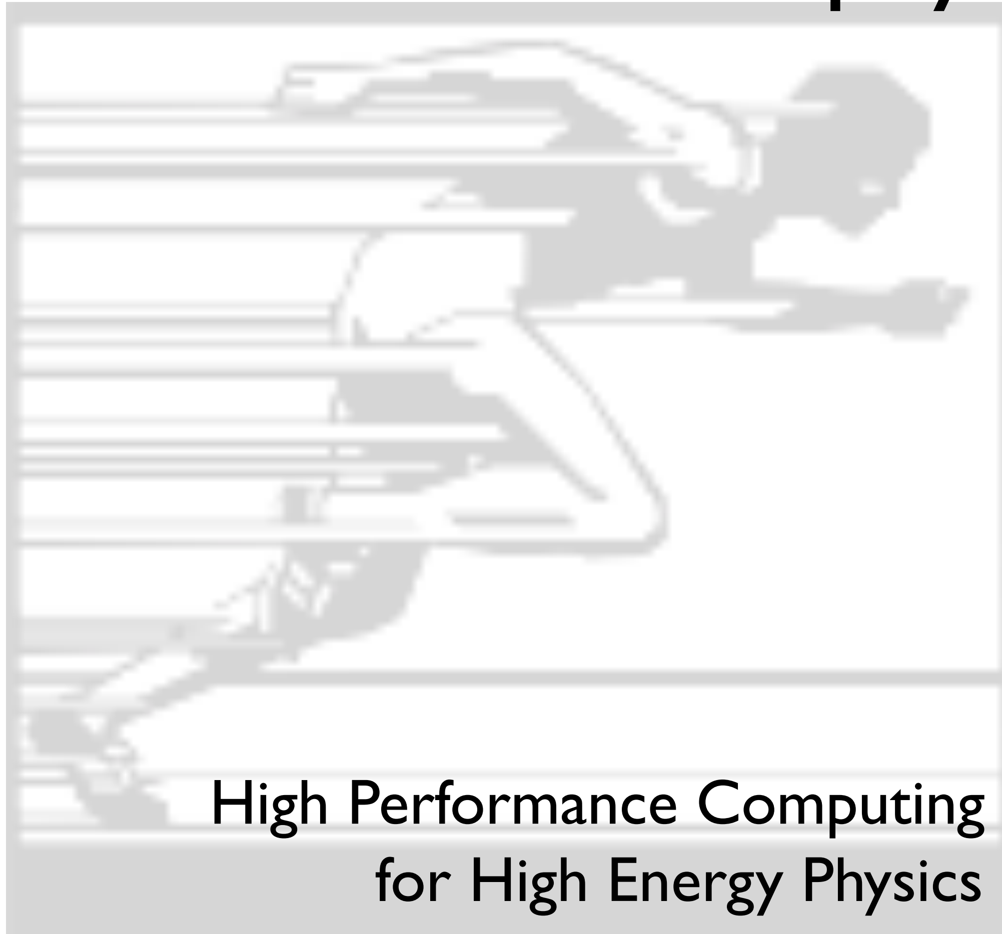
The challenge of adapting HEP physics-software to run on many-core cpus

CERN/TH, July `10

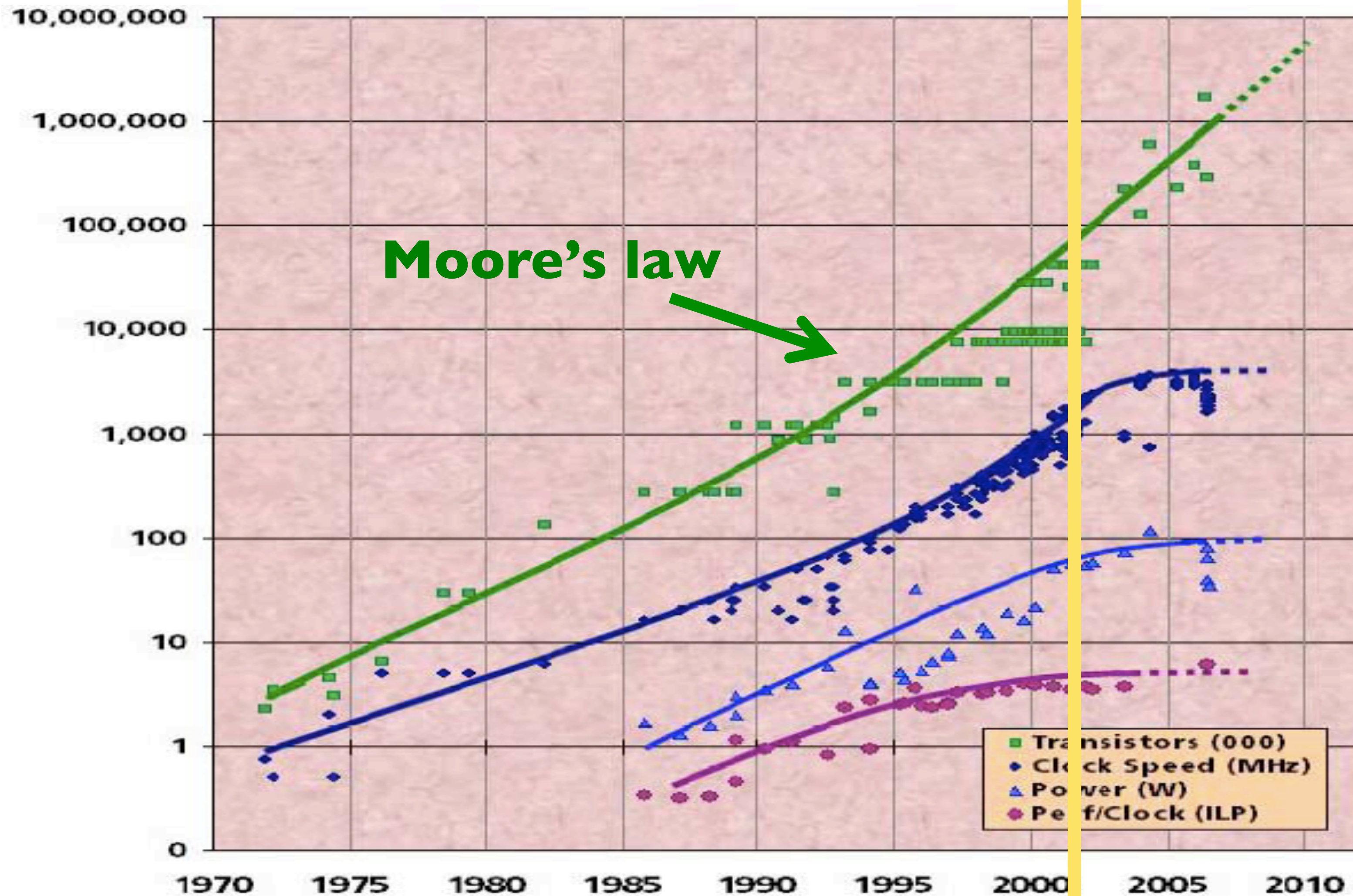


Vincenzo Innocente
CERN PH/SFT

MOTIVATIONS

Computing in the years Zero

Transistors used to increase *raw-power* | Increase *global power*



Consequence of the Moore's Law

Hardware continues to follow Moore's law

- More and more transistors available for computation
  - More (and more complex) execution units: hundreds of new instructions
  - Longer SIMD (Single Instruction Multiple Data) vectors
  - More hardware threading
  - More and more cores

The ‘three walls’

While hardware continued to follow Moore’s law, the perceived exponential grow of the “effective” computing power faded away in hitting three “walls”:

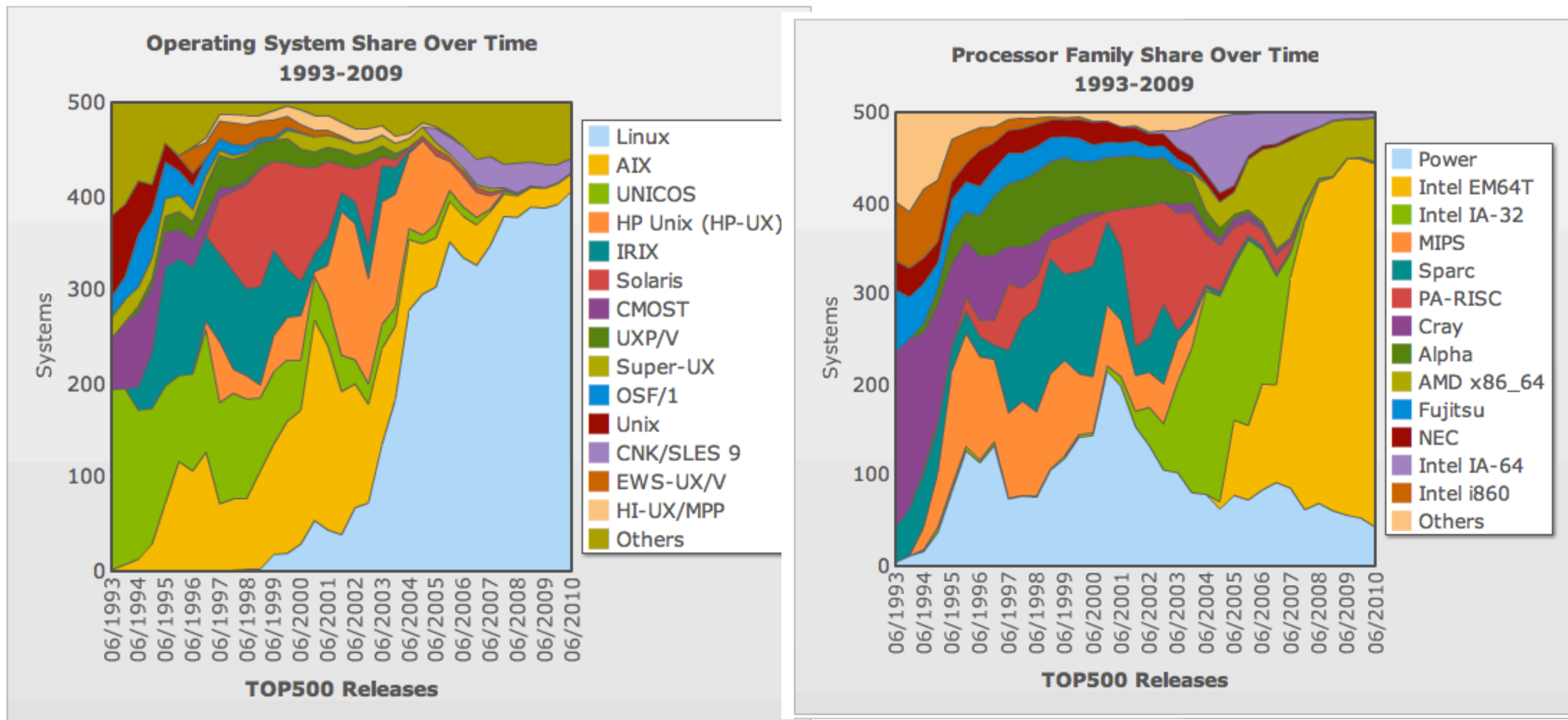
1. The memory wall
2. The power wall
3. The instruction level parallelism (micro-architecture) wall

Go Parallel: many-cores!

- A turning point was reached and a new technology emerged: **multicore**
  - » Keep frequency and consumption low
  - » Transistors used for multiple cores on a single chip: 2, 4, 6, 8 cores on a single chip
- Multiple hardware-threads on a single core
  - » simultaneous Multi-Threading (Intel Core i7 2 threads per core (6 cores), Sun UltraSPARC T2 8 threads per core (8 cores))
- Dedicated architectures:
  - » GPGPU: up to 240 threads (NVIDIA, ATI-AMD, Intel MIC)
  - » CELL
  - » FPGA (Reconfigurable computing)

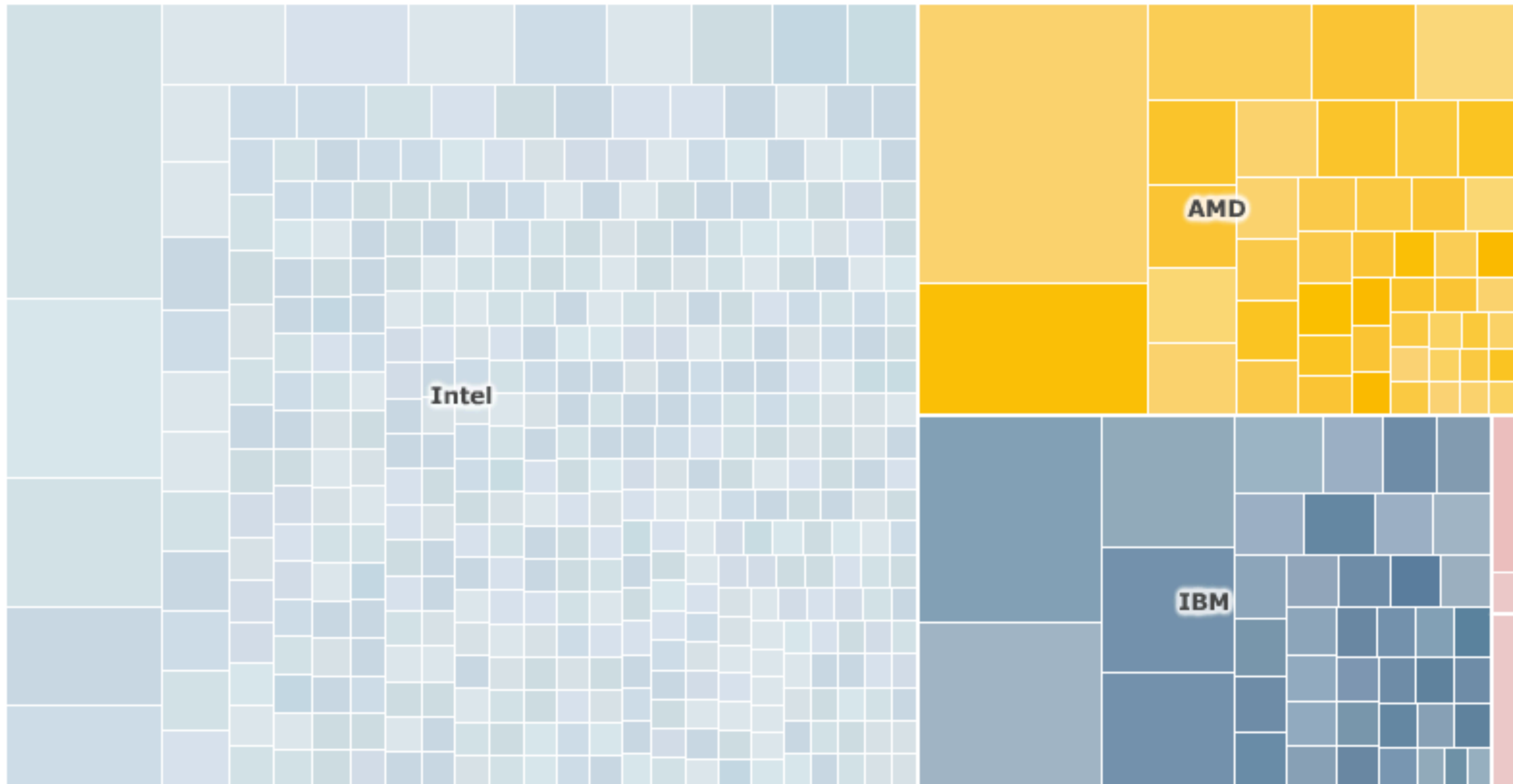
Top 500 1993-2010

Source http://www.top500.org/



Top 500 in 2010

Source BBC http://news.bbc.co.uk/2/hi/technology/10187248.stm



Moving to a new era

1990

- Many architectures
  - » Evolving fast
- Many OS, Compilers, libraries
  - » optimized to a given architecture
- Stead increase of single processor speed
  - » Faster clock
  - » flexible instruction pipelines
  - » Memory hierarchy
- High level software often unable to exploit all these goodies

2010

- One architecture
  - » Few vendor variants
- One Base Software System
- Little increase in single processor speed
- Opportunity to tune performances of application software
  - » Software specific to Pentium3 still optimal for latest INTEL and AMD cpus

HEP SOFTWARE IN THE MULTICORE ERA

HEP software on multicore: an R&D project *(WP8 in CERN/PH)*

The aim of the WP8 *R&D project is to investigate novel software solutions to efficiently exploit the new multi-core architecture of modern computers in our HEP environment*

Motivation:

industry trend in workstation and "medium range" computing

Activity divided in four "tracks"

» Technology Tracking & Tools

» System and core-lib optimization

» Framework Parallelization

» Algorithm Optimization and Parallelization

Coordination of activities already on-going in exps, IT, labs

The Challenge of Parallelization

Exploit all 7 “parallel” dimensions of modern computing architecture for HPC

–Inside a core (climb the ILP wall)

1. Superscalar: Fill the ports (maximize instruction per cycle)
2. Pipelined: Fill the stages (avoid stalls)
3. SIMD (vector): Fill the register width (exploit SSE, AVX)

–Inside a Box (climb the memory wall)

4. HW threads: Fill up a core (share core & caches)
5. Processor cores: Fill up a processor (share of low level resources)
6. Sockets: Fill up a box (share high level resources)

–LAN & WAN (climb the network wall)

7. Optimize scheduling and resource sharing on the Grid

HEP has been traditionally good (only) in the latter

Where are WE?

Experimental HEP is blessed by the natural parallelism of Event processing *(applies to MC integration as well!)*

- HEP code does not exploit the power of current processors
  - One instruction per cycle at best
  - Little or no use of vector units (SIMD)
  - Poor code locality
  - Abuse of the heap
- Running N jobs on N=8/12 cores still “efficient” but:
  - Memory (and to less extent cpu cycles) wasted in non sharing
    - “static” condition and geometry data
    - I/O buffers
    - Network and disk resources
  - Caches (memory on CPU chip) wasted and trashed
    - L1 cache local per core, L2 and L3 shared
    - Not locality of code and data

This situation is already bad today, will become only worse in future many-cores architecture

Code optimization

- Ample Opportunities for improving code performance
  - » Measure and analyze performance of current LHC physics application software on multi-core architectures
  - » Improve data and code locality (avoid trashing the caches)
  - » Effective use of vector/streaming instruction (SSE, future AVX)
  - » Exploit modern compiler's features (does the work for you!)
- *See Paolo Calafiura's talk @ CHEP09*:
  http://indico.cern.ch/contributionDisplay.py?contribId=517&sessionId=1&confId=35523
- Direct collaboration with INTEL experts established to help analyzing and improve the code

- All this is absolutely necessary, still not sufficient to take full benefits from the modern many-cores architectures
  - » NEED work on the code to have good parallelization

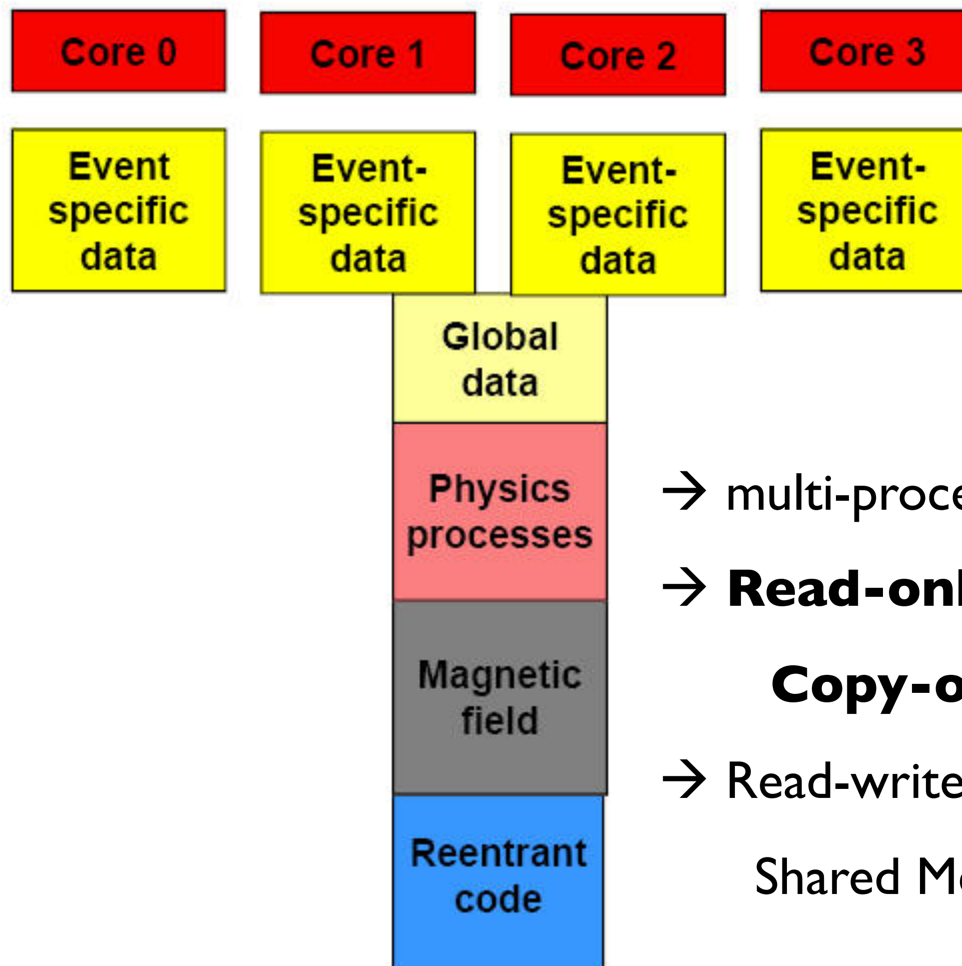
Instrument, measure, improve

- Experiment frameworks (CMSSW, Gaudi, Geant4) instrumented to capture performance counters in specific context (by module, by G4-volume, by G4-particle)
- All experiments, G4, Root successfully reduced memory allocation
- Use of streaming/vector instructions improved float algorithms used in reconstruction by factor 2 (theoretical max is 4)
  - Promising for double-precision in next generation INTEL/AMD cpus
- Speed-up observed when using auto-vectorization in gcc 4.5
- Work started to improve code locality (reduce instruction cache-misses)

Event parallelism

Opportunity: Reconstruction Memory-Footprint shows large condition data

How to share common data between different process?



→ multi-process vs multi-threaded

→ **Read-only:**

Copy-on-write, Shared Libraries

→ Read-write:

Shared Memory, Sockets, Files

Multithreaded Geant4 (Geant4MT)

- Event-level parallelism to simulate separate events by multiple threads
- Efficiency for future many-core CPUs
- Testing and validation on today's 4-, 8- and 24-core nodes
- Preliminary results available based on testing on `fullCMS bench1.g4`
- Patch parser.c of gcc to output static and global declarations in Geant4 source code and add the "__thread" keyword
- Separate and share read-only data members : Geant4 parameterised geomeries and replicas, Geant4 materials and particles, Geant4 physics tables, etc.
- Custom malloc library to support thread private allocation
- Modified G4Navigator to remove unnecessary updates to G4cout and G4cerr precision (shared variables)

"Multi-core & multi-threading: Tips on how to write "thread-safe" code in Geant4", Xin Dong and Gene Cooperman, *14th Geant4 Users and Collaboration Workshop Search*, `http://indico.cern.ch/sessionDisplay.py?sessionId=68\&slotId=0\&confId=44566#2009-` and `http://indico.cern.ch/conferenceDisplay.py?confId=44566`

Experimental Results on 24-core Intel Xeon 7400 Computer

By segregating read-write data members, large read-only memory chunks are formed. Copy-On-Write does not replicate those read-only chunks. (Geant4MT + COW)

- Separate Processes: No reduction for the memory footprint
- Geant4 + COW: Share geometries (no replica or parameterized geometry)
- Geant4MT + COW: Reduce the memory footprint
- Geant4MT: Reduce the memory footprint

Tested on `fullCMS bench1.g4` with 24 workers and 4000 events per worker (electromagnetics).

Implementation	Total Memory on master	Additional Memory per Worker	Total Memory (master + 24 workers)	Runtime
Separate Processes	250 MB	250 MB	6 GB	4575 s
Original Geant4 + COW	250 MB	70 MB	2G MB	4571 s
Geant4MT + COW	250 MB	20 MB	730 MB	4540 s
Geant4MT 24 threads	250 MB	20 MB	730 MB	4510 s

Performance After Output Privatization

Removal of writes to shared G4cout.precision on 4 Intel Xeon 7400 Dunnington

Number of Workers	# Instructions	L3 References	Before Removal		After Removal		
			L3 Misses	CPU Cycles	L3 Misses	Time	Speedup
1	1,598G	87415M	293M	1945G	308M	6547s	1
6	1,598G	87878M	326M	2100G	302M	1087s	6.02
12	1,598G	88713M	456M	3007G	302M	543s	12.06
24	1,599G	88852M	517M	3706G	294M	271s	24.16

Allocator comparison on 4 AMD Opteron 8346 HE

#Wks.	ptmalloc2		ptmalloc3		hoard		tcmalloc		tpmalloc	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	9923s	1	10601s	1	10503s	1	9918s	1	10090s	1
2	4886s	2.03	6397s	1.66	6316s	1.66	4980s	1.99	5024s	2.01
4	2377s	4.17	4108s	2.58	2685s	3.91	2564s	3.87	2504s	4.03
8	1264s	7.85	2345s	4.52	1321s	7.95	1184s	8.37	1248s	8.08
16	797s	12.46	1377s	7.70	691s	15.20	660s	15.02	623s	16.20

Algorithm Parallelization

- Ultimate performance gain will come from parallelizing **algorithms** used in current LHC physics application software
  - » Prototypes using posix-thread, OpenMP and parallel gcclib
  - » On going effort in collaboration with OpenLab and Root teams to provide basic thread-safe/multi-thread library components
    - Random number generators
    - Parallel minimization/fitting algorithms
    - Parallel/Vector linear algebra
- Positive and interesting experience with MINUIT
  - » Parallelization of parameter-fitting opens the opportunity to enlarge the region of multidimensional space used in physics analysis to essentially the whole data sample.

RooFit/Minuit Parallelization

- RooFit implements the possibility to split the likelihood calculation over different threads
  - » Likelihood calculation is done on sub-samples
  - » Then the results are collected and summed
  - » You gain a lot using multi-cores architecture over large data samples, scaling almost with a factor proportional to the number of threads
- However, if you have a lot of free parameters, the bottleneck become the minimization procedure
  - » Split the derivative calculation over several MPI processes
  - » Possible to apply an hybrid parallelization of likelihood and minimization using a Cartesian topology (see A.L. CHEP09 proceeding, to be published on …)
    - Improve the scalability for case with large number of parameters and large samples
- Code already inside ROOT (since 5.26), based on Minuit2 (the OO version of Minuit)

Parallel MINUIT *Alfio Lazzaro and Lorenzo Moneta*

– Minimization of Maximum Likelihood or χ^2 requires iterative computation of the gradient of the NLL function

$$\left.\frac{\partial NLL}{\partial \hat{\theta}}\right|_{\hat{\theta}_0} \approx \frac{NLL(\hat{\theta}_0 + \hat{\mathrm{d}}) - NLL(\hat{\theta}_0 - \hat{\mathrm{d}})}{2\hat{\mathrm{d}}}$$

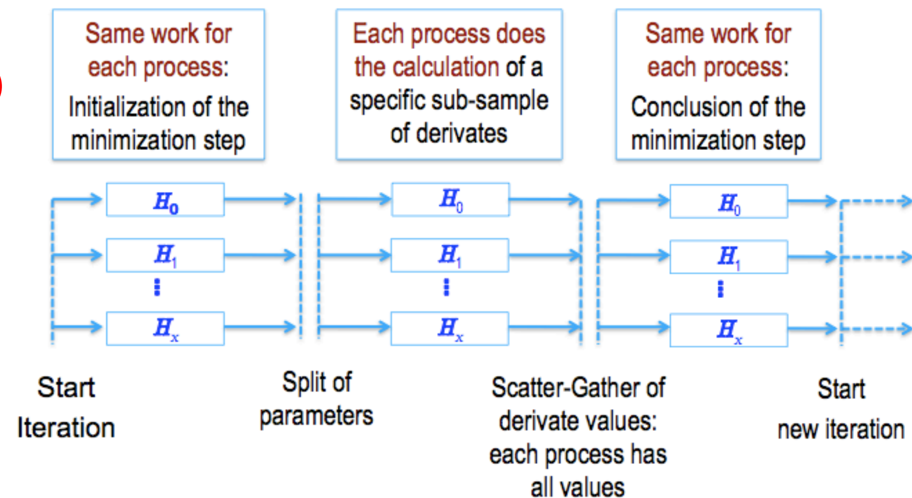
$$NLL = \ln\left(\sum_{j=1}^{s} n_j\right) - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j^i\right)$$

j species (signals, backgrounds)
n_j number of events for specie j
$\mathcal{P}_j$ probability density functions (PDFs)
N number total of events to fit

– Execution time scales with number θ free parameters and the number N of input events in the fit

– Two strategies for the parallelization of the gradient and NLL calculation:

1. Gradient **or** NLL calculation on the same multi-cores node (OpenMP)
1. Distribute Gradient on different nodes (MPI) **and** parallelize NLL calculation on each multi-cores node (pthreads): **hybrid solution**

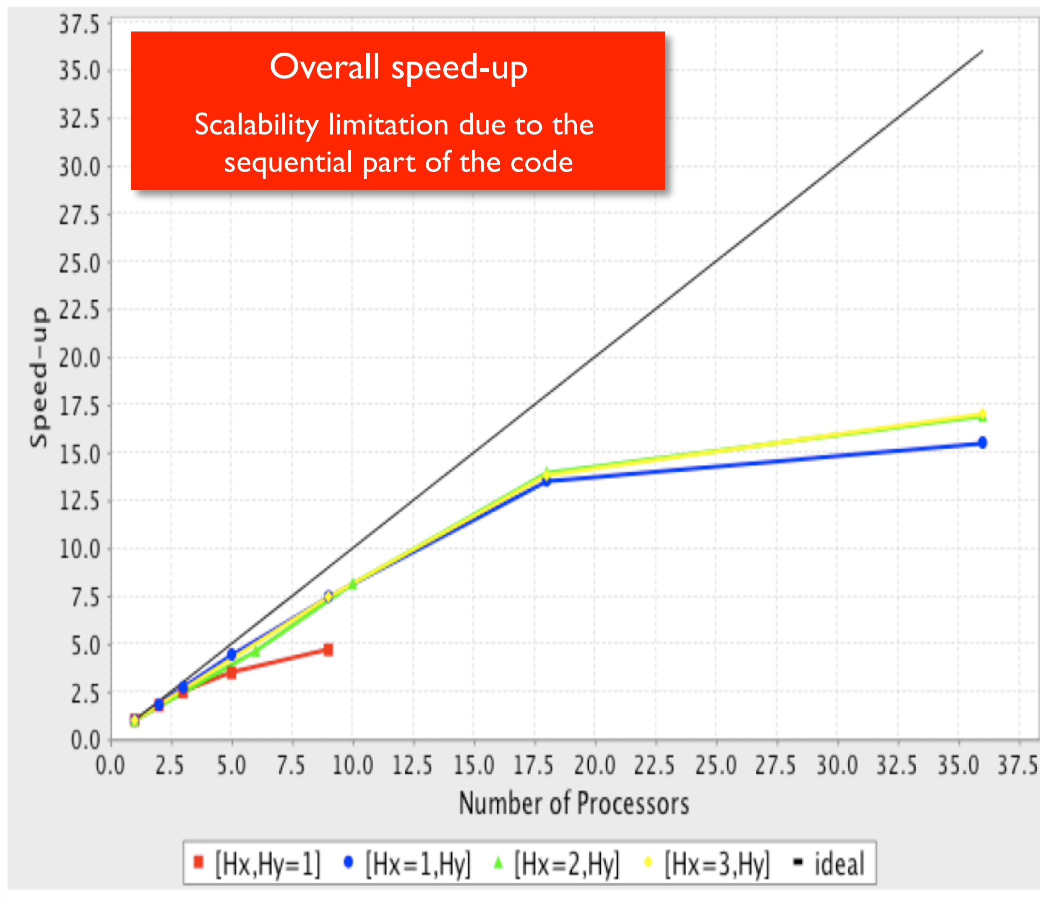


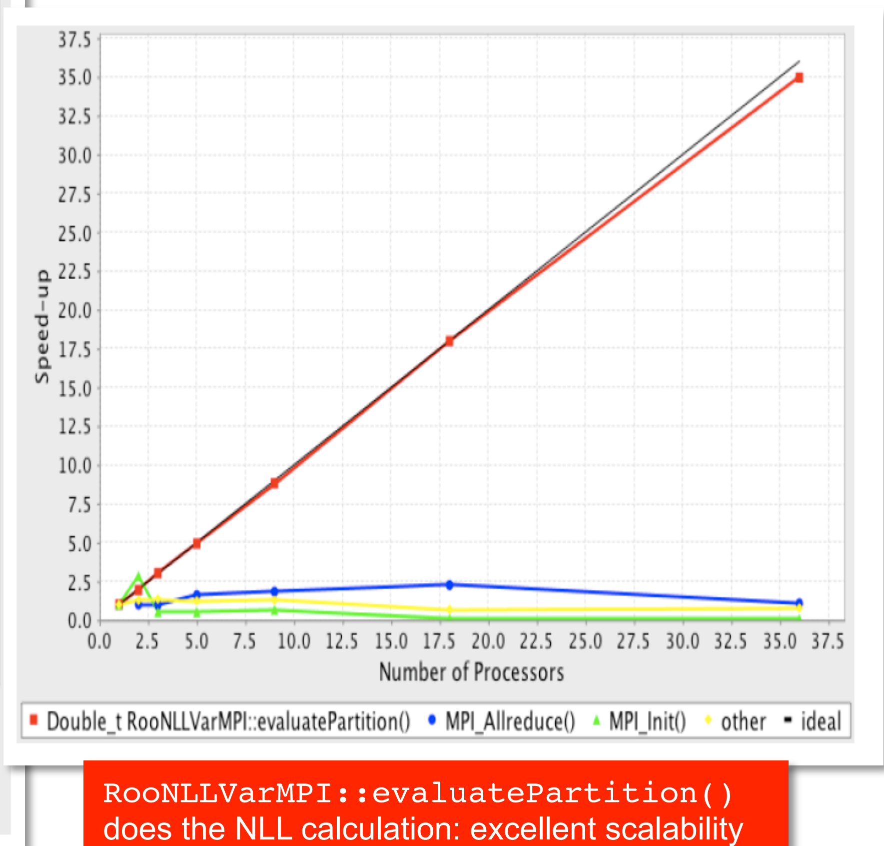
Test @ INFN CNAF cluster, Bologna (Italy)

3 variables, 600K events, 23 free parameters

PDFs per each variable: 2 Gaussians for signal, parabola for background

Sequential execution time (Intel Xeon @ 2.66GHz): **~80 minutes**





`RooNLLVarMPI::evaluatePartition()` does the NLL calculation: excellent scalability

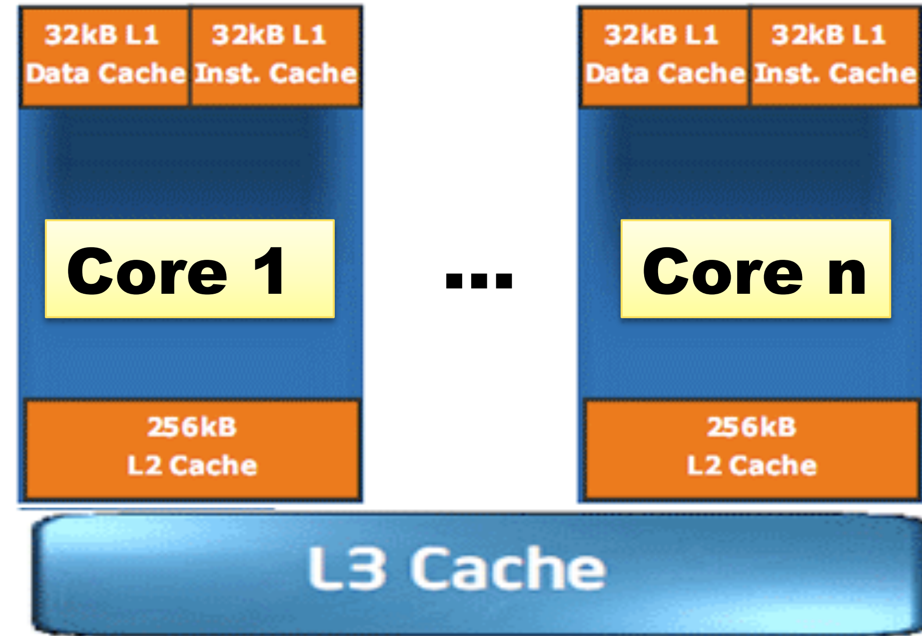
Summary

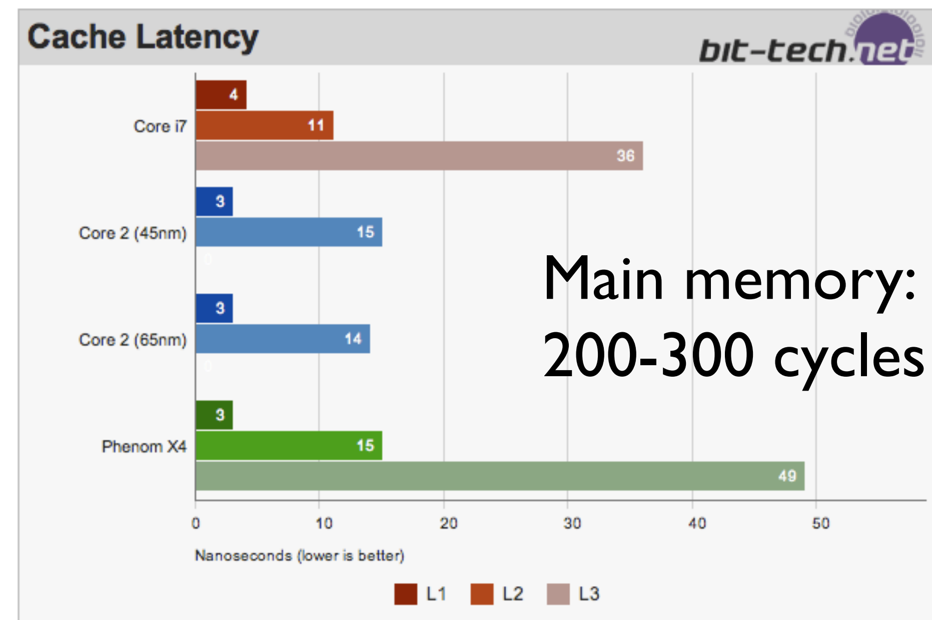
- The stagnant speed of single processors and the narrowing of the number of OSs and computing architectures modify the strategy to improve the performance of software applications
  - » Aggressive software optimization tailored to the processor in hand
  - » Parallelization
  - » Optimization of the use of “out-core” resources
- Experimental HEP is blessed by the natural parallelism of event processing:
  - » Very successful evolution of “frameworks” to multi-process with read-only shared memory
  - » Parallelize existing code using multi-thread proved to be “tricky”
  - » Exploiting this new processing model requires a new model in computing resources allocation as well:
    - The most promising solution is full node allocation

BACKUP SLIDES

The 'memory wall'

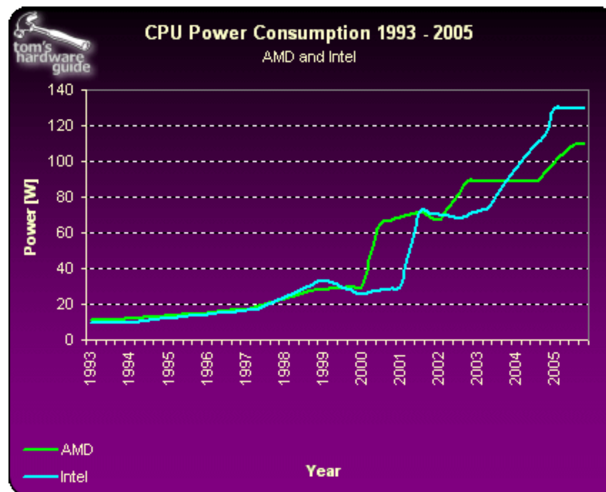
- Processor clock rates have been increasing faster than memory clock rates
- larger and faster "on chip" cache memories help alleviate the problem but does not solve it
- Latency in memory access is often the major performance issue in modern software applications

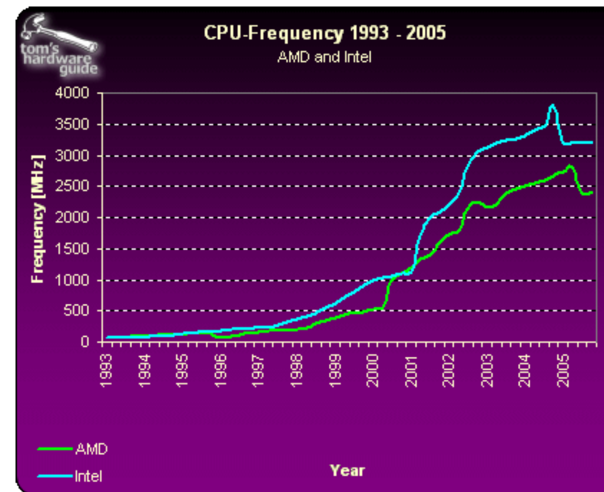




The 'power wall'

- Processors consume more and more power the faster they go
- Not linear:
  - 73% increase in power gives just 13% improvement in performance
  - (downclocking a processor by about 13% gives roughly half the power consumption)
- Many computing center are today limited by the total electrical power installed and the corresponding cooling/extraction power
- Green Computing!

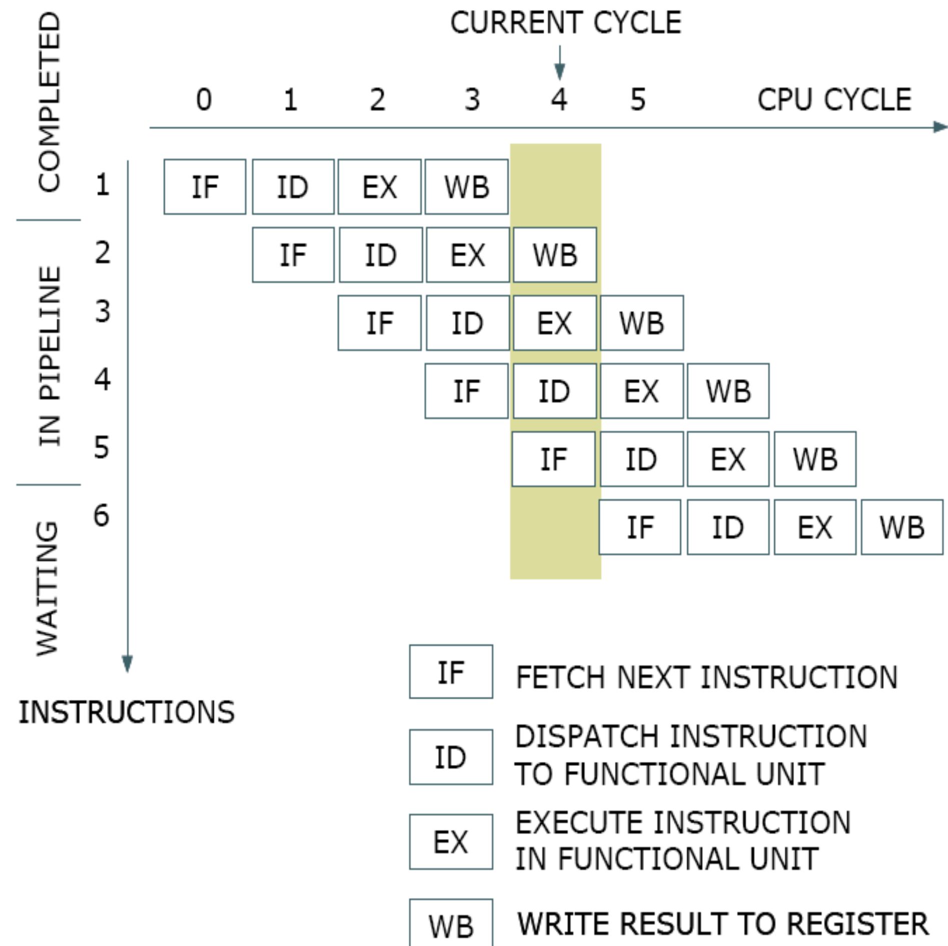




http://www.processor-comparison.com/power.html

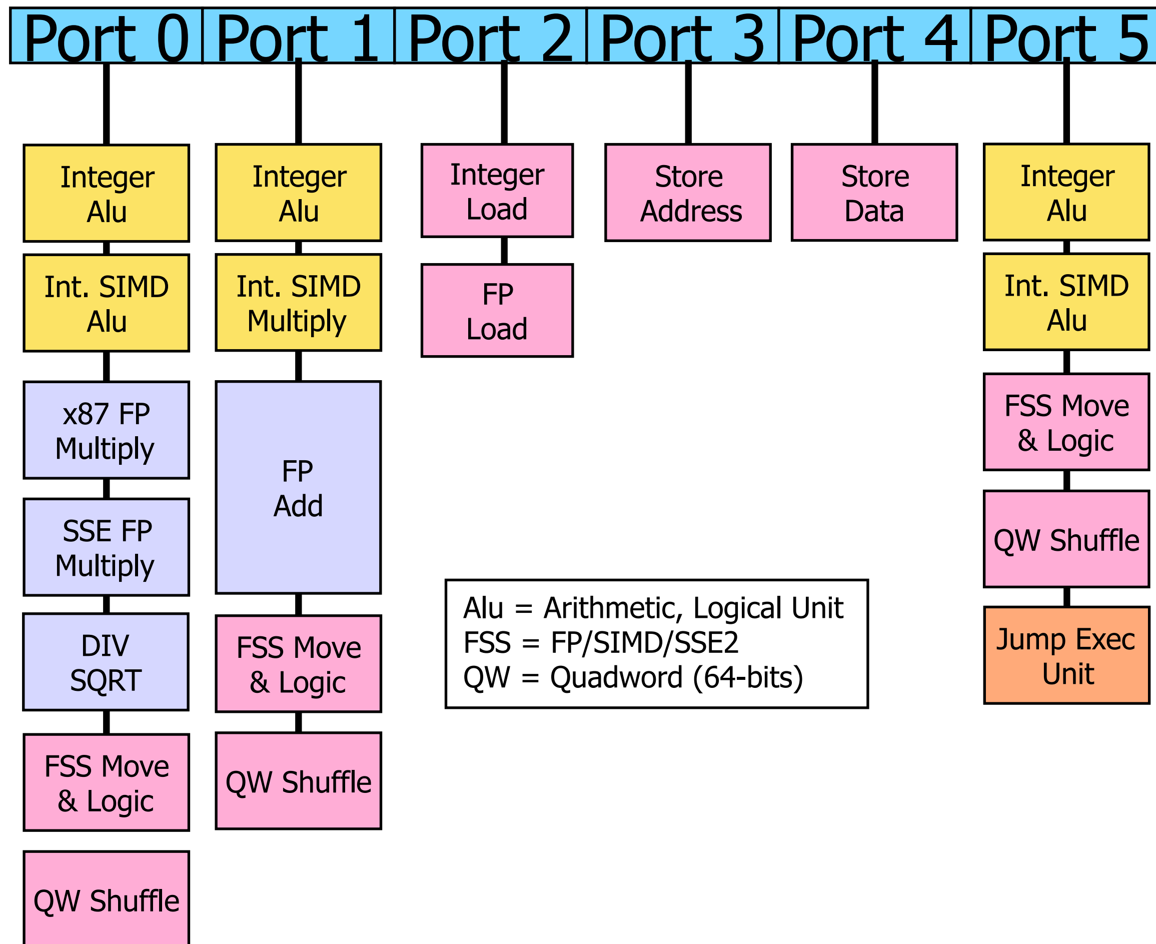
The 'Architecture walls'

- Longer and fatter parallel instruction pipelines has been a main architectural trend in `90s
- Hardware branch prediction, hardware speculative execution, instruction re-ordering (a.k.a. out-of-order execution), just-in-time compilation, hardware-threading are some notable examples of techniques to boost Instruction level parallelism (ILP)
- In practice inter-instruction data dependencies and run-time branching limit the amount of achievable ILP



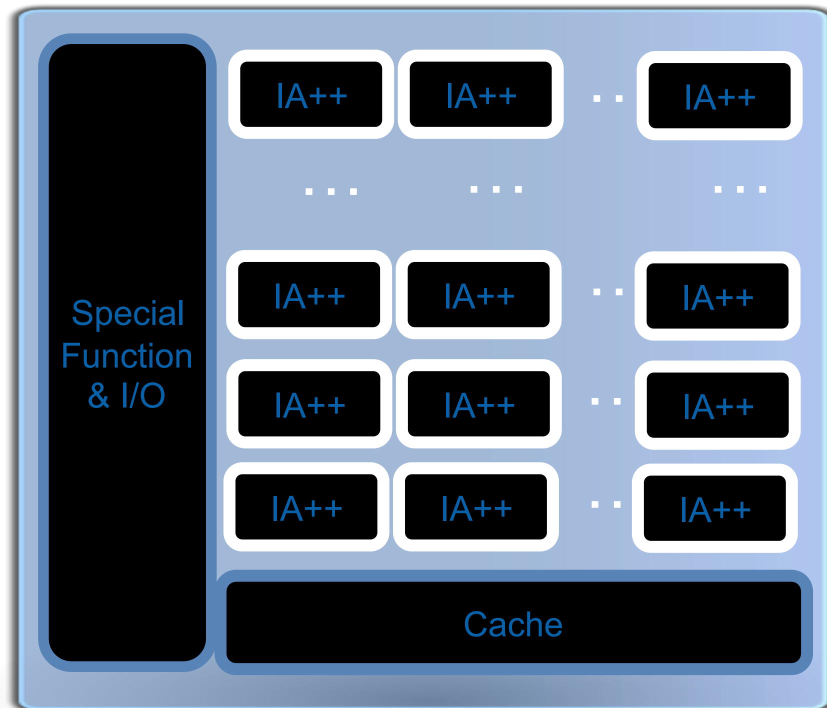
Core 2 execution ports

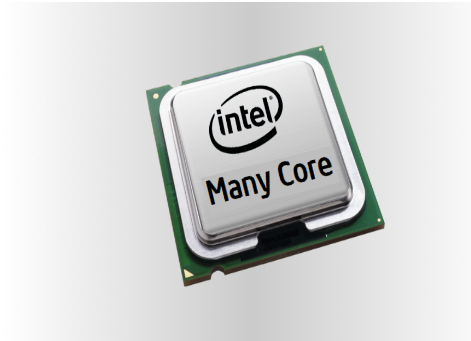
- Intel's Core microarchitecture can handle:
  - Four instructions in parallel:
  - Every cycle
  - Data width of 128 bits



Issue ports in the Core 2 micro-architecture (from Intel Manual No. 248966-016)

Bringing IA Programmability and Parallelism to High Performance & Throughput Computing

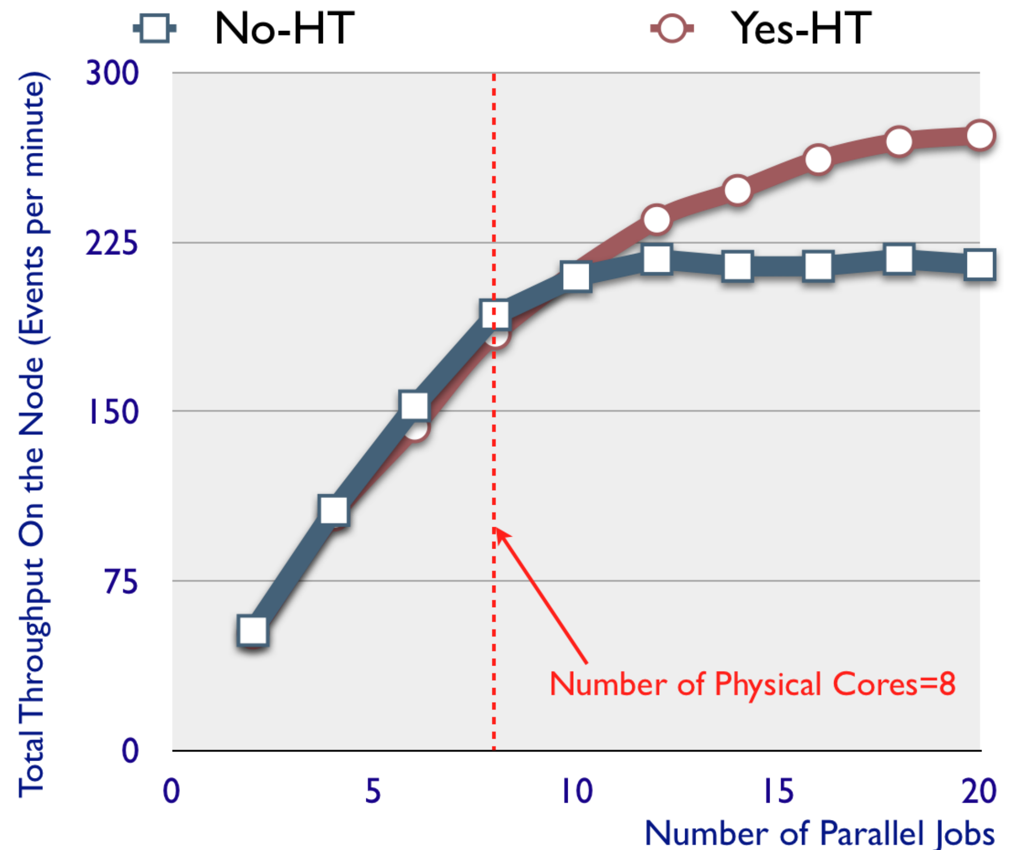




- Highly parallel, IA programmable architecture in development
- Ease of scaling for software ecosystem
- Array of enhanced IA cores
- New Cache Architecture
- New Vector Processing Unit
- Scalable to TFLOPS performance

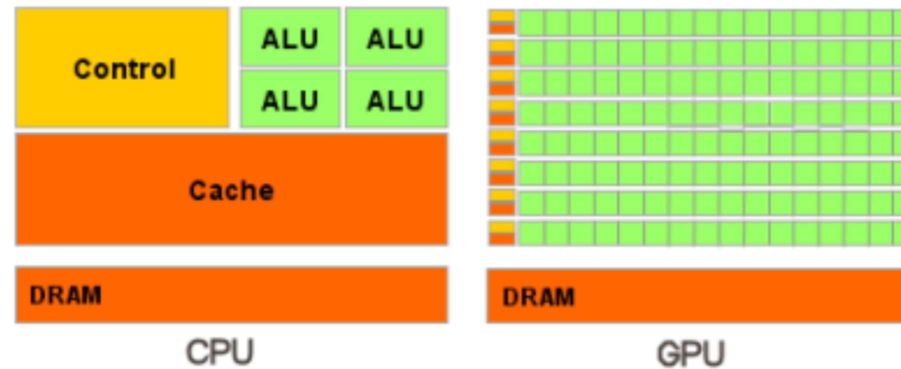
Parallel Job Performance with Hyper-Threading

- The Computer:
  - ✦ coors.lbl.gov
  - ✦ Dual-Xeon X5550@2.67G
  - ✦ 8 Cores in total, 24GB Mem
  - ✦ Hyper Threading
- The Jobs:
  - ✦ ATLAS Fast Reconstruction
  - ✦ 50 Events per job
  - ✦ Each job takes ~2 min.



- Tests:
  - ✦ For each N in (2, 4, 6, 8, 10, 12, 14, 16, 18, 20), run at the same time N parallel jobs, and measure the time each job takes. Repeat 10 times for more statistics for each N.
  - ✦ The throughput is the total number of events the Computer can process when running N parallel jobs.
  - ✦ This is to simulate the scenario of batch node in a cluster.
- Result:
  - ✦ With Hyper threading, one can stuff more jobs into the same node to achieve higher throughput
  - ✦ Meaning: if our clusters have HT-enabled CPUs, we can let the scheduler over commit jobs within the limit of memory. For this case, we can process 25% more events.

GPUs?



- A lot of interest is growing around GPUs
  - » Particular interesting is the case of NVIDIA cards using CUDA for programming
  - » Impressive performance (even 100x faster than a normal CPU), but high energy consumption (up to 200 Watts)
  - » A lot of project ongoing in HPC community. More and more example in HEP (wait for tomorrow talk…)
  - » Great performance using single floating point precision (IEEE 754 standard): up to 1 TFLOPS (w.r.t 10 GFLOPS of a standard CPU)
  - » Need to rewrite most of the code to benefit of this massive parallelism (thread parallelism), especially memory usage: it can be not straightforward…
  - » The situation can improve with OpenCL (*Tim Mattson visiting CERN next Monday)* and Intel Larrabee architecture (standard x86)