed to be constant

- We can simplify even more by ignoring the memory operations in the vectors X and Y
  - Speed in Mflops /element = $2 / (2^* t_f + t_m)$

Model validation

- Can this model be used to predict performances ?
  - DGEMV operation: optimized for target platforms
- Model accurate enough to compare machines
- But not accurate enough for the latest processors because of latency estimates

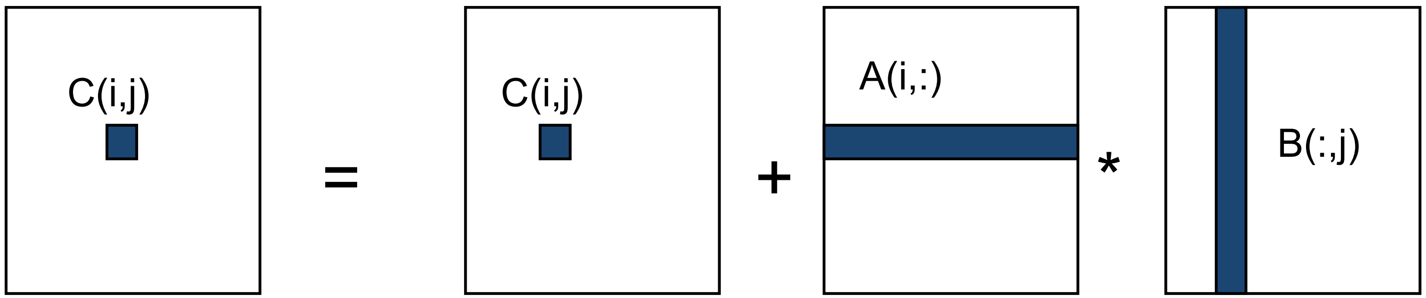


Naive matrix multiplication

```
// implements C = C + A*B
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

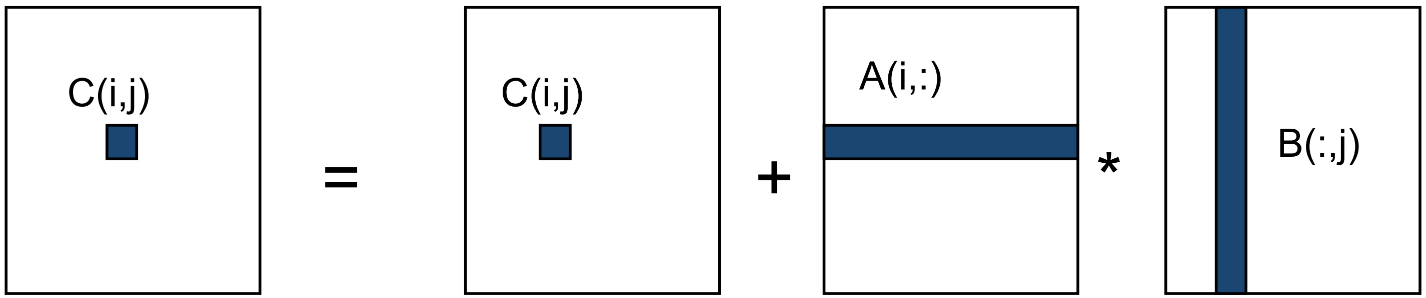
The algorithm has $2*n^3 = O(n^3)$ Flops and works on $3*n^2$ memory words

q potentially big as $2*n^3 / 3*n^2 = O(n)$



Naive matrix multiplication, contd.

```
// implements C = C + A*B
for i = 1 to n
   read row I of A in fast memory
   for j = 1 to n
      read C(i,j) in fast memory
      read colum j of B in fast memory
      for k = 1 to n
         C(i,j) = C(i,j) + A(i,k) * B(k,j)
      write C(i,j) in main memory
```



Naive matrix multiplication, contd.

Number of slow memory references for a multiplication of matrices without blocks

$m = n^3$ to read each column of B n times

$+ n^2$ to read each row of A once

$+ 2n^2$ to read and write each element of C once

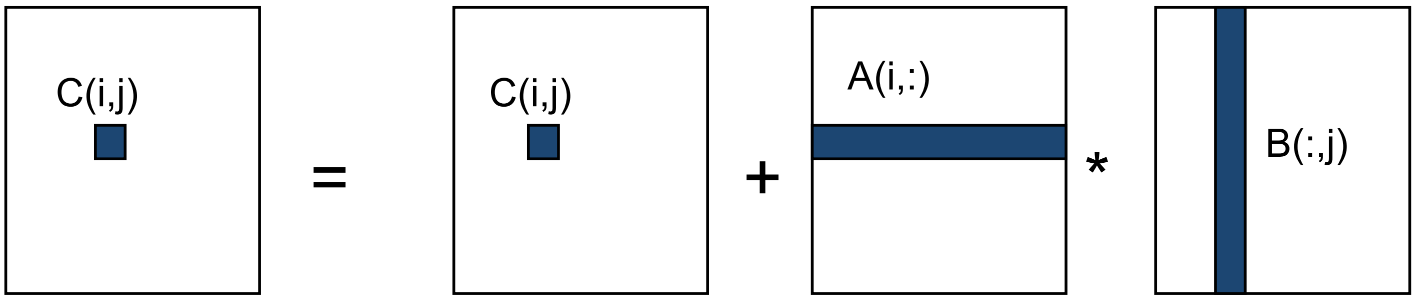
$= n^3 + 3n^2$

Then $q = f / m = 2n^3 / (n^3 + 3n^2)$

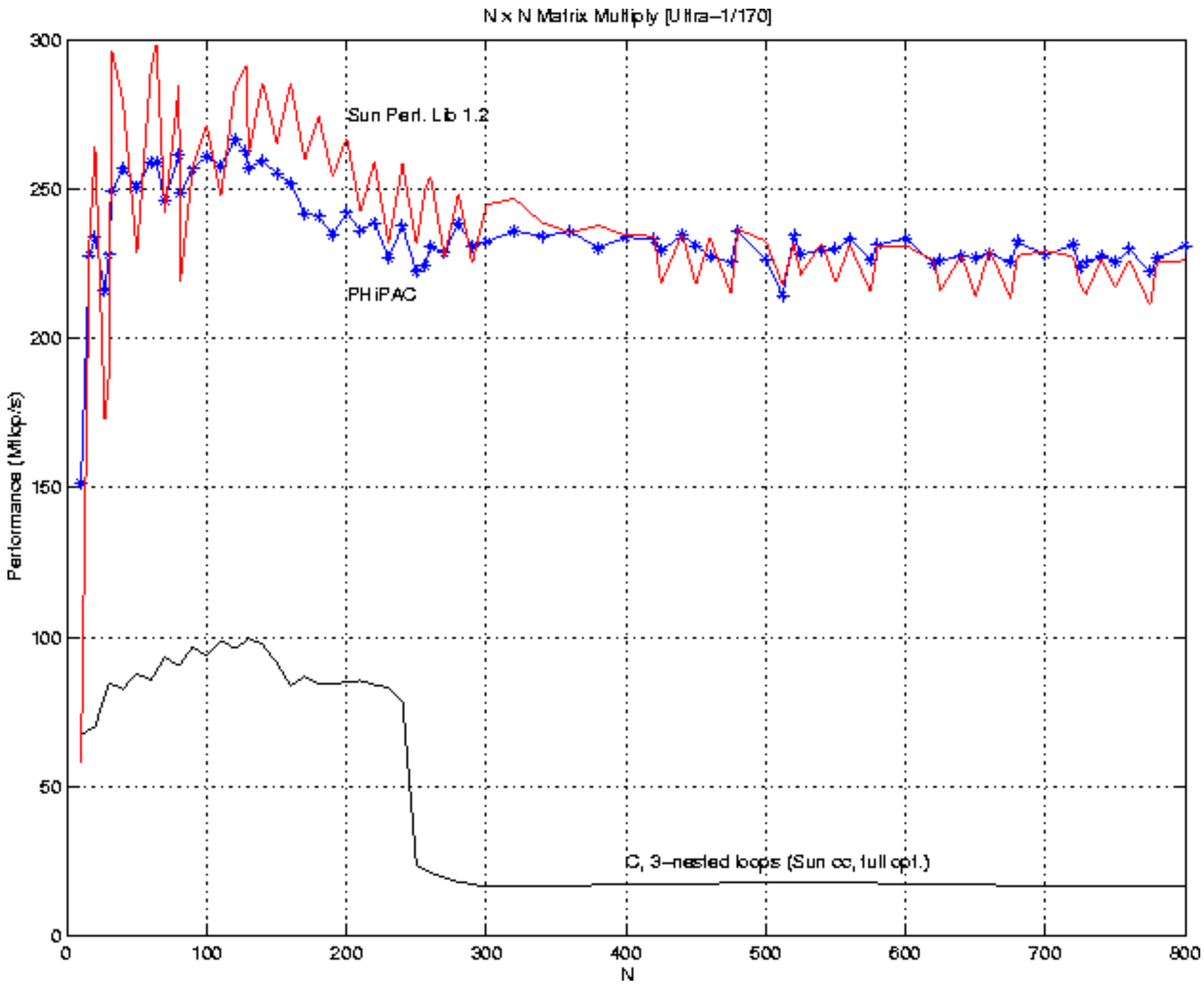
≈ 2 when n is big, no improvement compared to the matrix-vector product

The two internal loops are matrix-vector products of line i of A times B

Same if we exchange the 3 loops

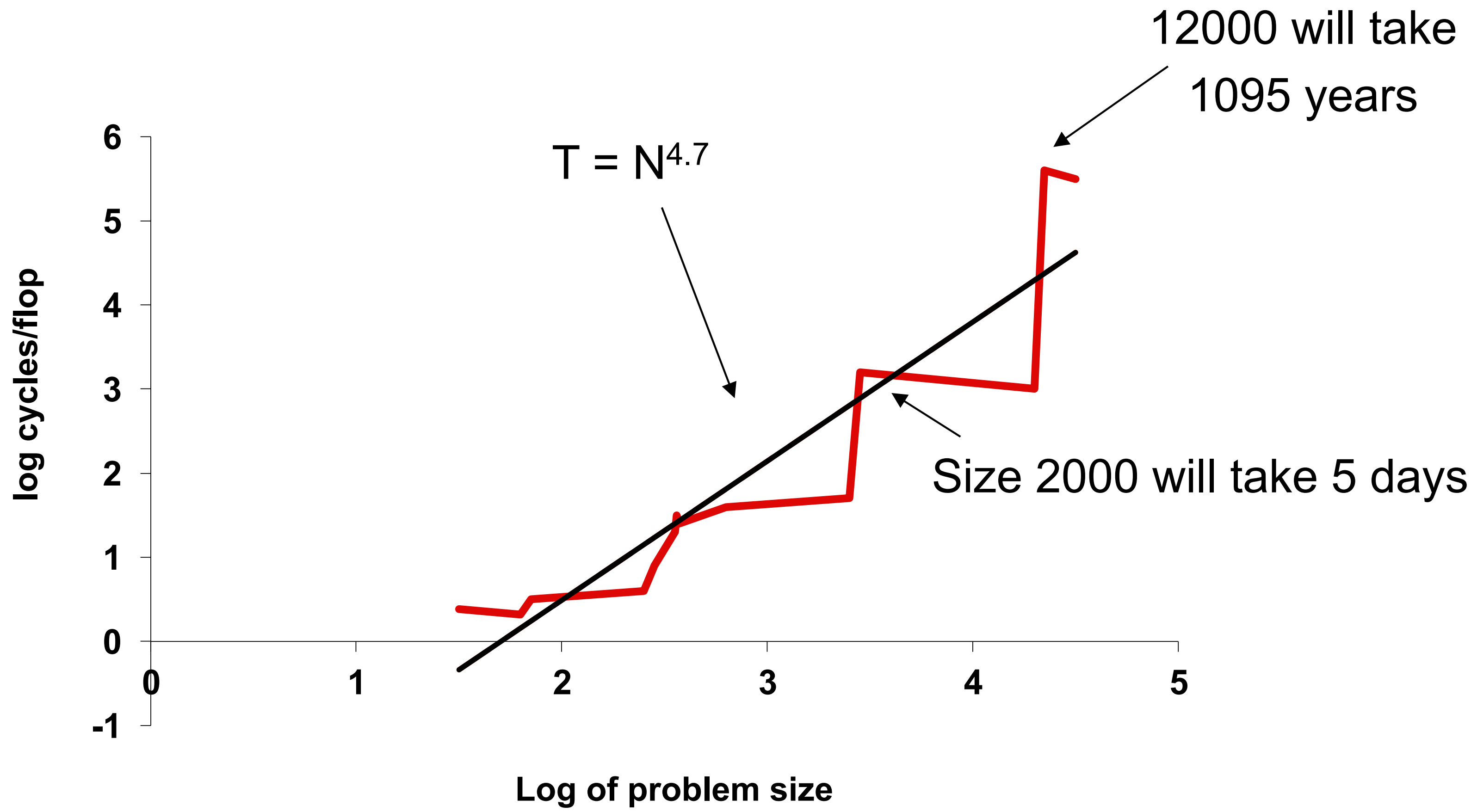


Performances of the naive matrix multiplication



Speed of a NxN matrix product on a Sun Ultra-1/170, peak performance = 330 MFlops

Naive matrix multiplication on RS/6000

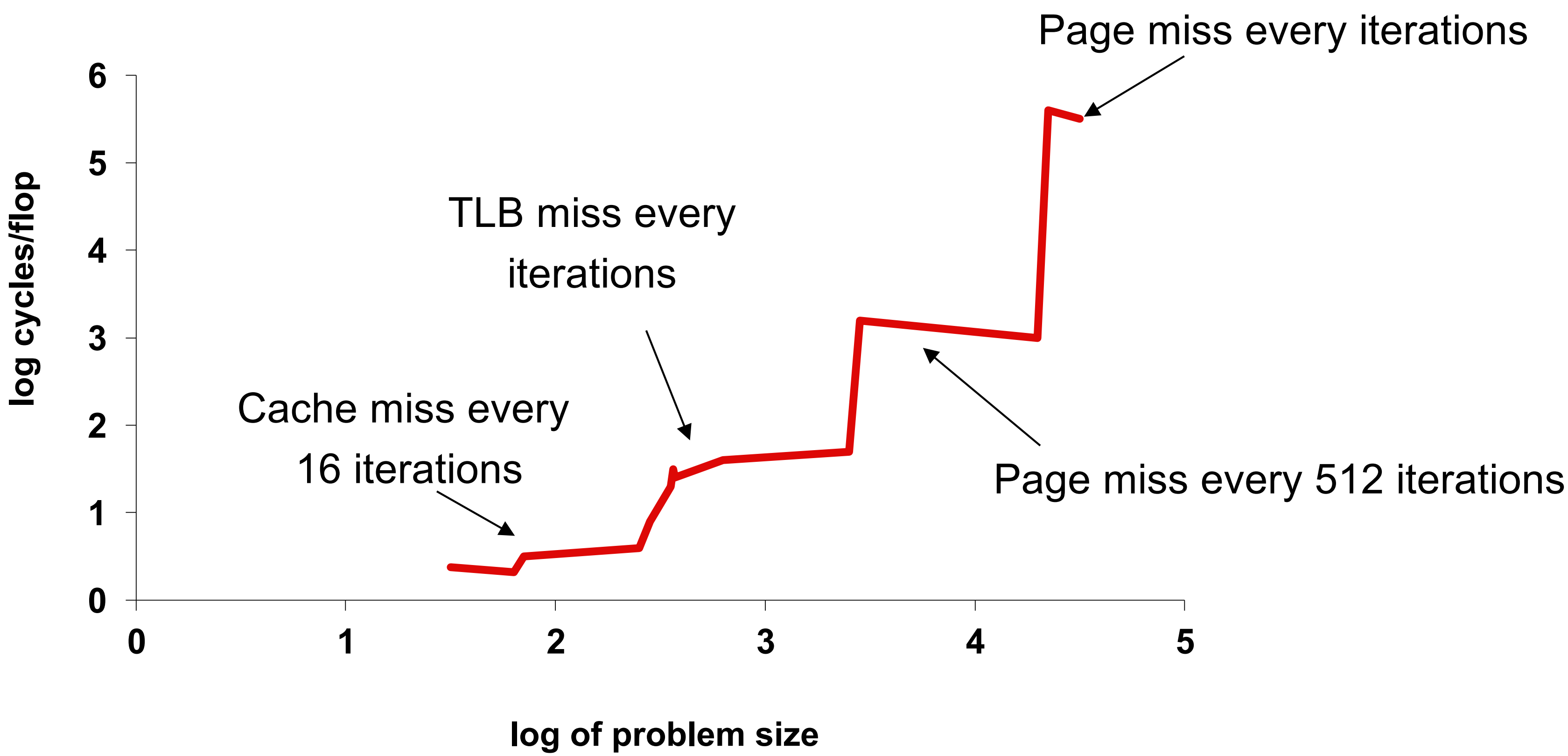


Performances in $O(N^3)$ with a number of cycle/flop constant

Performance is more like $O(N^{4.7})$

Slide source: Larry Carter, UCSD

Naive matrix multiplication on RS/6000, contd.

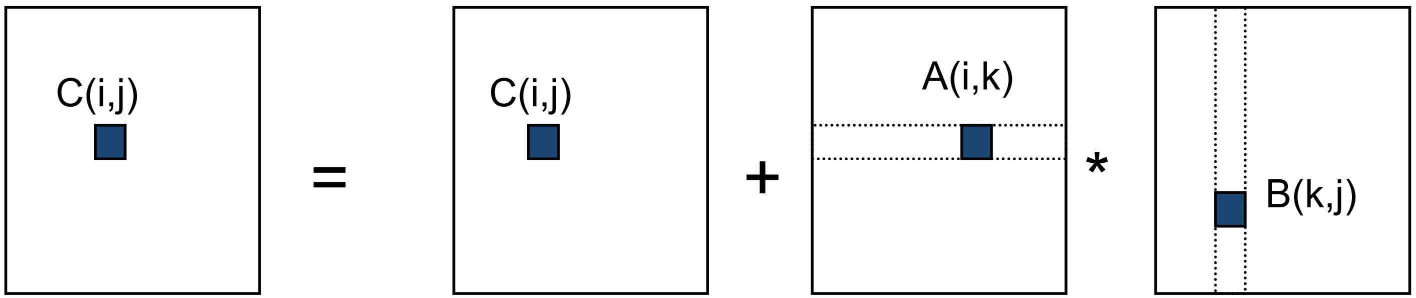


Slide source: Larry Carter, UCSD

Block partitioned matrix product

A,B,C matrices of size nxn partionned into sub-blocks of size bxb where b=n / N is called the **block size**

```
for i = 1 to N
  for j = 1 to N
     read block C(i,j) in fast memory
     for k = 1 to N
          read block A(i,k) in fast memory
          read block B(k,j) in fast memory
           C(i,j) = C(i,j) + A(i,k) * B(k,j) {matrix product over blocks}
     write block C(i,j) in main memory
```



Block partitioned matrix product, contd

Recall:

m is the volume of memory traffic between the fast memory and the slow (main) memory

The matrix has nxn elements, and NxN blocks of size bxb

f is the number of floating operations, $2n^3$ for the matrix product

q = f / m is our measure of algorithm efficiency in the memory system

Then:

- $m = N*n^2$ read each block of B N^3 times ($N^3 * b^2 = N^3 * (n/N)^2 = N*n^2$)
- $+ N*n^2$ read each block of A N^3 times
- $+ 2n^2$ read and write each block of C once
- $= (2N + 2) * n^2$

- Then the computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$
  $\approx n / N = b$ for large n
- The performance can be improved by increasing the size of the blocks b
- Can be much faster than the matrix-vector product (q = 2)

Analysis to understand algorithms

- The block algorithm has a computational intensity $q \approx b$
- The larger the block size, the more efficient our algorithm
- **Limit**: All blocks of A, B, and C must be able to hold in fast memory (cache)
- If we assume that our fast memory has a size M_{fast}

$$3b^2 \leq M_{fast}, \quad \text{then } q \approx b \leq (M_{fast}/3)^{1/2}$$

- To construct a machine for executing a matrix product at ½ of the peak performance, we need a fast memory of size

$$M_{fast} \geq 3b^2 \approx 3q^2 = 3(t_m/t_f)^2$$

- This size is reasonable for an L1 cache but not for register sets
- **Note:** this analysis assumes that it is possible to schedule the instructions perfectly

		required
	t_m/t_f	**KB**
Ultra 2i	24.8	14.8
Ultra 3	14	4.7
Pentium 3	6.25	0.9
Pentium3M	10	2.4
Power3	8.75	1.8
Power4	15	5.4
Itanium1	36	31.1
Itanium2	5.5	0.7

Limitations on optimization of the matrix product

- The block algorithm changes the order in which values are accumulated on each C [I, j] by applying commutativity and associativity
- The previous analysis has shown that the block algorithm has a computational intensity of

$$q \approx b \leq (M_{fast}/3)^{1/2}$$

- There is a lower bound which says that one can not do better than this bound (using only associativity)

- Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (which uses only associativity) is limited to

$$q = O(\ (M_{fast})^{1/2}\ )$$

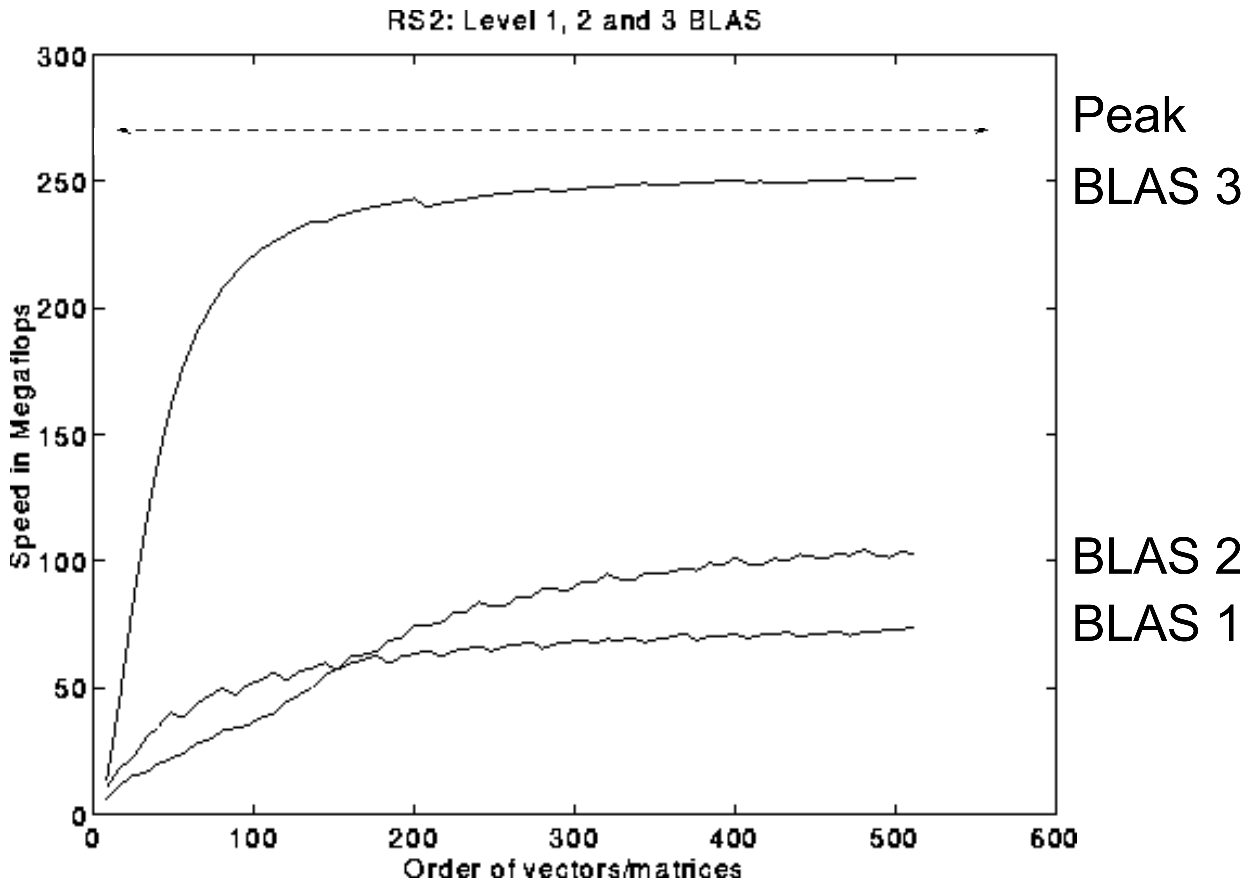
I/O complexity: The red-blue pebble game, J.W. Hong, H.T. Kung, STOC, ACM Press, pp: 326-333, (1981)

Basic Linear Algebra (BLAS)

- Standard interface
  - www.netlib.org/blas, www.netlib.org/blas/blast--forum
- Machine vendors, optimized implementations
- History
  - BLAS1 (1970's):
    - Vector operations: dot product, saxpy ($y=\alpha*x+y$), etc
    - m=2*n, f=2*n, q ~1 or less
  - BLAS2 (middle of 1980's)
    - Matrix-vector operations: matrix-vector product, etc
    - m=n^2, f=2*n^2, q~2, less overhead
    - Faster than BLAS1
  - BLAS3 (en of 1980's)
    - Matrix-matrux operations: product, etc
    - m <= 3n^2, f=O(n^3), then q=f/m can reach n
      Then BLAS3 are potentially faster than BLAS2 operations
- Best algorithms use BLAS3 operations when possible (LAPACK & ScaLAPACK) `www.netlib.org/{lapack,scalapack}`

BLAS on IBM RS6000/590

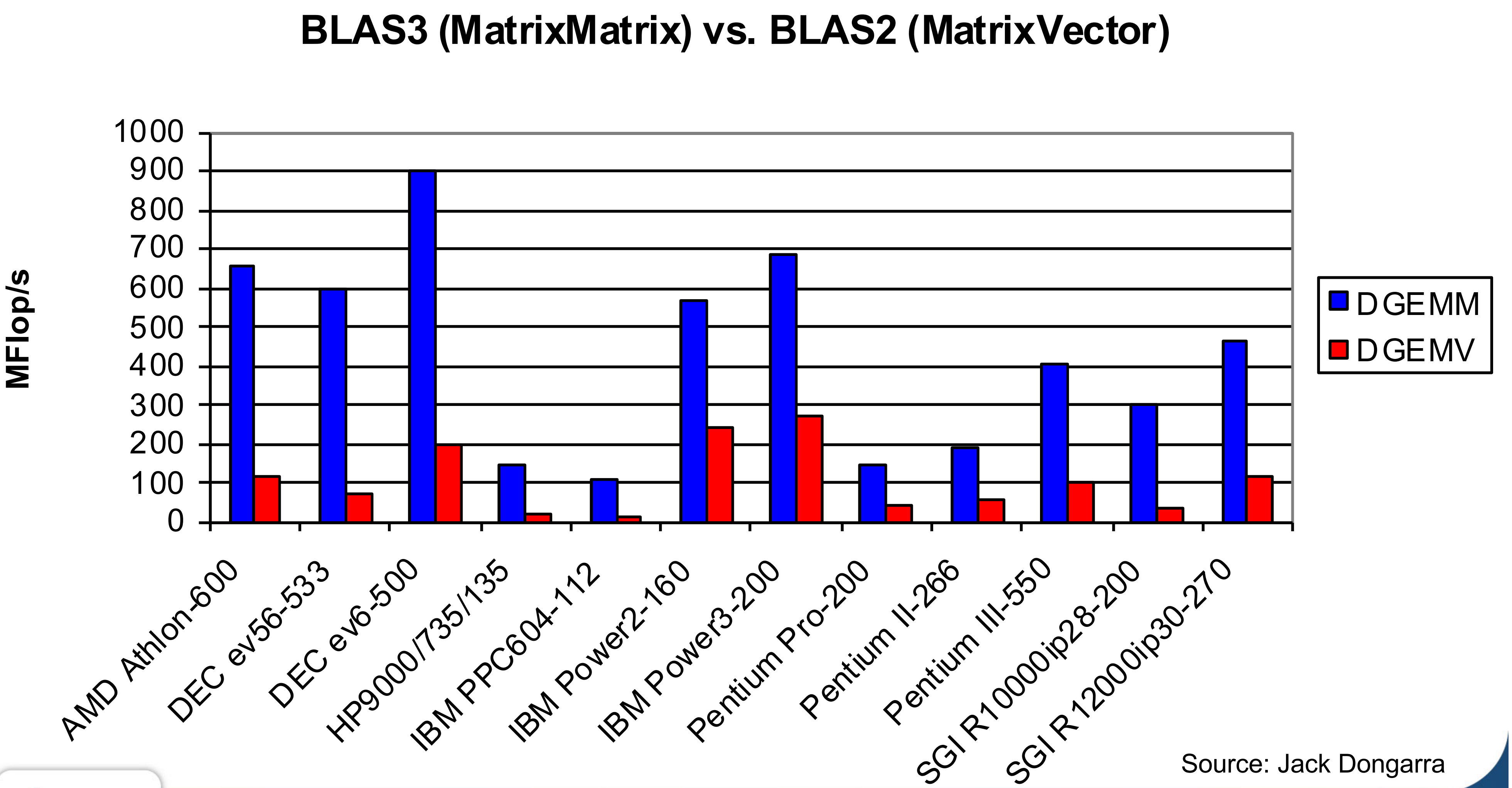
Peak performance = 266 Mflops



BLAS 3 (nxn matrix product) vs BLAS 2 (nxn matrix-vector product) vs BLAS 1 (saxpy of n vectors)

Dense linear algebra: BLAS2 vs BLAS3

- BLAS2 and BLAS3 have very different computational intensities and therefore very different performances



Source: Jack Dongarra

Recursion: cache oblivious algorithms

- The block algorithm requires finding a good block size
- The cache oblivious algorithms represent an alternative
- Treat a nxn matrix product as a set of smaller problems that may fit into the caches
- **Case for A (nxm) * B (mxp)**
  - **Case 1:** m>= max{n,p}: split A horizontally
  - **Case 2 :** n>= max{m,p}: split A vertically et B horizontally
  - **Case 3 :** p>= max{m,n}: split B vertically

$$\boxed{\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}}$$

Case 1

$$\boxed{(A_1, A_2) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = (A_1 B_1 + A_2 B_2)}$$

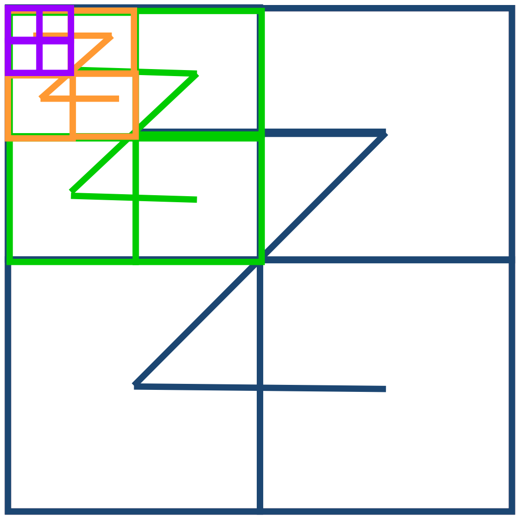
Case 2

$$\boxed{A(B_1, B_2) = (A\ B_1, A\ B_2)}$$

Case 3

Storage of recursive data

- A related idea is to use a recursive structure for the matrix
  - Improves the locality of the data with a data structure independent of the machine
  - Can minimize latency with multiple levels of memory hierarchy
- There are several recursive decompositions in the order of the subblocks
- The figure shows the Z-Morton arrangement ("space filling curve")
- Search for articles on "cache oblivious algorithms" and "recursive layouts"



Advantages

- Recursive storage works pretty well with all cache sizes

Drawbacks

- The calculation of the indexes to find A [i, j] is expensive

Automatic tuning of linear algebra kernel libraries

- **An ideal world**
  - Write numerical codes to Matlab and get performance close to peak performance

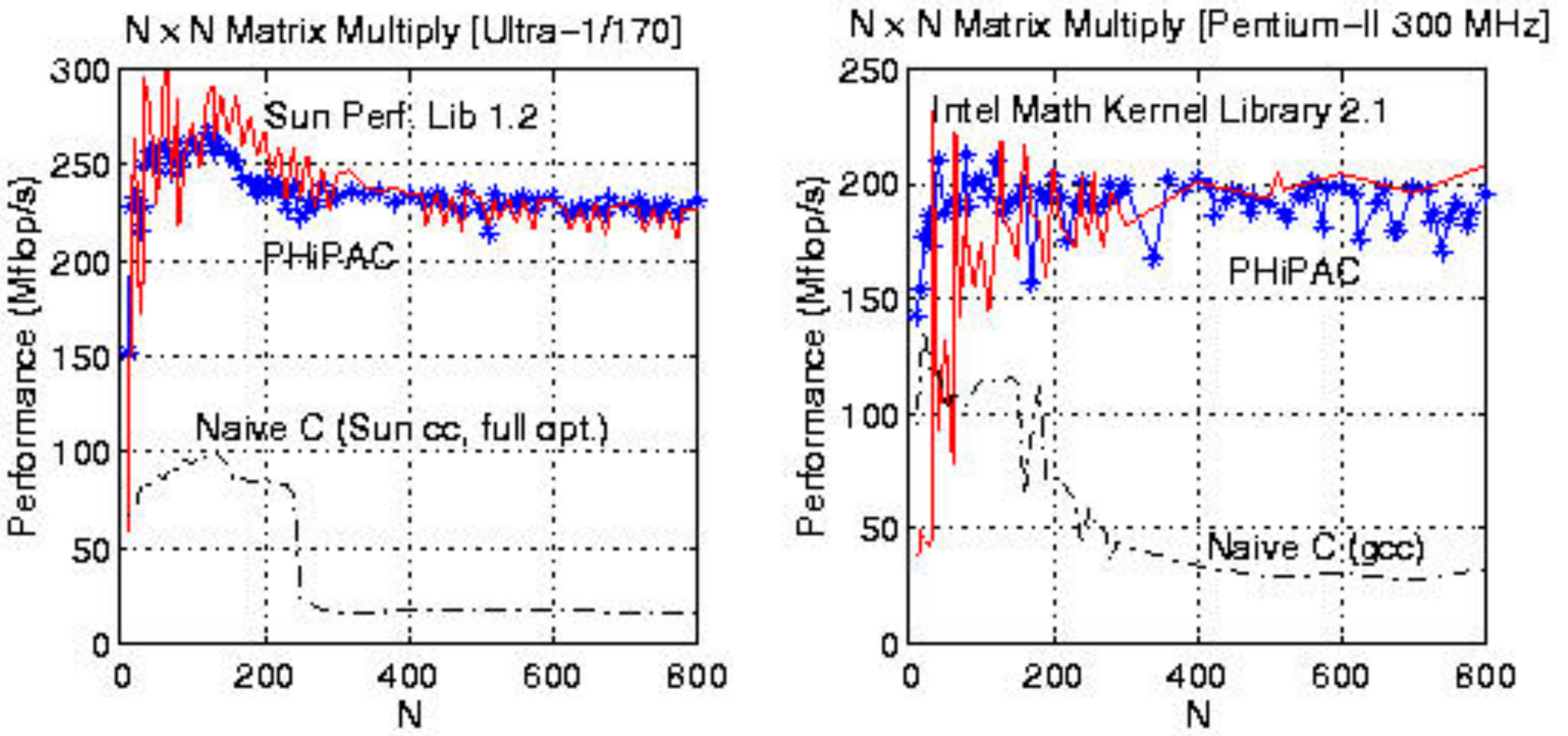
- **The Sad Reality**
  - The best algorithms depend on the target architectures and compilers used
  - Development "by hand" of codes optimized for a given application and for a given architecture
  - Difficult to understand and model the behavior of architectures and compilers

- Can we automate the generation of high-performance codes according to the target architectures?
  - A program is left to generate a large number of code variations and one takes the most efficient

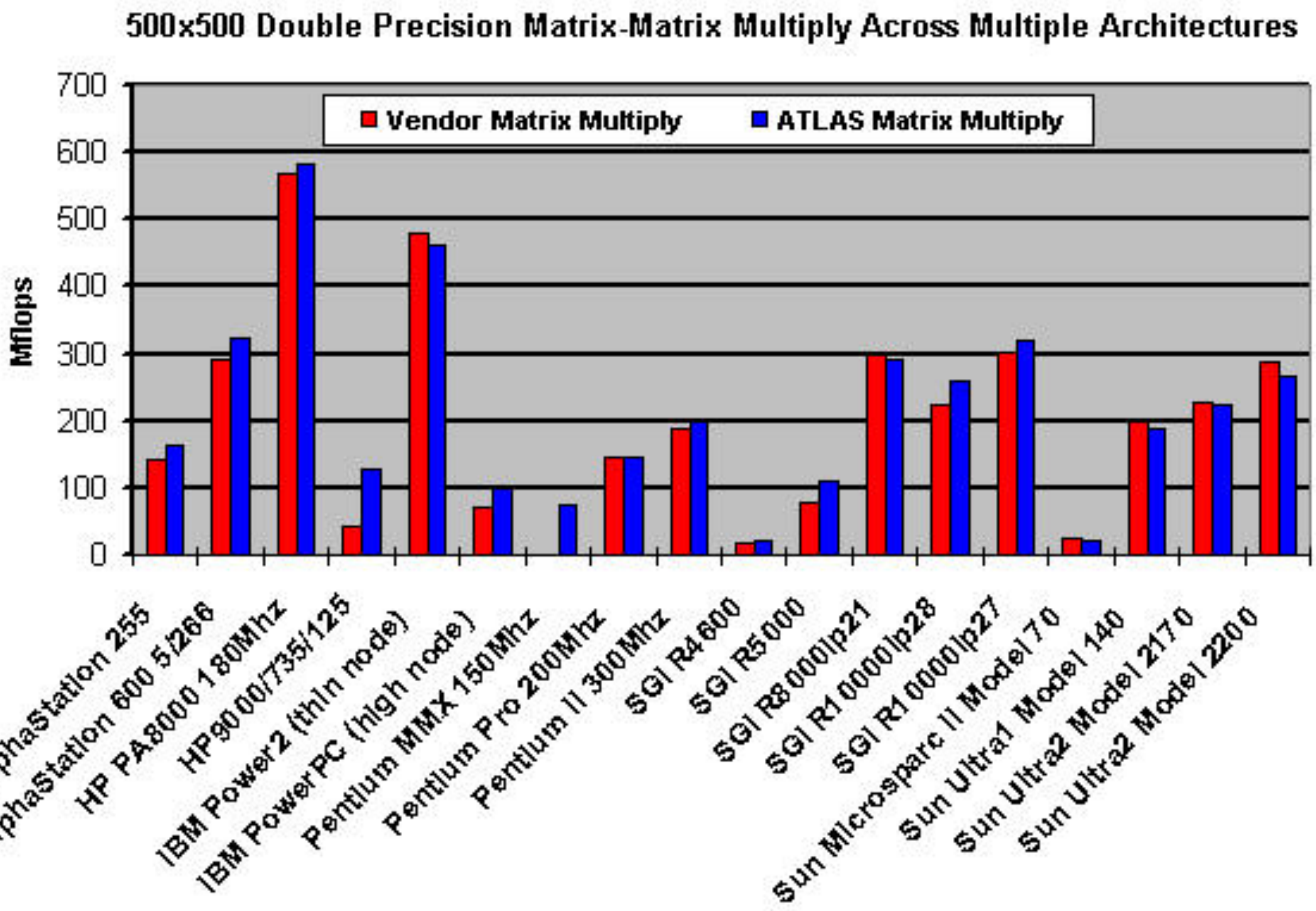
Some examples

- Dense BLAS
  - Sequential
  - PHiPAC (UCB), then ATLAS (UTK)
  - Now in Matlab, several other versions
  - math-atlas.sourceforge.net/
- Fast Fourier Transform (FFT) & its variations
  - FFTW (MIT)
  - Both sequential and parallel
  - 1999 Wilkinson Software Prize
  - www.fftw.org
- Digital Signal Processing
  - SPIRAL: www.spiral.net (CMU)
- Collective communication operations in MPI (UCB, UTK)

PHiPAC (Berkeley)



ATLAS (UTK)

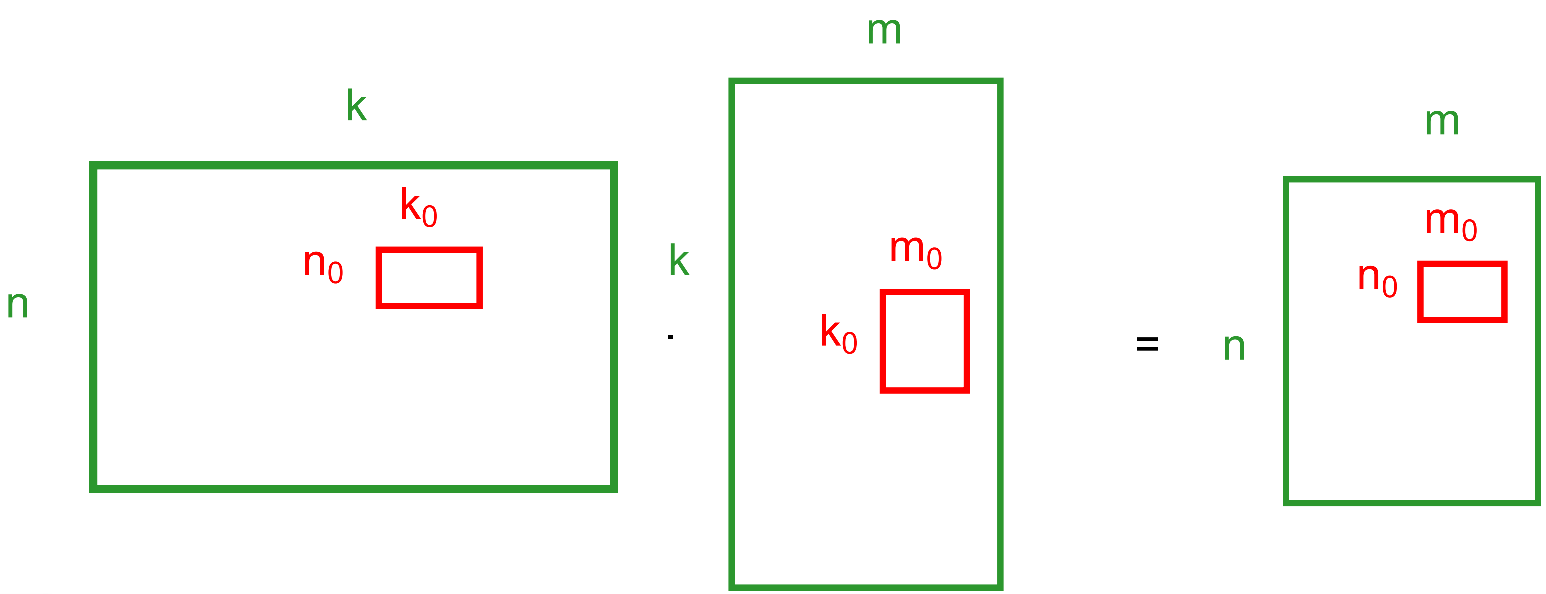


500x500 Double Precision Matrix-Matrix Multiply Across Multiple Architectures
Vendor Matrix Multiply
ATLAS Matrix Multiply
Mflops
700
600
500
400
300
200
100
0
DEC AlphaStation 255
DEC AlphaStation 600 5/266
HP PA8000 180Mhz
HP9000/735/125
IBM Power2 (thin node)
IBM PowerPC (high node)
Pentium MMX 150Mhz
Pentium Pro 200Mhz
Pentium II 300Mhz
SGI R4600
SGI R5000
SGI R8000ip21
SGI R10000ip28
SGI R10000ip27
Sun Microsparc II Model 70
Sun Ultra1 Model 140
Sun Ultra2 Model 2170
Sun Ultra2 Model 2200

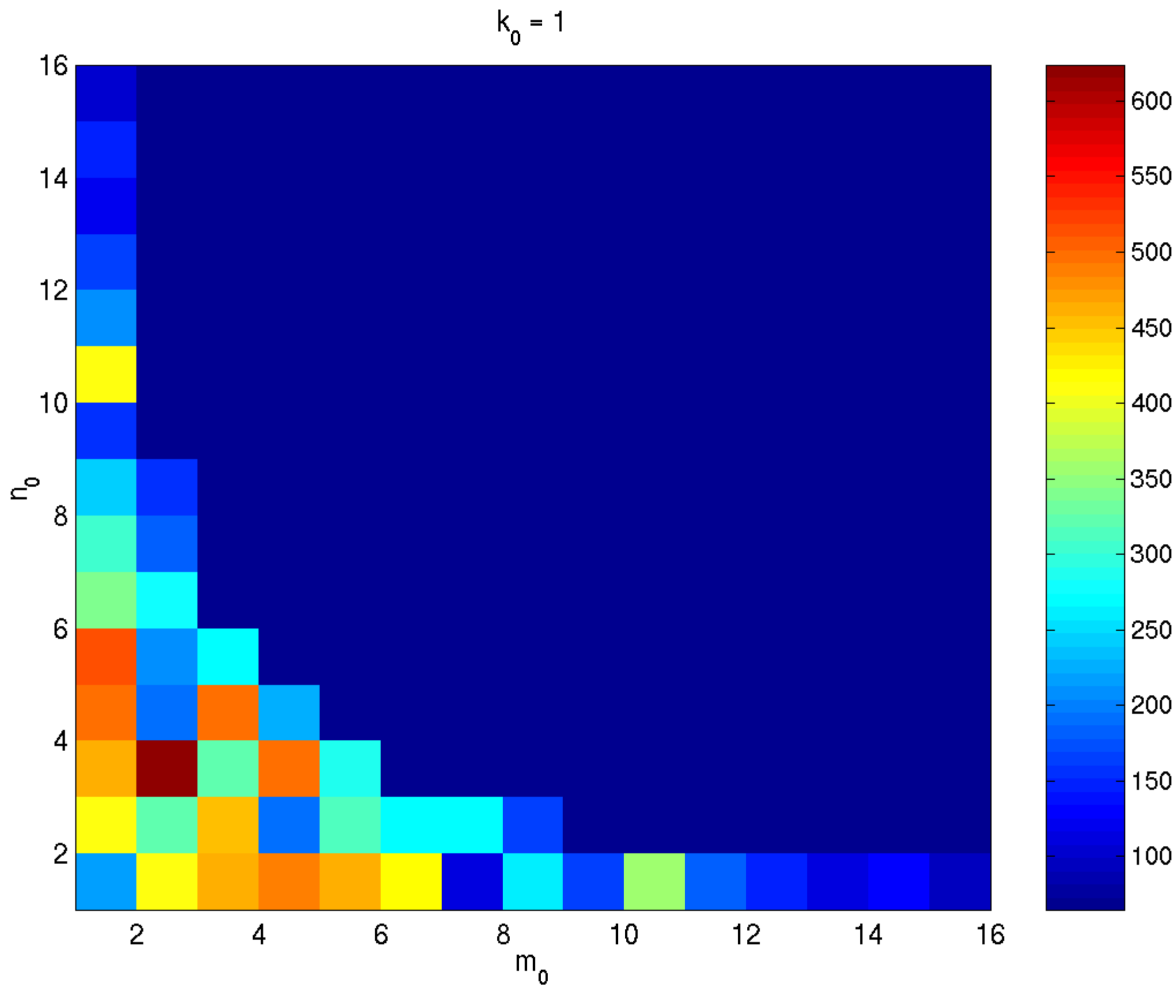
How does it work?

- What do BLAS, FFT, signal processing, reductions have in common?
  - One can perform the off-line tuning: once per architecture
  - We can take the time we want (hours, weeks ...)
  - At run time, the choice of algorithm depends only on a few parameters
    - Matrix sizes, FFT size, ...

- Computation of block size for registers in the matrix product



Computation of the block size for the matrix product



333 MHz Sun Ultra 2i

2-D section of a 3-D space;

Color-coded implementations of performance in Mflop/s

16 registers, but 2x3 tile faster

Optimizations

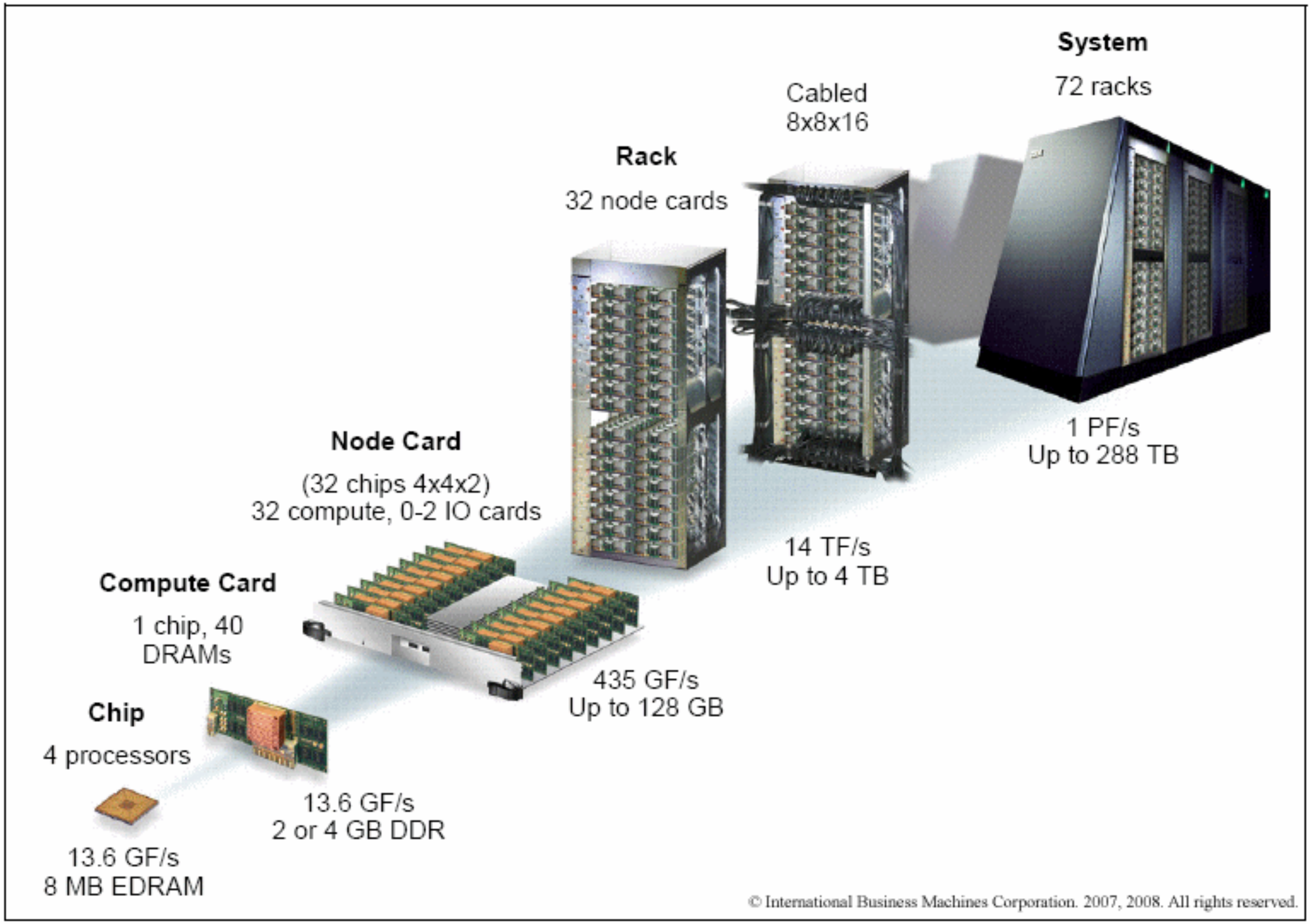
- Use block algorithms for registers
  - Loop unrolling, using named "registers" variables
- Use block algorithms for multi-level caches
- Harness the fine grain parallelism of processors
  - Superscalar instructions; pipelines
- Complex interactions with compilers

- **Several projects on the subject**
  - ParLab: `parlab.eecs.berkeley.edu`
  - BeBOP: `bebop.cs.berkeley.edu`
  - PHiPAC: `www.icsi.berkeley.edu/~bilmes/phipac`

    in particular `tr-98-035.ps.gz`
  - ATLAS: www.netlib.org/atlas
  - BOAST

CONCLUSIONS

Parallel machines today



https://computing.llnl.gov/tutorials/bgp/images/bgpScalingArch.gif

Getting performance on these machines

- In general, we only talk about computing power
- But it is also necessary to take into account the memory bandwidth and the latency
  - It is necessary to be able to fill the speed of the processor (s)
  - Hierarchical memory and caches
- The I / O bandwidth to the drives grows linearly with the number of processors

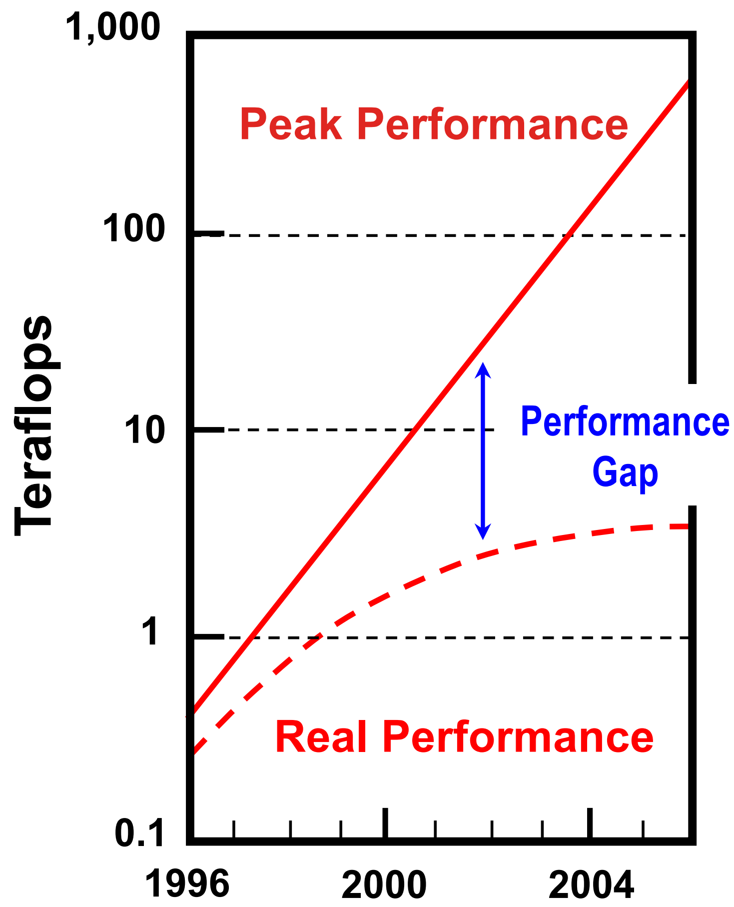
Improve real performance

Peak performance increases exponentially

- In 1990's, peak performance was increased 100x;
- In the years 2000, they will increase 1000x

But the efficiency (performance related to peak performance)

- 40-50% on vector supercomputers of the 1990s
- Now close to 5-10% on today's supercomputers



Reduce the gap ...

- Algorithms that obtain performance on a single processor and are extensible over several thousand
- More efficient programming models and tools for massively parallel machines

Parallelism in 2017

- All processor vendors produce multicore processors
  - All machines ~~will soon be~~ are parallel
  - To continue to double the power it is necessary to double the parallelism
- New processor architectures start to show up in HPC platforms
  - FPGA, low power consumption processors
- What applications will (well) benefit from parallelism?
  - Will they have to be redeveloped from scratch?
- Will all programmers have to be parallel machine programmers?
  - New software models are needed
  - Try to hide the parallelism to the maximum
  - Understand it!
- The industry is betting on these changes ...
- ... but still a lot of work to do

Challenges to be taken

- **Parallel applications are often very sophisticated**
  - Adaptive algorithms that require dynamic balancing
- **Multi-level parallelism is difficult to manage**
  - Massive use of task graph parallelism in modern numerical applications
- **The size of the new machines gives problems of efficiency**
  - Scalability Issues
  - Serialization and load imbalance
  - Bottlenecks in communications and/or input/output
  - Inefficient or no sufficient parallelization
  - Faults and/or breakdowns
  - Energy management
- **Difficulty in getting the best performance on the nodes themselves**
  - Contention for shared memory
  - Utilization of the memory hierarchy multicore processors
  - Influence of the operating system

Conclusions

- **The set of parallel machines consists of a (very) large set of elements**
  - from parallel units in processors
  - up to data centers connected worldwide
- **Field of study of parallelism**
  - Architectures
  - Algorithms
  - Software, compilers,
  - Libraries
  - Environments
- **Significant historical changes**
  - Until the 1990s, reserved for large simulation calculations
  - Today, parallelism in all processors (from smartphones to supercomputers), parallelism in everyday life (iPad, Google!)

Inria
INVENTEURS DU MONDE NUMÉRIQUE