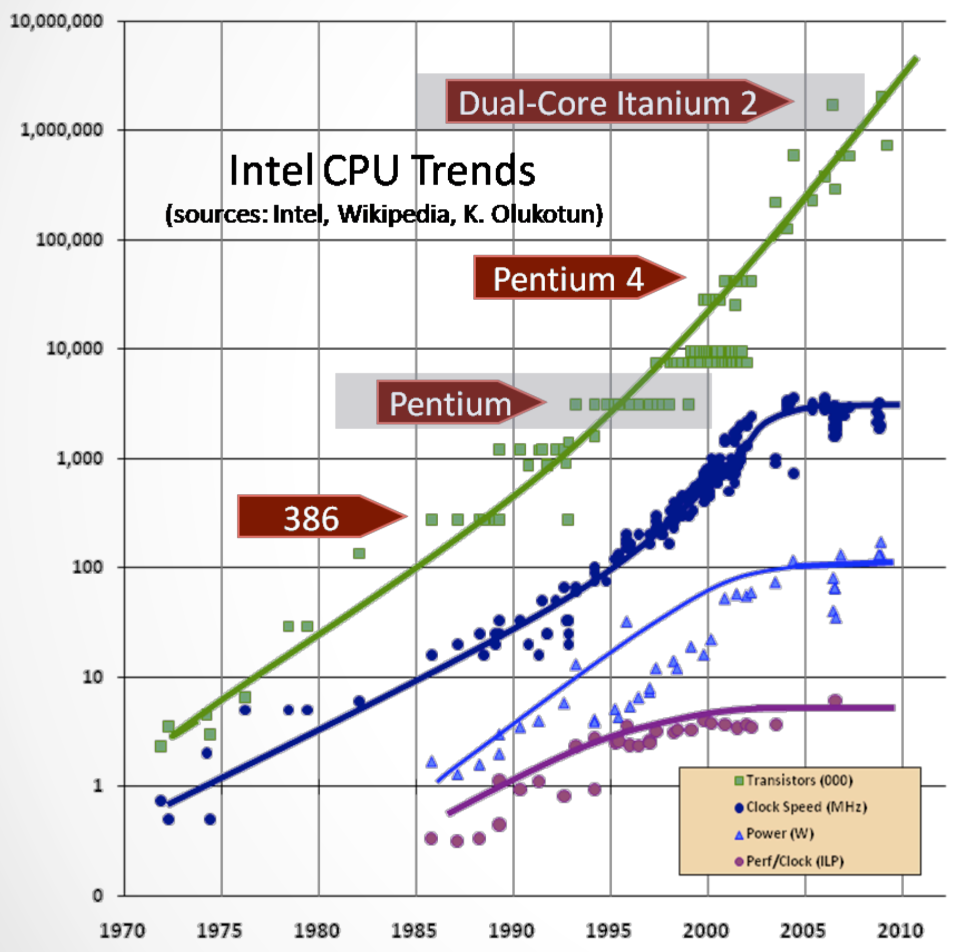
Future Directions For Software Tools

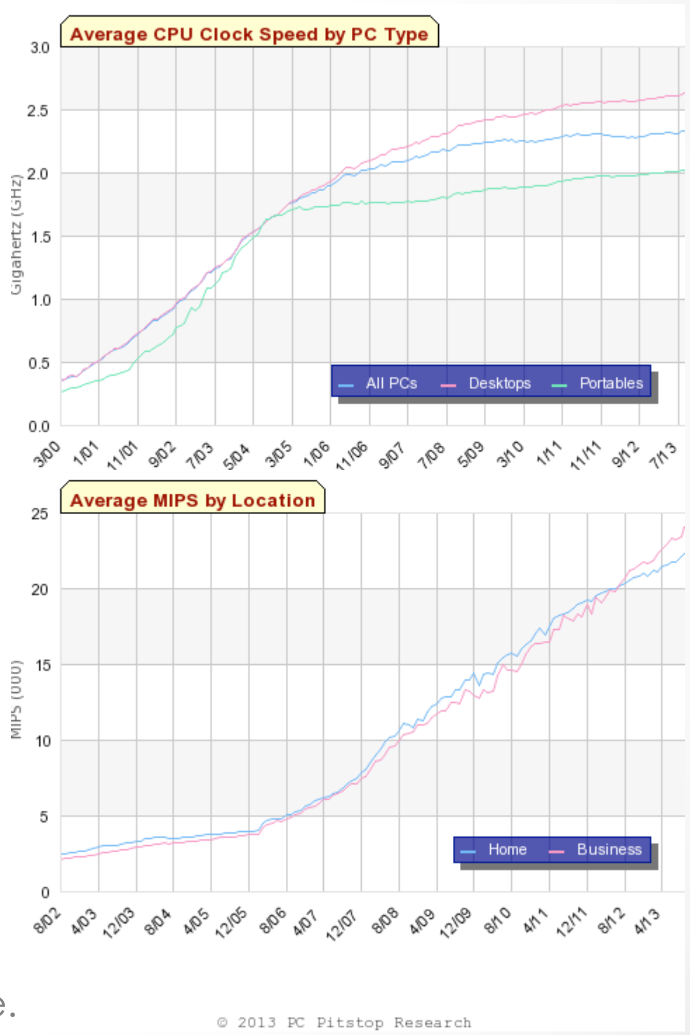
Philippe Canal, Fermilab

Outline

- Hardware evolution
- Software implications
- Review of (some) current (and past) efforts
- Common Libraries
  - ROOT, Geant4
- Conclusions

Game Changer

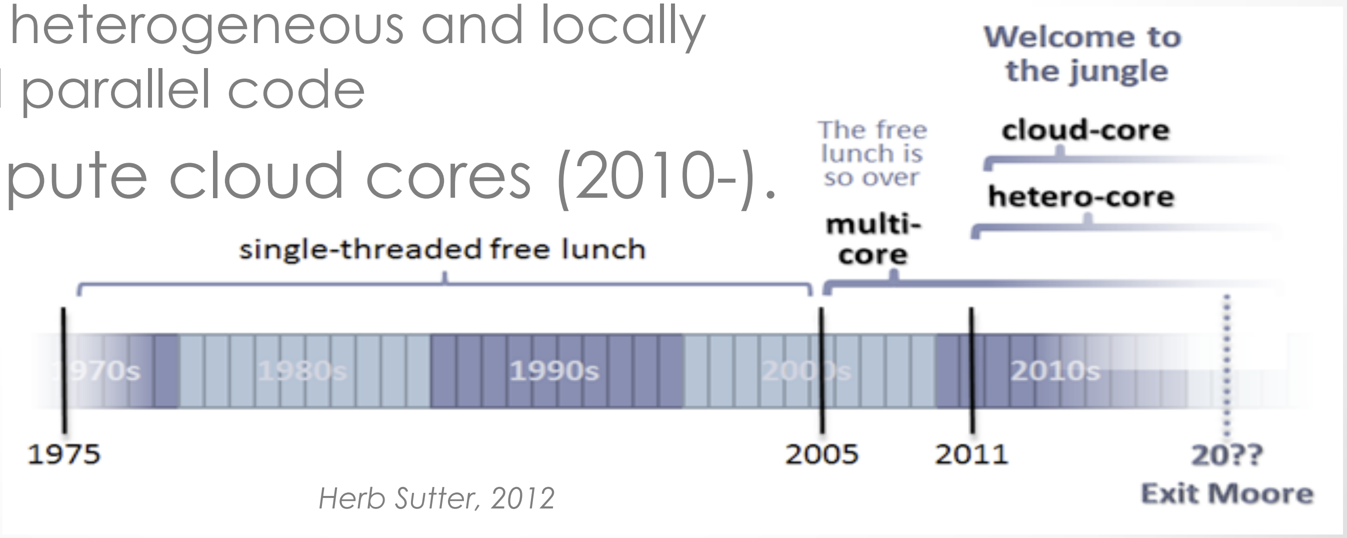




Clock Speed plateaued but MIPS continue to increase.

Cores Capabilities

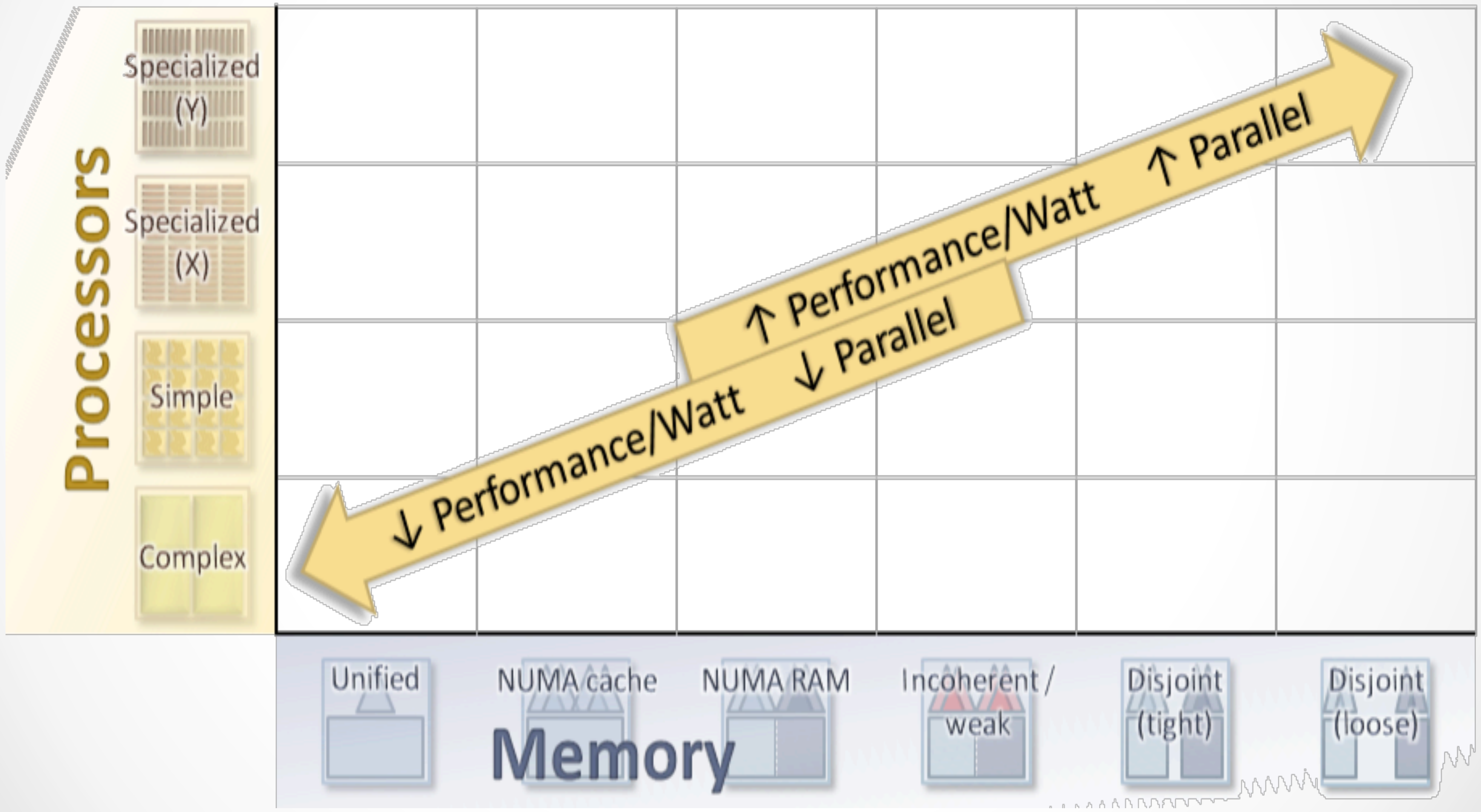
- Unicore
  - Free lunch, same exe just run faster on new hardware
- Multi-core (2005-) ; Many-core (2012-)
  - Must write parallel code ; must write very parallel code.
  - Or memory available must scale with number of cores.
- Heterogeneous cores (2009-)
  - Must write heterogeneous and locally distributed parallel code
- Elastic compute cloud cores (2010-).



Memory Architectures

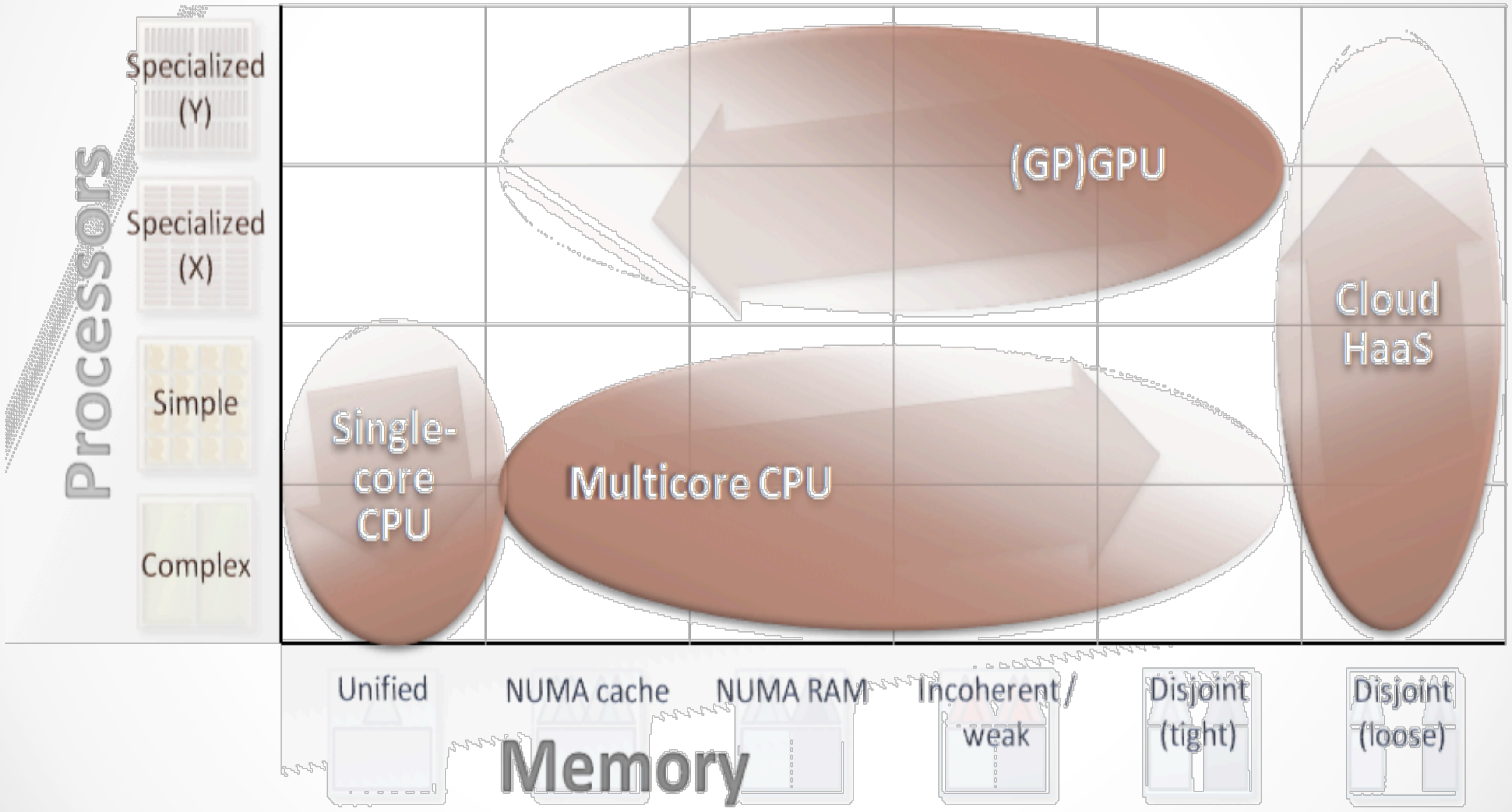
- Unified Memory
  - Concerns: locality, access order
- Non Uniform Memory Access cache
  - Concerns: locality, layout
- NUMA RAM
  - Examples: bladed servers, SMP desktop, newer GPGPU
  - Concerns: copying over slower links
- *Incoherent and weak memory*
  - *Examples: PowerPC, ARM*
  - *Concerns: explicit synchronization*
- Disjoint but tightly coupled
  - Examples: older GPGPU
  - Concerns: copying
- Disjoint and loosely coupled
  - Examples: Grids, Clouds
  - Concerns: reliability (of nodes) and latency

Charting The Landscape



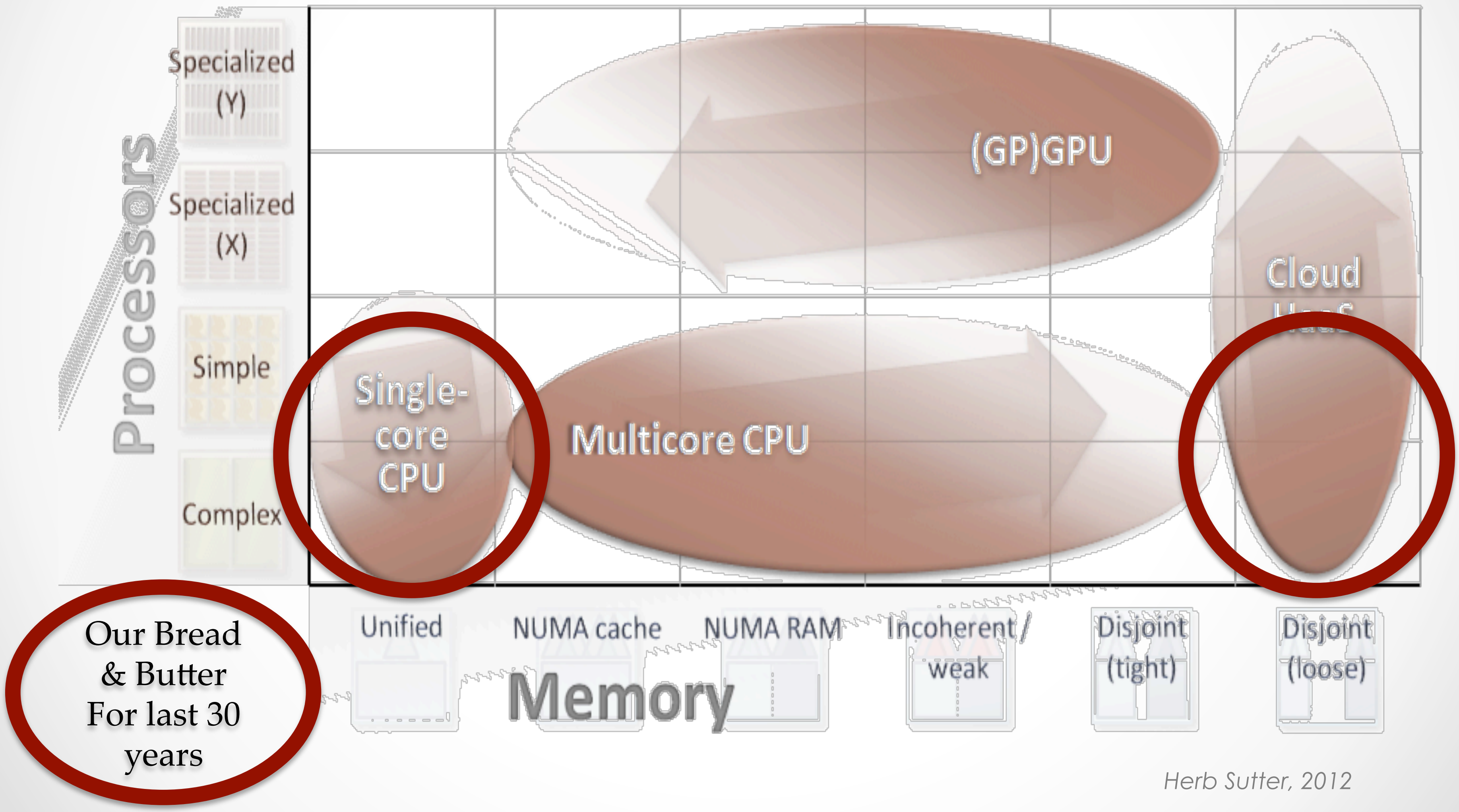
Herb Sutter, 2012

Filling The Landscape



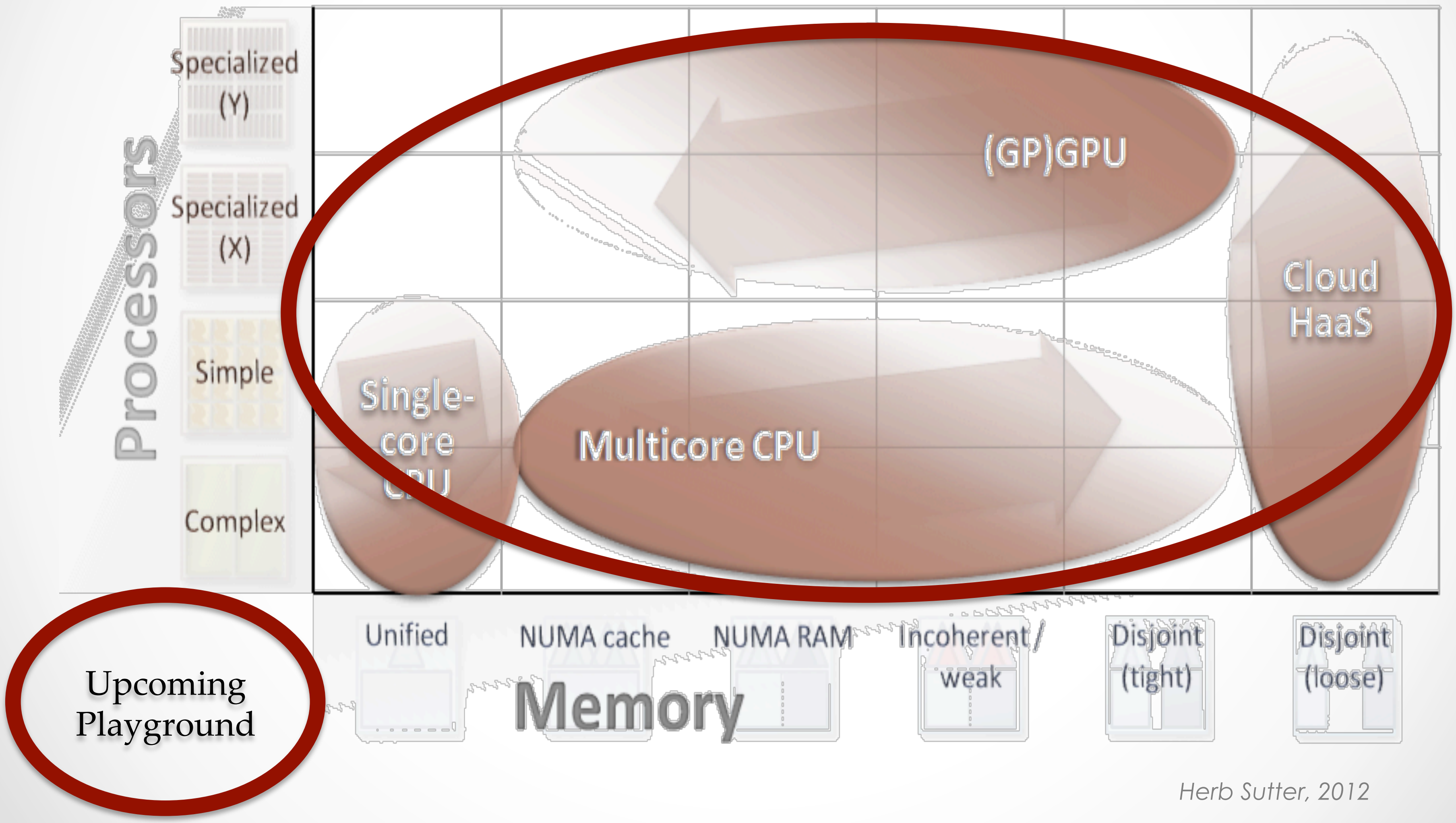
Herb Sutter, 2012

Filling The Landscape



Herb Sutter, 2012

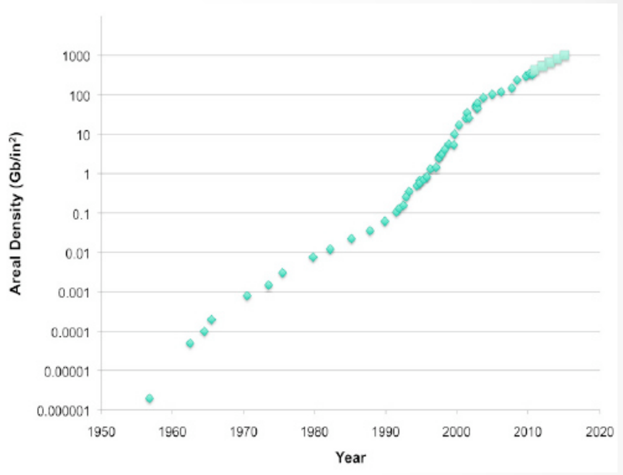
Filling The Landscape

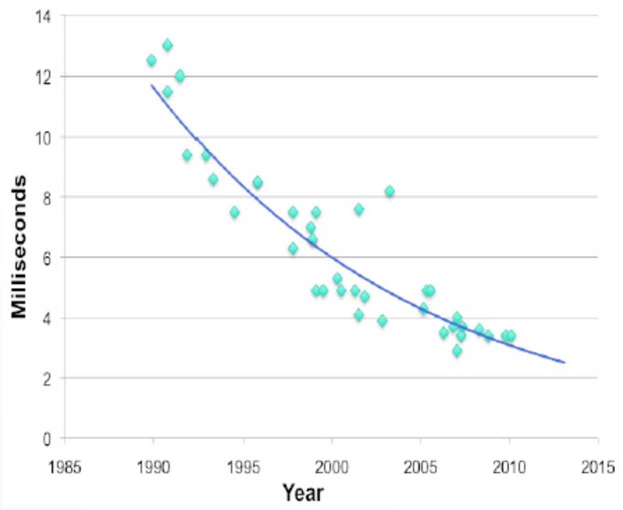


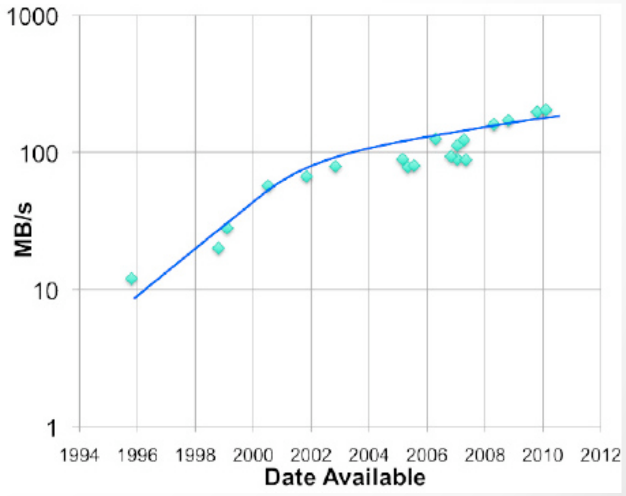
Herb Sutter, 2012

Disk Trends

- Area density and throughput somewhat plateauing
- Latencies decreasing
- SSD mainstream, added to multi-tier storage solution



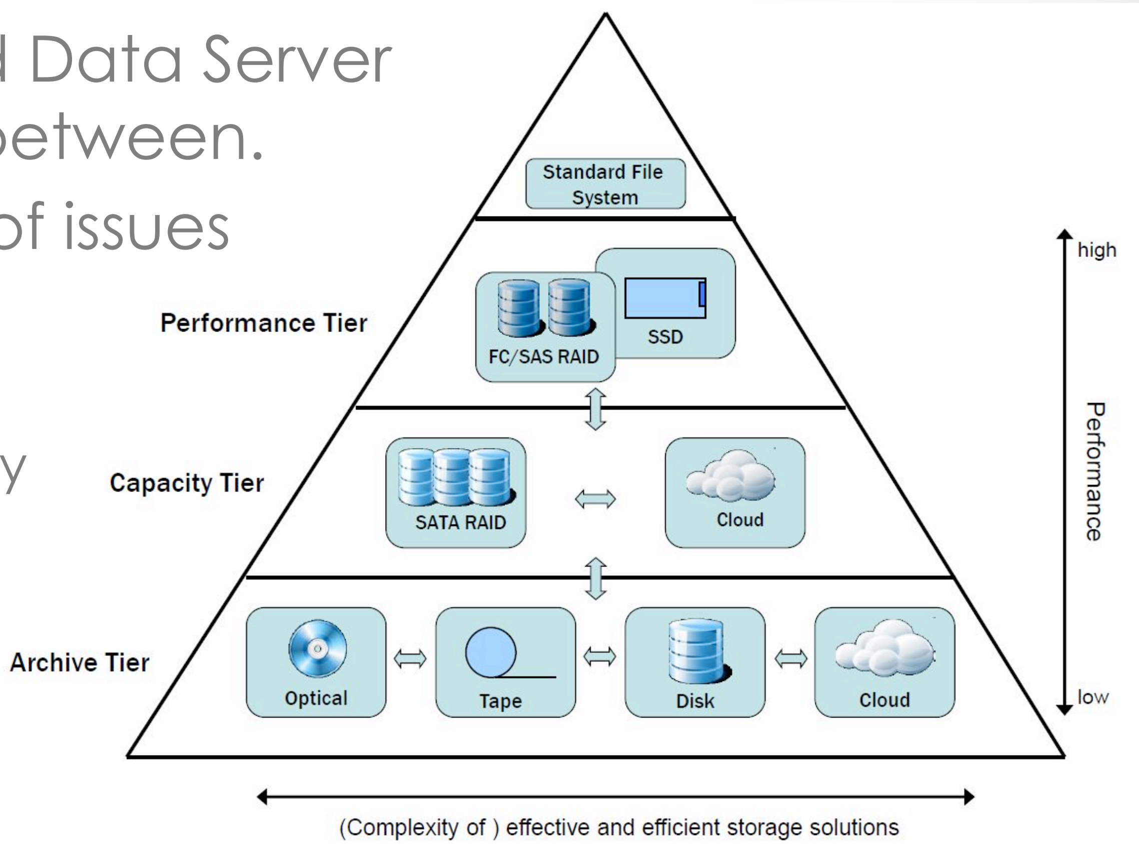




Jean-Jacques Maleval, 2011

Disk Hierarchy

- Similar to CPU/Memory hierarchy
- From SSD, HDD to
  Globally Distributed Data Server
  and everything in between.
- Same large range of issues
  - Latency
  - Copying
  - Reliability/Availability



What It Means For Software Dvpt

- Code/Libraries will need to become aware of, if not made for using
  - heterogeneous cores/memory/disk
  - and non-local cores/memory/disk
- Efficiency and performance optimization will become even more important and more complicated
  - Layer of latency, bandwidth
  - High variability of capability of cores
  - Eventual plateauing when Moore's Law ends
- Might spell the end of compile-once-run-everywhere
  - As heterogeneity increase the need for just-in-time recompilation will increase

Heterogeneous Programming

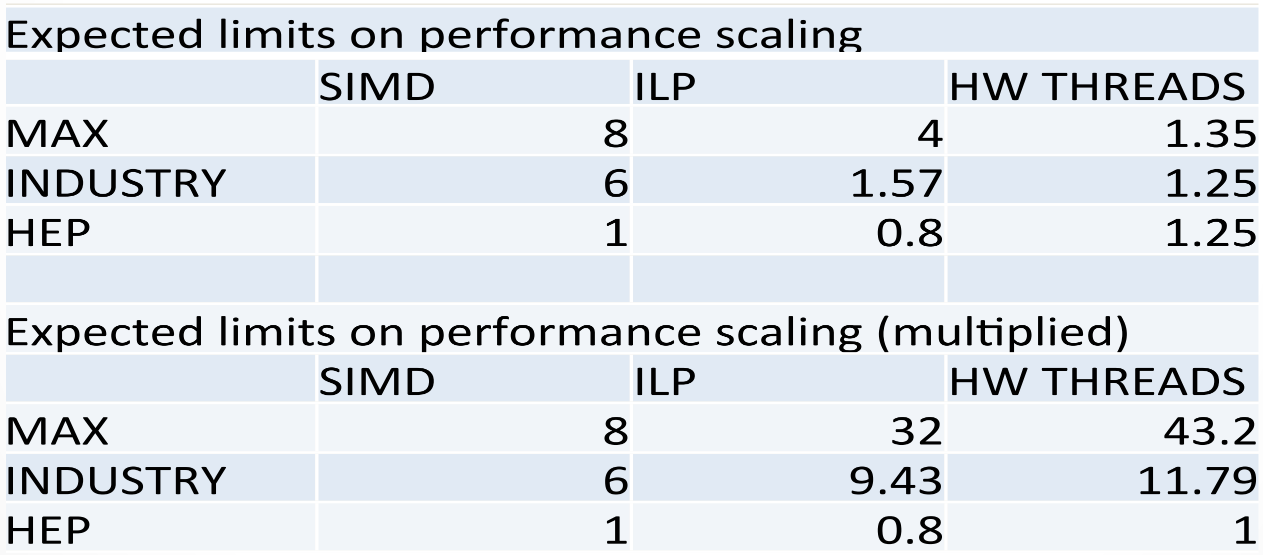
- Currently requires special handling
    - #pragma (OpenACC, OpenMP, MPI, etc.)
    - Libraries, Toolkits, scripts (TBB, MKL, ArBB, Vc, VDT, etc.)
    - 'Special' Language (Cuda, OpenCL, Cilk, etc.)
- Future versions of mainstream languages will adapt (or become marginalized).
    - For example C++11 (finally!) standardized thread behavior
    - PyCuda, mpi4py, etc.
    - Rootbeer, Java run-time compiler for GPU
- New languages designed with concurrency at their core
    - Go, Erlang
    - Becoming more popular
    - But we have large body of existing code

Level Of Parallelism

- Macro
  - Multi-sites
    - Currently using Grid and Clouds often driven by experiment controlled layer
  - Multi-node
  - Multi-socket
  - Multi-core
    - Most often also using Grids/Clouds, OpenMP, MPI
    - Existing solution limited by decrease in memory per core
- Micro
  - Hardware threading, Instruction Level Parallelism (ILP)
    - On some platforms (GPU for examples), threads need to perform similar operations on different data for maximum throughput
  - Instruction Pipelining
  - Vectors Processing Units

HEP/NPP & Micro Parallelism

- Our code underutilize (even current) hardware



Expected limits on performance scaling			
	SIMD	ILP	HW THREADS
MAX	8	4	1.35
INDUSTRY	6	1.57	1.25
HEP	1	0.8	1.25
Expected limits on performance scaling (multiplied)			
	SIMD	ILP	HW THREADS
MAX	8	32	43.2
INDUSTRY	6	9.43	11.79
HEP	1	0.8	1

OpenLab, Chep 2012

What Can Limit Micro Parallelism?

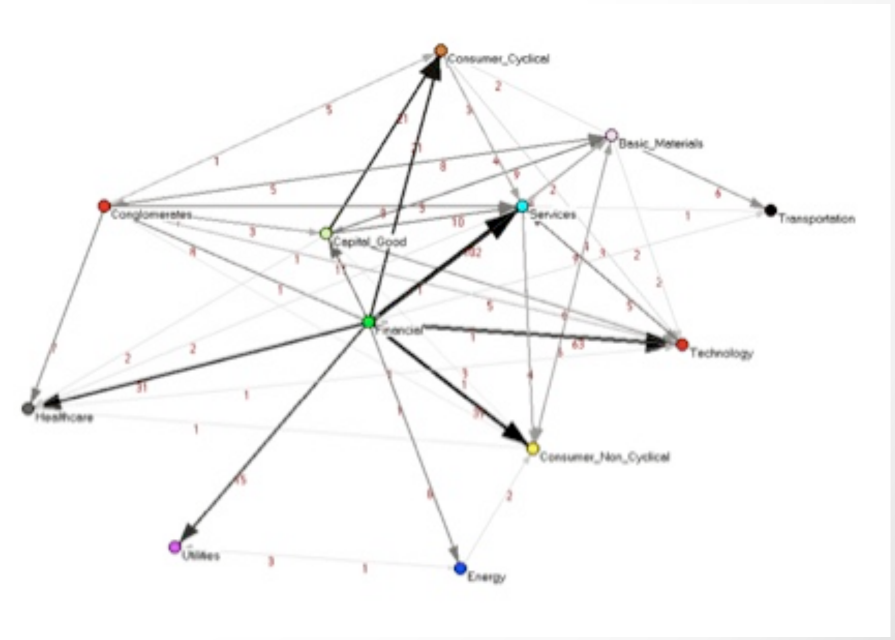
- No intrinsic vector access pattern
  - Most high level algorithm/infrastructure push one event/tracks/unit of work at unit.
  - Individual low-level algorithms do not take significant amount of time (very few exceptions)
    - So vectorizing only those helps only marginally
    - For example for standalone a CMS Geant4 simulation no individual routines takes more than a few percent of the run-time

What Can Limit Micro Parallelism?

- Too many conditional branches and virtual functions
  - In core ROOT I/O replacing switch statement and reducing ifs statement improved performance by 25%
  - In NVidia GPU, threads within a warp must stay in sync to be executed at the same time (on the same mini-core). Divergence due to ifs statement lead to essentially serializing those threads.

What Can Limit Micro Parallelism?

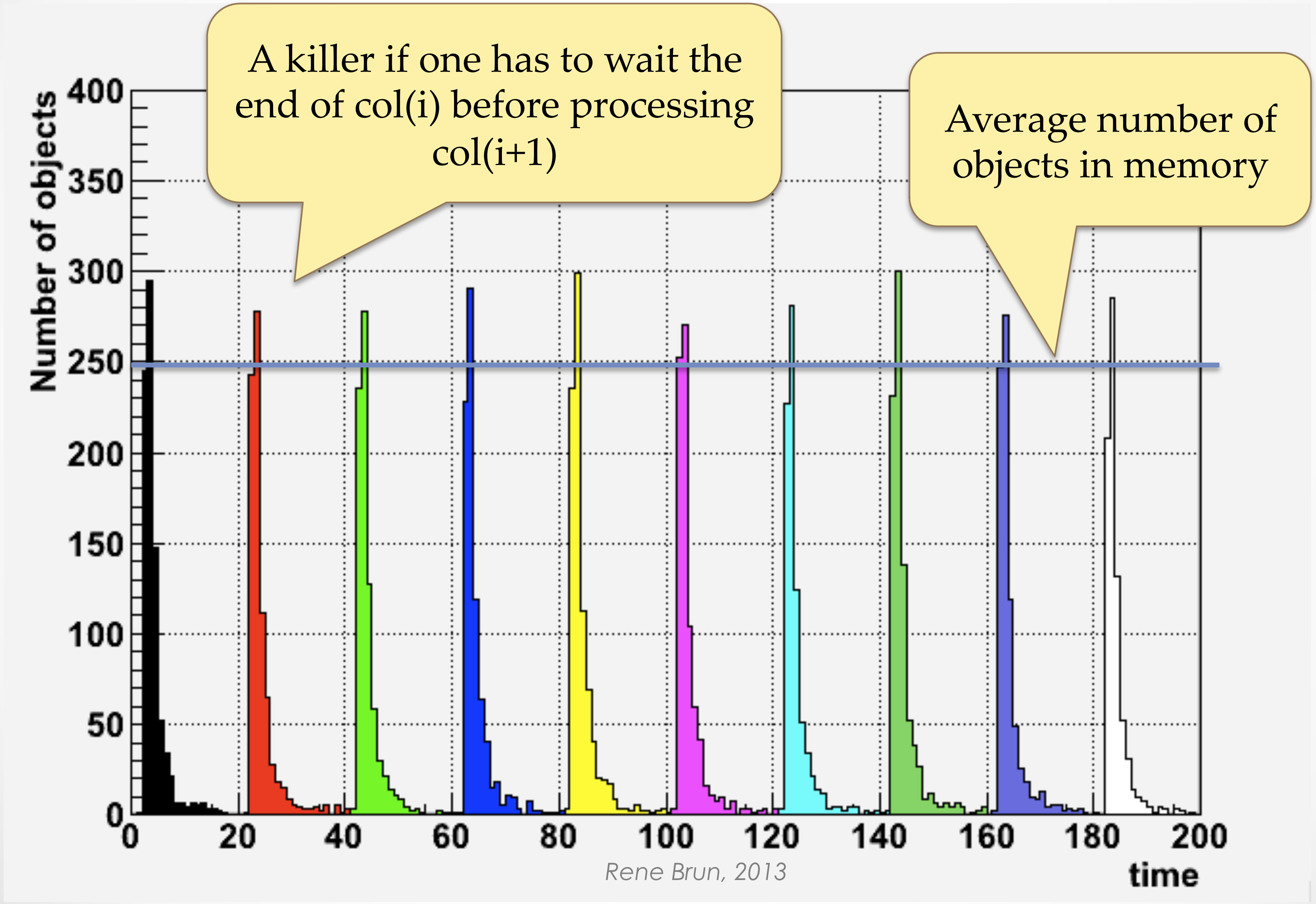
- Data dependencies between different iterations of a loop
- Memory bandwidth limitation (lack of caching)
- Indirect addressing
- If and switch statements, early loop termination
- Outer loop with outline function calls



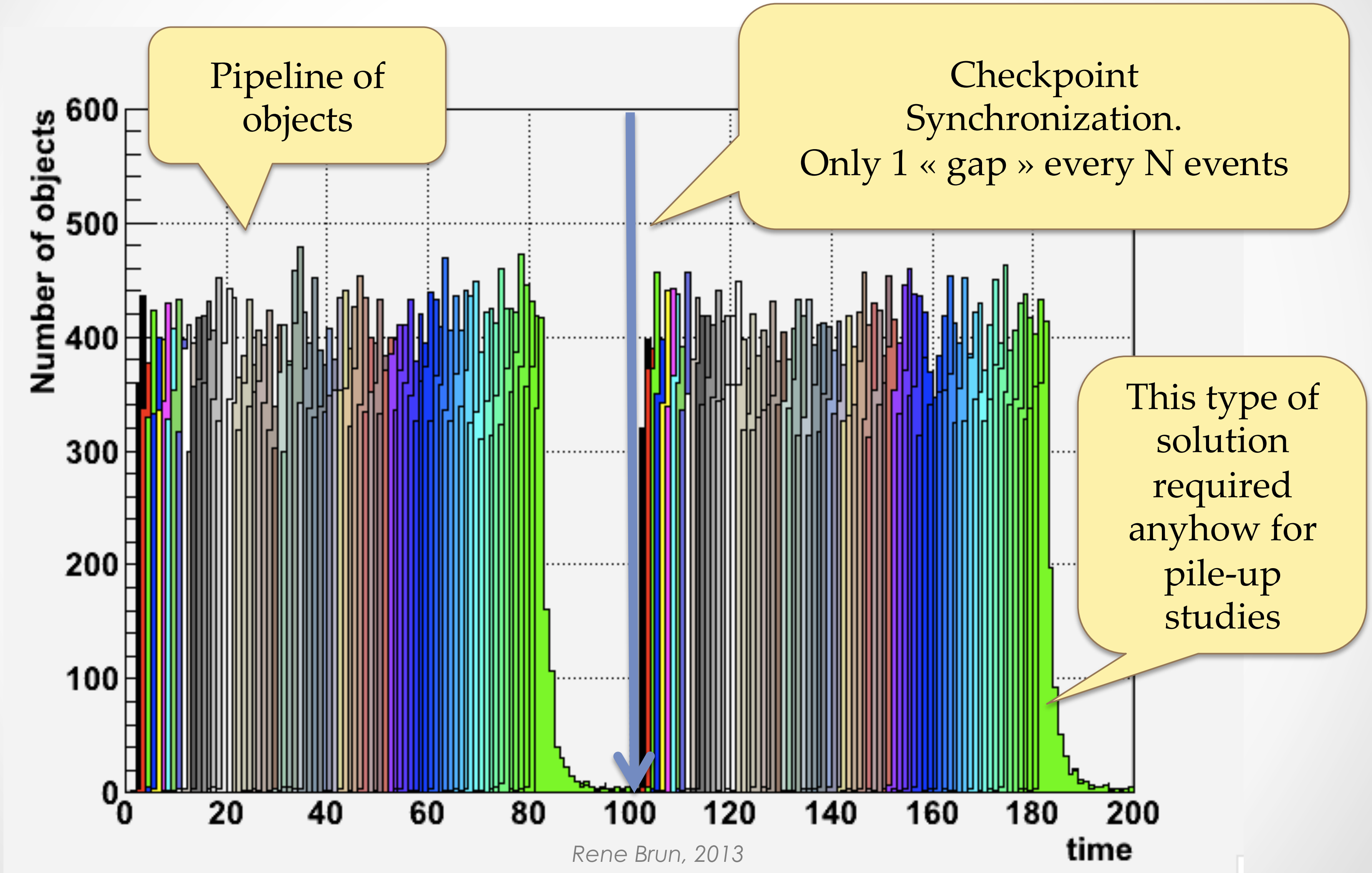
Transitions Bottlenecks

- Many potential serial points
  - End of Event
  - End of Luminosity
  - End of Run
  - Etc.
- Delaying is okay but
  - Memory constraint
- Removing is ‘better’ but much ‘harder’!
  - Some quantity must intrinsically be accumulated/calculated at each transition

Tails


A killer if one has to wait the end of col(i) before processing col(i+1)
Average number of objects in memory
Number of objects
400
350
300
250
200
150
100
50
0
0
20
40
60
80
100
120
140
160
180
200
time
Rene Brun, 2013

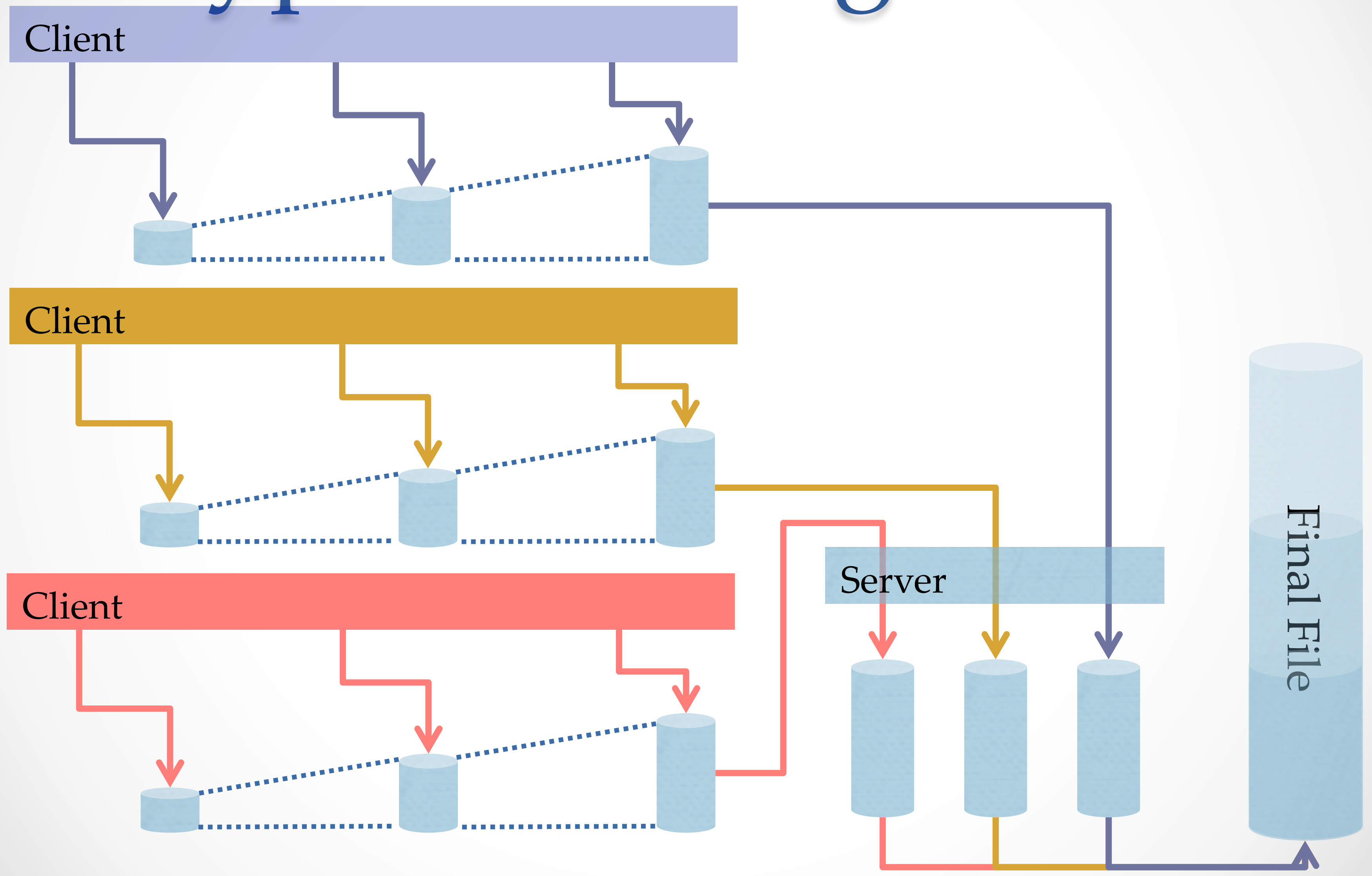
A better solution



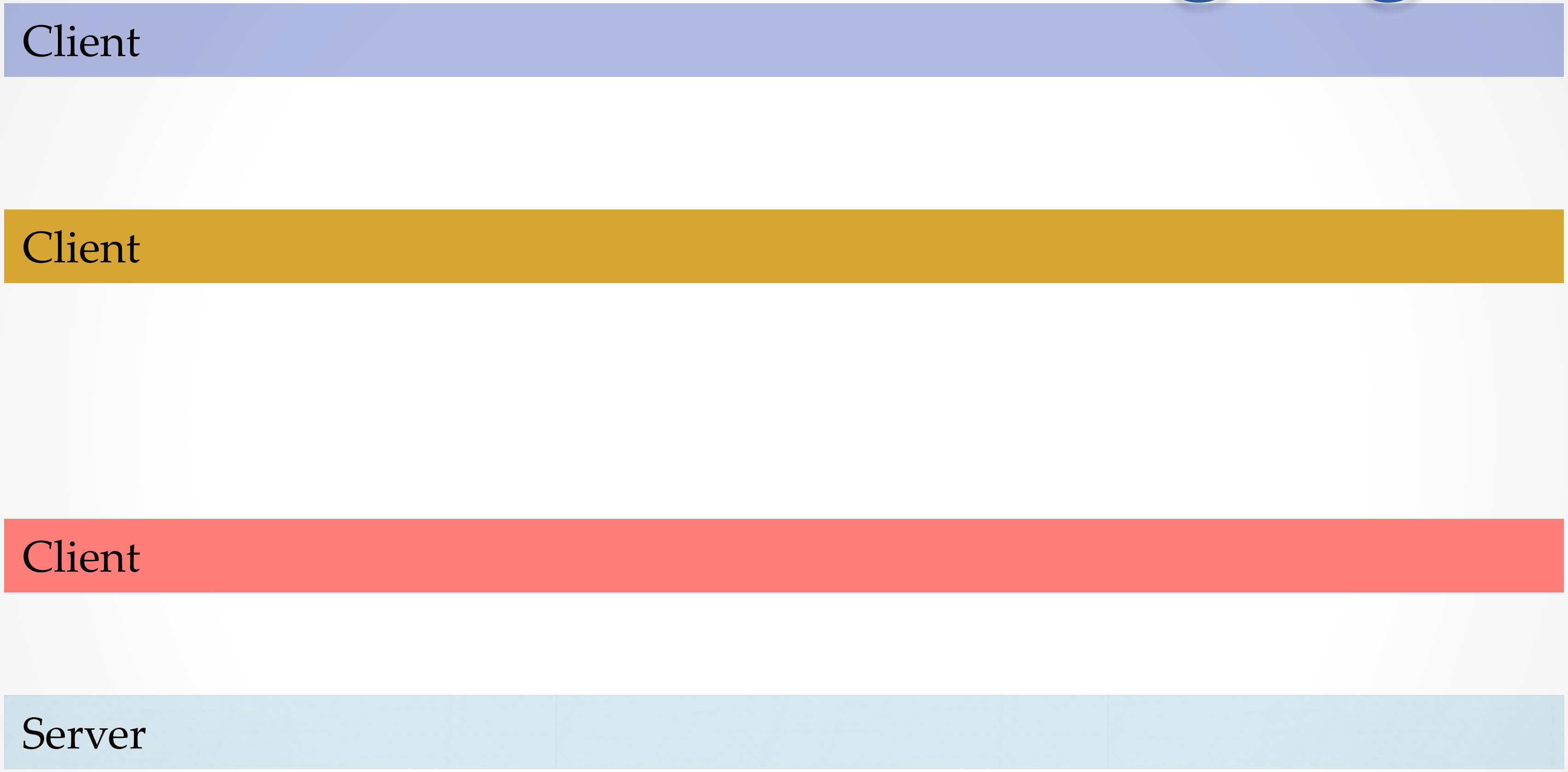
The I/O Trap

- We read/write large amount of data
- Many opportunities to become a bottleneck
  - Threads writing in single TTrees.
  - Processes writing to single local disk
  - Experiment Data Management copying files from execution node to data server
  - Merging files to avoid having too many files
- In addition, the number of disks and spindles is not increasing as fast as processing power
  - Hidden serialization for example when using whole node allocation and fork on write

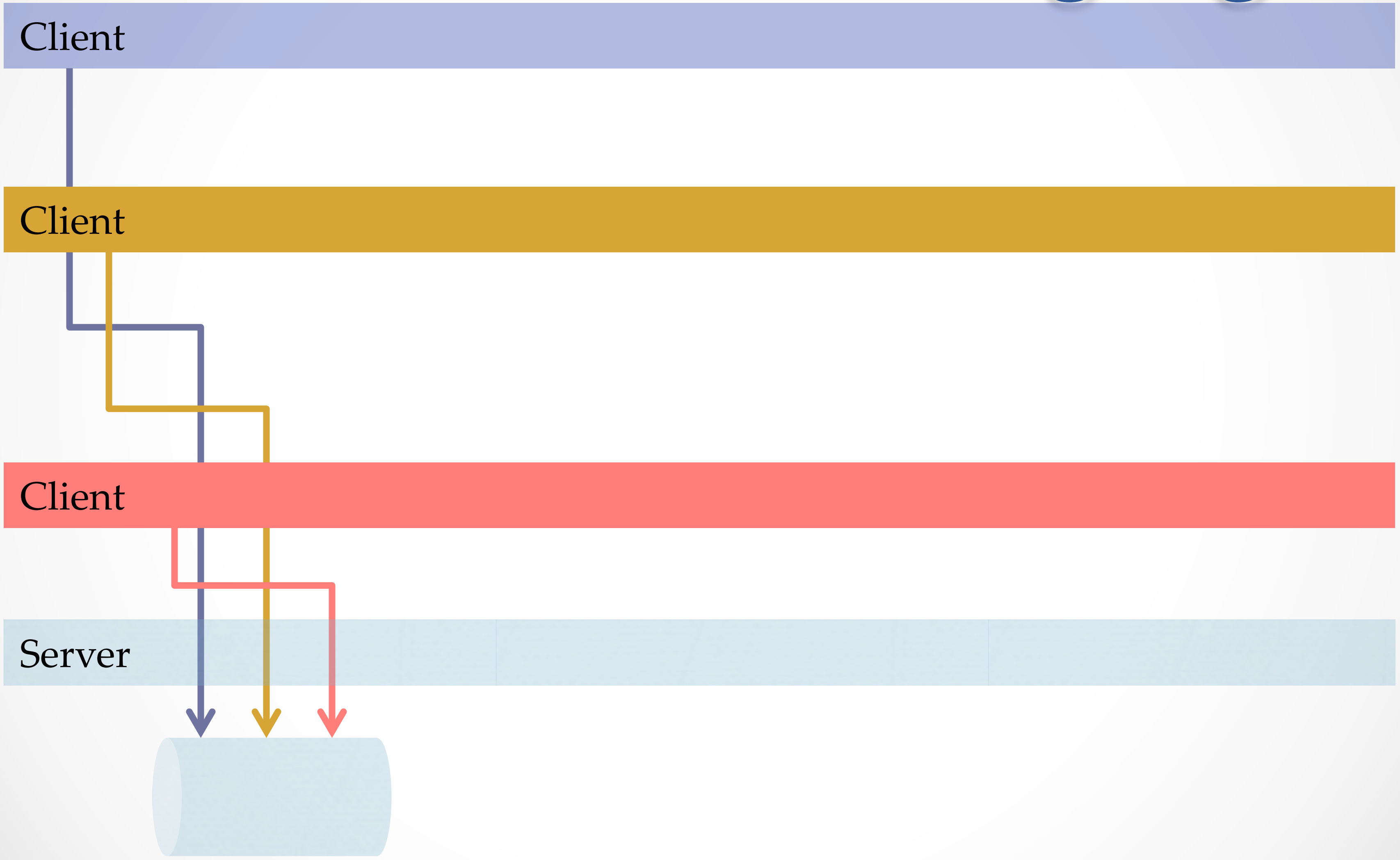
Typical Arrangement



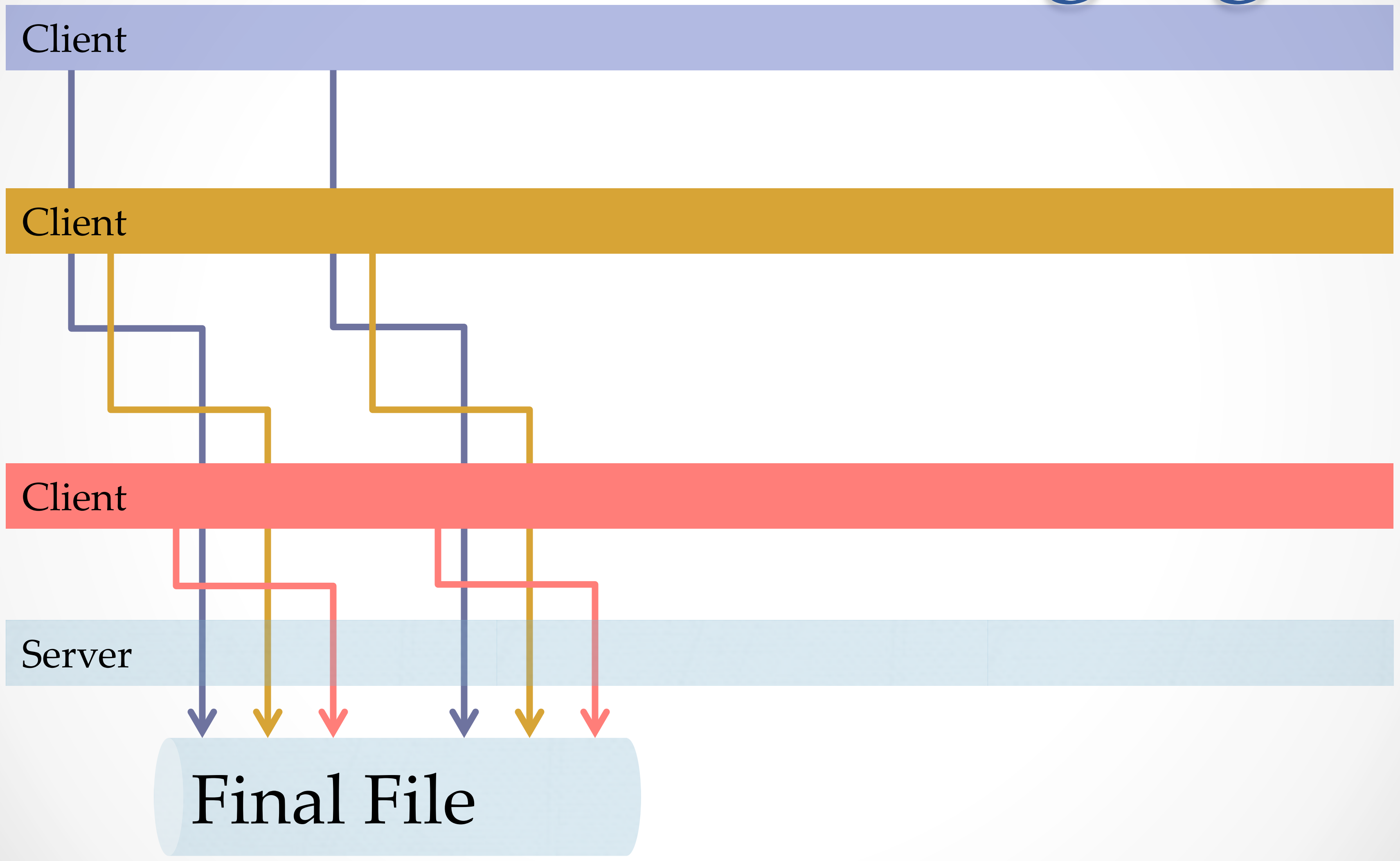
With Parallel Merging



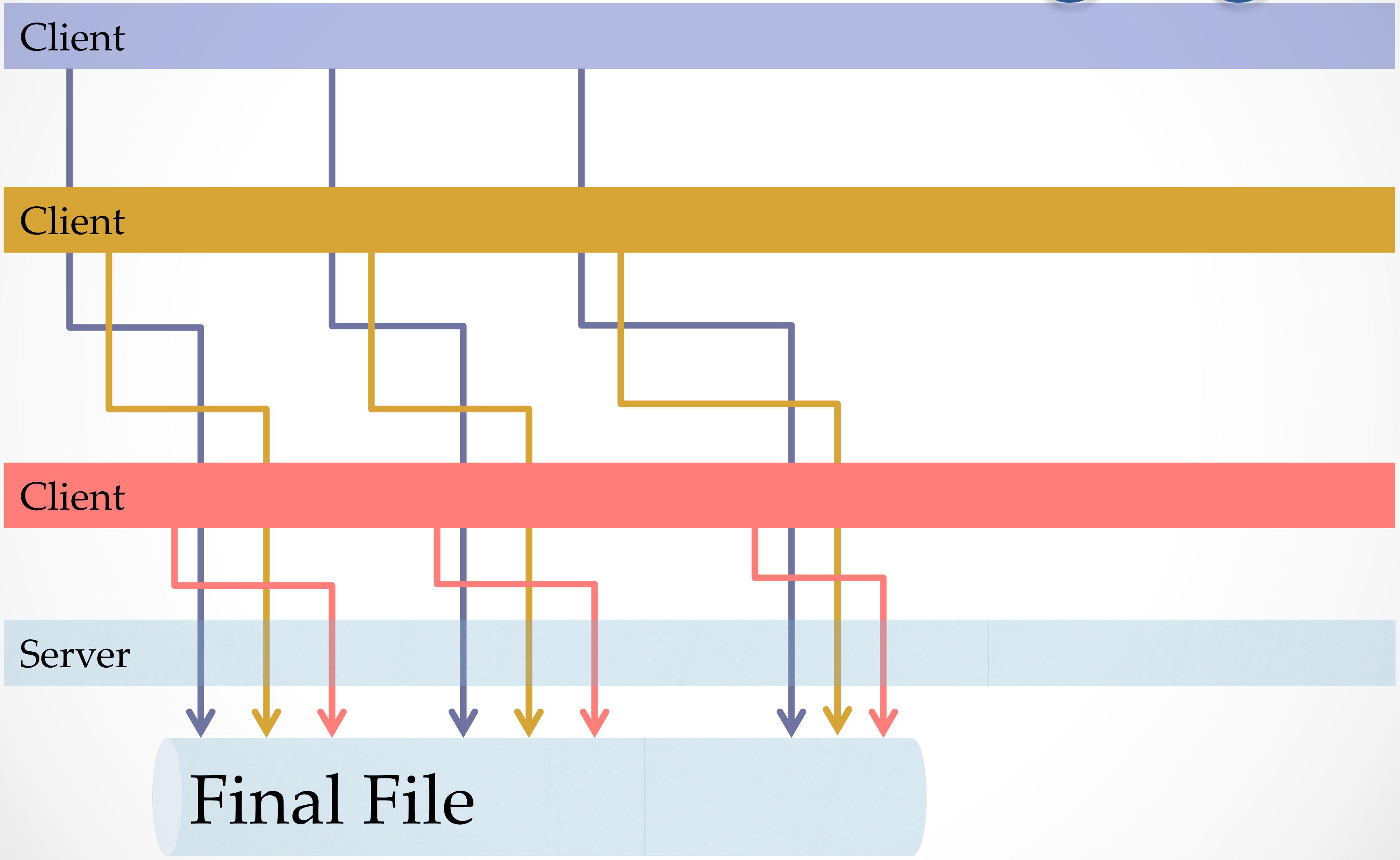
With Parallel Merging



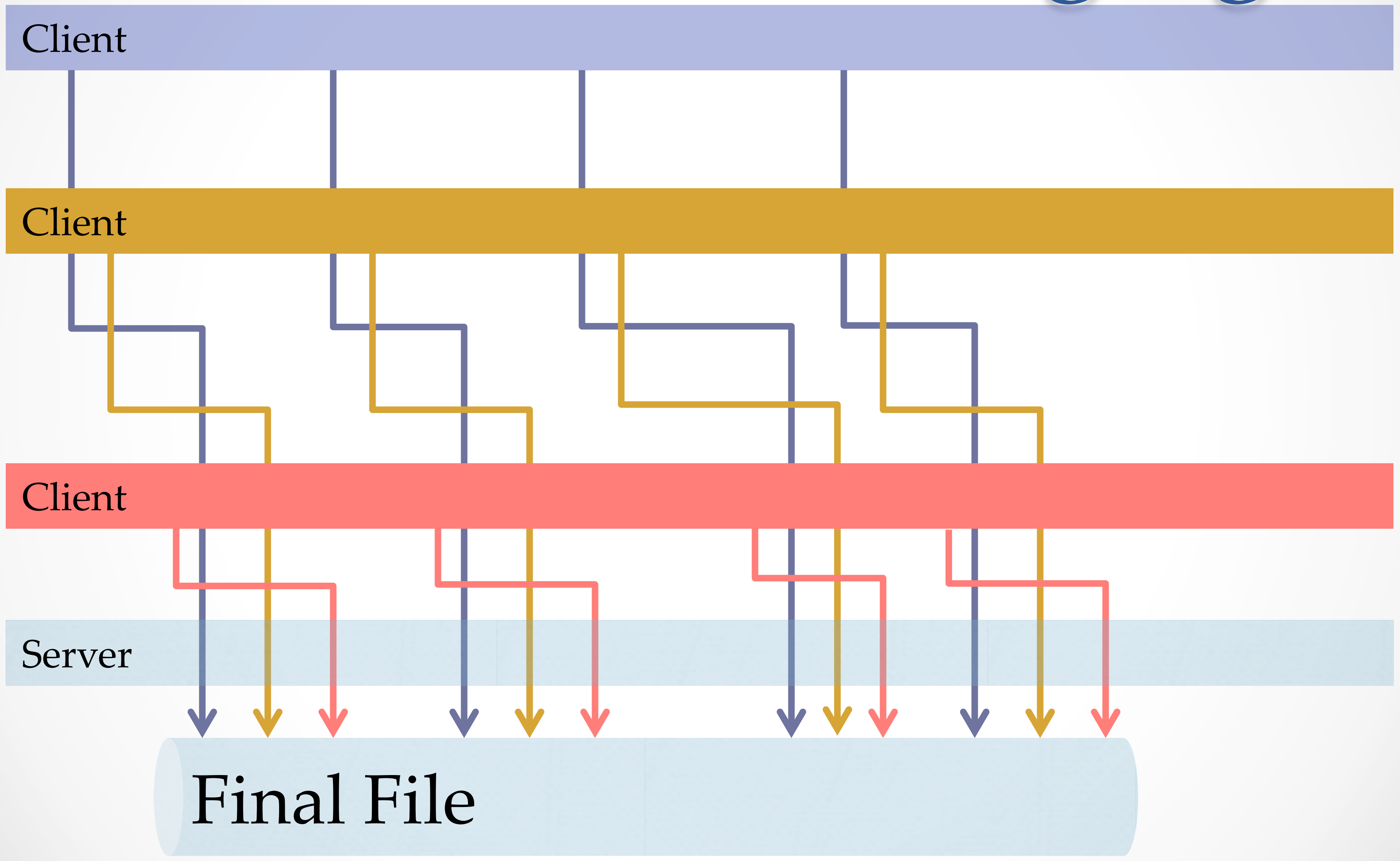
With Parallel Merging



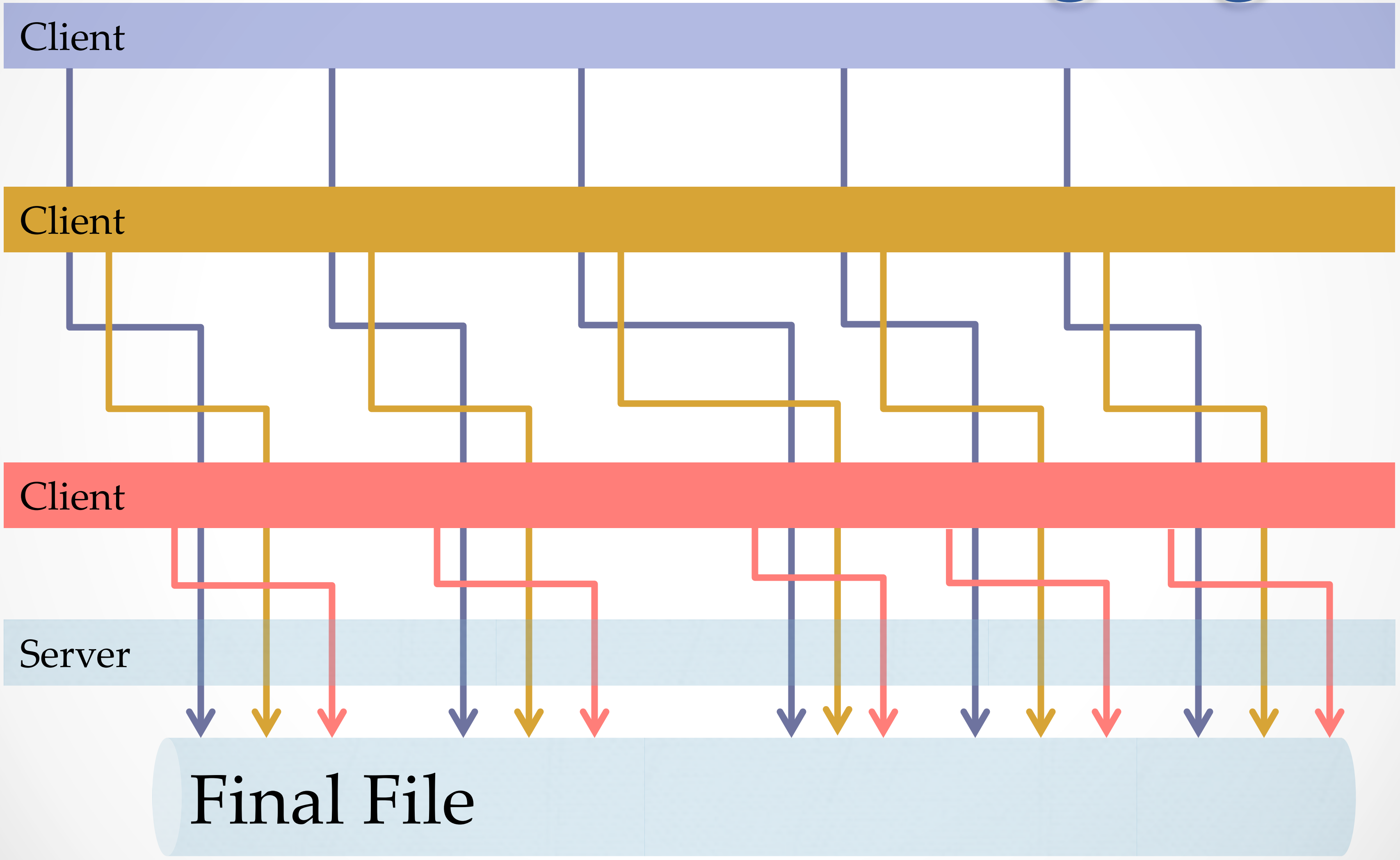
With Parallel Merging



With Parallel Merging

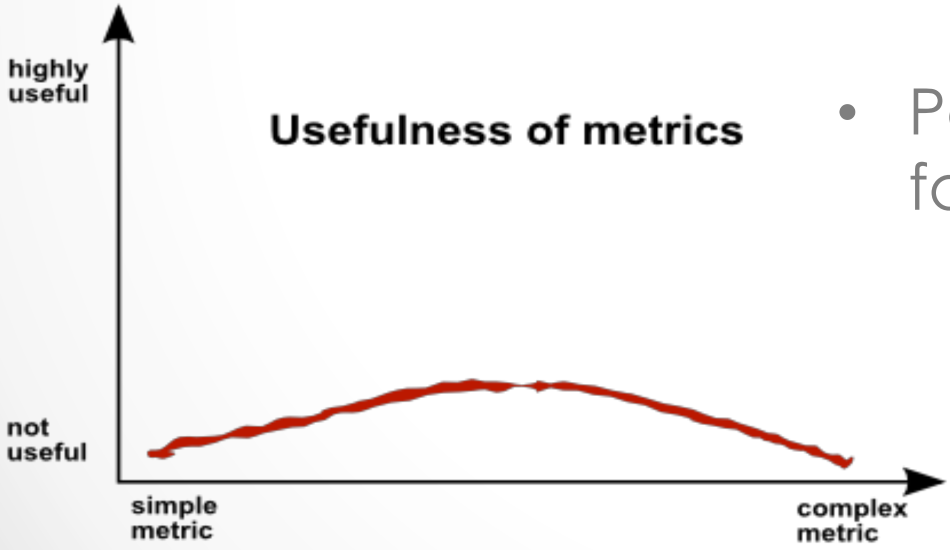


With Parallel Merging



Cost Of Precision

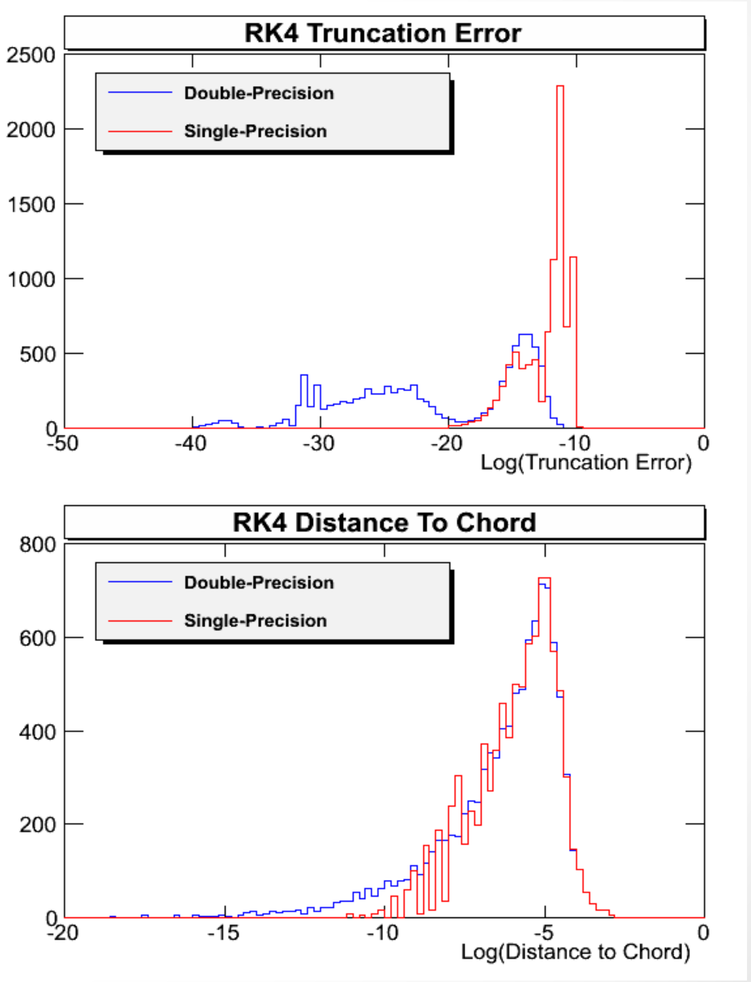
- Precision is expensive
  - Higher precision requires more memory
  - SIMD vectors with less elements
  - Polynomial approximation take longer



- Is precision (double) always justified?
  - Inputs sometimes not as accurate
  - Difficult to predict or track the final accuracy
  - Not trivial to know the final accuracy requirement.
- Potential Gains of factor 2-3

GPU Floating-point Consideration

- Cost for double-precision
  - memory throughput (x2)
  - possible registers spilling
  - cycles for arithmetic instructions (x2/x3 in M2090/K20)
  - performance in classical RK4: float/double = 2.24 (M2090)
  - not negotiable for precision and accuracy
- Possibilities for single-precision
  - input physics tables
  - B-field map (texture)
  - local coordination

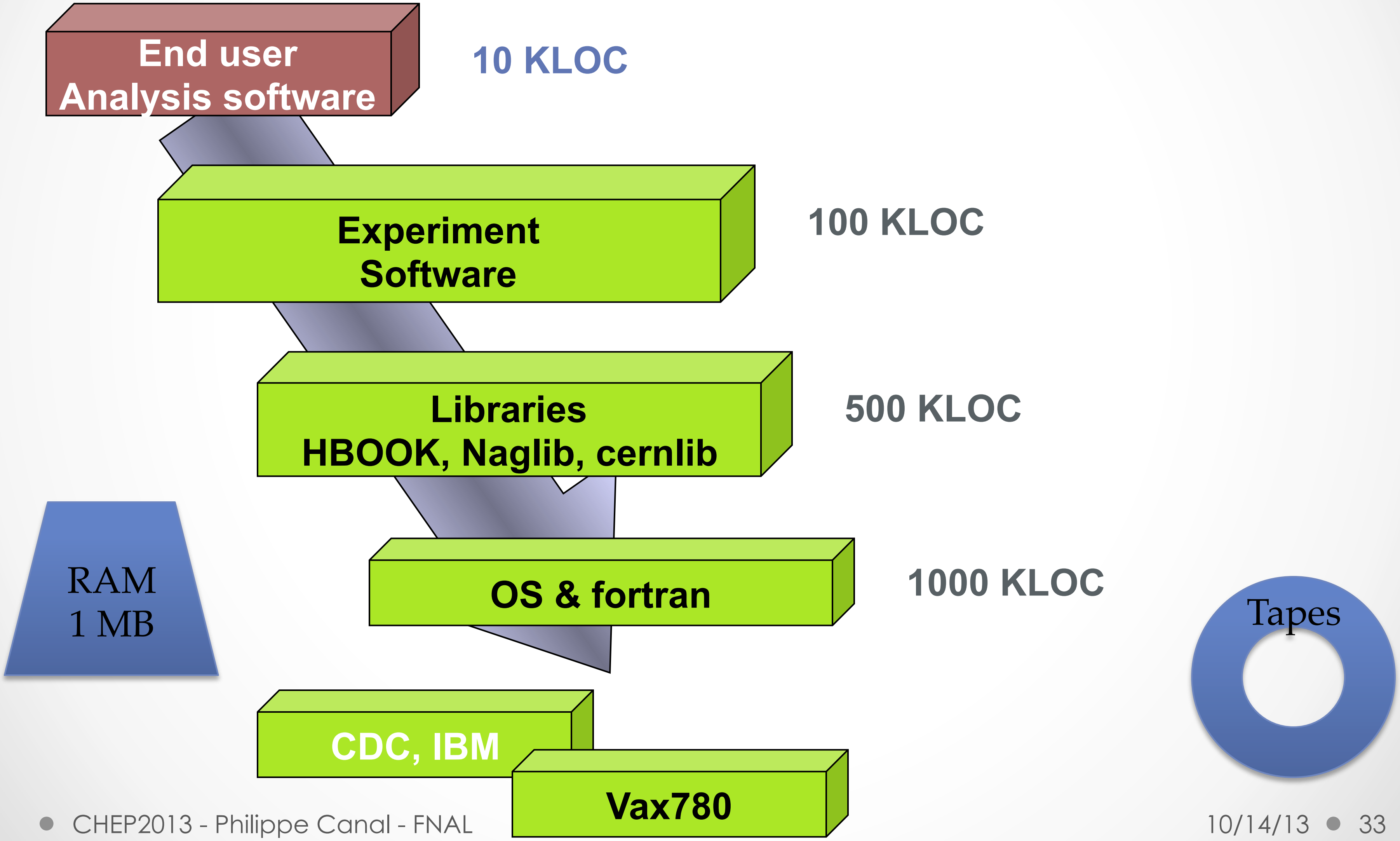


FNAL Detector Simulation GPU Prototype, 2013

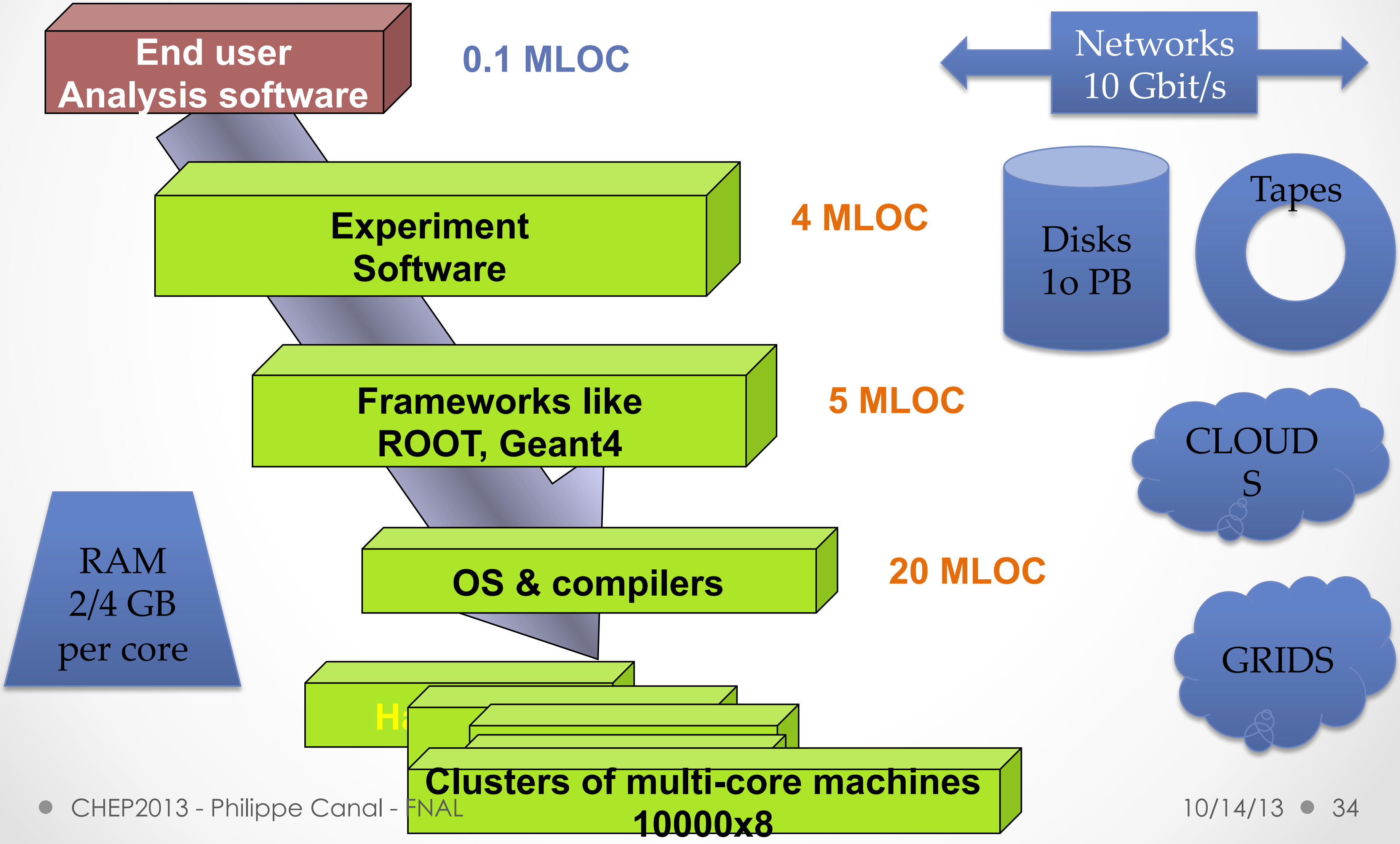
HEP/NPP & HPC

- We have skipped several trains
  - Vectorisation (IBM VM, Cray X-MP, CRAY/YMP, CYBER205,ETA10)
    - Tried for dectector simulation but not successful enough to justify the extra development cost.
  - Low parallelism (IBM VM, Cray X-MP)
  - Moderate parallelism (GPMIMD machine)
  - High parallelism (IBM SP2)
  - Heterogeneous parallelism
- Trivial (job-level) parallelism and evolution of clock cycle has been enough
- But now the incremental *bangs-per-bucks* for us is a monotonic decreasing function and this will affects also throughput
- Time to bite the bullet

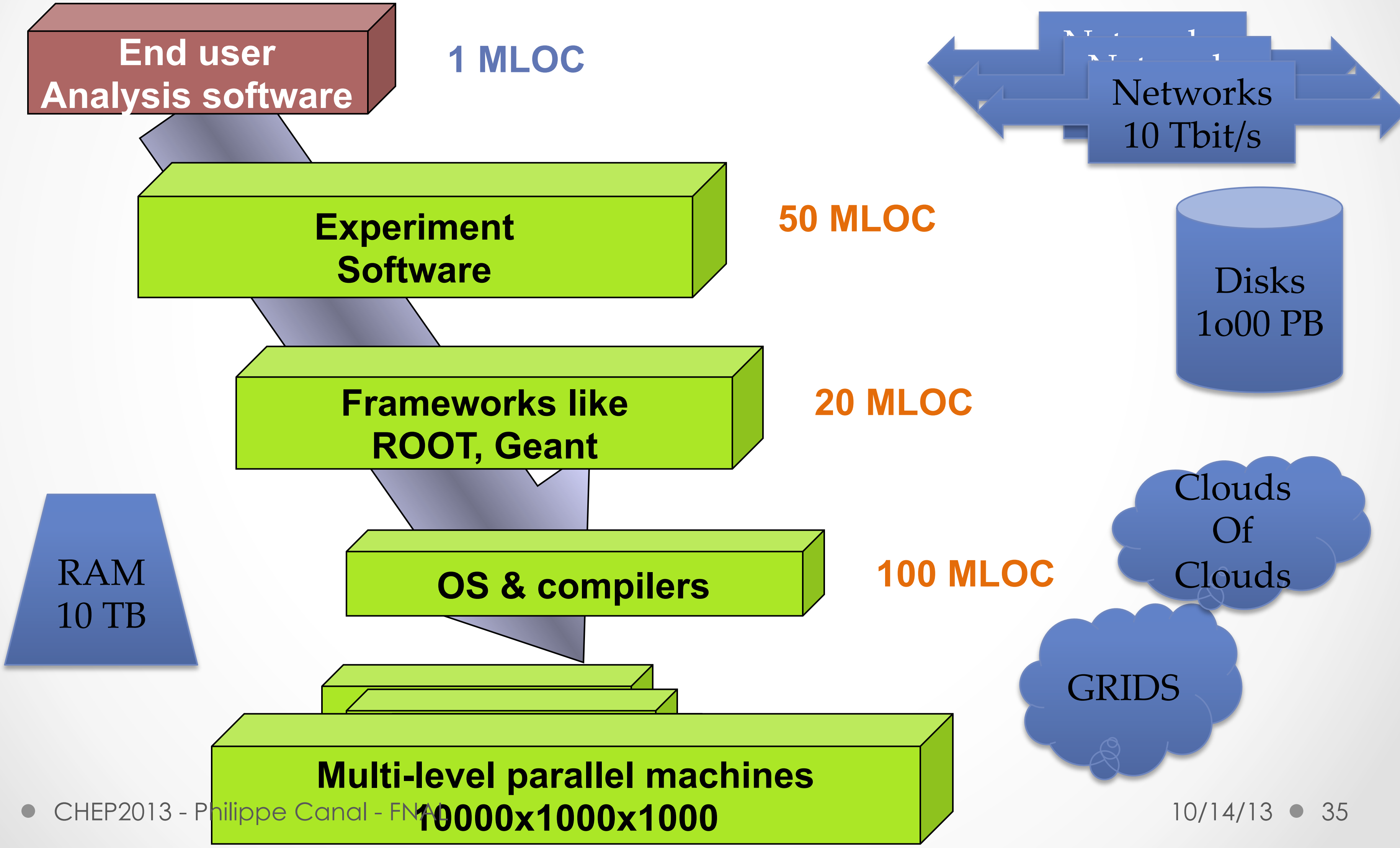
Systems in 1980



Systems today



Systems in 2030 ?



Distributed Computing

- **GRIDS**
  - OSG, WLCG
  - CRAB, GANGA, DIRAC
  - Focus (mainly) on one core / one process
  - Deal with resource allocation and WAN data placement

- **CLOUDS**
  - FutureGrid (multi-cloud project), Public Clouds (Amazon, etc.), Private and Institutional Clouds, etc.
  - Extra flexibility for provisioning (EaaS, Environment as a Service)

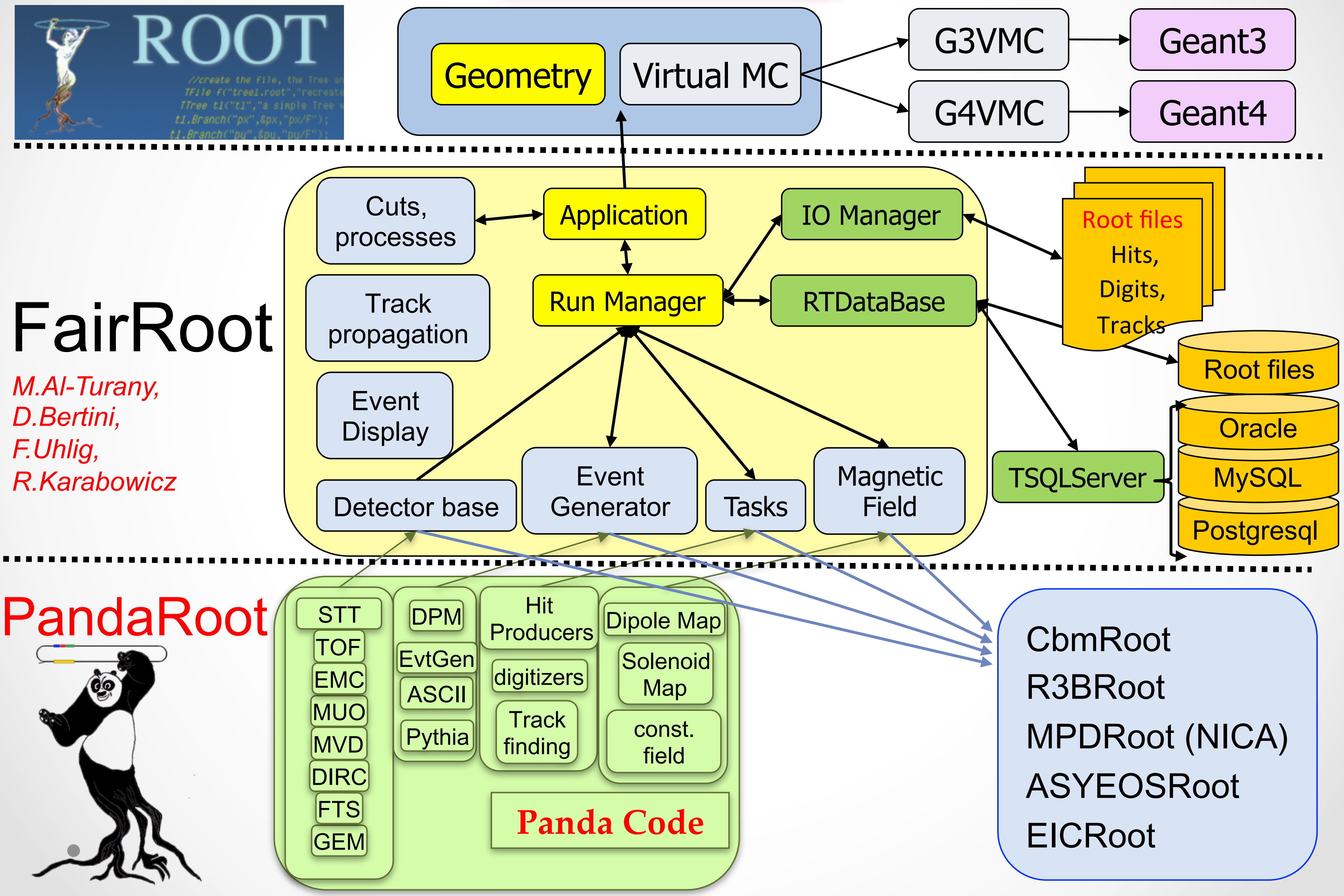
Distributed Computing Evolving

- **Many** existing and varied solutions, including for example:

- ClaRA @ Jefferson lab
    - Implementation of the SOA
    - Data processing major components as services with multilingual support
    - Physics application design/composition based on services
    - **Supports both traditional and cloud computing models**
    - **Multi-Threaded**
    - Communicate data through shared memory and /or pub-sub messaging system

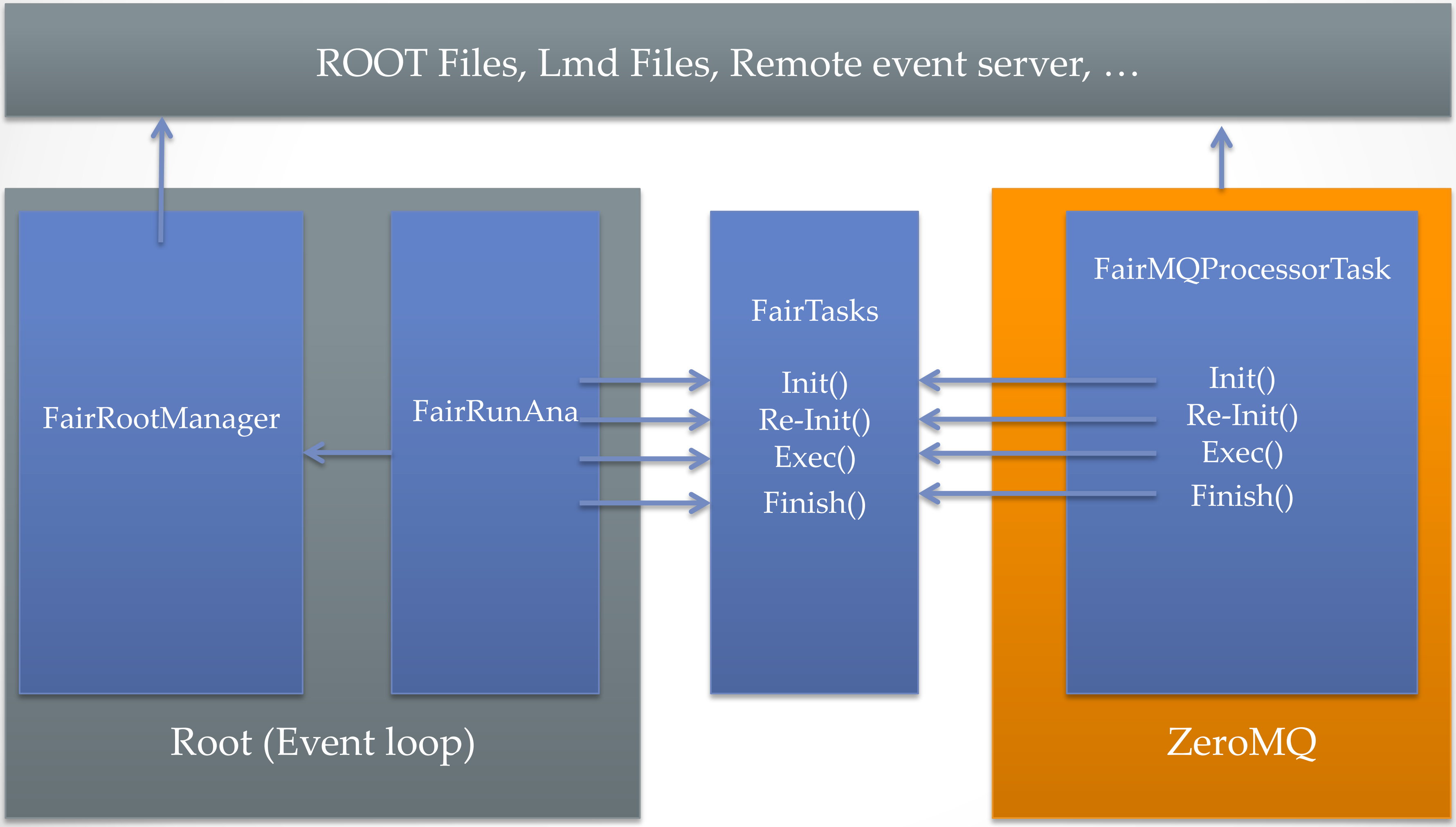
- FairRoot design for distributed processes
    - Highly flexible: different data paths should be modeled.
    - Adaptive: Sub-systems are continuously under development and improvement
    - Should works for simulated and real data
    - **It should support all possible hardware where the algorithms could run (CPU, GPU, FPGA)**
    - **It has to scale to any size! With minimum or ideally no effort.**

Code Design

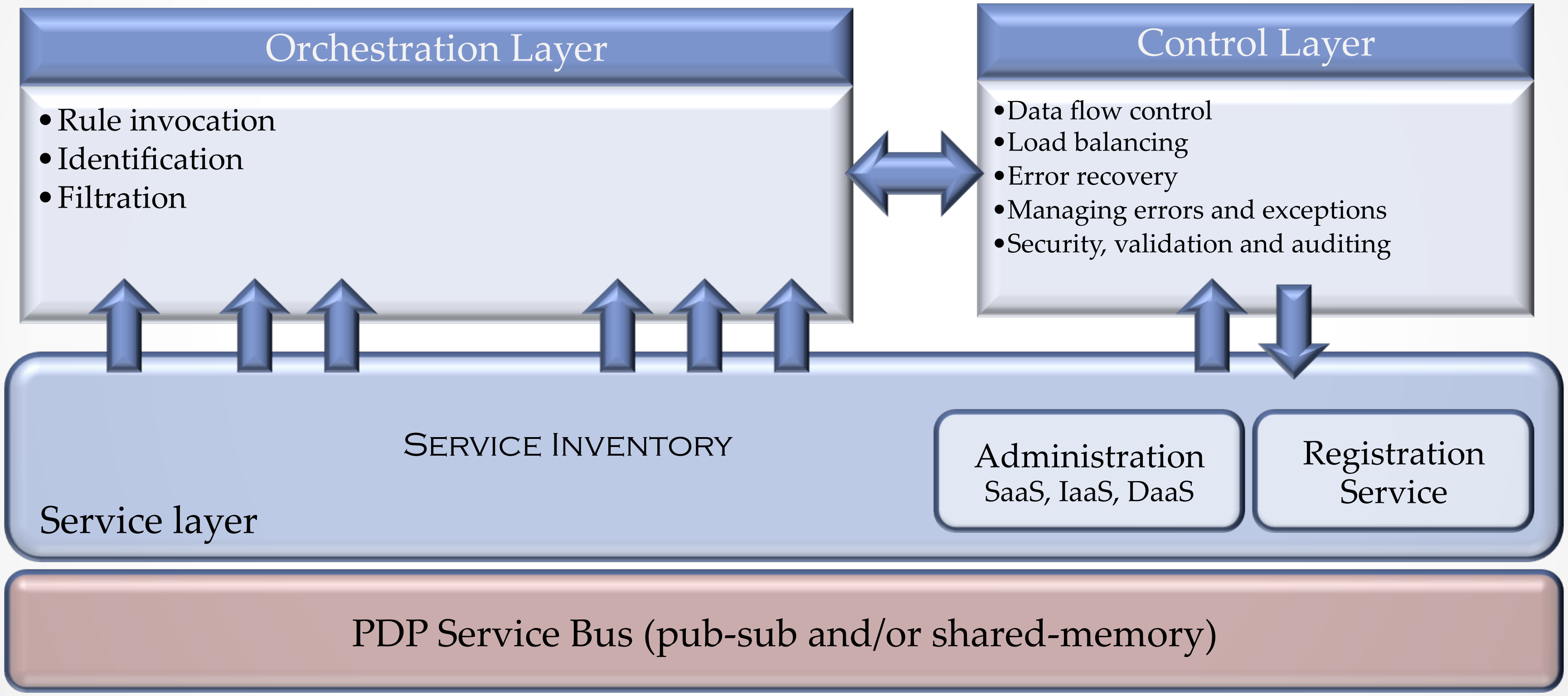


M.Al-Turany, D.Bertini, F.Uhlig, R.Karabowicz

FairROOT - Integrating the existing software:

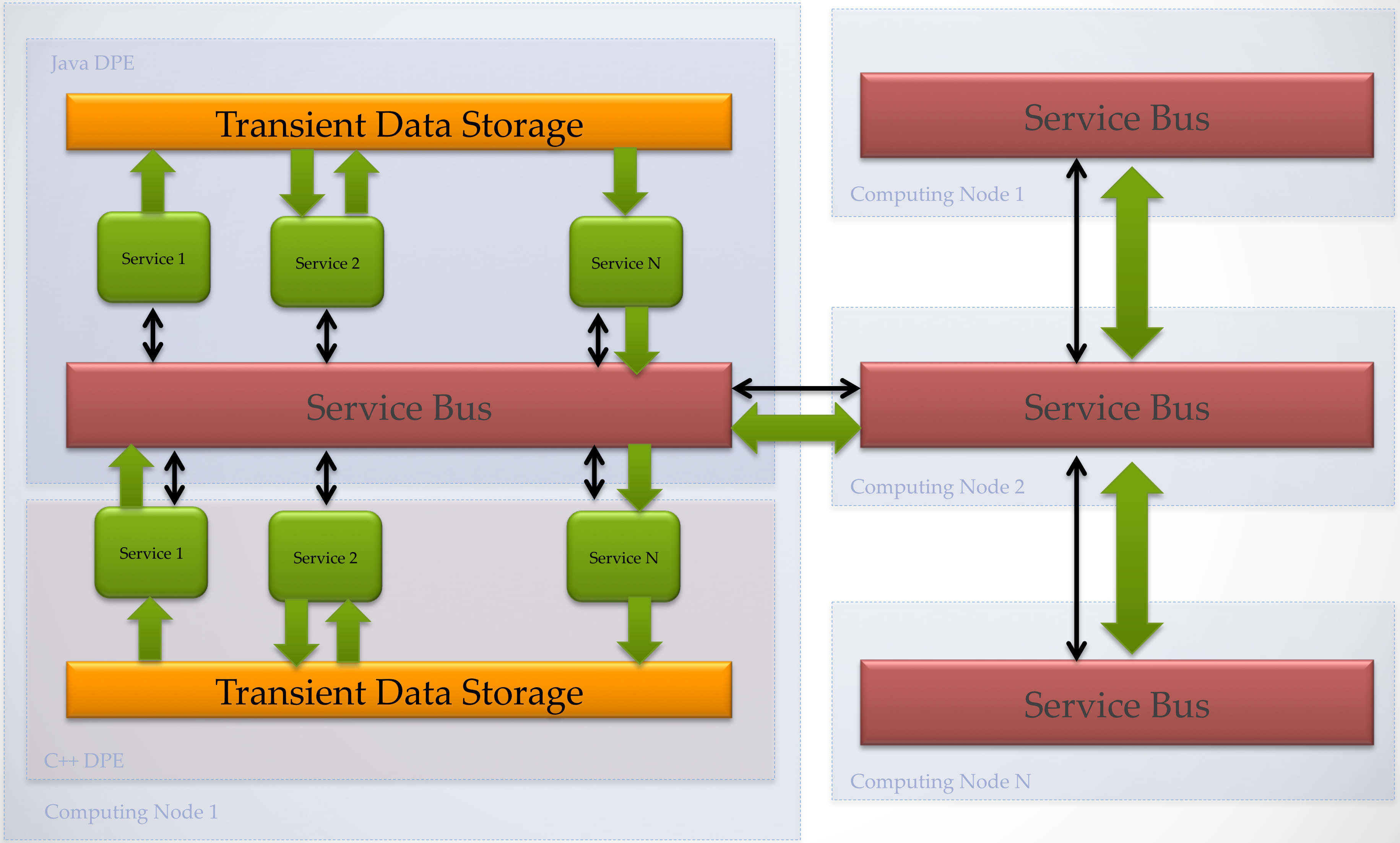


ClaRA Design Architecture



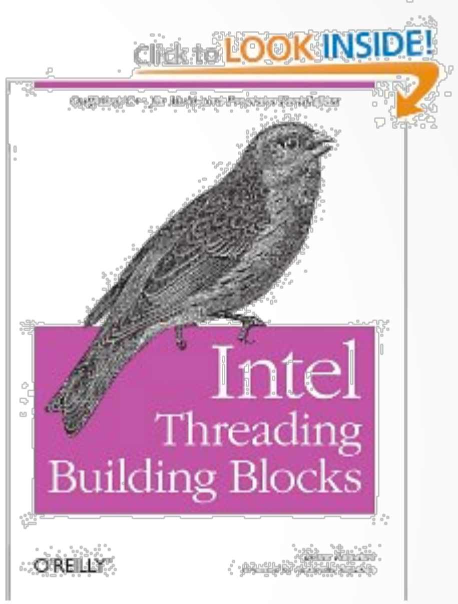
V. Gyurjyan S. Mancilla
Jefferson Lab

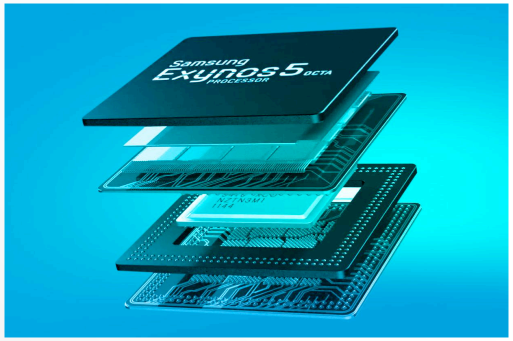
ClaRA - Service Communication

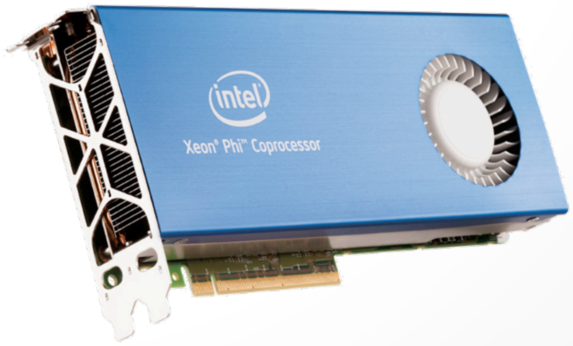


LHC On-going Efforts

- TBB Frameworks
  - CMSSW, ART
- White boards
  - ATLAS, LHCb, i.e. Gaudi, also use TBB
- Port to ARM, Intel MIC / Xeon Phi

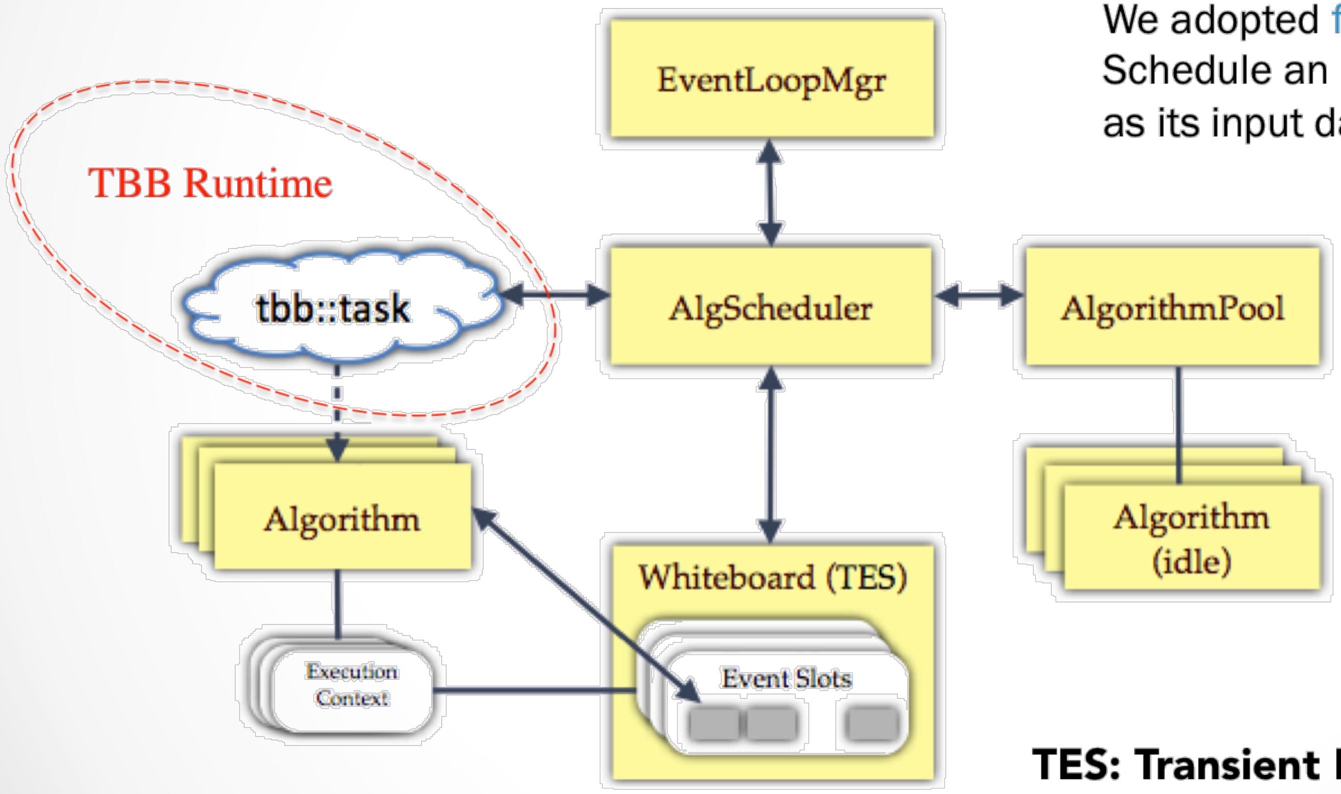






Concurrent Gaudi Component Overview

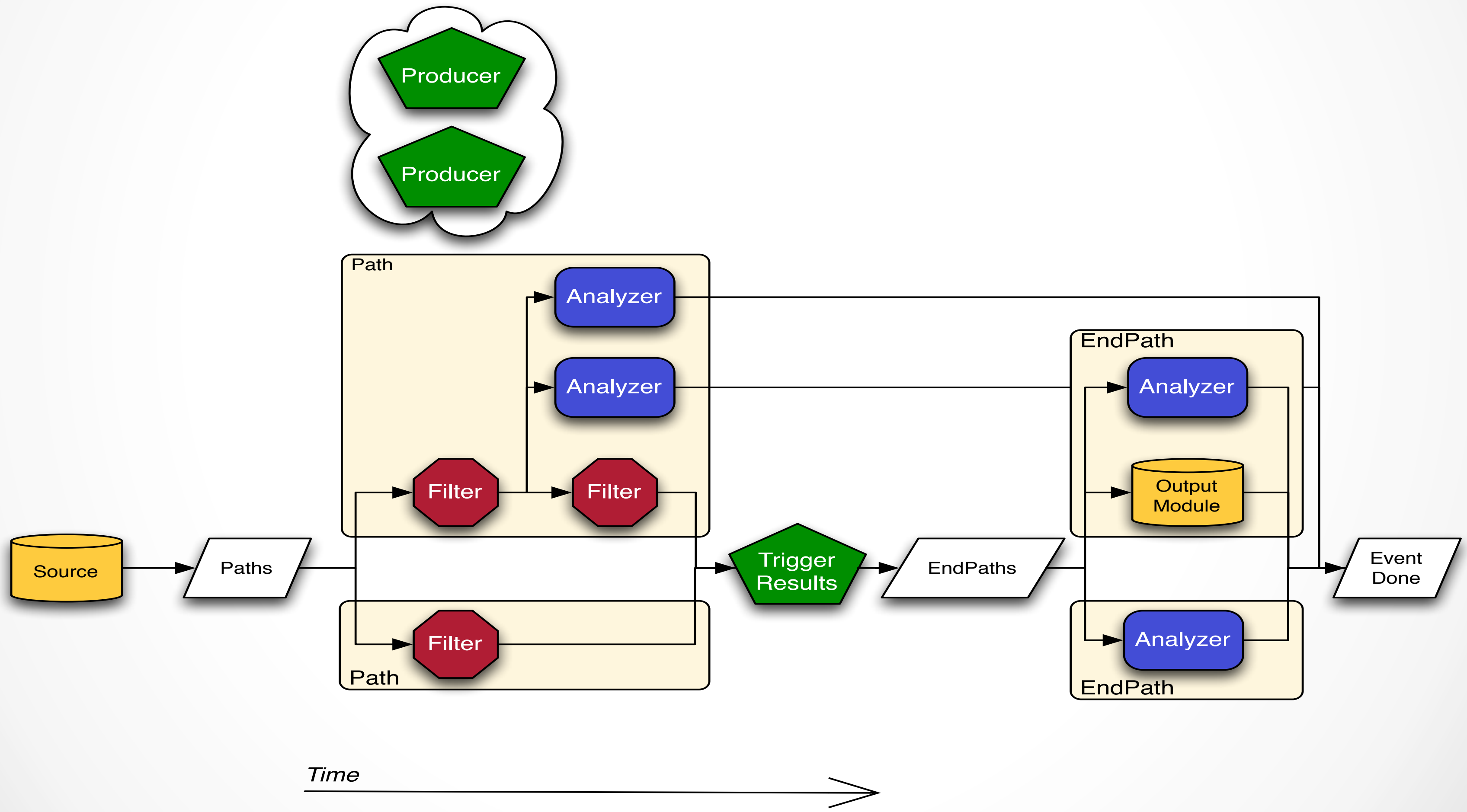
We adopted forward scheduling: Schedule an algorithm as soon as its input data is available



TES: Transient Event Store

8

CMSSW Concurrent Modules



Algorithm Parallelism

- Many Efforts
  - CMS Vertex Clustering
  - Triplet Seeding in CMS
  - Real-time use of GPUs in NA62 Experiment
  - LHCb Pixel tracking using GPU
  - TBB in ATLAS Liquid Argon Calibration
  - Tracking with Cellular Automata in CMS, Alice, CBM, STAR
  - CBM MUCH Trigger in CUDA
  - Cluster Transformation in Alice HLT
  - STAR High Level Tracking Trigger
  - Track Finding in the Silicon Vertex Detector of Belle 2
  - KFParticle Package for Vertexing and Particle Finding
  - Etc., etc.

Vectorization Efforts

- Auto-vectorization
  - Manual re-code work is needed to help the compiler (think-C)
  - Factor ~2 in specific algorithms has been reported (CMS, ATLAS)
- Vector libraries
  - Linear algebra libraries (e.g. Eigen 3)
  - Vector data types – many objects in parallel (e.g. VC)
  - Transcendental functions (e.g. VDT speedup 2x-3x)
  - In general small effort required to introduce these libraries
- Intrinsics
  - Code at hardware level (code maintenance is an issue)
- Language extensions
  - Cilk++ (not explored so far)

Current Trends

- Recurrent thread
  - Use of TBB in particular and task based system in general for 'higher' level application

- Framework effort current concentrate on task level concurrency but each module treats only one event at a time.
  - Vectorization limited to a few (significant) modules/algorithms
  - Allows evolution rather than revolution
  - But still overall underutilization of (vector) hardware.

Current Trends

- Simplifying the end developers' life.
  - They should not (usually) have to worry about locks and race conditions.
  - Need to come up with simple rules to follow
    - avoid concurrent data access in 'their' part of the code

- However need to get them started to design for vector/parallelism

Need For Coordination

- Many efforts to parallelize both Low Level Algorithms and experiment frameworks
- Both will try to use all computing resources
- Need run-time Coordination to avoid risk of over-subscription and mutual negative effect on cache/ memory coherency
- Effort is very significant but our resources are diminishing, coordination and sharing of code is becoming more important

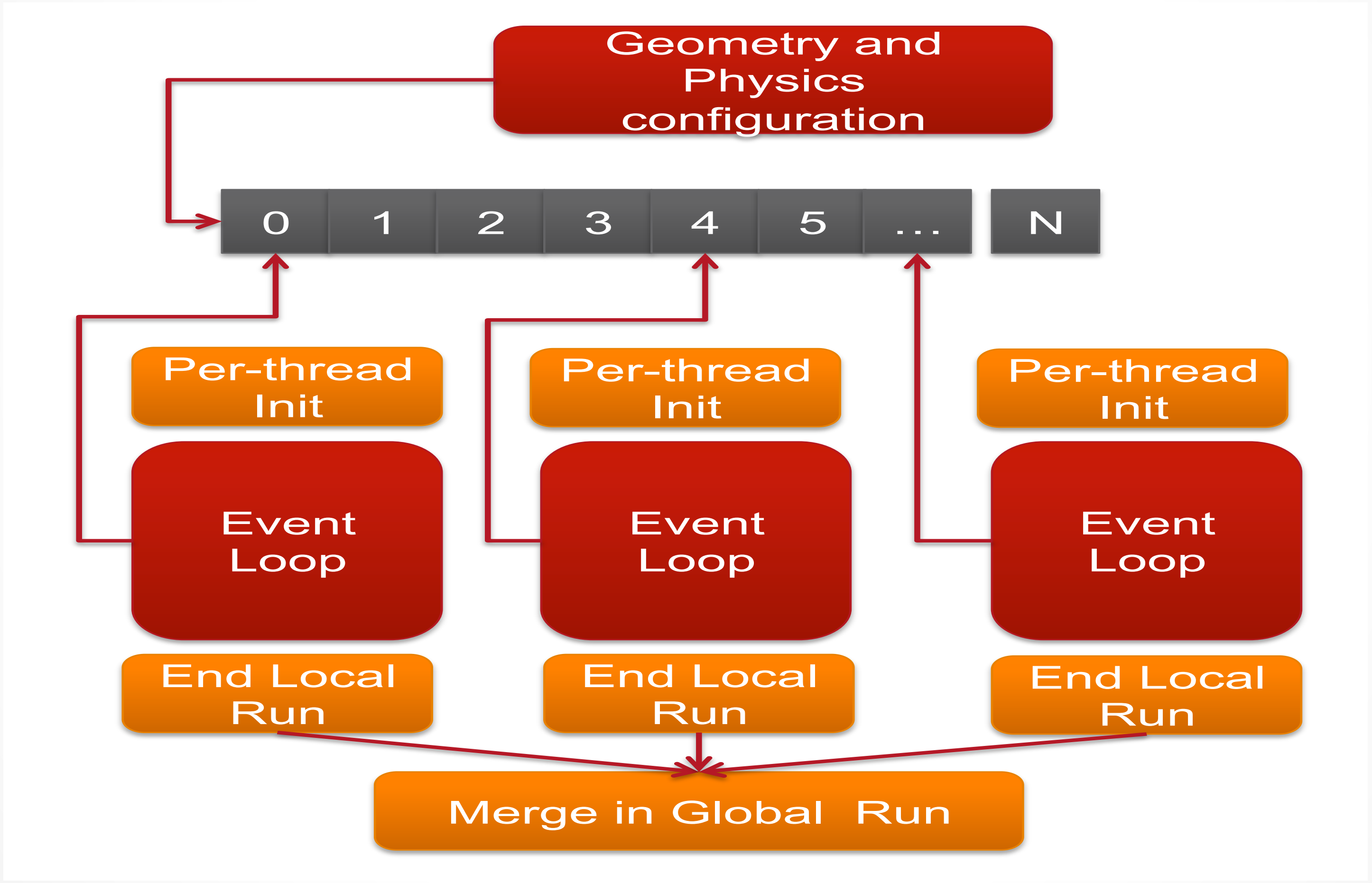
Sharing Progress

- Developers Gatherings
  - Concurrency Forum
  - Workshop For Future Challenges in Tracking and Trigger Concepts
  - Not quite yet at the level of common/shared developments
    - A few exceptions like the Vector Class library.
  - Not yet fully engaging similar effort outside HEP/NPP
    - DOE's Advanced Scientific Computing Research
    - Software Sustainability Institute at University of Edinburgh
    - etc.
- Existing de-facto HEP/NPP standards
  - Eg. Geant4, ROOT
  - Improvements (and bottlenecks) have magnified effects
    - Including as a code copy/paste source.
  - Must lead by example
    - But backward compatibility constraints/challenges

Geant4 Version 10

- **Event level parallelism**
- **Designed to minimize changes in user-code**
  - Maintain API changes at minimum
- Focusing on “**lock-free**” code
  - linearity of speed-up (w.r.t. #threads) is the metric they are currently concentrating on (then we’ll optimize absolute throughput)
  - Good results obtained for both metrics anyway
- Enforce use of **POSIX standards** to allow for integration with user preferred parallelization frameworks (e.g. MPI,TBB, ...)
- Basic Design Choice
  - Thread-safety implemented via **Thread Local Storage**
  - “Split-class” mechanism: reduce memory consumption
    - Read-only part of most memory consuming objects shared between thread
    - Geometry, PhysicsTables
    - Rest is thread-private

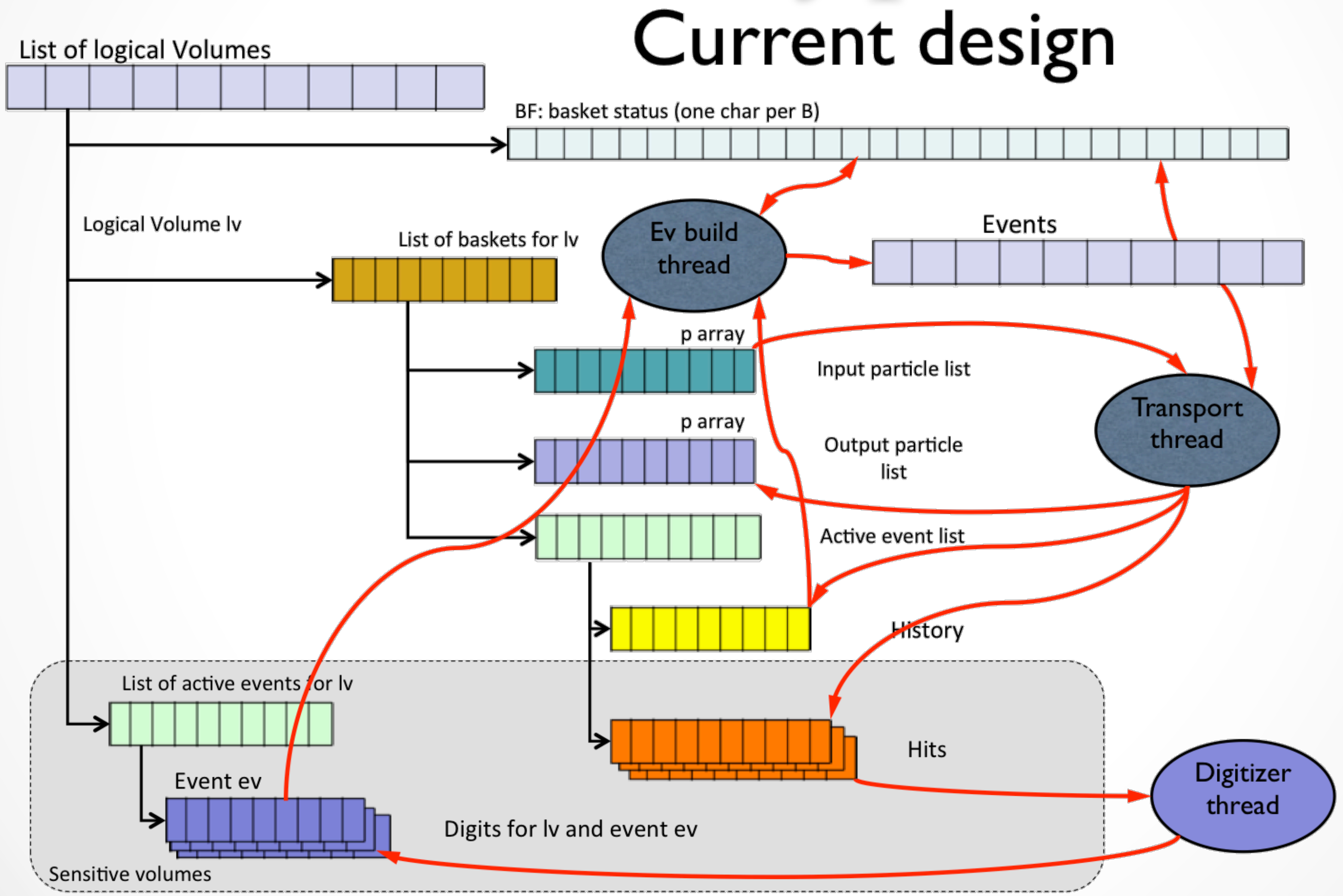
Geant4 Version 10



Simulation Vector Prototype

- Strategy
  - Explore from a performance perspective, no constraints from existing code
  - **Expose the parallelism at all levels, from coarse granularity to micro-parallelism at the algorithm level**
  - Integrate from the beginning slow and fast simulation in order to optimize both in the same framework
  - Explore if-and-how existing physics code can be optimized in this framework
  - Improvements (in geometry for instance) and techniques are expected to feed back into reconstruction
  - Explore executing on coprocessors (GPU, Intel MIC)

Simulation Vector Prototype



ROOT & Concurrency

- Proof, POD, xrootd
  - Exploiting core level parallelism and Grid/Clouds
  - Process level parallelism
  - Deal with on node/site level data locality.
- Concurrency and vectorization in math
  - Introduction of the Vc (Vector Class) library
  - Uses Vc in vector and matrix library
  - Vectorization of Fitting (using Vc and/or VDT libraries)
  - Exploring OpenMP / Intel TBB for multi-threads for fitting and numerical integration
- Concurrent geometry navigation
  - Adding multi-threading and vectorization in concert with Simulation Vector Prototype

ROOT 6 & Concurrency

- Core of ROOT based on Interpreter
- CINT was inherently thread adverse
  - database and execution were intermingled for performance reasons.
- ROOT 6 introduces Cling
  - Based on LLVM and Clang.
  - Cling has clear separation of database engine and execution engine allowing to lock them independently
  - Enables support for sturdy multi-thread I/O
- Documentation effort to express thread-safe (by correctly marking them as const methods)
- Proper Just-In-Time compiler opens up a large set of run-time optimization

ROOT I/O & Concurrency

- Parallel File Merging
  - Address end-of-job tail
- TTreeCache, Asynchronous prefetching
  - Address local and remote I/O latencies
- Support for cloud storage
  - HDFS, Amazon S3, CloudFront, Google Storage
- I/O internal engine and data structures
  - Ready to be easily extended to support data bunching
  - Can also use JIT to optimize hot-spot

Fusion-io ioDrive2
1.5 GB/s
242K 4K-IOPS
68µs latency

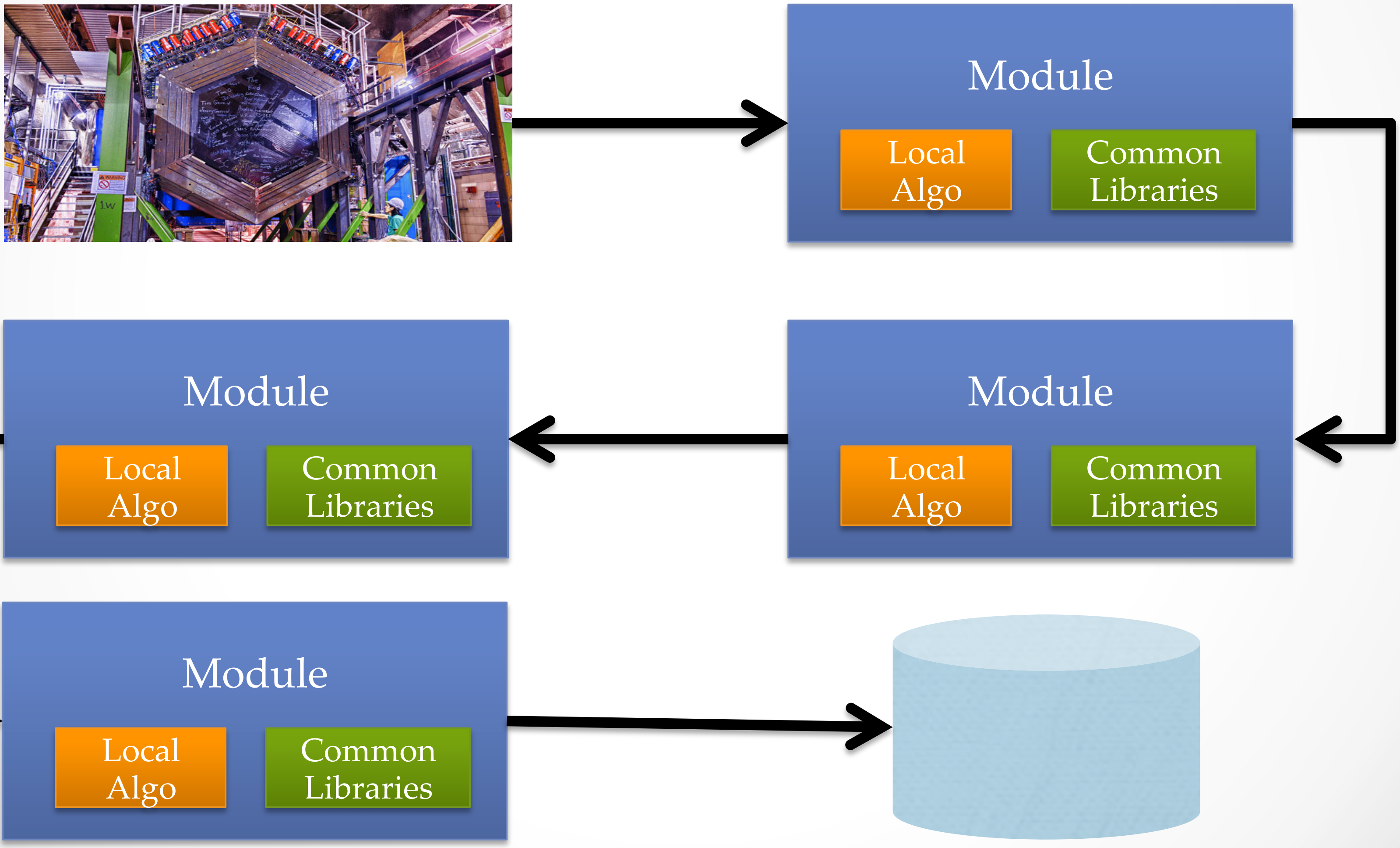
Meanwhile

- In the meantime, progress in ROOT and Geant4 needs to continue on other fronts
- New and improved features, physics processes, math and statistics algorithms and presentations
- Display on new form factor and OS
- Program of works for both is long and healthy

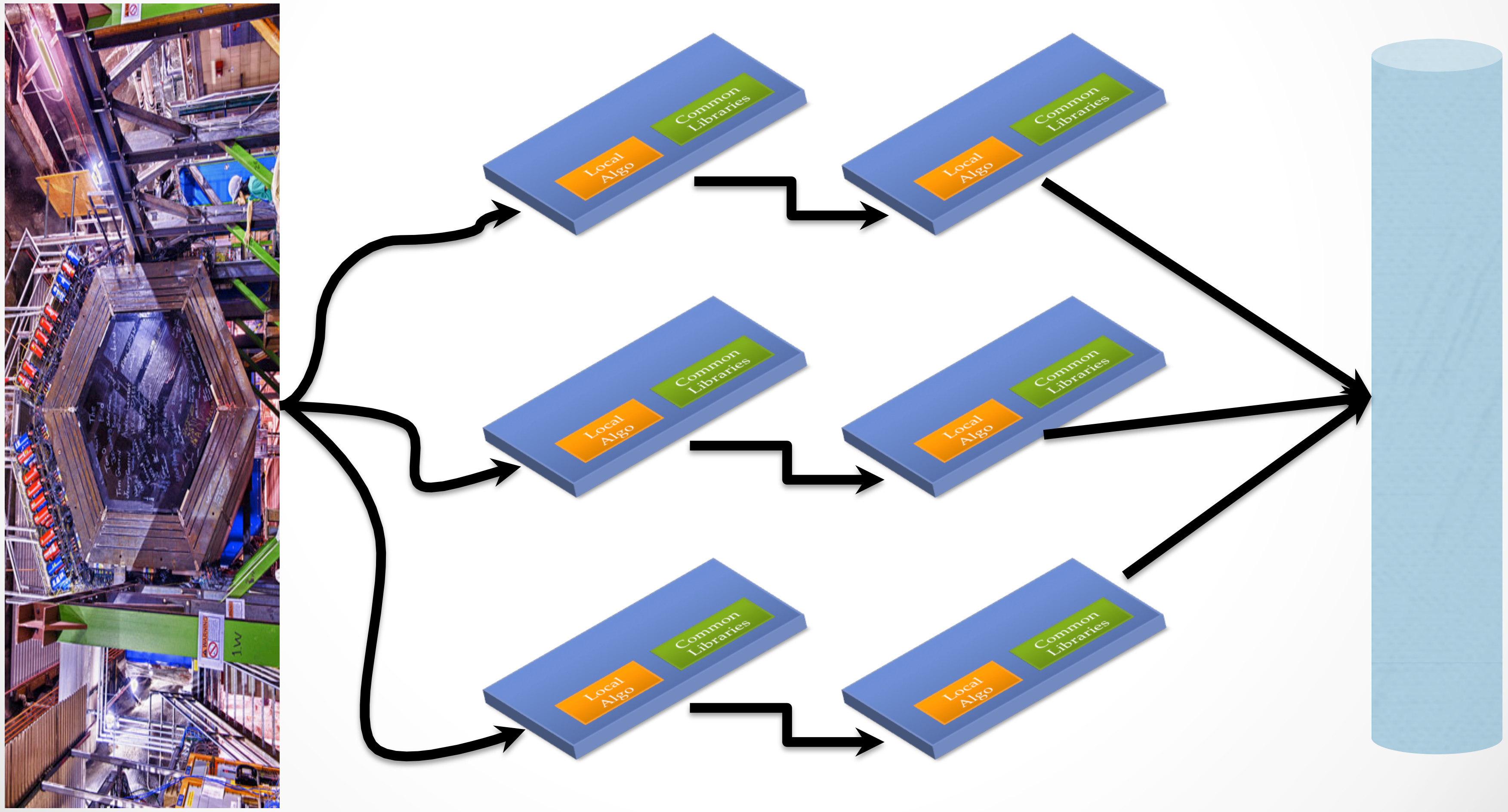
Needs Vectorization From Start to Finish

- We have been addressing islands
  - *Low level algorithm*
  - *Framework*
  - *Large Libraries (Simulation)*
- Islands are growing but will misconnect without coordination
  - *Mismatch between simulation GPU prototype and vector prototype in term of granularity and data layout*
  - *A single large algorithm can kill overall performance of a multi-thread framework but taking over all core and swapping out of the carefully plan/located module*
  - *Let's not forget tail handling and Amdahl's law …*

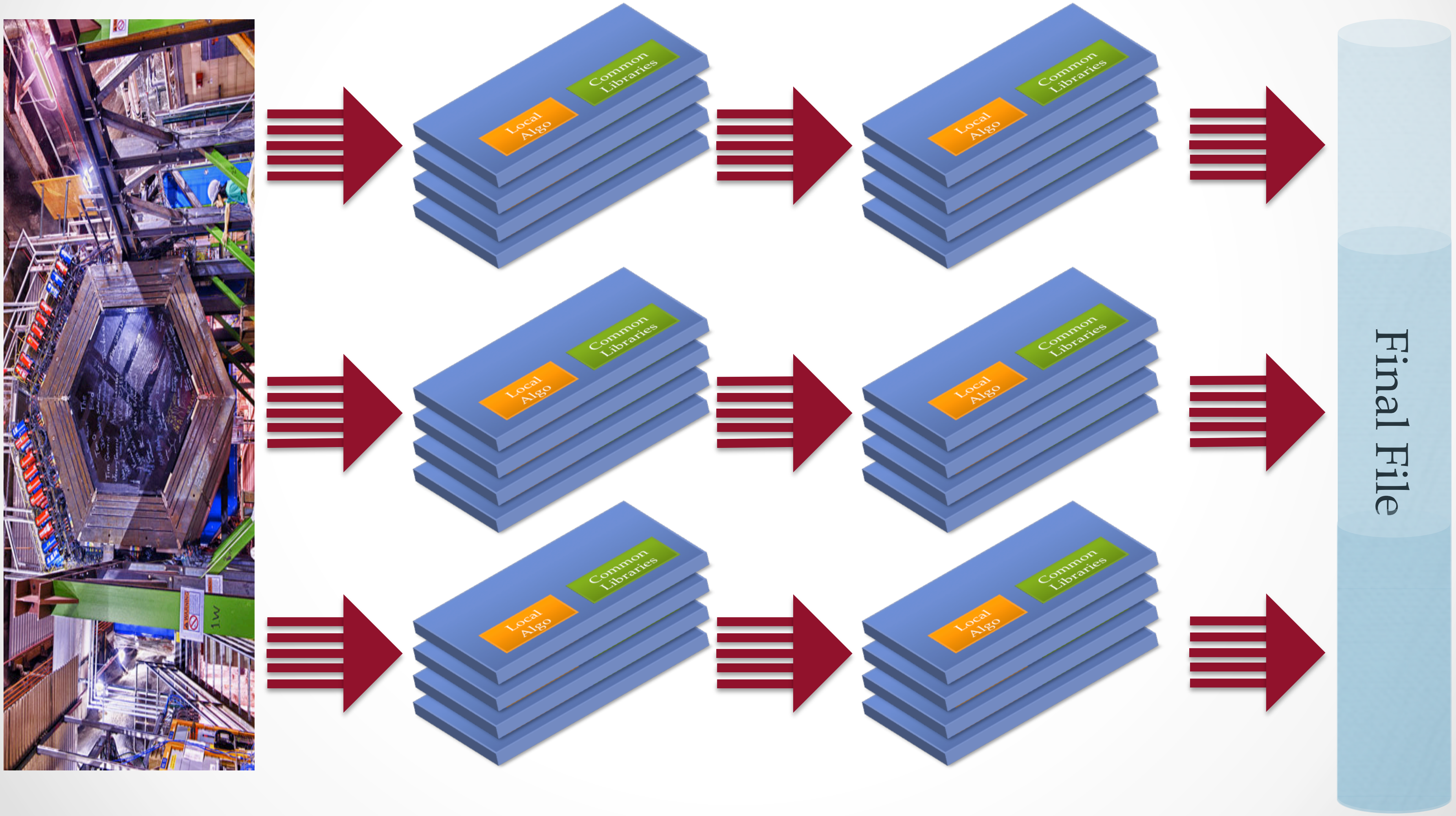
Classic Frameworks



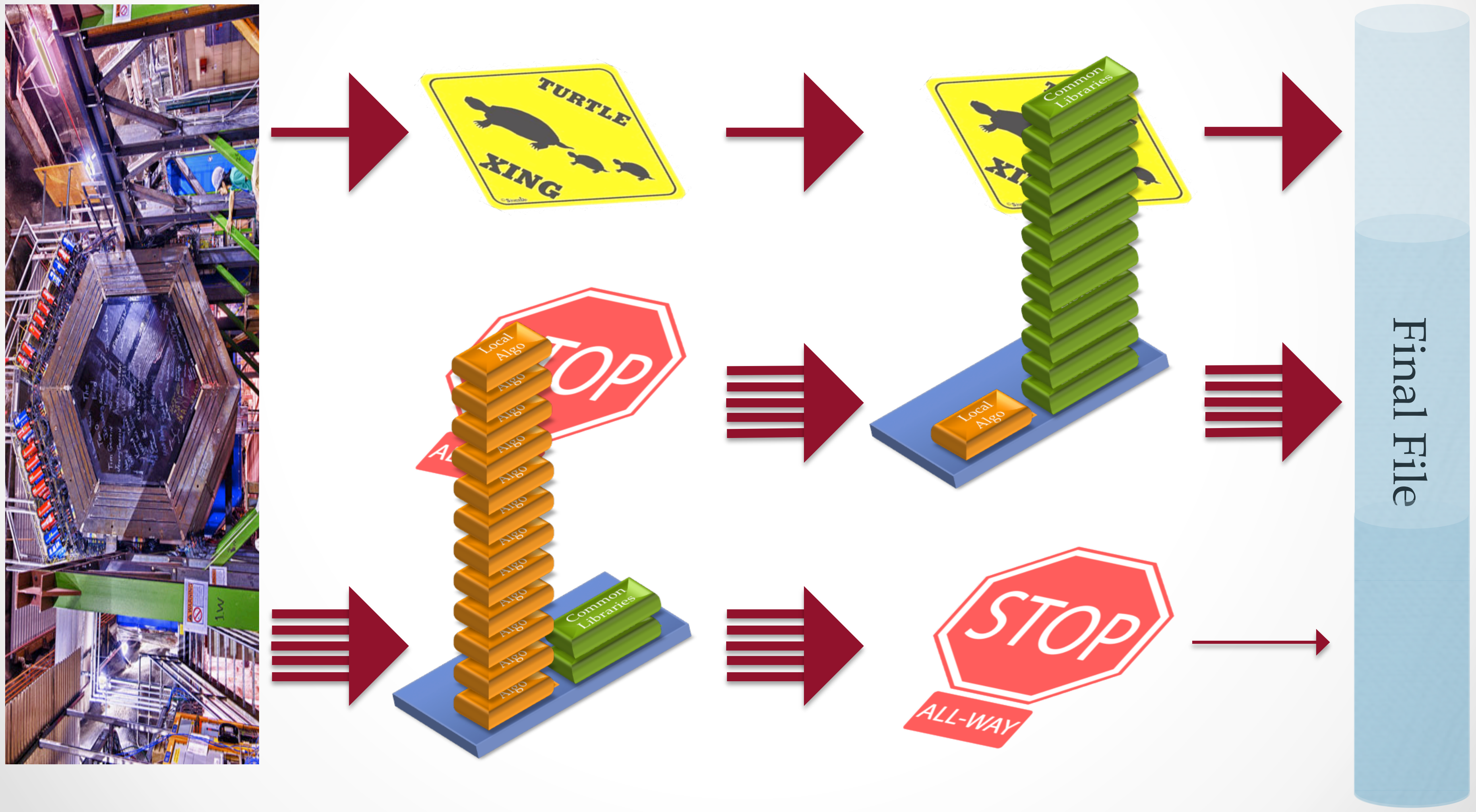
Parallel Frameworks



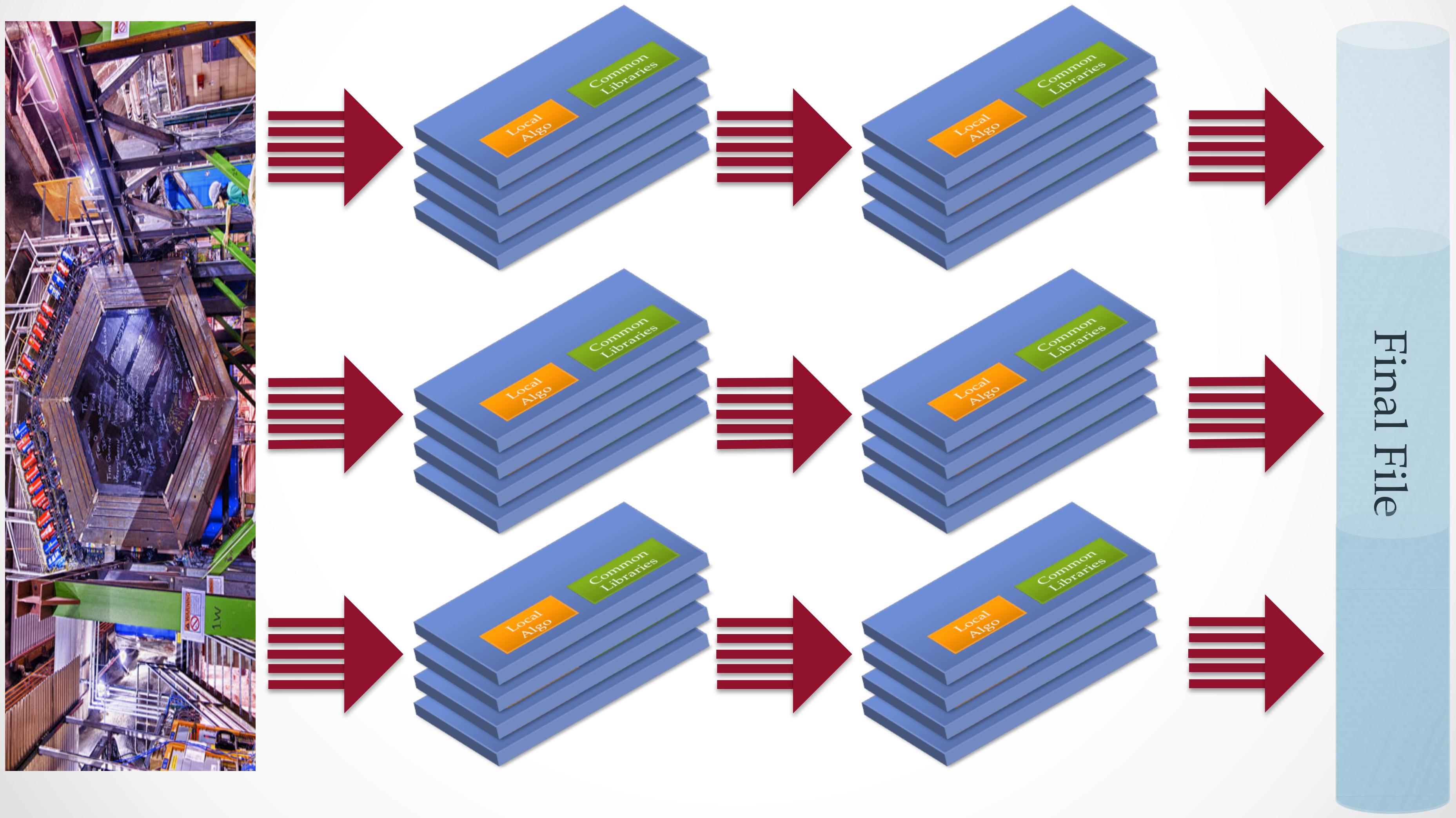
Parallel/Vector Frameworks



Lack Of Coordinations



Parallel/Vector Frameworks



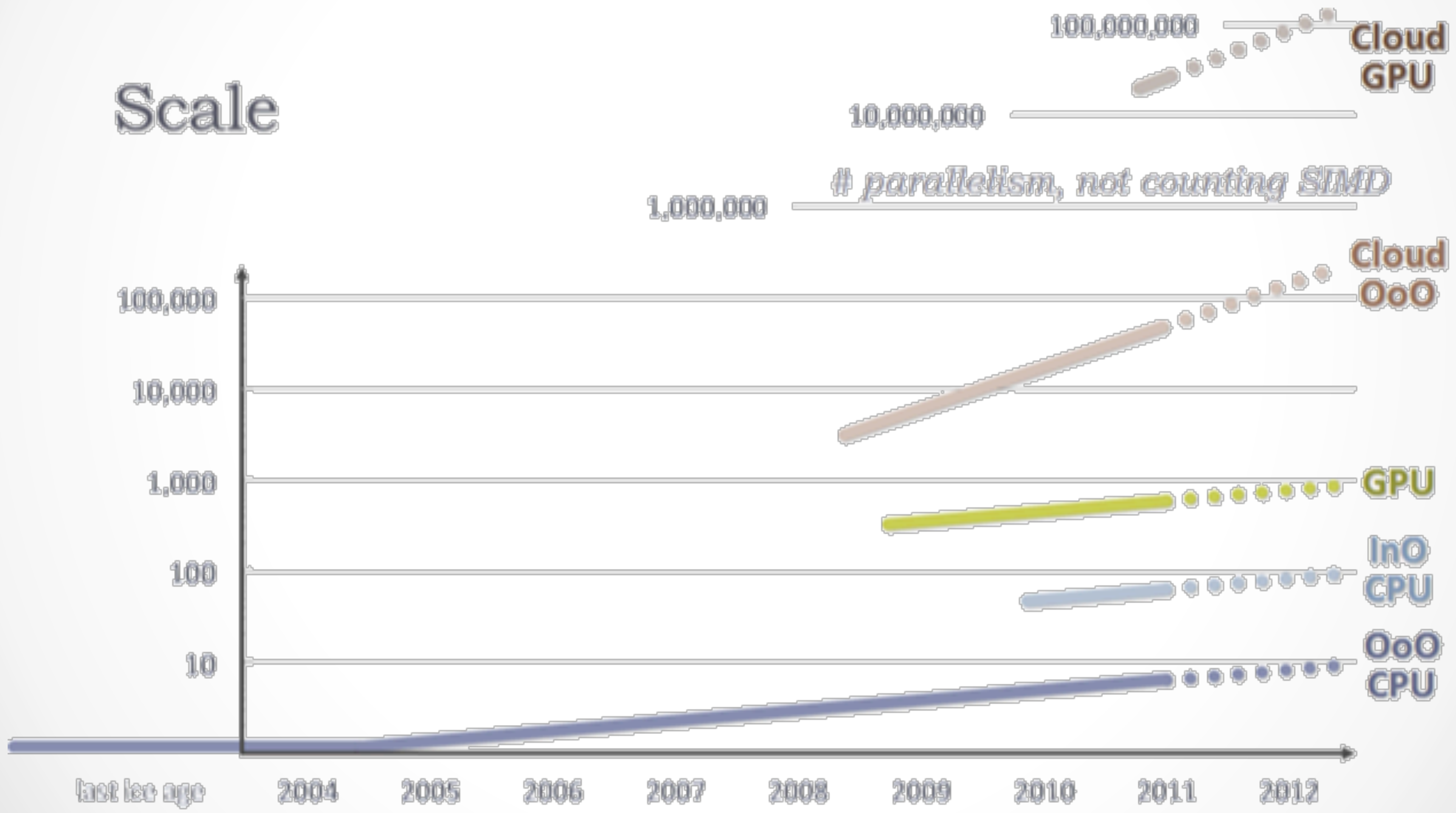
Conclusions

- A software (r)evolution has (finally) started
- Need to invest significantly in common software tools and common solutions
- Need new design and coding paradigm
  - Design *for* parallelism and vectorization

"We stand today on the edge of a new frontier – the frontier of the 1960's - a frontier of unknown opportunities and perils - a frontier of unfulfilled hopes and threats."
John F. Kennedy

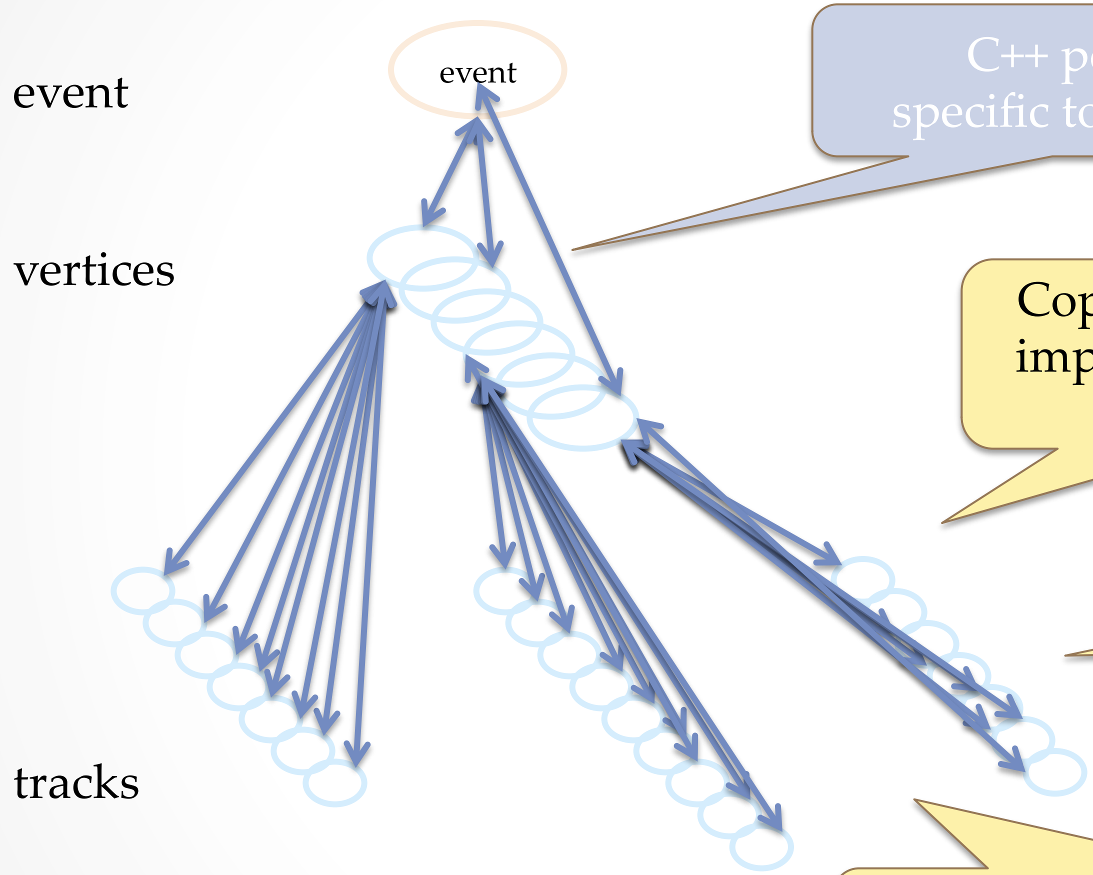
Backup slides

Scales



Herb Sutter, 2012

Data Structures & parallelism



C++ pointers specific to a process

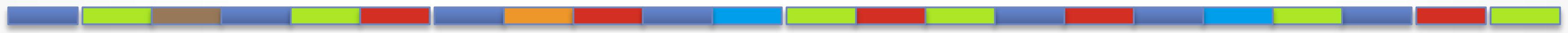
Copying the structure implies a relocation of all pointers

I/O is a nightmare

Update of the structure from a different thread implies a lock/mutex

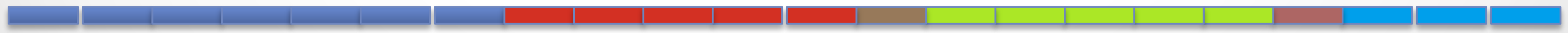
Data Structures & Locality

sparse data structures defeat the system memory caches



Group object elements/ collections such that the storage matches the traversal processes

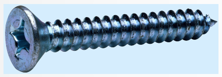
For example: group the cross-sections for all processes per material instead of all materials per process



Parallelism: key points

Minimize the sequential/synchronization parts (Amdhal law): Very difficult

Run the same code (processes) on all cores to optimize the memory use (code and read-only data sharing)


Job-level is better than event-level parallelism for offline systems.

Use the good-old principle of data locality to minimize the cache misses.

Exploit the vector capabilities but be careful with the new/delete/gather/scatter problem

Reorganize your code to reduce tails

Language and Tools

- C++11 threads
- Intel TBB (Thread Building Block)
- ArBB (Array Building Blocks)
- Intel Cilk++
- OpenCL (Open Computing Language)
- OpenACC (Directive for Accelerators)
- NVIDIA CUDA (Compute Unified Device Architecture)
- Vector classes (Vc)
- VDT (**V**ectorize**d** Ma**t**hematical Library)

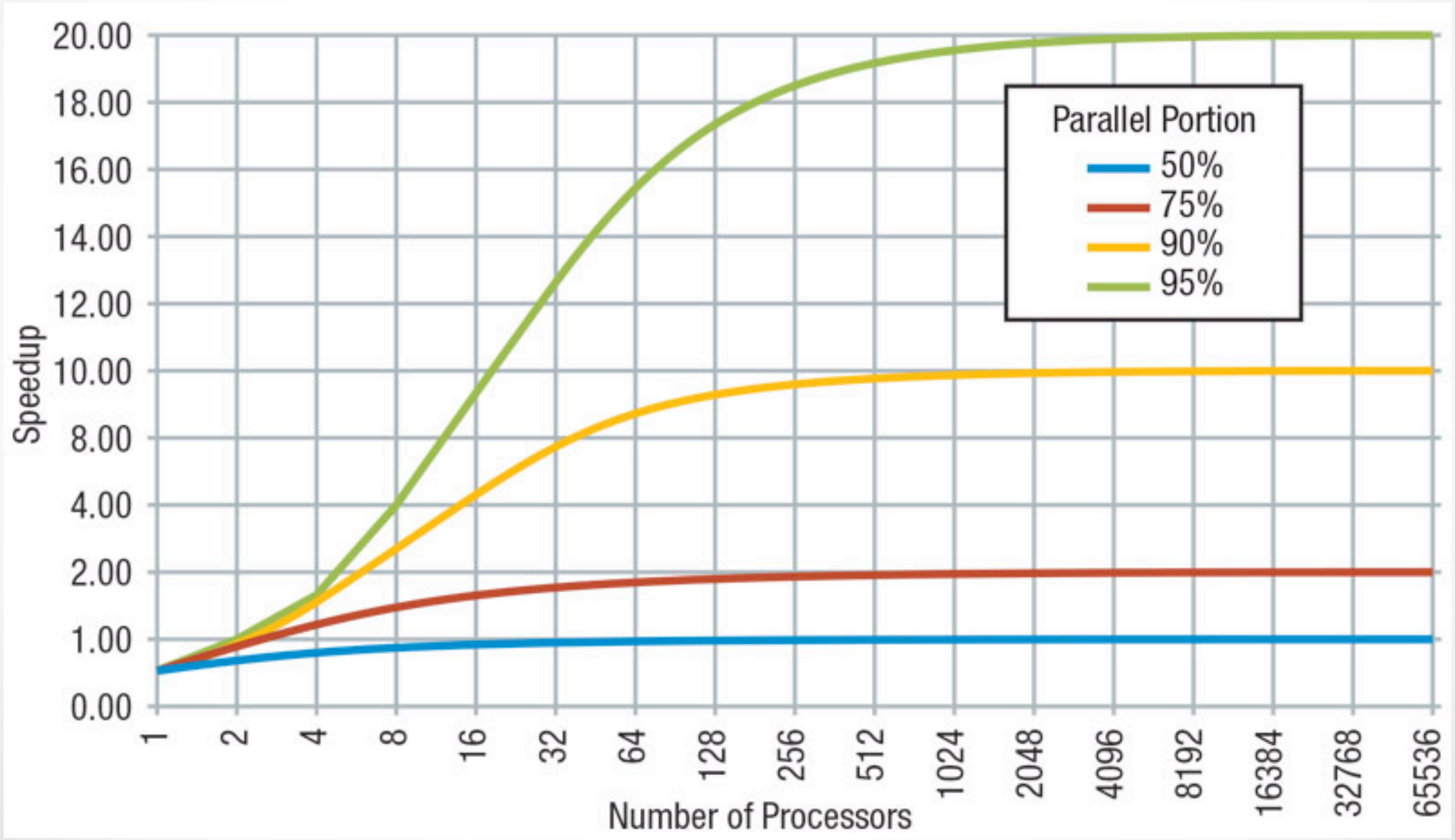
Analysis And Vectorization?

- The levels of analysis
  - TTree::Draw, implemented as TSelector using TTreeFormula
  - Single Core TSelector
  - Multi Core TSelector via ProofLite
  - Multi Node TSelector via Proof
  - Multi Site TSelector via POD
  - TSelector often substituted by experiment frameworks.
- Currently based on the one-event at time paradigm
  - Vectorization only available in user code for 'complex' inner event analysis
  - But most often each event are ***independent***
- Significant performance gain plausible by introducing vectorization through the data flow but
  - Often unzipping bound (followed by unboxing, the I/O)
  - Will require significant library **and some user** code changes.

Other Disk Trends

- To boost performance, many customers are using flash memory within servers, as well as solid-state drives in storage arrays, to cache speed-sensitive data before writing it to slower, but less expensive and higher-capacity hard drives.
- This new platform, he says, is not only an "order of magnitude faster" than its older storage but delivers high performance, availability and disaster recovery without the need for extensive management. The performance gain achieved by writing data to six storage nodes without transferring it over the network means storing multiple copies of the same data. However, says Piesche, the low price of commodity disk and servers make the trade-off worthwhile.

Amdahl's Law



Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
4.00
2.00
1.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors
Parallel Portion
50%
75%
90%
95%