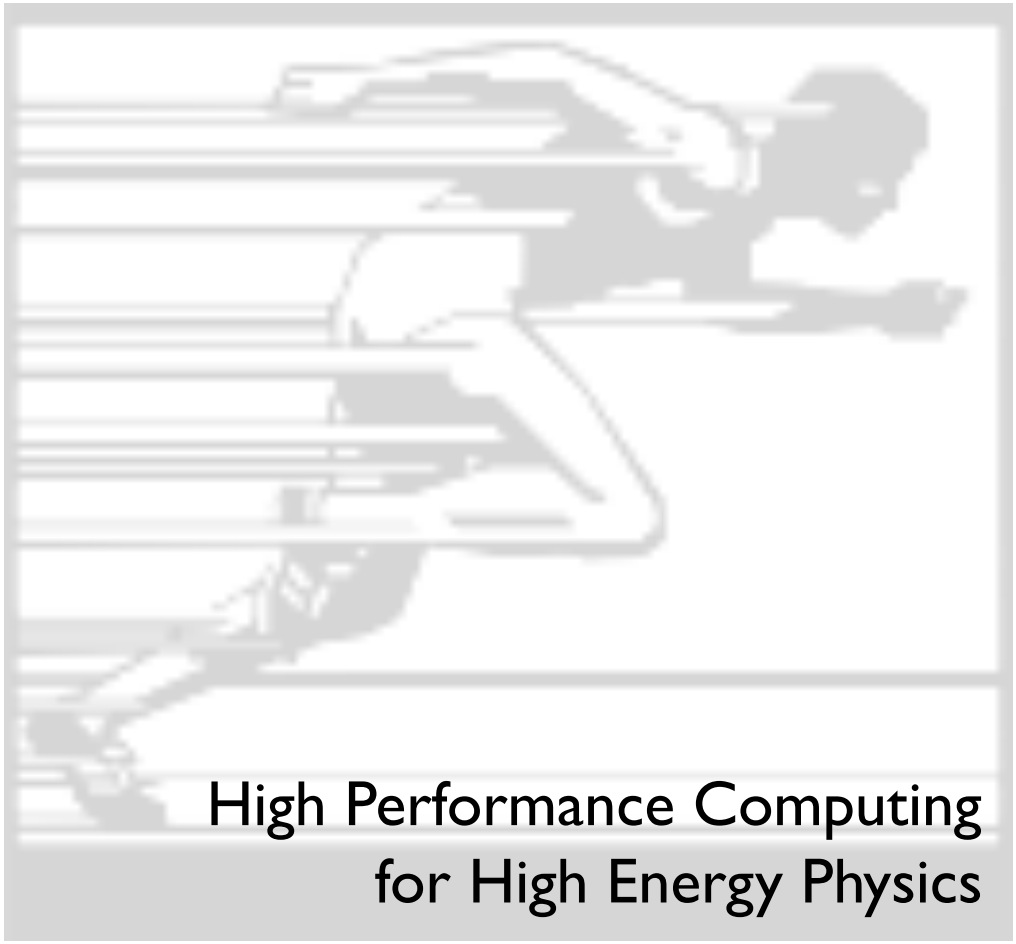
The challenge of adapting HEP physics software applications to run on many-core cpus



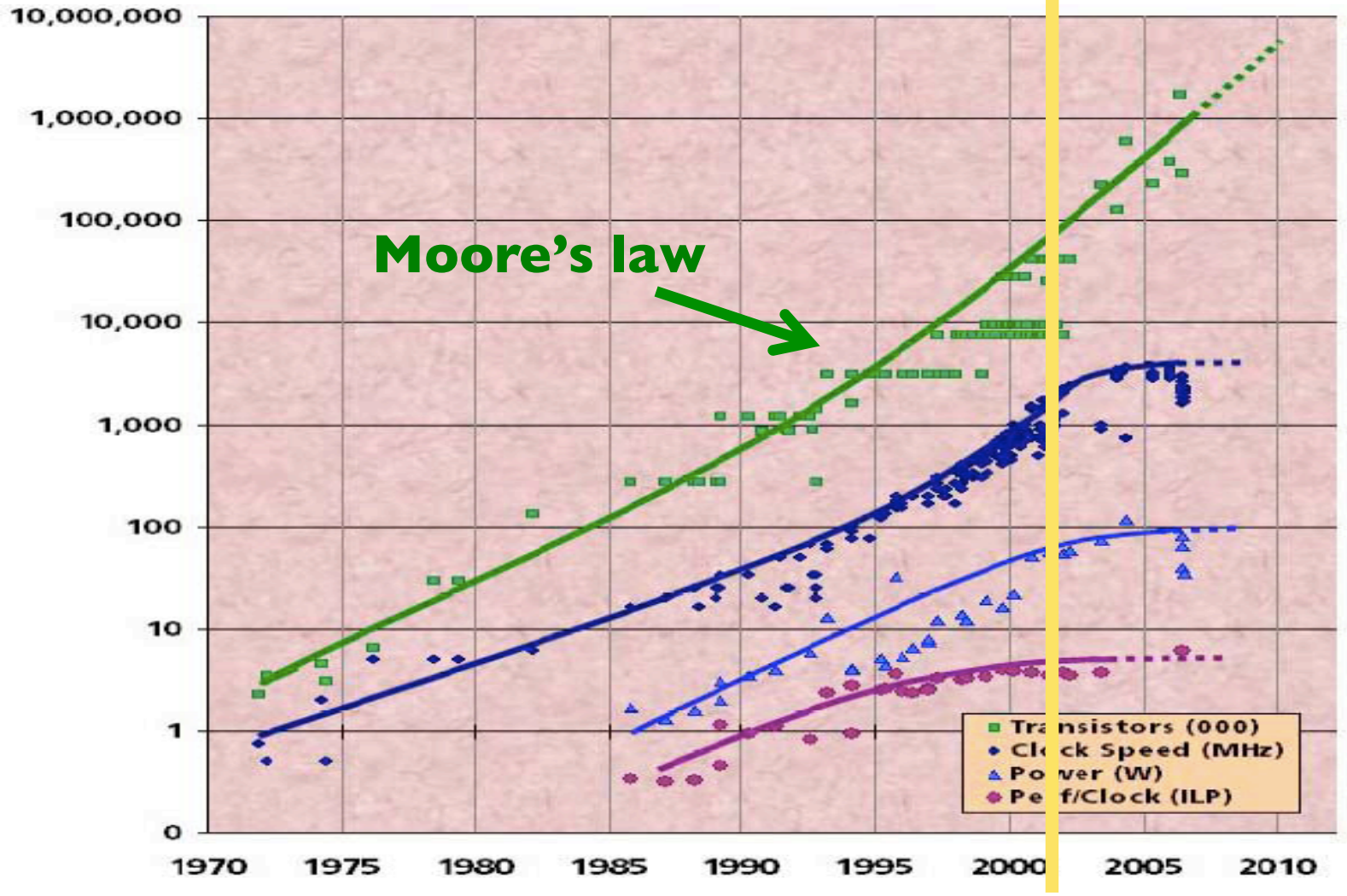
SuperB Workshop, March `10

Vincenzo Innocente
CERN

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

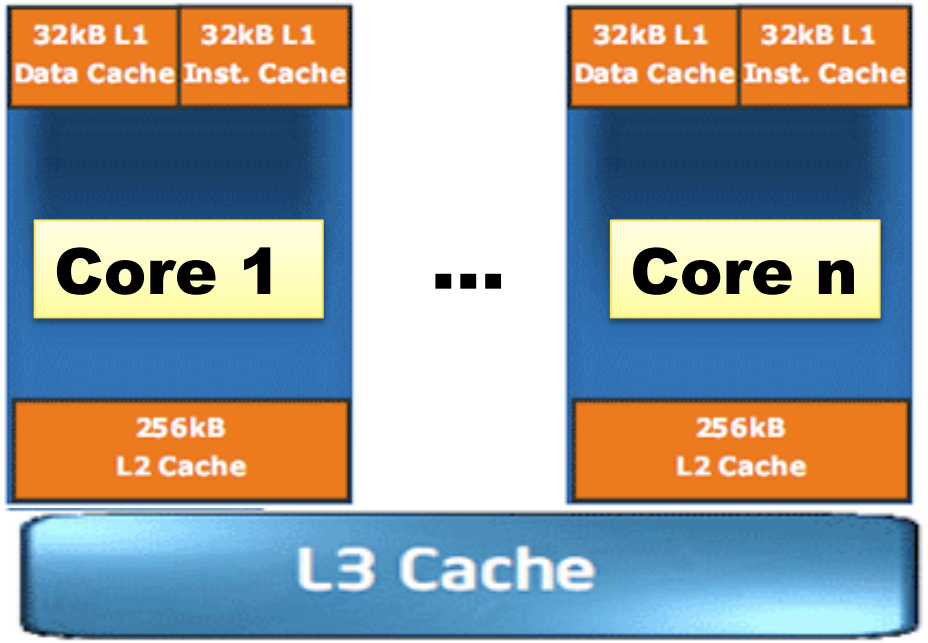
The ‘three walls’

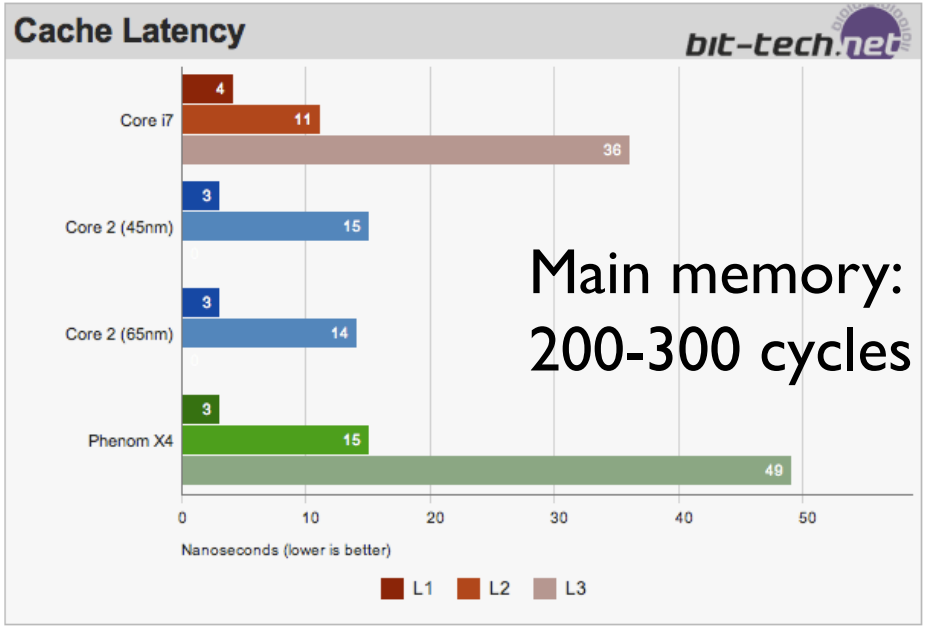
While hardware continued to follow Moore’s law, the perceived exponential grow of the “effective” computing power faded away in hitting three “walls”:

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

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

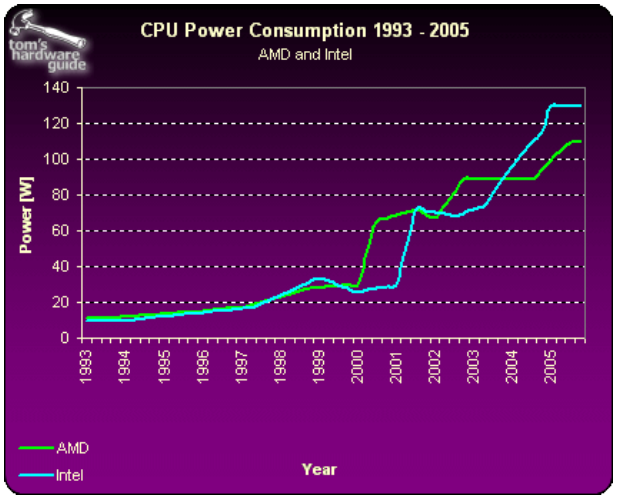


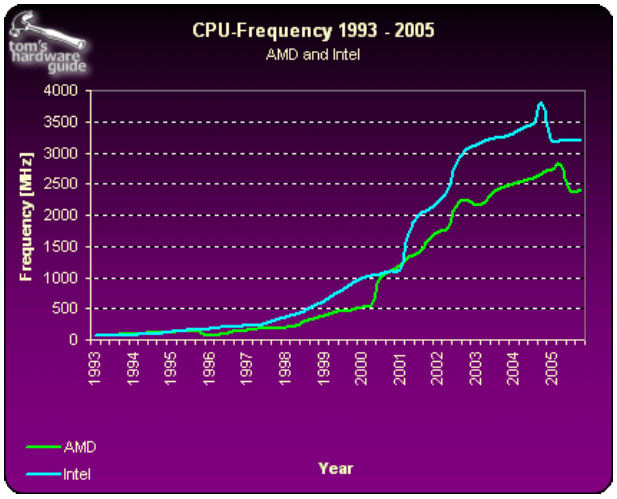


Main memory: 200-300 cycles

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!

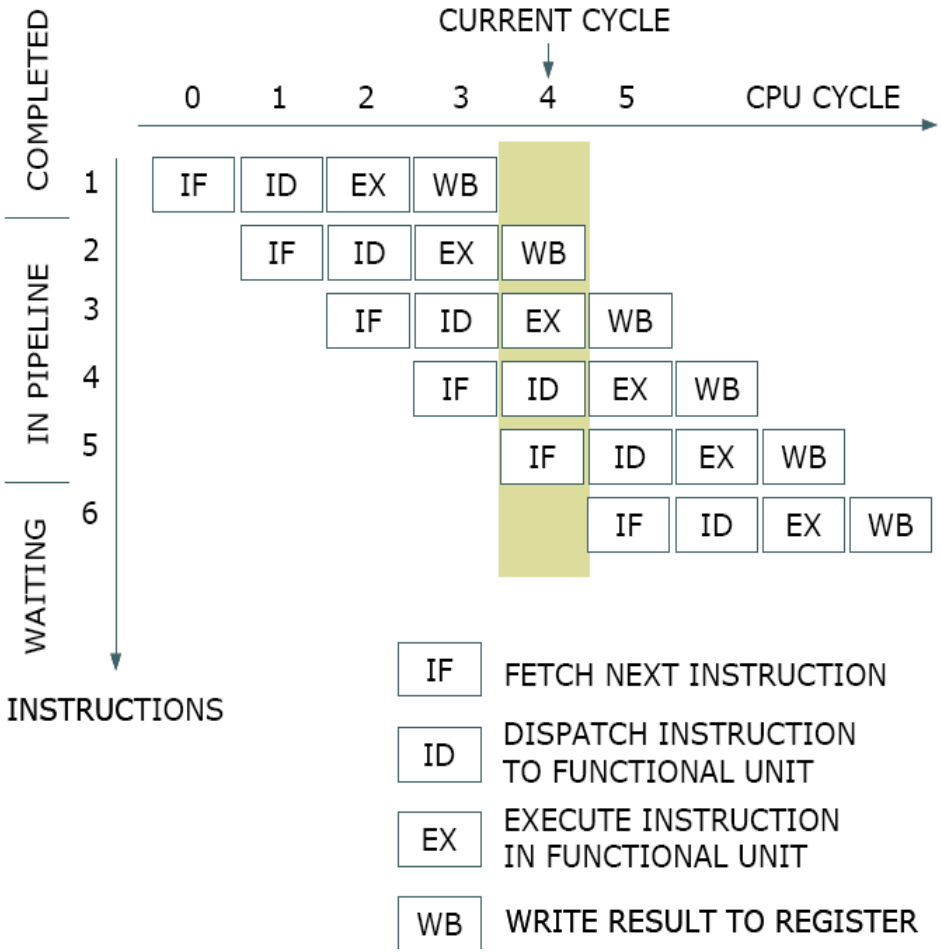




http://www.processor-comparison.com/power.html

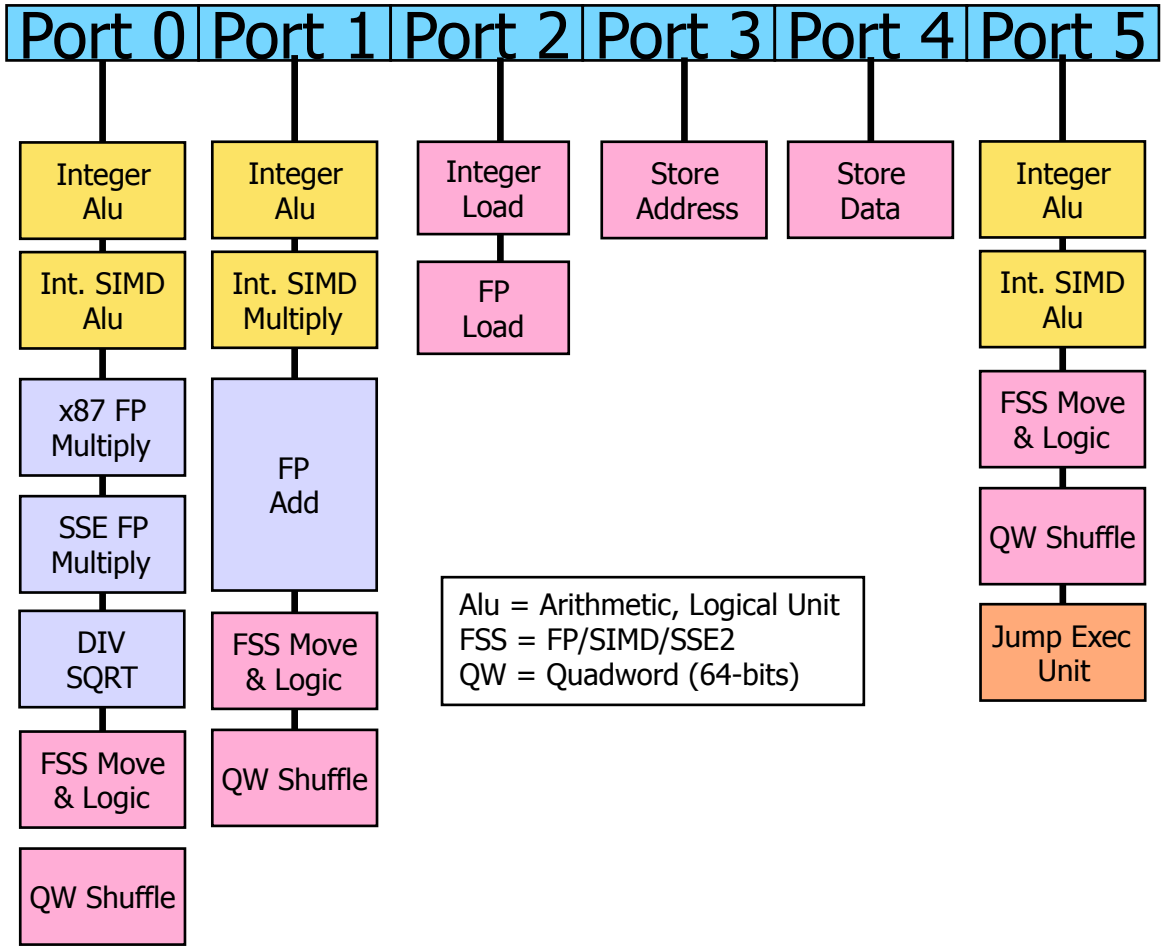
The ‘Architecture walls’

- 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)

Some Observations Regarding Benefits vs. Efforts

- The number of transistors on a chip doubles every 24 month
- Processor architectures changed from area limited to power limited

- ILP useful for ~4 parallel instructions
- Instruction pipeline useful if ≤ 30 stages
- Power consumption grows about cubically with frequency
- Processing capability grows faster than memory speed

- Single core performance is growing slower than it used to be

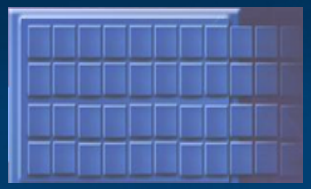
- SMT/HT can mitigate the situation for certain workloads
- CMP/Multi-Core seems to be a reasonable “compromise”
- Opportunity for performance to increase faster than Moore’s Law

Go Parallel: many-cores!

– A turning point was reached and a new technology emerged: **multicore**
  » Keep low frequency and consumption
  » 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 (4 cores), Sun UltraSPARC T2 8 threads per core (8 cores))
– Dedicated architectures:
  » GPGPU: up to 240 threads (NVIDIA, ATI-AMD, Intel Larrabee)
  » CELL
  » FPGA (Reconfigurable computing)

Industry Trend to Multi/Many-Core

*Intel **Tera-Scale** Computing Research Program: www.intel.com/go/terascale*



Many-Core

Multi-Core

Dual-Core

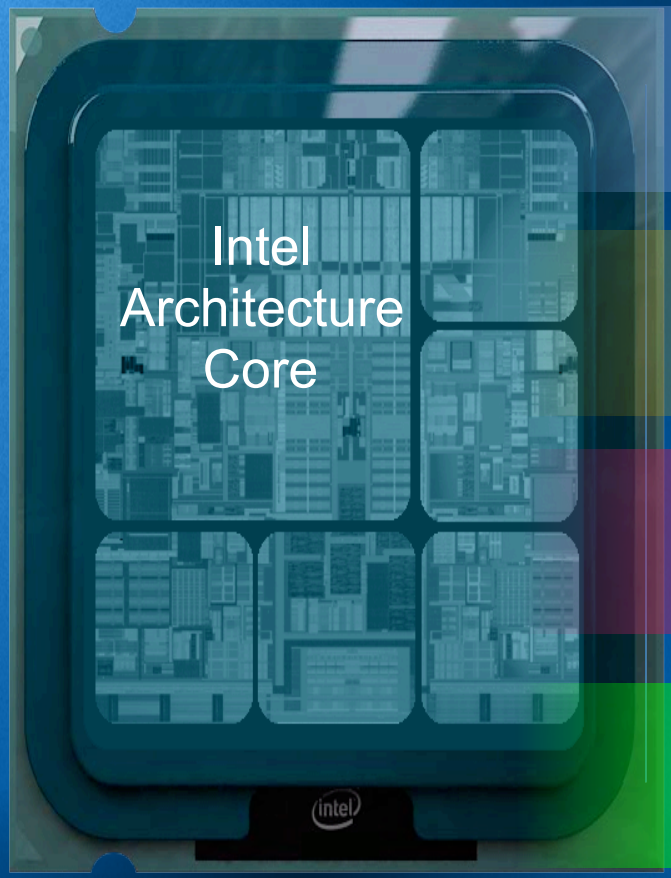
Hyper-Threading

Multi Processor

QUAD-CORE

Energy Efficient Petascale with Multi-threaded Cores

Going Forward



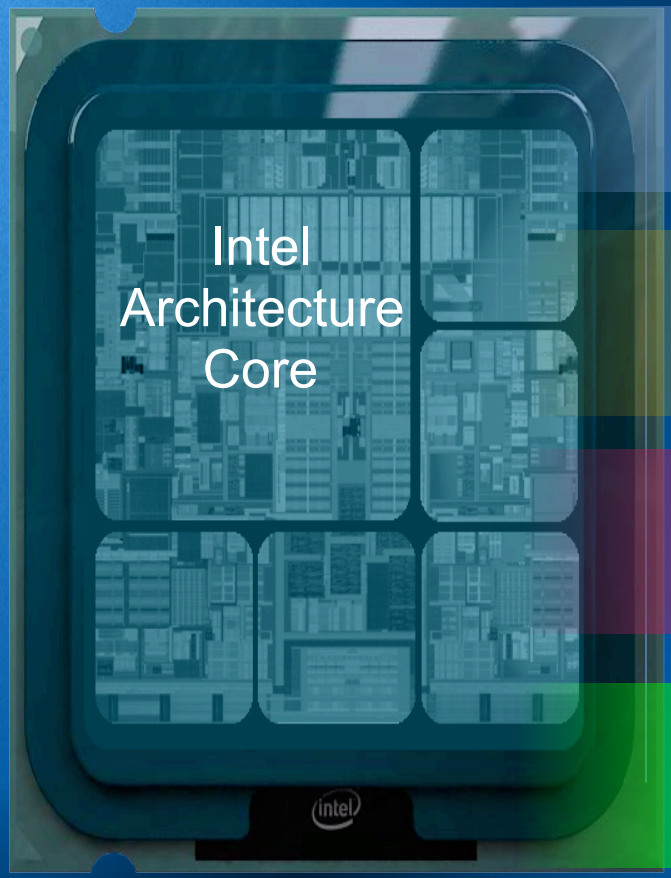
New Materials and Designs

Core Enhancements

Multi to Many-Core

Platform Enhancements

Going Forward



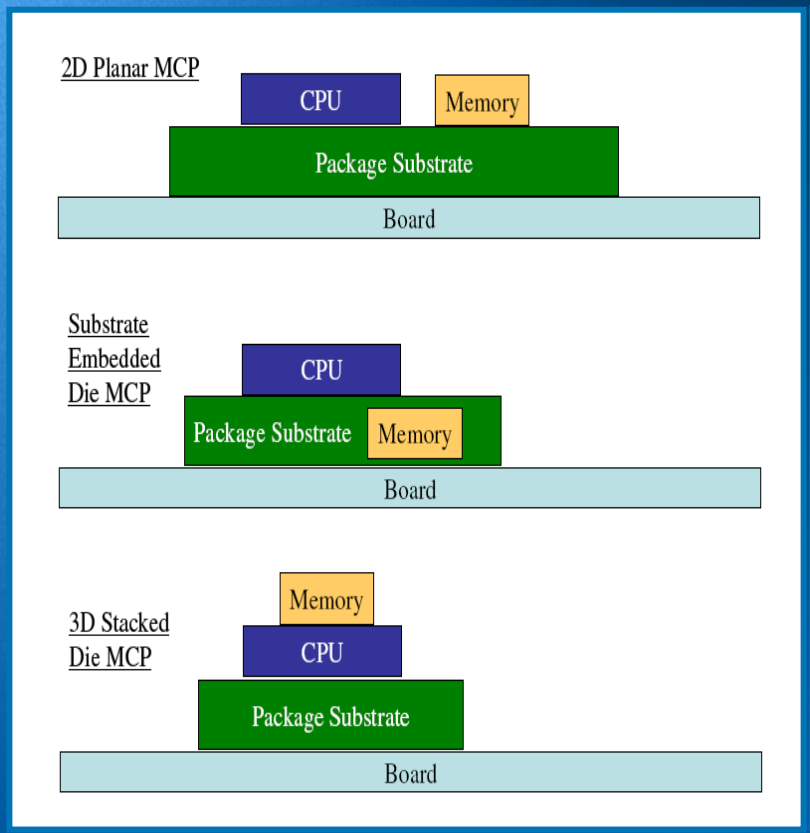
Tri-Gate, Nanotubes →

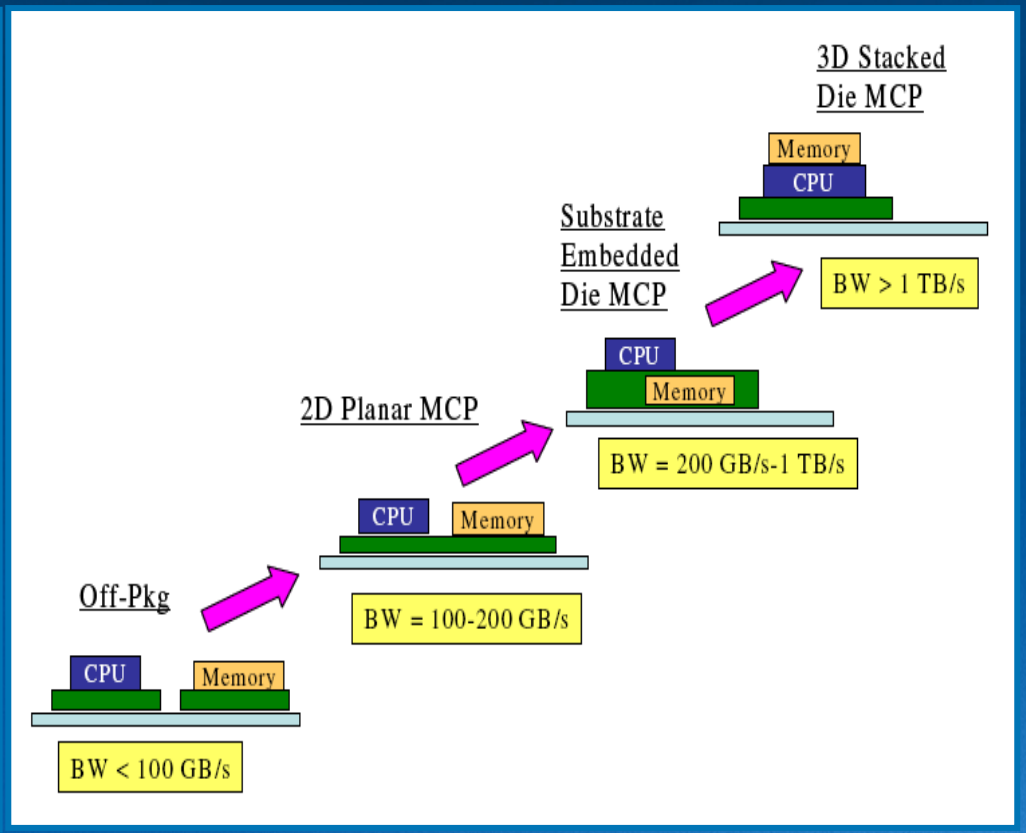
MMX → SSE → AVX →

Dual → Quad → Octo →

PCIe, IMC, QPI, SOC →

Memory and CPU package architectures for addressing bandwidth challenges





Package Technology to Address the Memory Bandwidth Challenge for Tera-scale Computing, Intel Technology Journal, Volume 11, Issue 3, 2007

Future Vision, does not represent real Intel product

Tick/Tock: Our Model for Sustained Microprocessor Leadership

Intel® Core™	Penryn	Nehalem	Westmere	Sandy Bridge
NEW Microarchitecture	Compaction/ Derivative	NEW Microarchitecture	Compaction/ Derivative	NEW Microarchitecture
65nm	**45nm**	45nm	**32nm**	32nm
2006	2007	2008	2009	2010

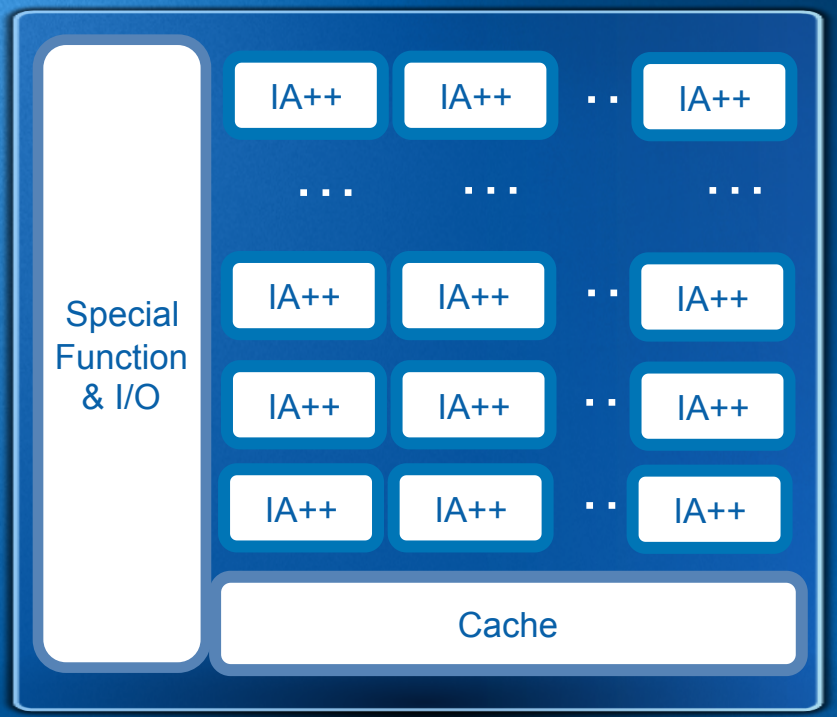
Forecast

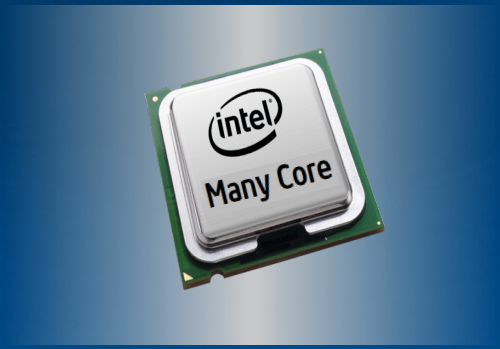
e.g. Intel® QuickPath Architecture

e.g. Intel® AVX

Future options subject to change without notice.

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

Future options subject to change without notice.

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?

- 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 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 (thread/core affinity)
- This situation is already bad today, will become only worse in future many-cores architectures

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"

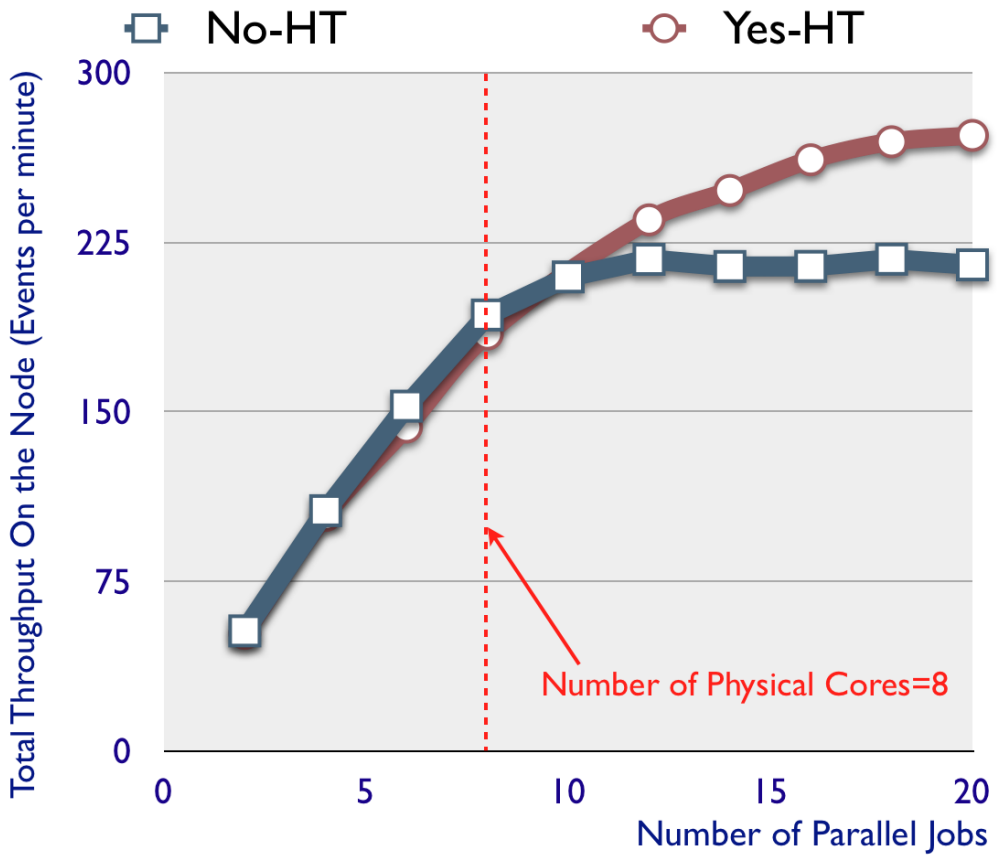
- » Technology Tracking & Tools
- » System and core-lib optimization
- » Framework Parallelization
- » Algorithm Parallelization

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

Summary of activity in 2008/2009

- Collaboration established with experiments, OpenLab, Geant4 and ROOT
  - » Close interaction with experiments (bi-weekly meetings, reports in AF)
  - » Workshops each "six" months (April, October 2008, june 2009, next june 2010)
- Survey of HW and SW technologies
  - » Target multi-core (8-16/box) in the short term, many-core (96+/box) in near future
  - » Optimize use of CPU/Memory architecture
  - » Exploit modern OS and compiler features (copy-on-write, MPI, OpenMP)
- Prototype solutions
  - » In the experiments and common projects (ROOT, Geant4)
    - Improved Root-Math, ProofLite, MT-G4, flork&COW in ATLAS and CMS
  - » In the R&D project itself
    - Parallel GAUDI, perfmon instrumentation of Gaudi and CMSSW

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.

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 instruction (improve ILP)
  - » 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

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

Floating Point Math on new CPUs

- Moore's law and "progress" in compiler technology have made HEP to focus mostly in (excessive?) accuracy
  - » Why spending effort in careful optimizations if in the meanwhile computers gets much faster, hardware changes, compiler gets less stupid?
  - » **The time of free lunches is over!**
- **We need to re-establish excellence in numerical computation competence in our field!**
  - » We shall master approximations, vectorization, numerical methods beyond simple FORmula TRANslation
- Each new computing architecture changes the balance between accuracy and speed in particular for FP
  - » The code emitted even for a trivial "`1/sqrt(x)`" depends on machine, OS, compiler, compiler-option, even on the way we write it!
- One size does not fit all: need of accuracy and speed depends on the context
  - » Cannot use global switches, replacement libraries, etc
  - » We shall be able to select the required accuracy for each single use-case

Optimization of sequential FP algos

```
// Energy loss and variance according to Bethe and Heitler, see also
// Comp. Phys. Comm. 79 (1994) 157.
//
double p = localP.mag();
double normalisedPath = fabs(p/localP.z())*materialConstants.radLen();
double z = exp(-normalisedPath);
double varz = (exp(-normalisedPath*log(3.)/log(2.))-
                                exp(-2*normalisedPath));

if ( propDir==oppositeToMomentum ) {
  // for backward propagation: delta(1/p) is linear in z=p_outside/p_inside
    theDeltaP += -p*(1/z-1);
  theDeltaCov(0,0) += varz/p/p;
}else {
   // for forward propagation: calculate in p (linear in 1/z=p_inside/p_outside)
    theDeltaP += p*(z-1);
  double f = 1./p/z;
  theDeltaCov(0,0) += f*f*varz;
}
```

This code will not become faster in future (gcc 4.5 will help a bit with compiler-time transcendentals)

IF speed is a concern
- accuracy needs to be tuned
- vectorization/parallelization

have to be accounted for

Vectorization: Features & Challenges

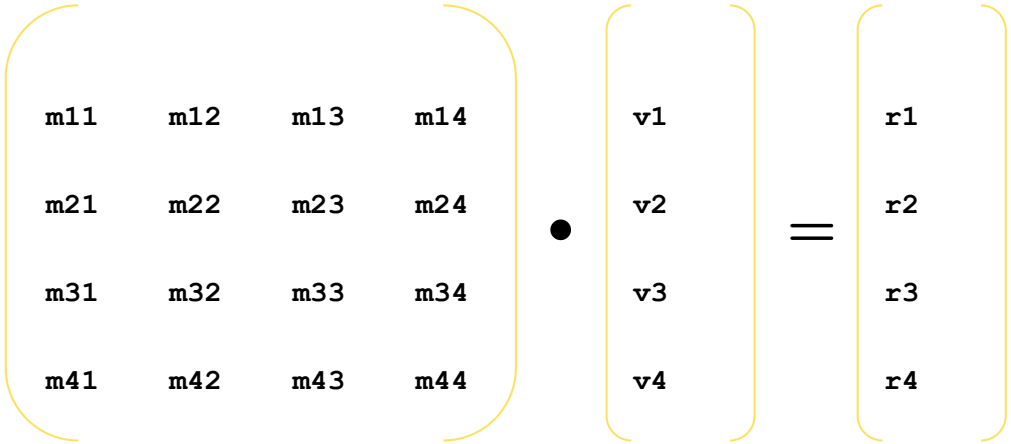
- SIMD computational width
  - 128 bits (current) → 2 doubles or 4 floats
  - 256 bits (2010 - Sandy Bridge) → 4 doubles or 8 floats
  - 512 bits (2011 - Larrabee) → 8 doubles or 16 floats
  - 1024 bits (coming soon) → 16 doubles or 32 floats

- It makes sense to start **now** to take it into consideration developing *pilot projects* in production environment

- Possible ways of doing it:
  - Assembly
  - **Instrinsics**
  - Autovectorization
  - SIMD-aware programming languages

- Challenges using intrinsics:
  - Reorganize your data to scale to increasing vector width
  - Rethink your algorithm (do not readapt it!)

Trivial example: matrix-vector multiplication




```
__m128 t0 = _mm_set1_ps(v0());
__m128 t1 = _mm_set1_ps(v1());
__m128 t2 = _mm_set1_ps(v2());
__m128 t3 = _mm_set1_ps(v3());

t0 = _mm_mul_ps(m0, t0);
t1 = _mm_mul_ps(m1, t1);
t2 = _mm_mul_ps(m2, t2);
t3 = _mm_mul_ps(m3, t3);

__m128 rs = _mm_add_ps(t0,
            _mm_add_ps(t1,
            _mm_add_ps(t2, t3)));
```

Standard Procedure

```
r1 = v1*m11+v2*m12+v3*m13+v4*m14
r2 = v1*m21+v2*m22+v3*m23+v4*m24
r3 = v1*m31+v2*m32+v3*m33+v4*m34
r4 = v1*m41+v2*m42+v3*m43+v4*m44
```

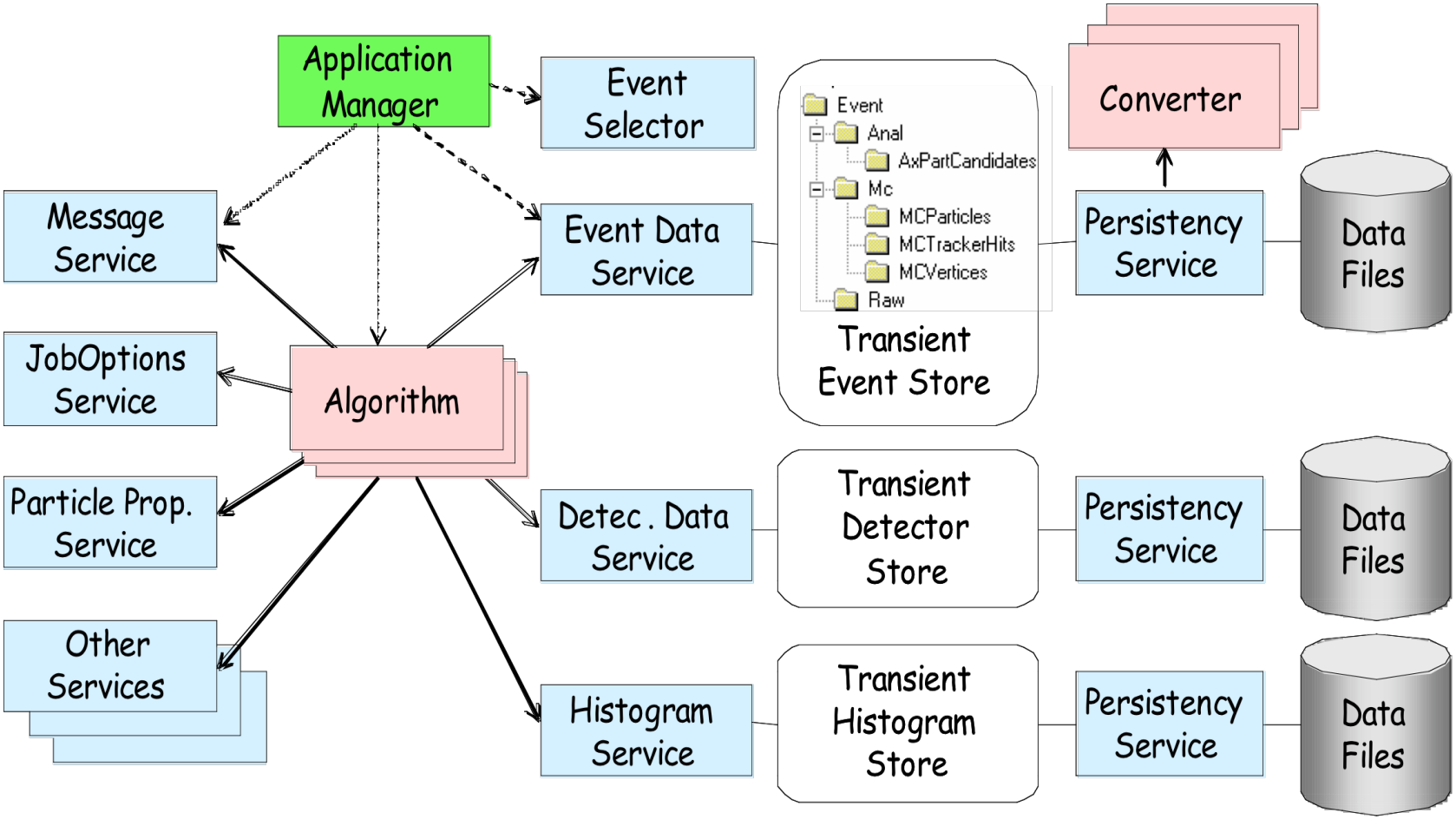
Vector Procedure

```
t1 = [m11, m21, m31, m41] * v1
t2 = [m12, m22, m32, m42] * v2
t3 = [m13, m23, m33, m43] * v3
t4 = [m14, m24, m34, m44] * v4
R = t1 + t2 + t3 + t4
```

Observed speed-up in this example: ~2x (instead of 4x) using floats

PARALLEL ARCHITECTURES FOR HEP EVENT PROCESSING

HEP Application



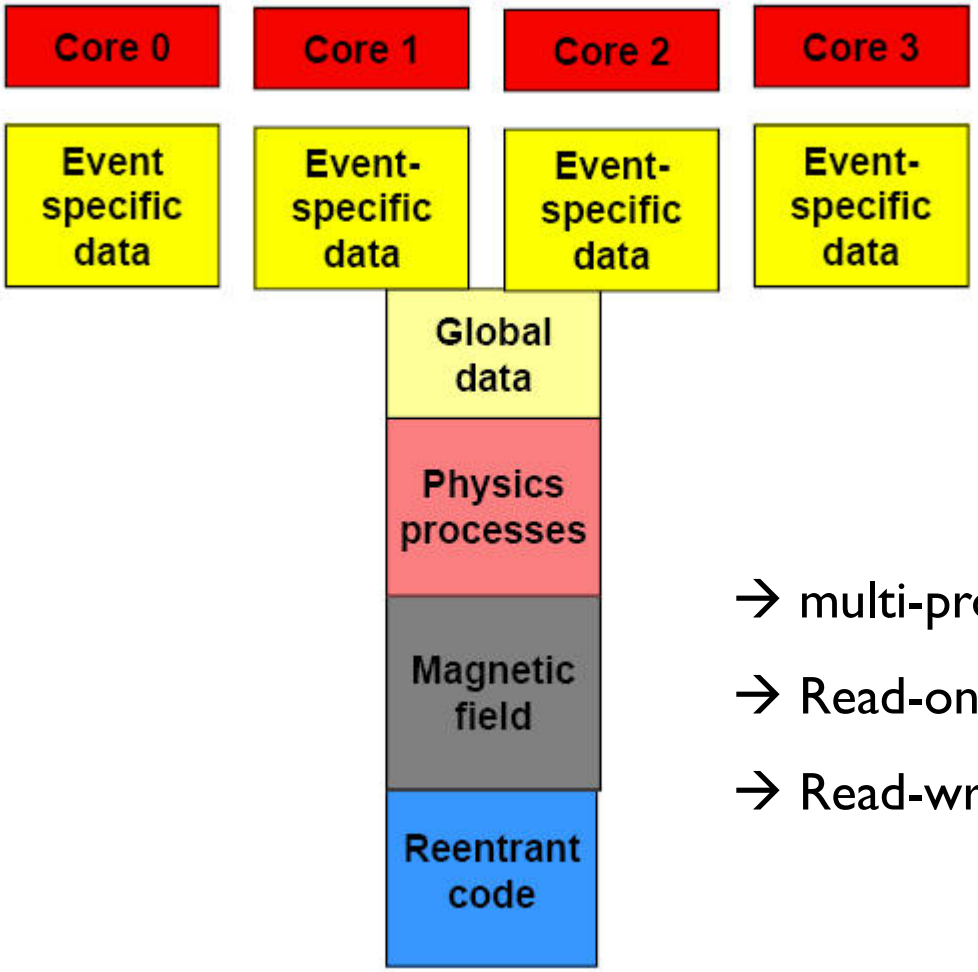
Experience and requirements

- Complex and dispersed "legacy" software
  - » Difficult to manage/share/tune resources (memory, I/O): better to rely in the support from OS and compiler
  - » Coding and maintaining thread-safe software at user-level is hard
  - » Need automatic tools to identify code to be made thread-aware
    - Geant4: 10K lines modified! (**thread-parallel** Geant4)
    - Not enough, many hidden (optimization) details
- "Simple" multi-process seems more promising
  - » ATLAS: fork() (exploit copy-on-write), shmem (needs library support)
  - » LHCb: python
  - » PROOF-lite
- Other limitations are at the door (I/O, communication, memory)
  - » Proof: client-server communication overhead in a single box
  - » Proof-lite: I/O bound >2 processes per disk
  - » Online (Atlas, CMS) limit in in/out-bound connections to one box

Event parallelism

Opportunity: Reconstruction Memory-Footprint shows large condition data

How to share common data between different process?



CMS:
1GB total Memory Footprint
Event Size 1 MB
Sharable data 250MB
Shared code 130MB
Private Data 400MB !!

→ multi-process vs multi-threaded

→ Read-only: Copy-on-write, Shared Libraries

→ Read-write: Shared Memory, sockets, files

Exploit Copy on Write (COW)

See Sebastien Binet's talk @ CHEP09

- Modern OS share read-only pages among processes dynamically
  - » A memory page is copied and made private to a process only when modified
- Prototype in Atlas and LHCb
  - » Encouraging results as memory sharing is concerned (50% shared)
  - » Concerns about I/O (need to merge output from multiple processes)

Memory (ATLAS)
One process: *700MB VMem and 420MB RSS*
COW:
(before) evt 0: private: 004 MB | shared: 310 MB
(before) evt 1: private: 235 MB | shared: 265 MB
. . .
(before) evt50: private: 250 MB | shared: 263 MB

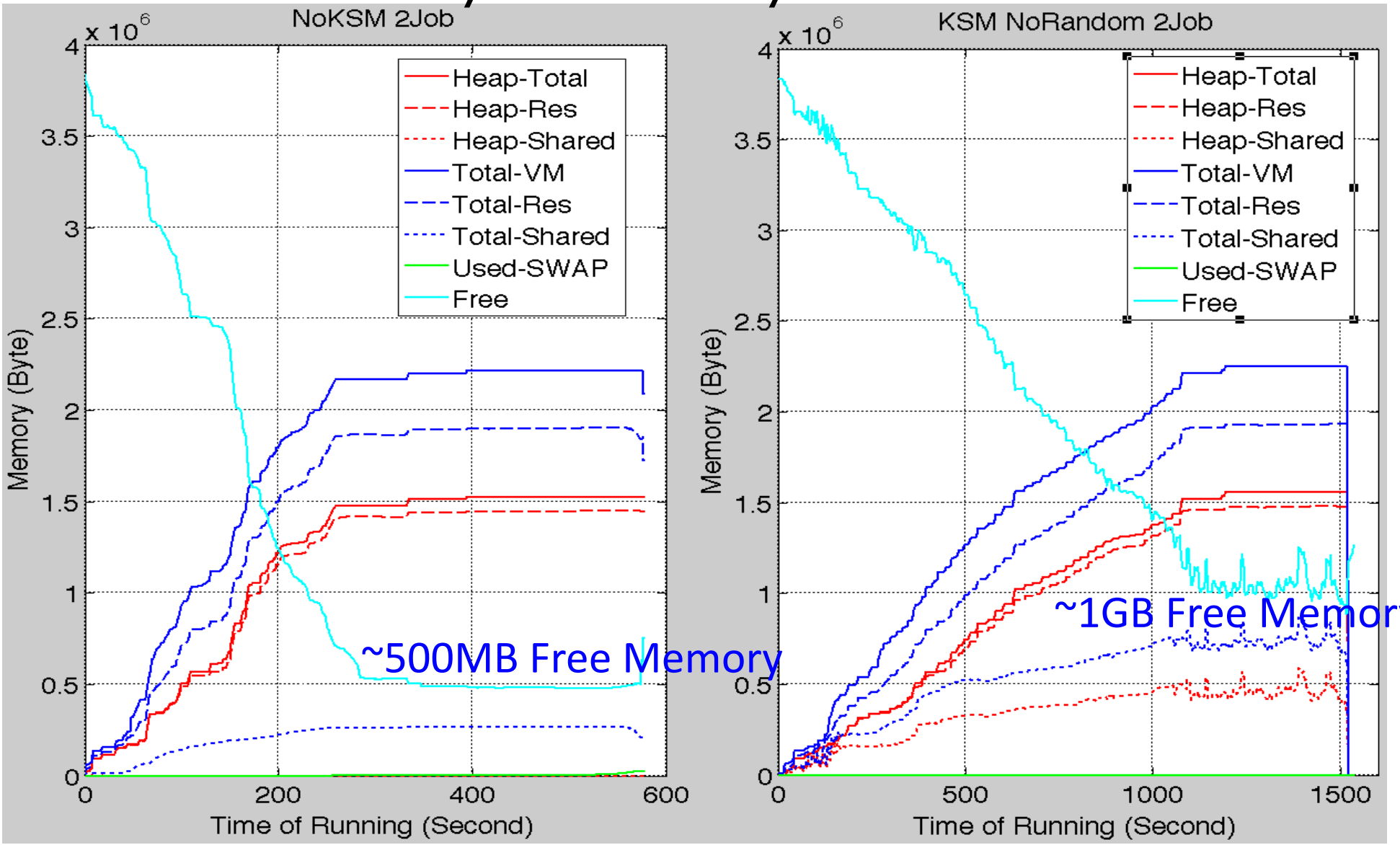
Exploit "Kernel Shared Memory"

- KSM is a linux driver that allows dynamically sharing identical memory pages between one or more processes.
  - » It has been developed as a backend of KVM to help memory sharing between virtual machines running on the same host.
  - » KSM scans just memory that was registered with it. Essentially this means that each memory allocation, sensible to be shared, need to be followed by a call to a registry function.
- Test performed "retrofitting" TCMalloc with KSM
  - » Just one single line of code added!
- CMS reconstruction of real data (Cosmics with full detector)
  - » No code change
  - » 400MB private data; 250MB shared data; 130MB shared code
- ATLAS
  - » No code change
  - » In a Reconstruction job of 1.6GB VM, up to 1GB can be shared with KSM

ATLAS results by Yushu Yao

Comparing FREE Memory (light blue) when running 2 jobs

KSM frees 500MB System Memory



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

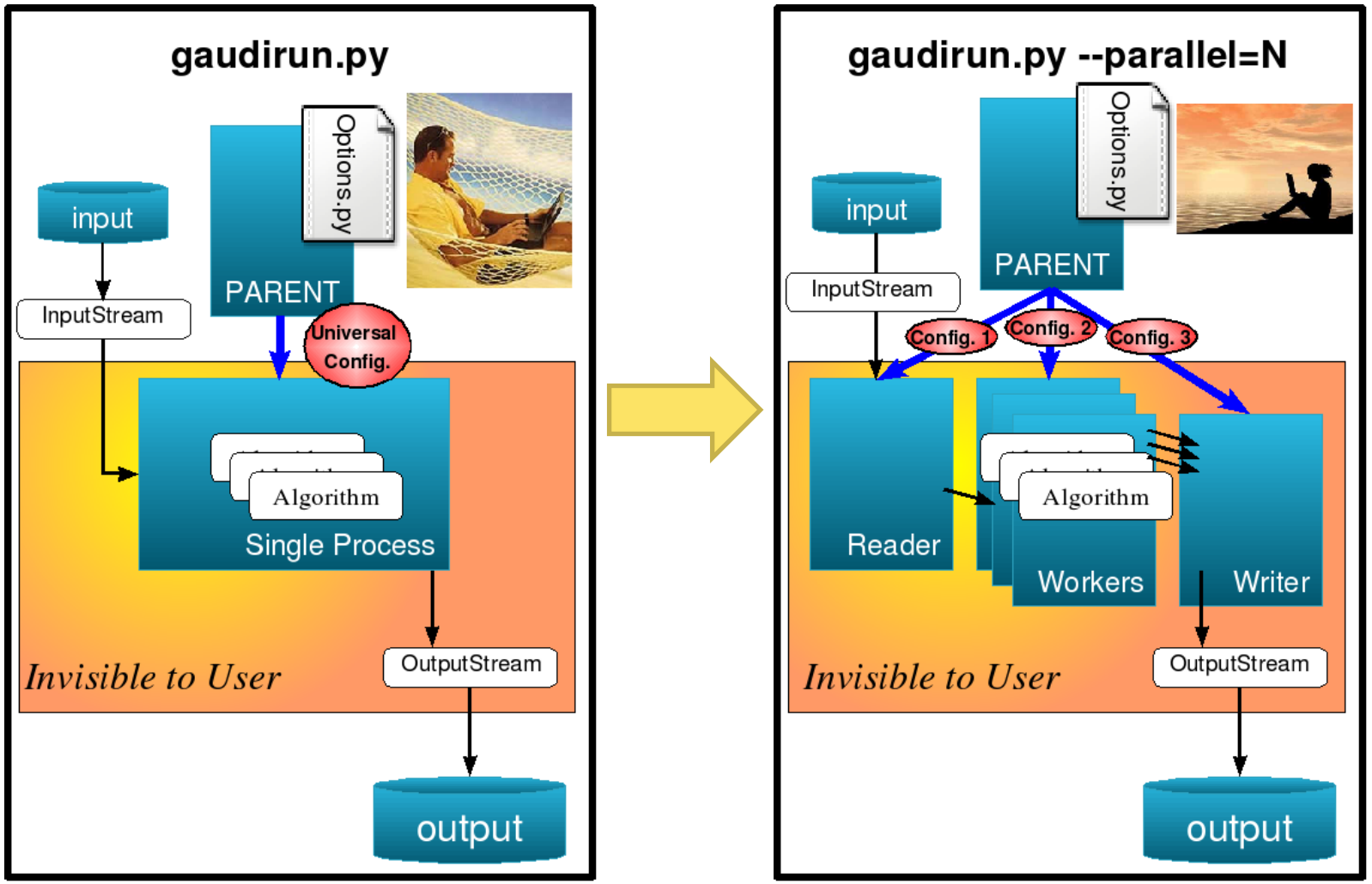
By segregating read-write data members, large read-only memory chunks are formed. Copy-On-Write does not replicate those read-only chunks. (Geant4MT + COW)

- 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

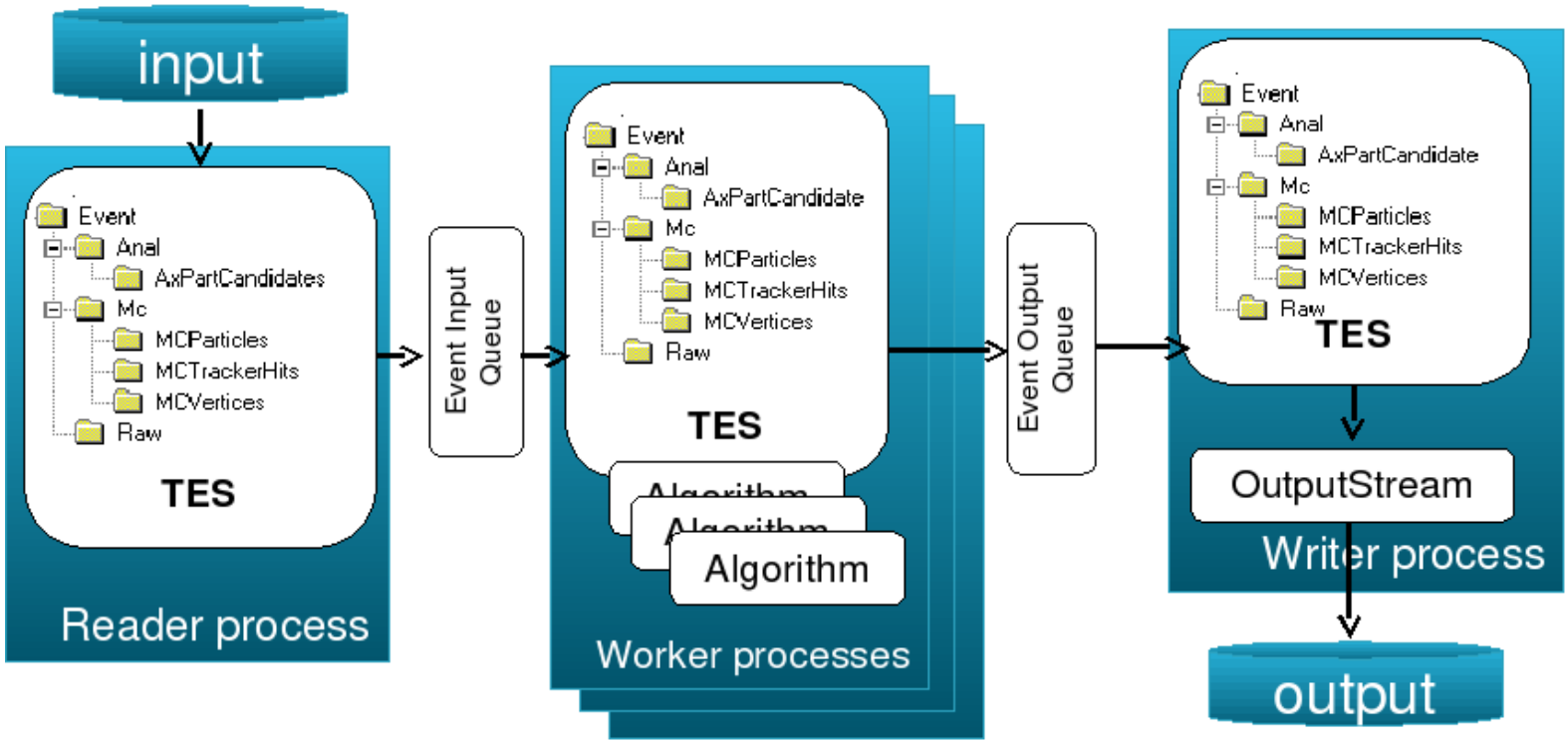
Tested on `fullCMS bench1.g4` with 24 workers and 4000 events per worker (electromagnetics).

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

Parallelization of Gaudi Framework



GaudiPython Parallel: Specifics



- TES transferred by Serialise/Deserialise
- Auto-optimisation of data transfer

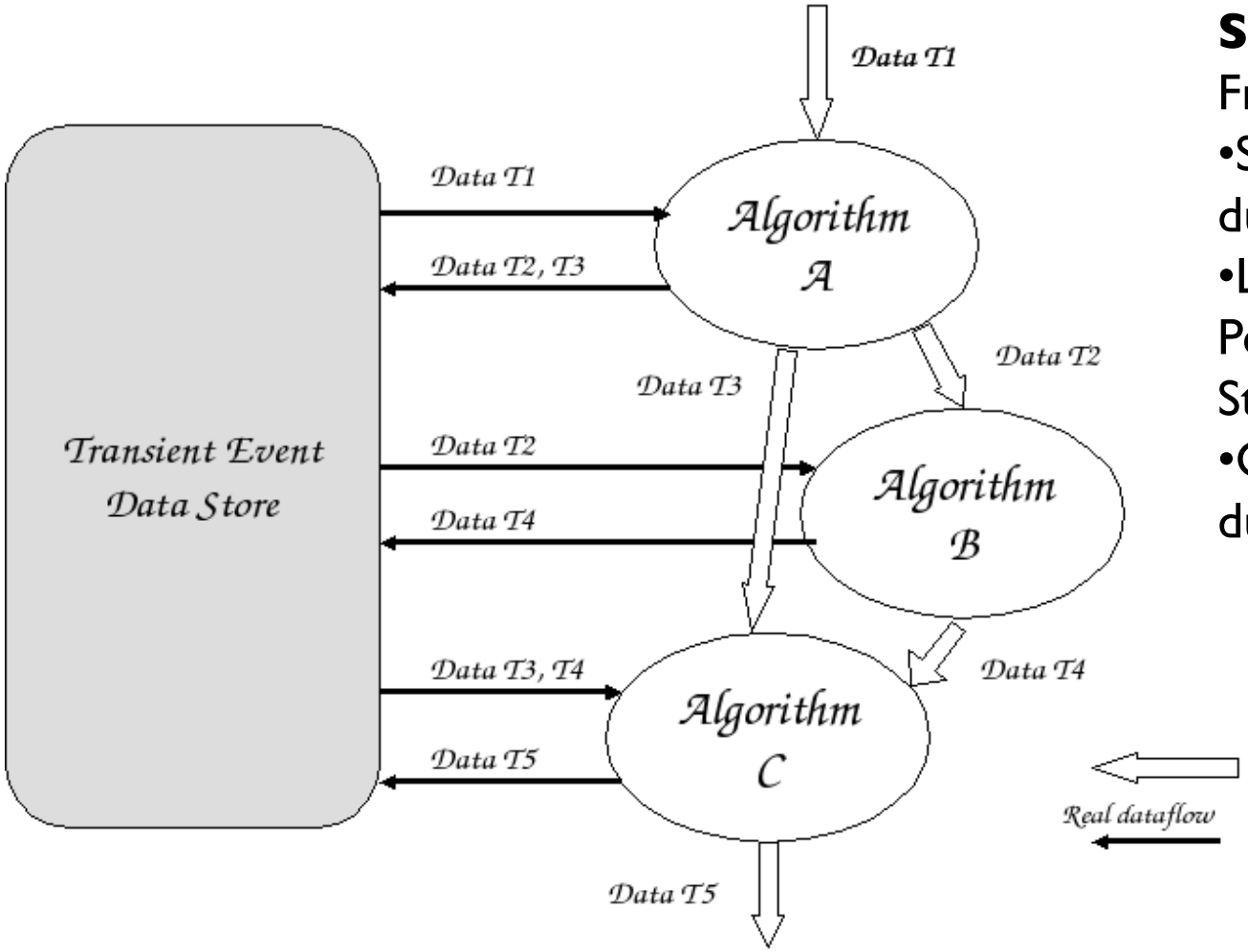
GaudiPython Parallel

- Reconstruction (Brunel)
  - » FEST-2009-Data.py : 1000 Events
    - From $BRUNELOPTS

Run Type	CPU%	T_elapsed	T_init	T_run	Speedup
Serial		1334	47	1287	1
parallel=5		317	47	280	4.6

- ~1.5s/event
  - Parallel Overhead 3%
  - Speedup Near-Linear

Gaudi : HEP Event Processing



- **Transient Event Store** : Part of Framework
- Stores *DataObjects* during processing
- Loaded from Persistent Storage at Start
- Constantly modified during run

HEP data processing: beyond event-by-event processing

- No need of a coherent event state:
  - » The "Event" occupies a small part of the resident memory
    - Small overhead to keep several event in memory at once
  - » Algorithms
    - read specific event-fragments, store new fragments: **never modify existing ones**
    - Dependencies are known: algorithms can be scheduled in parallel
    - We can distribute algorithms among cores, improving data and code locality in caches
  - » Storage:
    - "Event-Fragments" map root branches: independent of each other
    - Can be streamed as soon as created, no need to wait for the full event to be ready
- Conditions shared among events and (some) algorithms
  - » Event parallelism will profit of coherent shared conditions
  - » Few conditions are used by different algorithms
  - » Algorithm parallelism can make conditions private to each one\

No work started yet: opportunity for SuperB leveraging Babar code and data

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.

Parallel MINUIT *A. L. and Lorenzo Moneta*

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

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

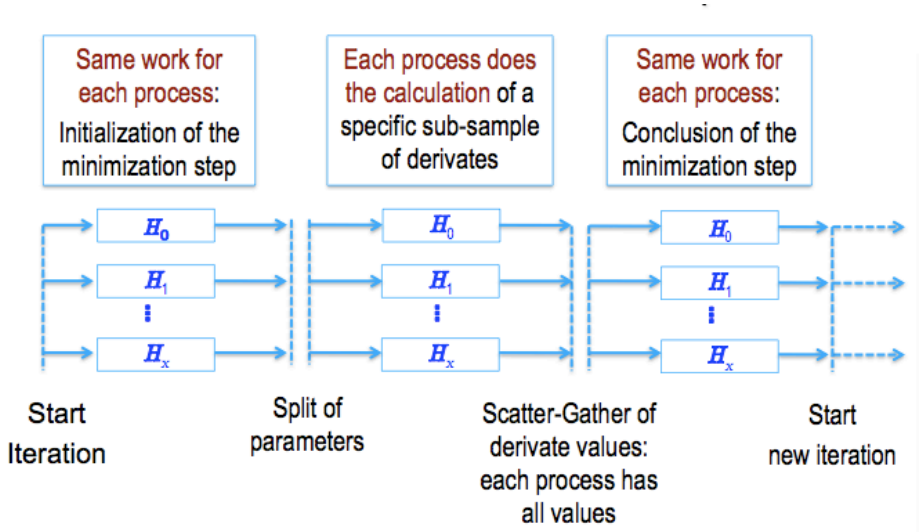
$$NLL = \ln\left(\sum_{j=1}^{s} n_j\right) - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j^i\right)$$

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

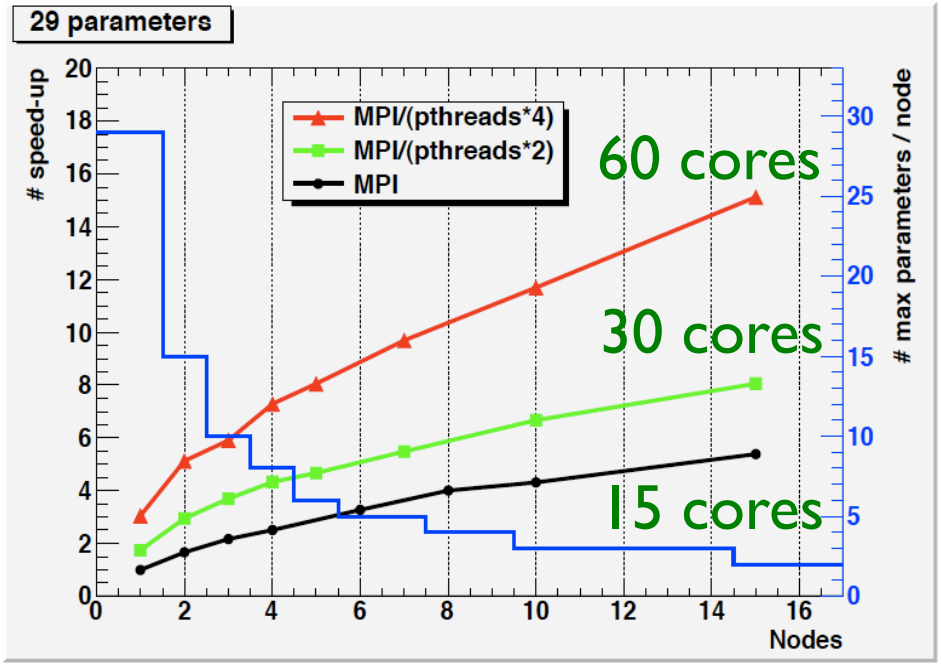
– Two strategies for the parallelization of the gradient and NLL calculation:

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**



Minuit Parallelization – Example

– Waiting time for fit to converge down from several days to a night (Babar examples)
  » iteration on results back to a human timeframe!



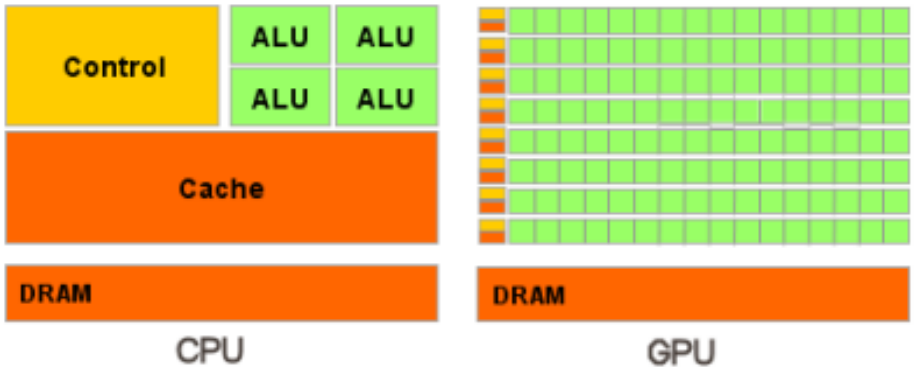
Parallelism implementation

- High grain parallelism need to be implemented using multi-thread
- Low level multi-threading is well established at OS and posix level (now also in C++0x std)
- At user level many implementation technologies exist
  - » Intel alone today proposes 4/5 different technologies!
  - » Watch out for compatibility with GGPU approaches
- A serious investigation of the technology trends and of what best fit our use cases is required before starting to fill up the code with *pragmas* and not standard keywords

Outlook

- Recent progress shows that we shall be able to exploit next generation multicore with "small" changes to HEP code
  - » Exploit copy-on-write (COW) in multi-processing (MP)
  - » Develop an affordable solution for the sharing of the output file
  - » Leverage Geant4 experience to explore multi-thread (MT) solutions
- Continue optimization of memory hierarchy usage
  - » Study data and code "locality" including "core-affinity"
- Expand Minuit experience to other areas of "final" data analysis, such as machine learning techniques
  - » Investigating the possibility to use GPUs and custom FPGAs solutions
- "Learn" how to run MT/MP jobs on the grid
  - » workshop at CERN, June 2009: http://indico.cern.ch/conferenceDisplay.py?confId=56353
  - » Tests ongoing with CERN/IT (just got two machines with a dedicated queue)

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)

Explore new Frontier of parallel computing

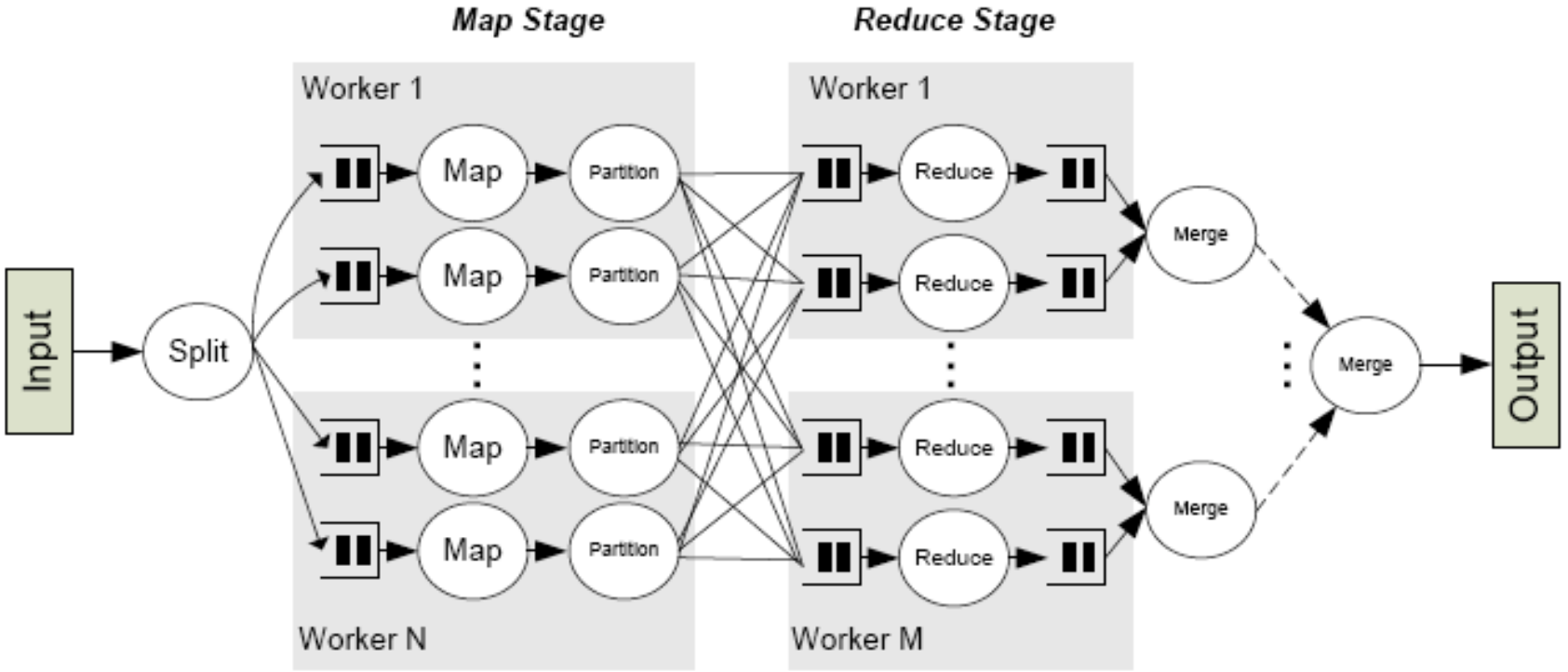
- Hardware and software technologies may come to the rescue in many areas
  - » We shall be ready to exploit them
- Scaling to many-core processors (96-core processors foreseen for next year) will require innovative solutions
  - » MP and MT beyond event level
  - » Fine grain parallelism (OpenCL, custom solutions?)
  - » Parallel I/O
- Possible use of GPUs for massive parallelization
- But, Amdahl docet, algorithm concept have to change to take advantages on parallelism: think parallel, write parallel!

BACKUP

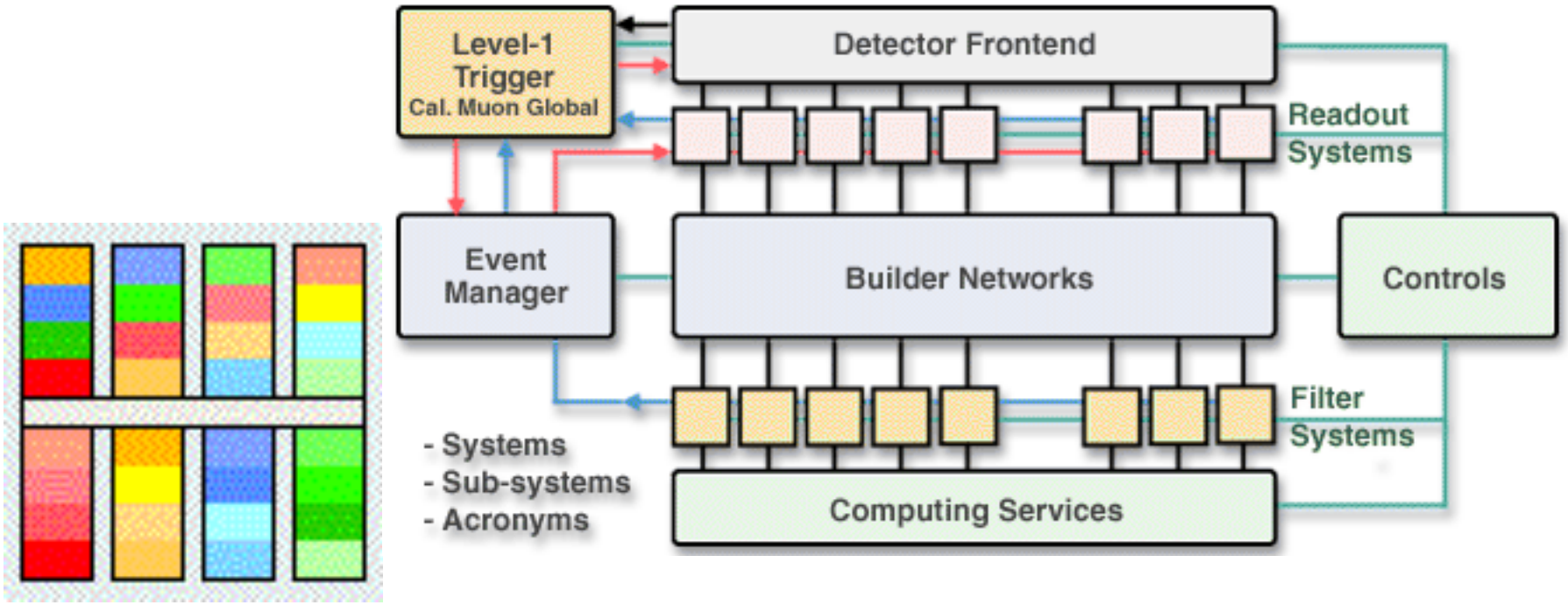
Map/Reduce

Established, yet evolving paradigm

- » Original (google, Hadhoop) Map/Reduce takes a set of *input key/value pairs, and* produces a set of *output key/value pairs.*
- » Many implementations
- » *Map (written by the user)*



Event Building!



Map: Detector frontend assign event-id to each fragment
DAQ dispatch all fragment with same id to a given filter node
Reduce: filter node assemble the event and process it