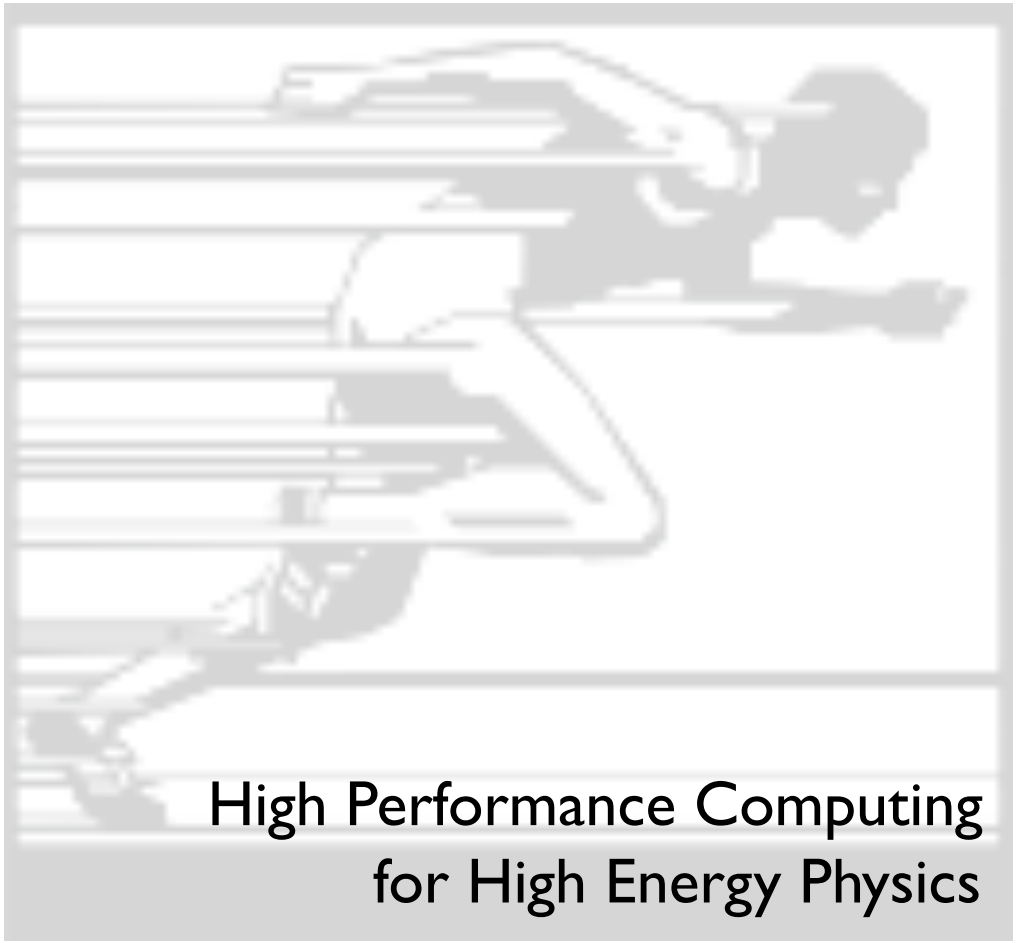
The challenge of adapting HEP physics software applications to run on many-core cpus



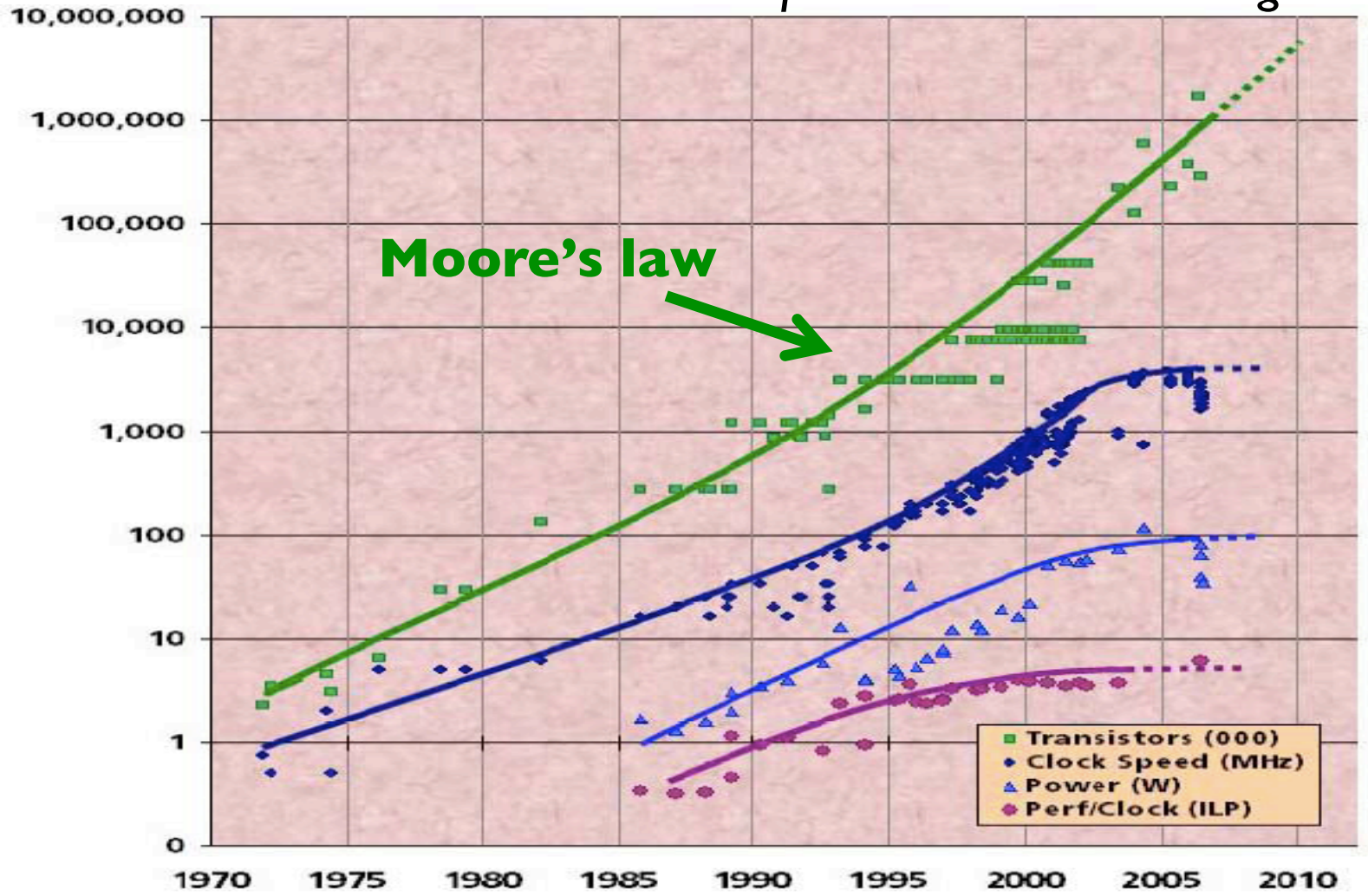
CHEP, March '09

Vincenzo Innocente
CERN

Computing in the years zero

Transistors used to increase *raw-power*

Increase *global power*



The 'three walls’

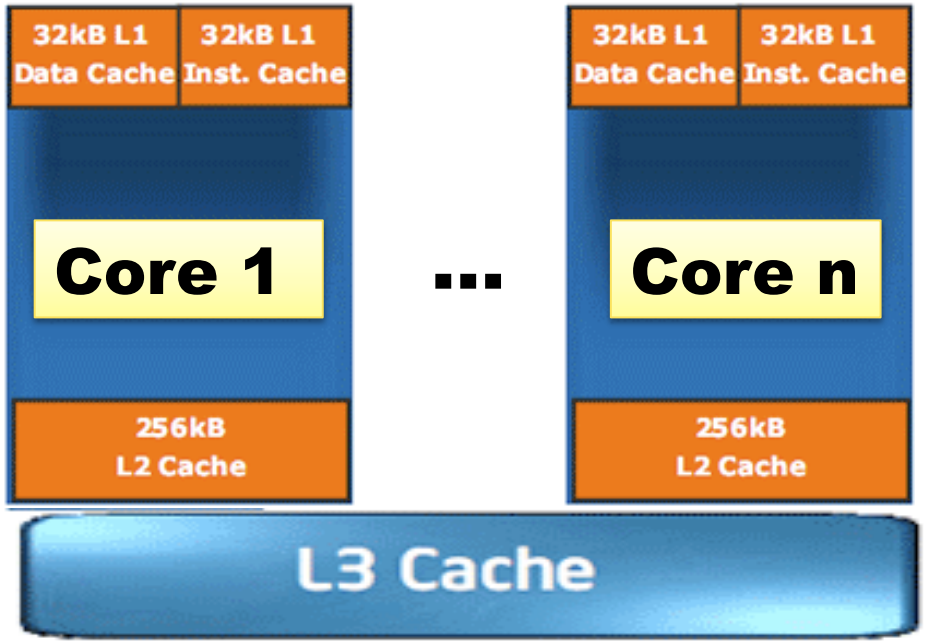
While hardware continued to follow Moore’s law, the perceived exponential grow of the “effective” computing power faded away in hitting three “walls”:

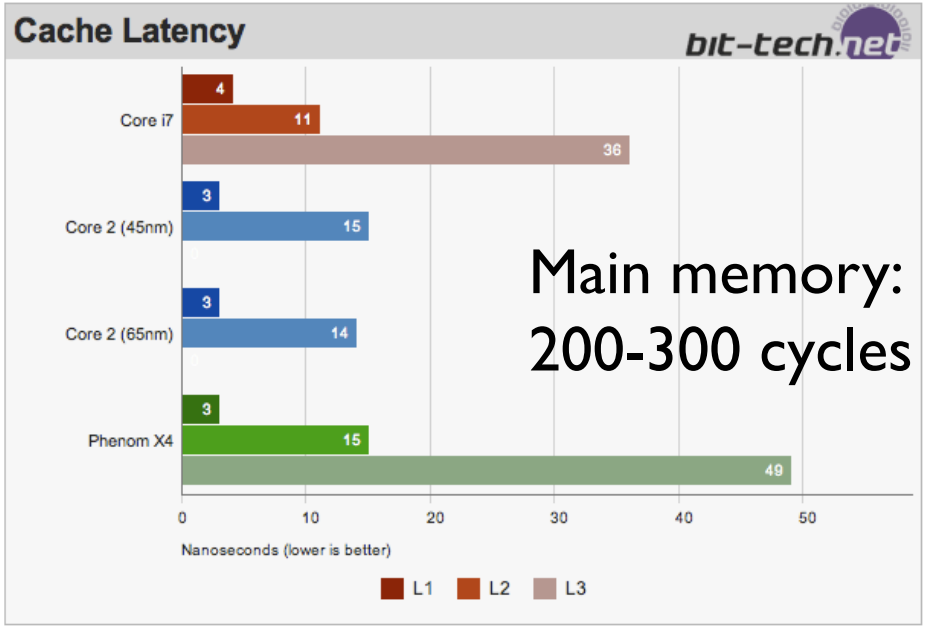
– The memory wall
– The power wall
– The instruction level parallelism (micro-architecture) wall

A turning point was reached and a new paradigm emerged: **multicore**

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

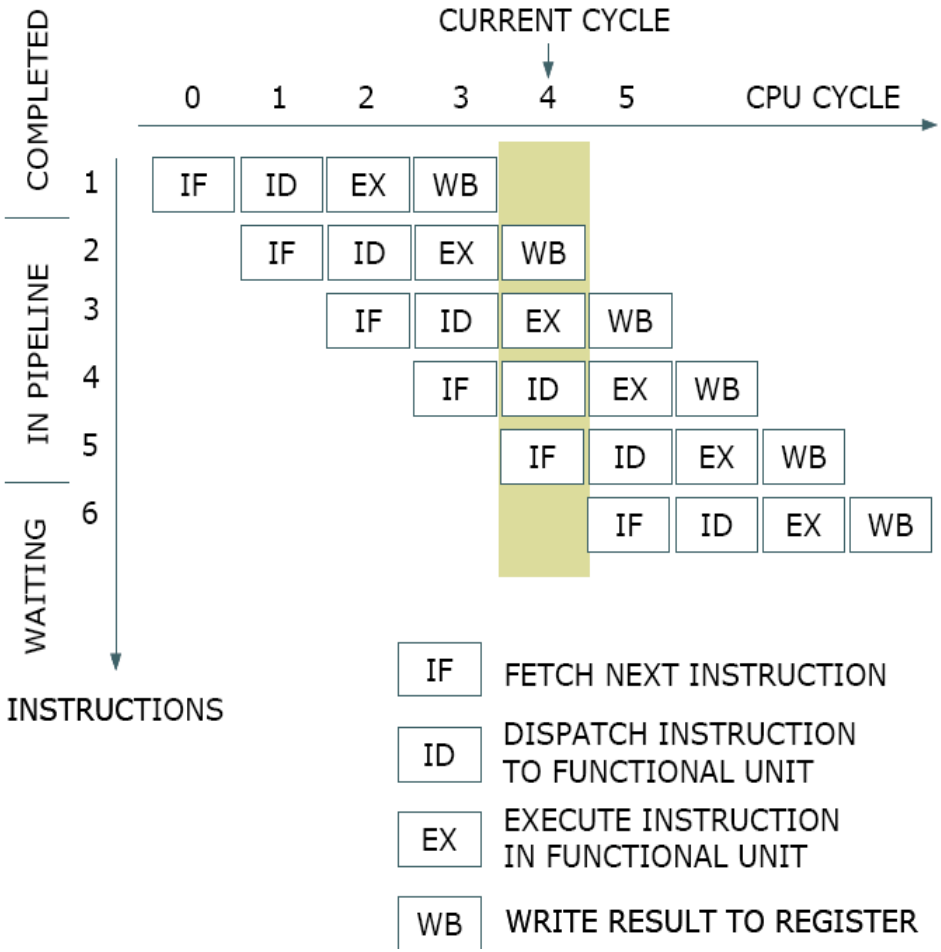
The ‘power wall’

- 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.

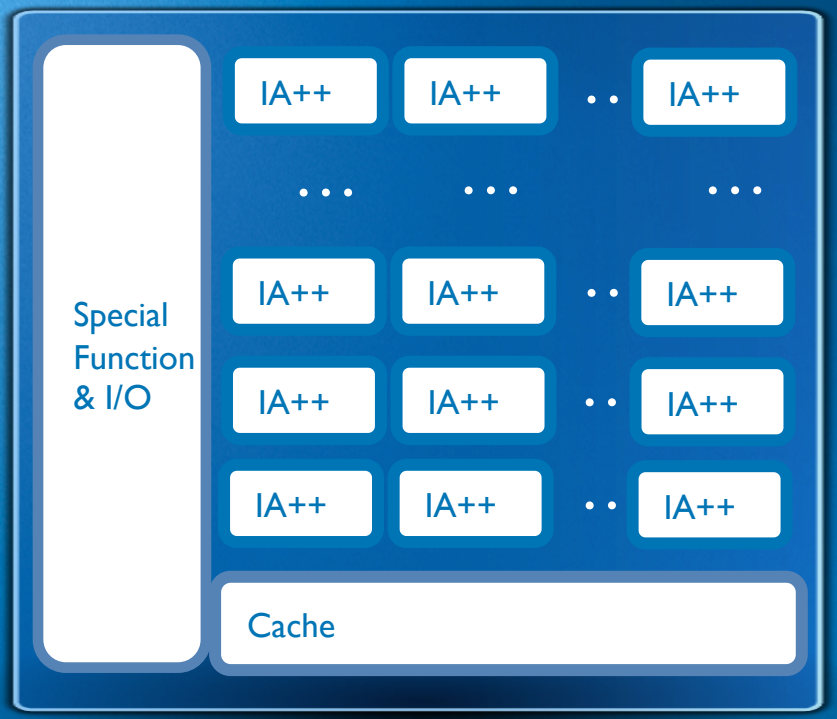
- How else increase the number of instruction per unit-time: **Go parallel!**

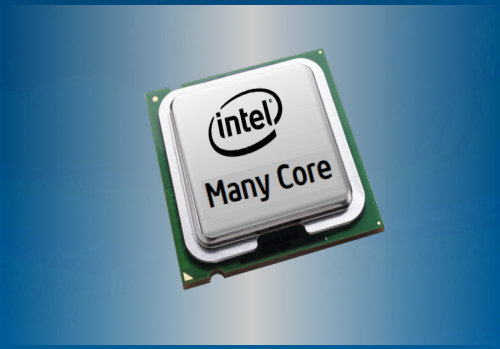
The ‘Architecture walls’

- 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 ILP
- In practice inter-instruction data dependencies and run-time branching limit the amount of achievable ILP



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)
– 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?

See talks by P.Elmer, G.Eulisse, S. Binet

- 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
    - Not locality of code and data
- This situation is already bad today, will become only worse in future architectures

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*

- All this is absolutely necessary, still not sufficient

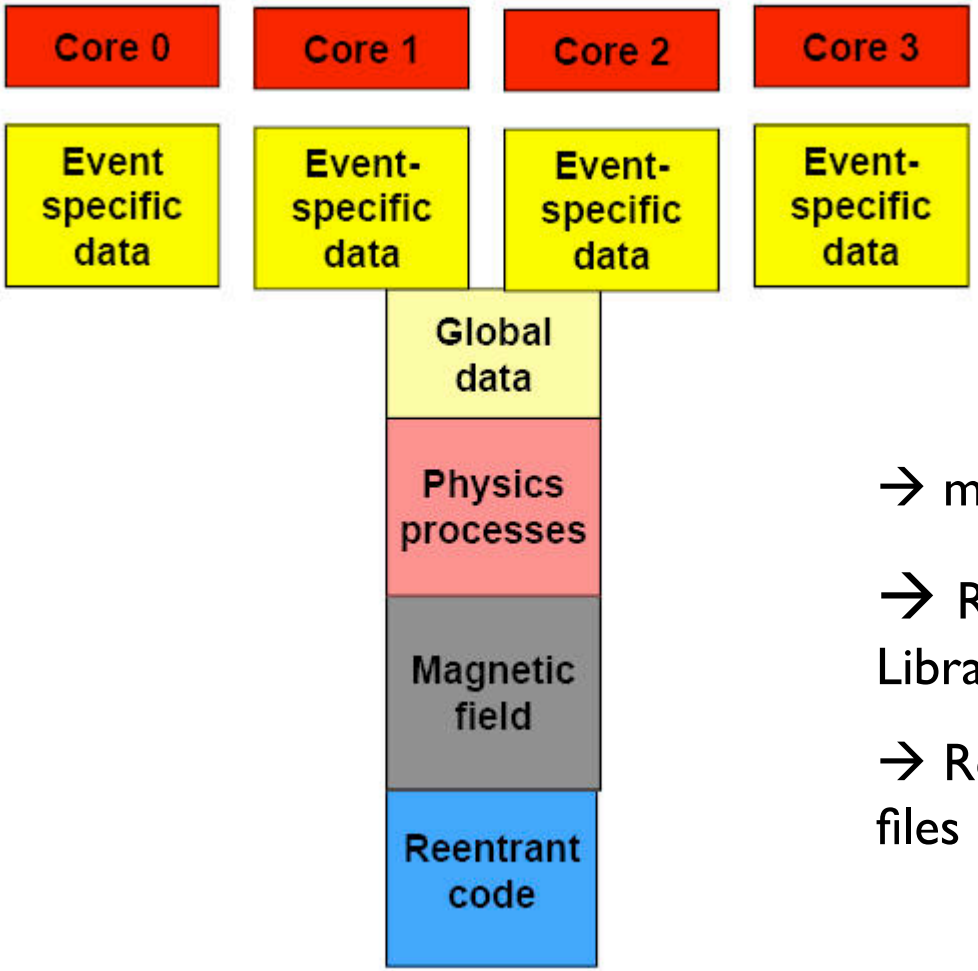
HEP software on multicore: a R&D effort

- Collaboration among experiments, IT-departments, projects such as OpenLab, Geant4, ROOT, Grid
- Target multi-core (8-24/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
  - » SSE/AltiVec, OpenCL
- Prototype solutions
  - » *Adapt legacy software*
  - » *Look for innovative solution for the future*

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

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!
    - Not enough, many hidden (optimization) details such as state-caches
- "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

Exploit Copy on Write

See Sebastien Binet's talk

- 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)

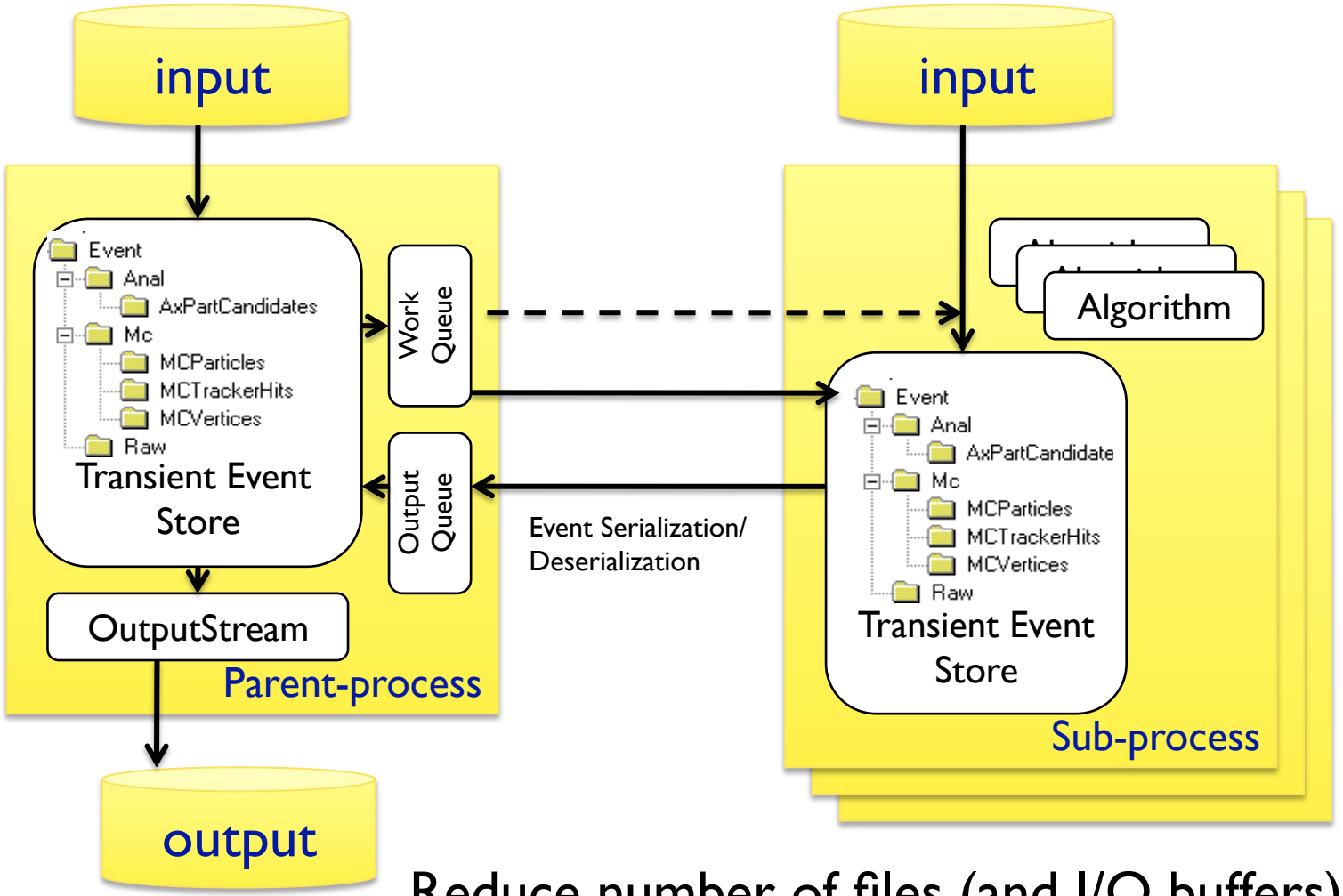
Memory (ATLAS)
One process: *700MB VMem and 420MB RSS*
COW
(before) evt 0: private: 004 MB | shared: 310 MB
(before) evt 1: private: 235 MB | shared: 265 MB
. . .
(before) evt50: private: 250 MB | shared: 263 MB

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

Handling Event Input/Output

See talk by Pere Mato & Eoin Smith

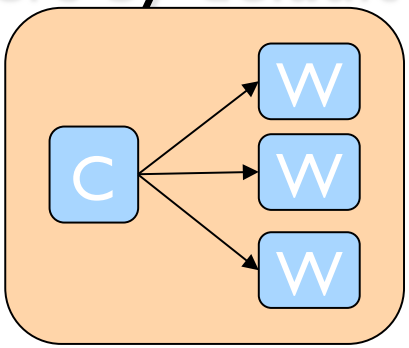


Reduce number of files (and I/O buffers) by 1-2 orders of magnitude

PROOF Lite

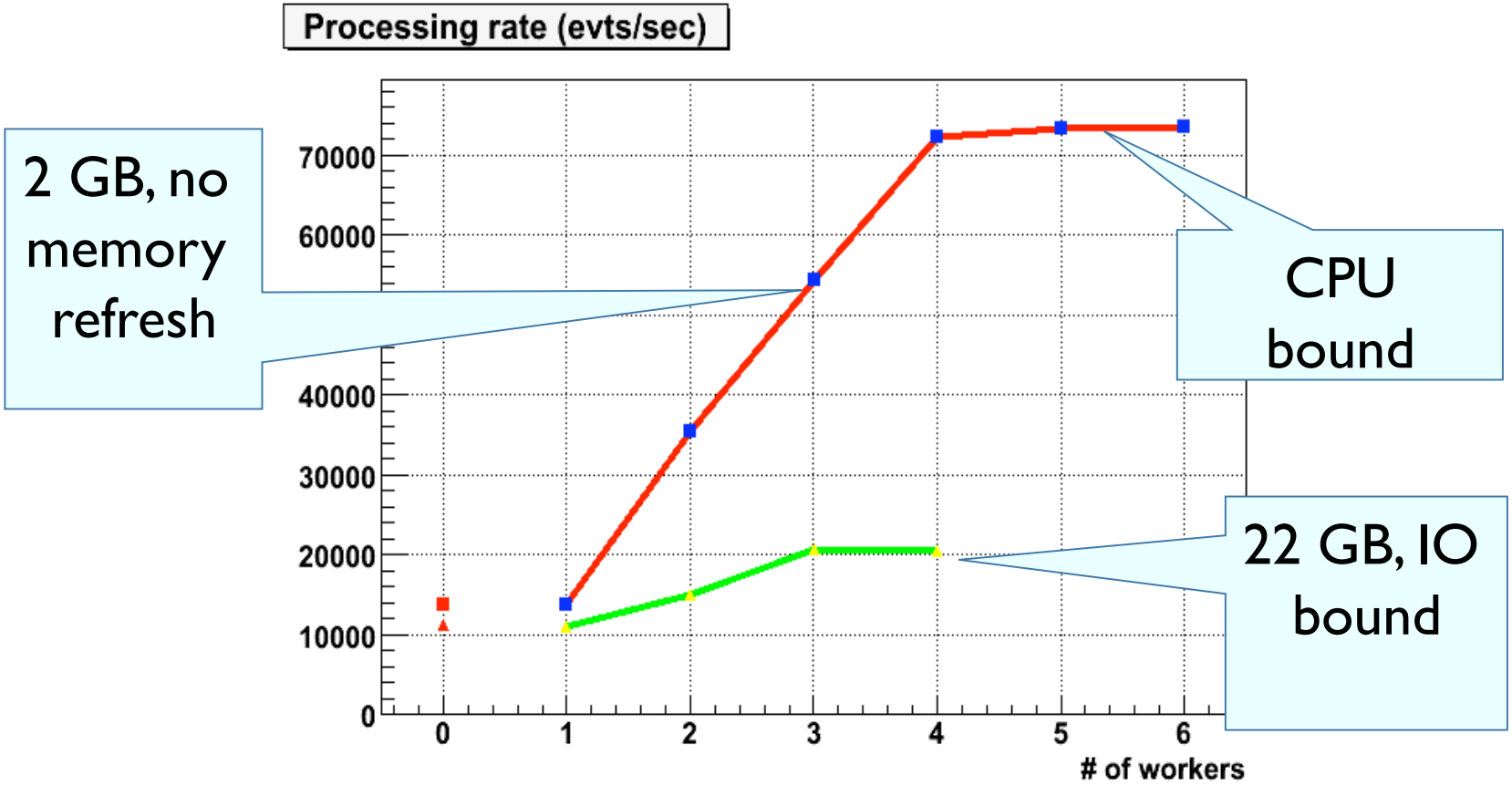
See talk by Gerry Ganis & Fons Rademakers

- PROOF Lite is a realization of PROOF in 2 tiers
  - The client starts and controls directly the workers
  - Communication goes via UNIX sockets
- No need of daemons:
  - workers are started via a call to 'system' and call back the client to establish the connection
- Starts N_{CPU} workers by default



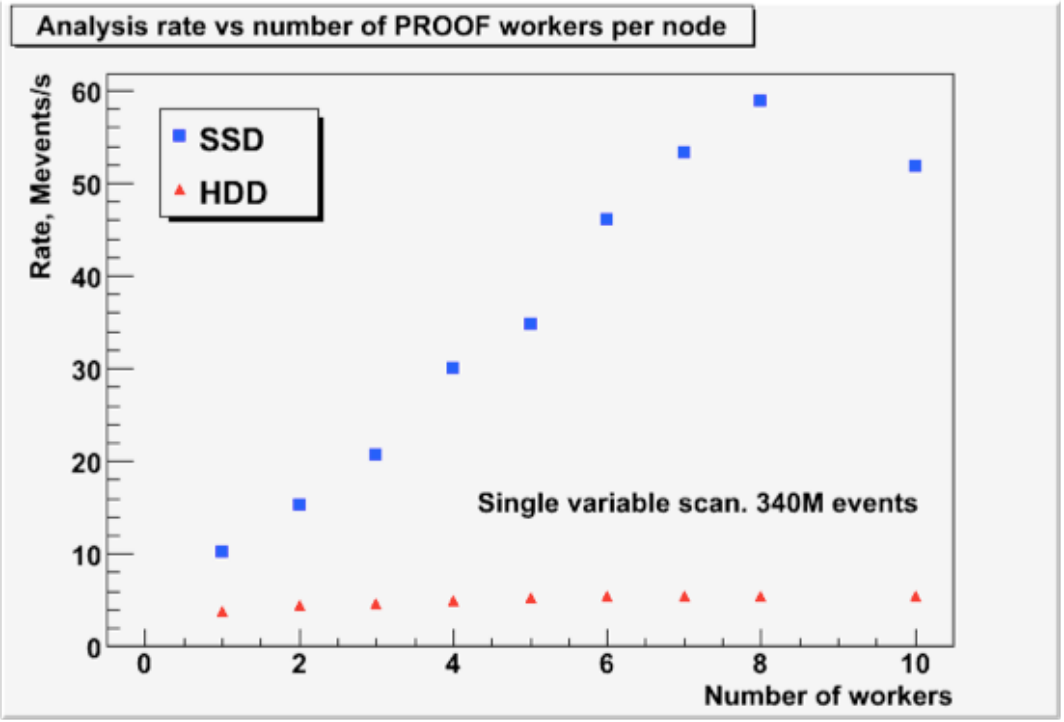
Scaling processing a tree, example (4core box)

- Datasets: 2 GB (fits in memory), 22 GB



SSD vs HDD on 8 Node Cluster

See Sergey Panitkin's talk



Solid State Disk:
120GB for 400Euro

- Aggregate (8 node farm) analysis rate as a function of number of workers per node
- Almost linear scaling with number of nodes

Progress toward a **thread-parallel** Geant4

Gene Cooperman and Xin Dong (NEU Boston)

- Master/Worker paradigm
- Event-level parallelism: separate events on different threads
  - only 1 RAM : increase sharing of memory between threads
- Phase I : code sharing, but no data sharing *Done*
- Phase II : sharing of geometry, materials, particles, production cuts *Done, undergoing validation*
- Phase III : sharing of data for EM physics processes *In Progress*
  - ❑ Physics tables are read-only, but small caches and different API
- Phase IV : other physics processes *Todo*
- Phase V : General Software Schema: new releases of sequential Geant4 drive corresponding multi-threaded releases *In Progress*
  - Patch **parser.c** of **gcc** to identify static and globals declarations in G4
  - Currently 10,000 lines of G4 modified automatically + 100 lines by hand

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
  - » Effort 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.

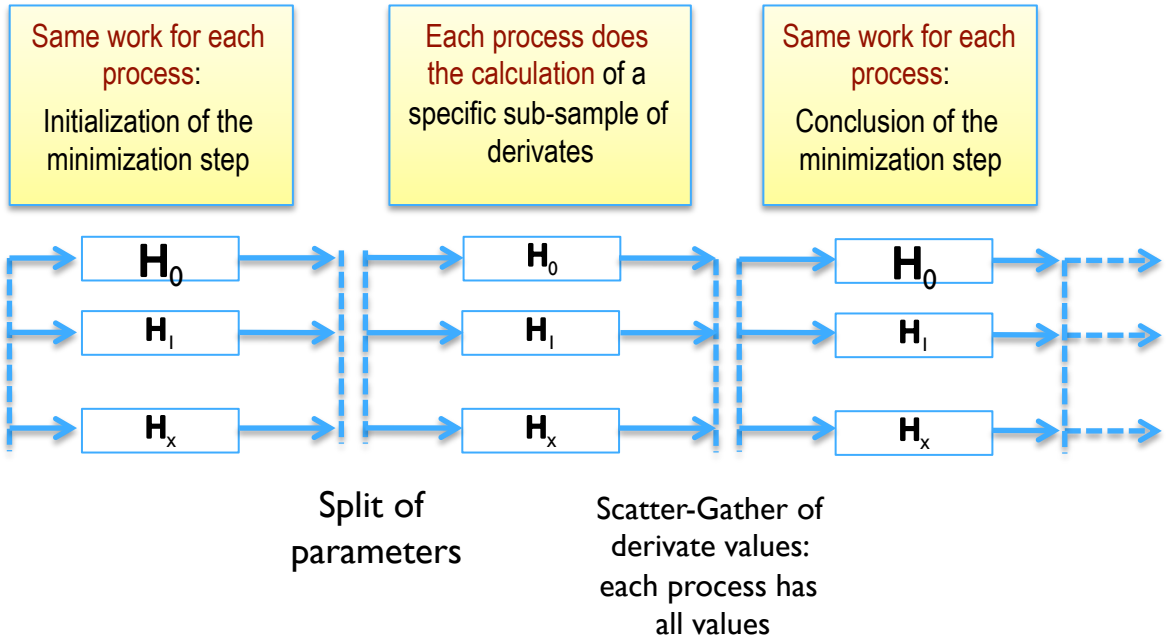
Parallel MINUIT

see poster by Alfio Lazzaro, Lorenzo Moneta

–Minimization of Maximum Likelihood or χ^2 requires iterative computation of the gradient of the NLL $\left.\frac{\partial NLL}{\partial \hat{\theta}}\right|_{\hat{\theta}_0} \approx \frac{NLL(\hat{\theta}_0 + \hat{\mathrm{d}}) - NLL(\hat{\theta}_0 - \hat{\mathrm{d}})}{2\hat{\mathrm{d}}}$

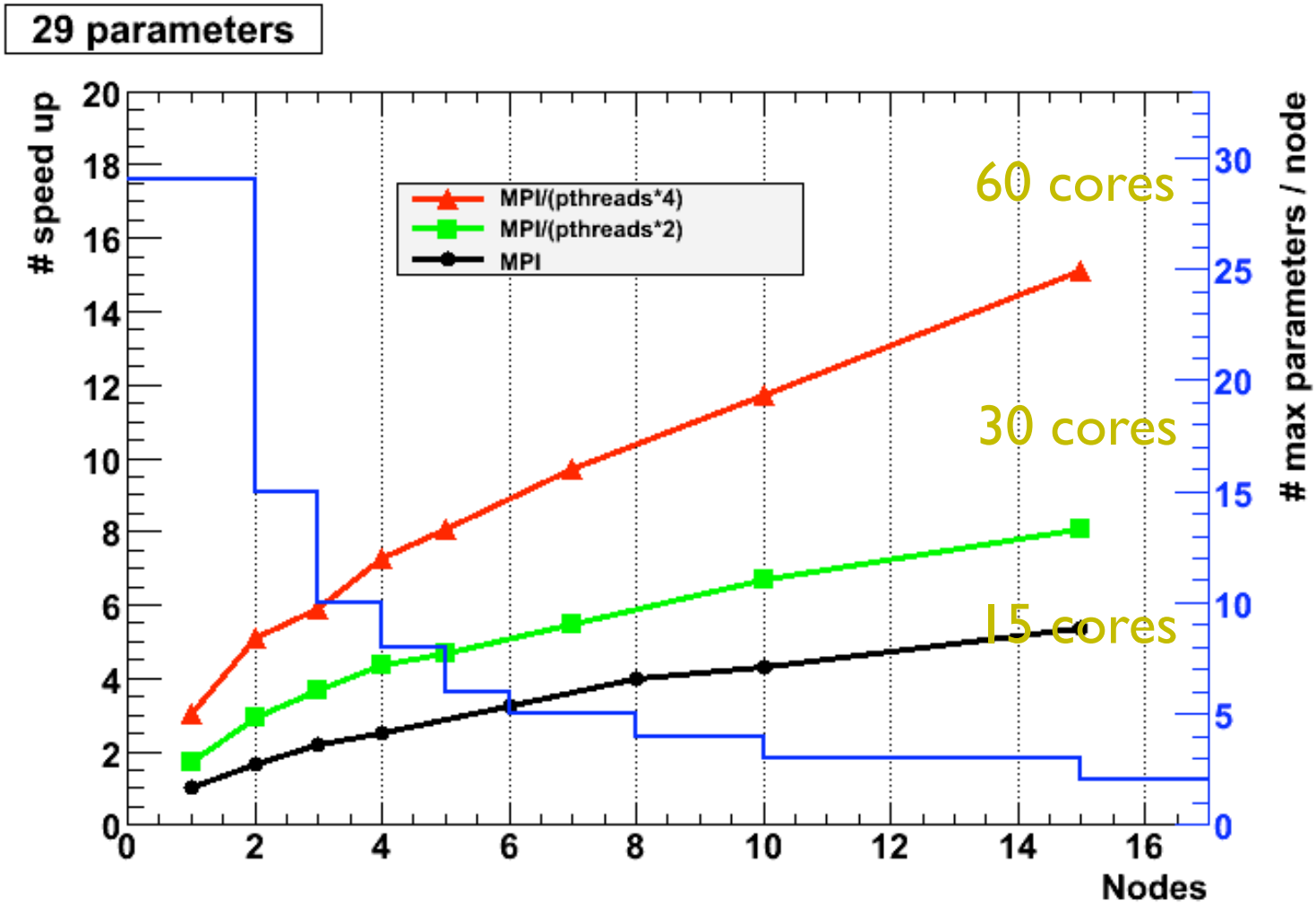
–Two strategies for the parallelization of the Gradient and NLL calculation:

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



Minuit Parallelization – Example (2008)

– Waiting time for fit to converge down from a week to a night:
  » iteration on results back to a human timeframe!



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
- "Learn" how to run MT/MP jobs on the grid

Explore new Frontier of parallel computing

– Hardware and software technologies (SSD, KSM, OpenCL) 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
  » Use of “spare” cycles/core for ancillary and speculative computations
– Algorithm concept have to change: Think Parallel!