Programming in Multi-cores Era

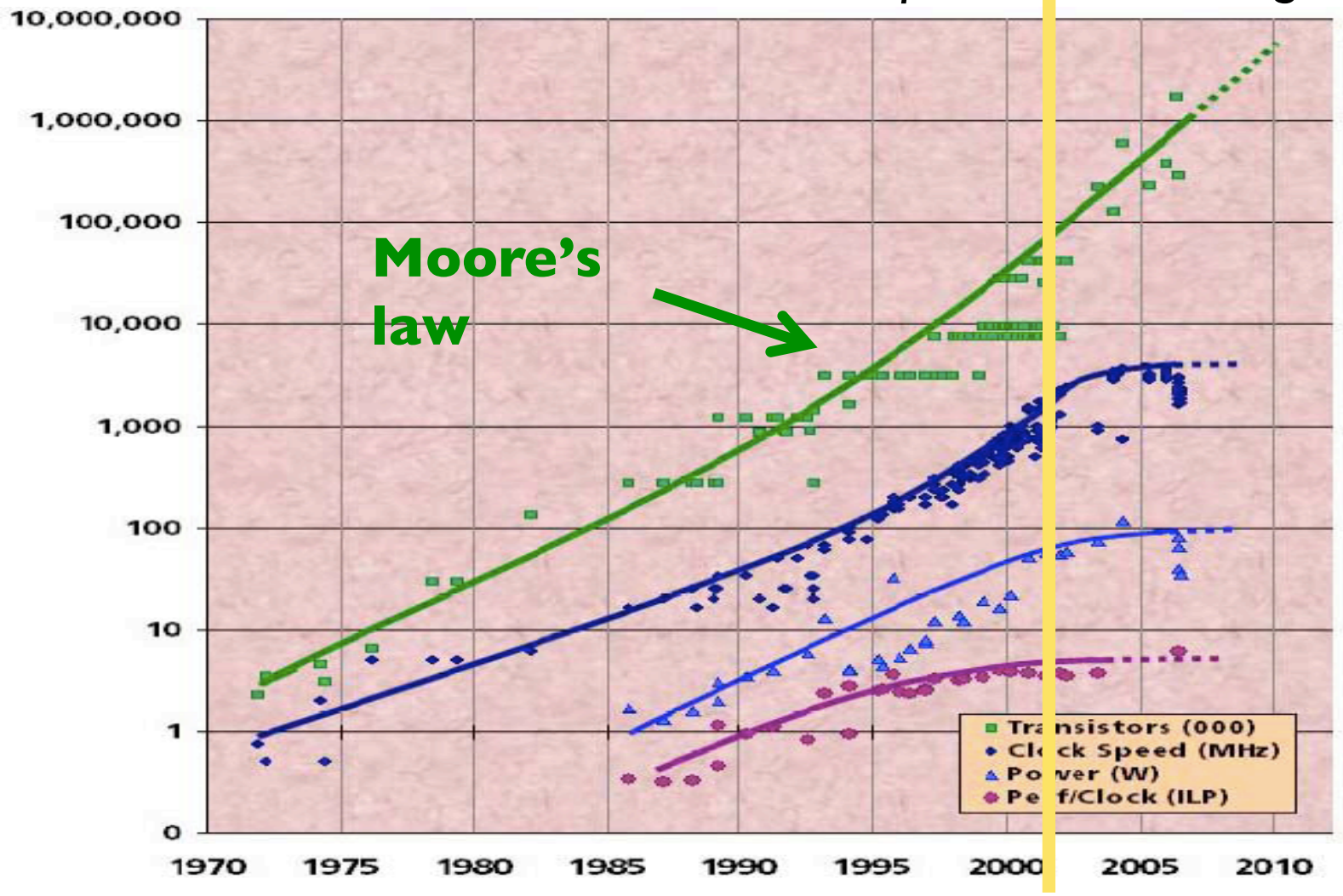
Alfio Lazzaro
CERN (European Organization for Nuclear Research) Openlab
Geneva

Jaipur (India), February 22nd, 2010

Introduction: why multi-cores?

Computing in the years

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

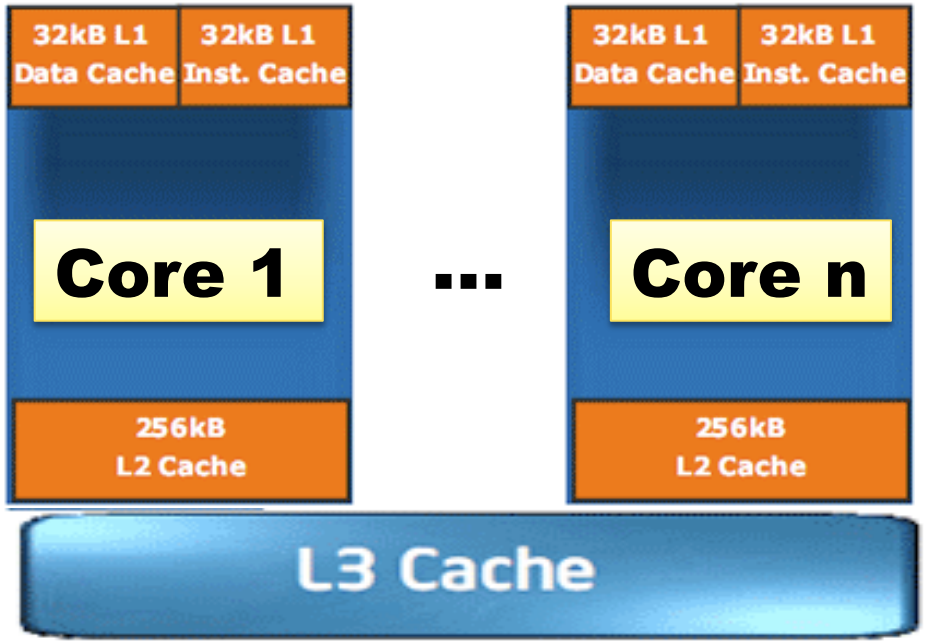
The 'three walls'

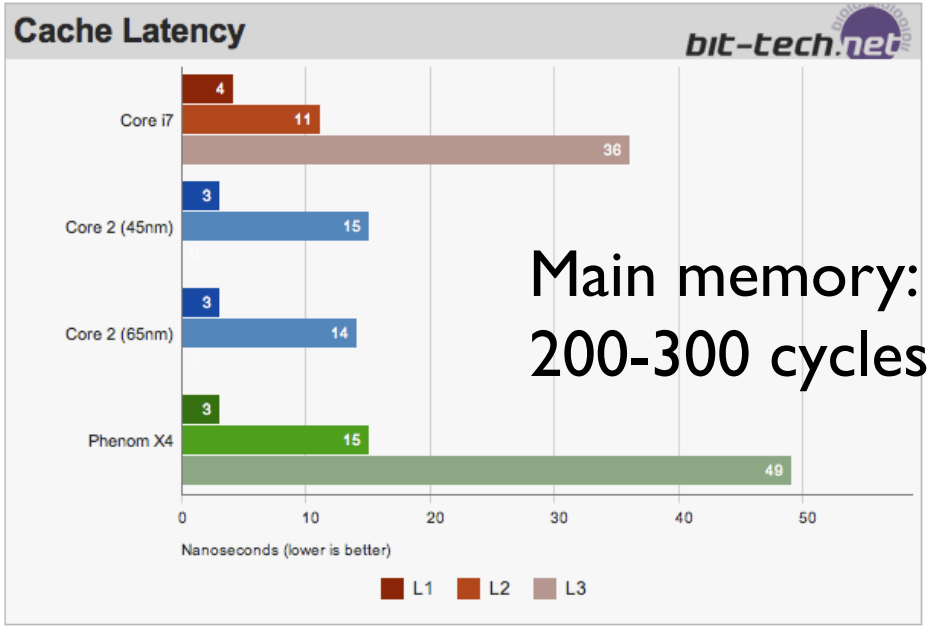
While hardware continued to follow Moore's law, the perceived exponential grow of the "effective" computing power faded away in hitting three "walls":

1. The memory wall
2. The power wall
3. The instruction level parallelism (ILP) 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

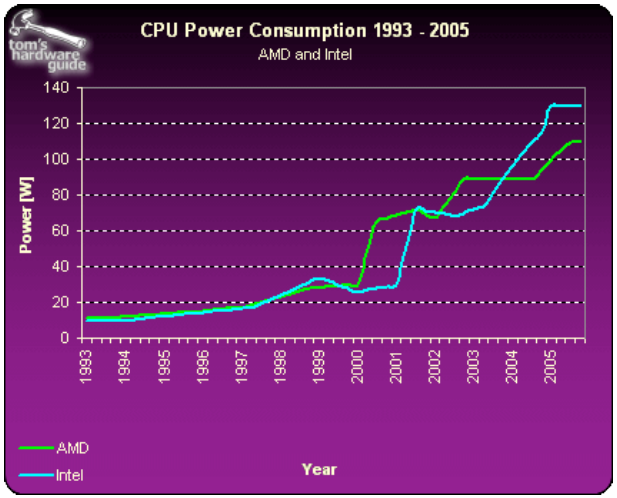


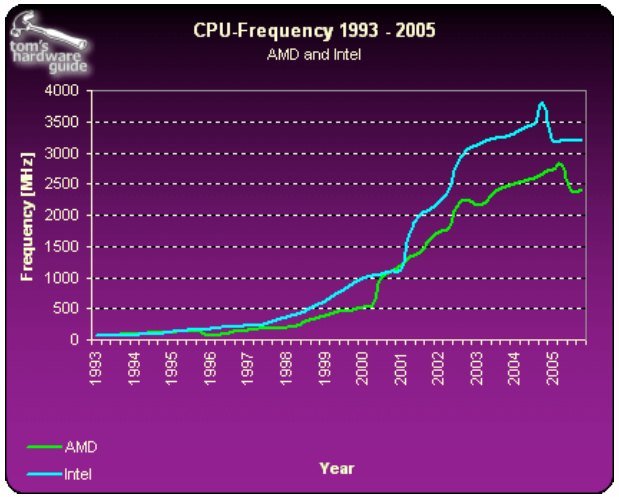


Main memory: 200-300 cycles

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!

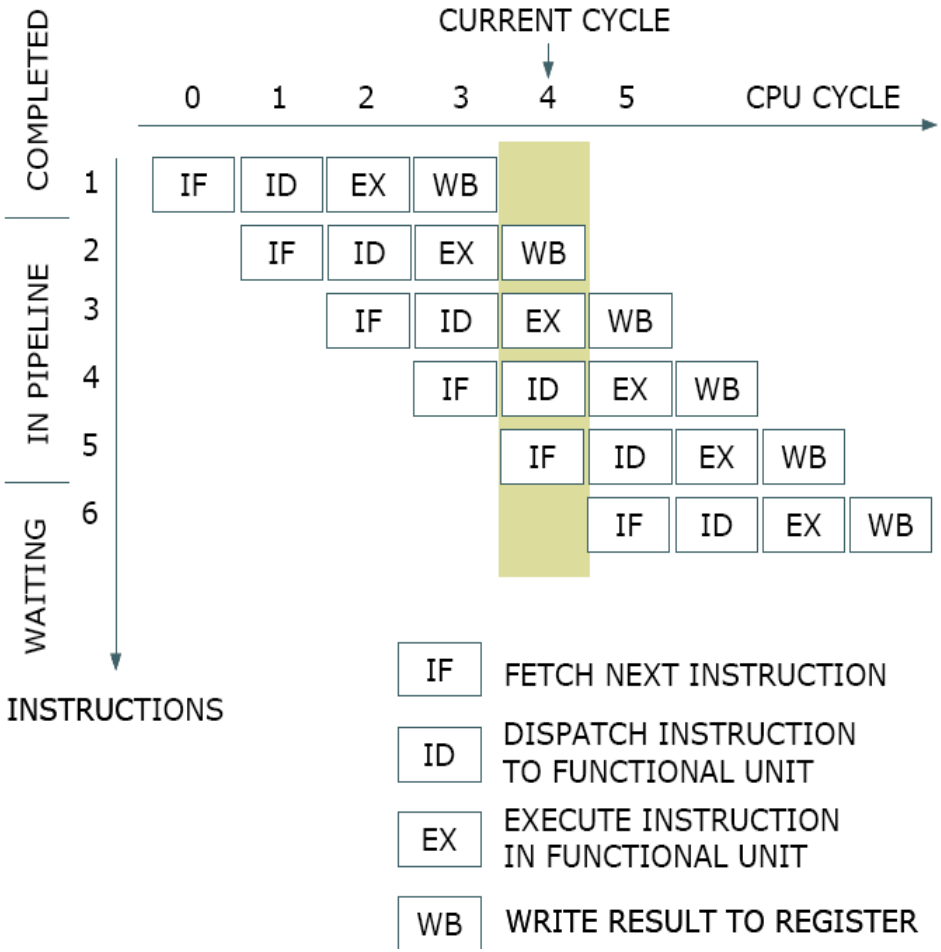




http://www.processor-comparison.com/power.html

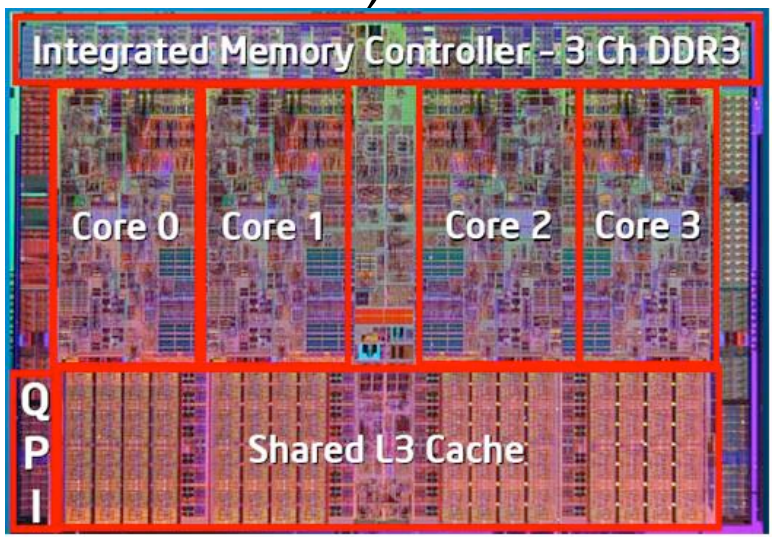
The ‘Architecture walls’

- 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



Think Parallel!

- 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
- Dedicated architectures:
  - » GPGPU (NVIDIA, ATI-AMD, Intel Larrabee)
  - » IBM CELL
  - » FPGA (Reconfigurable computing)



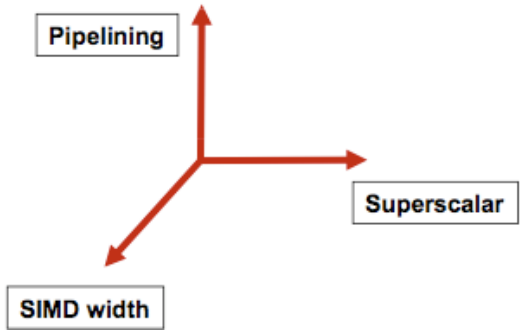
Parallelization: definitions

The Challenge of Parallelization

Exploit all 7 "parallel" dimensions of modern computing architecture for High Performance Computing (HPC)

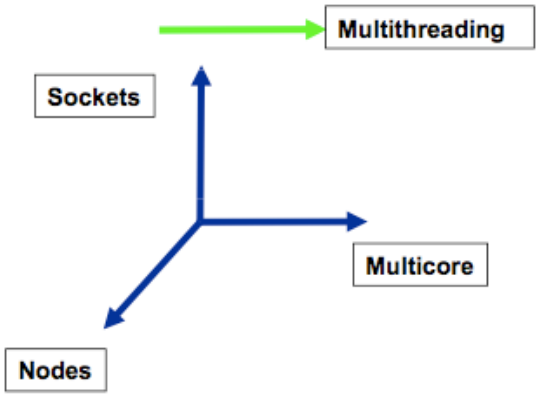
–Inside a core (climb the ILP wall)

1. Superscalar: Fill the ports (maximize instruction 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 level resources)
6. Sockets: Fill up a box (share high level resources)



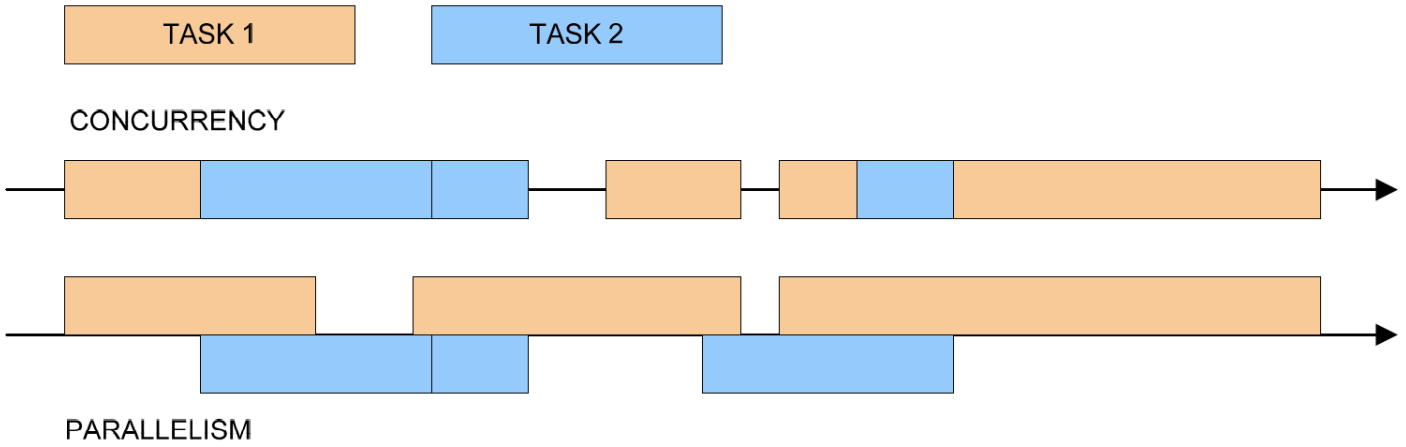
–LAN & WAN (climb the network wall)

7. Optimize scheduling and resource sharing on the Grid

In this lecture

Definition of concurrency/parallelism

- Concurrent programming: the program can be logically split in independent parts (threads)
  - » Concurrent programs can be executed sequentially on a single CPU by interleaving the execution steps of each computational process
  - » Benefits can arise from the use of I/O resources
    - Example: a thread is waiting for a resource reply (e.g. data form disk), so another thread can be executed by the CPU
    - Keep CPU busy as much as possible
- Parallel execution: Independent parts of a program execute simultaneously



Other Some Basic Definitions

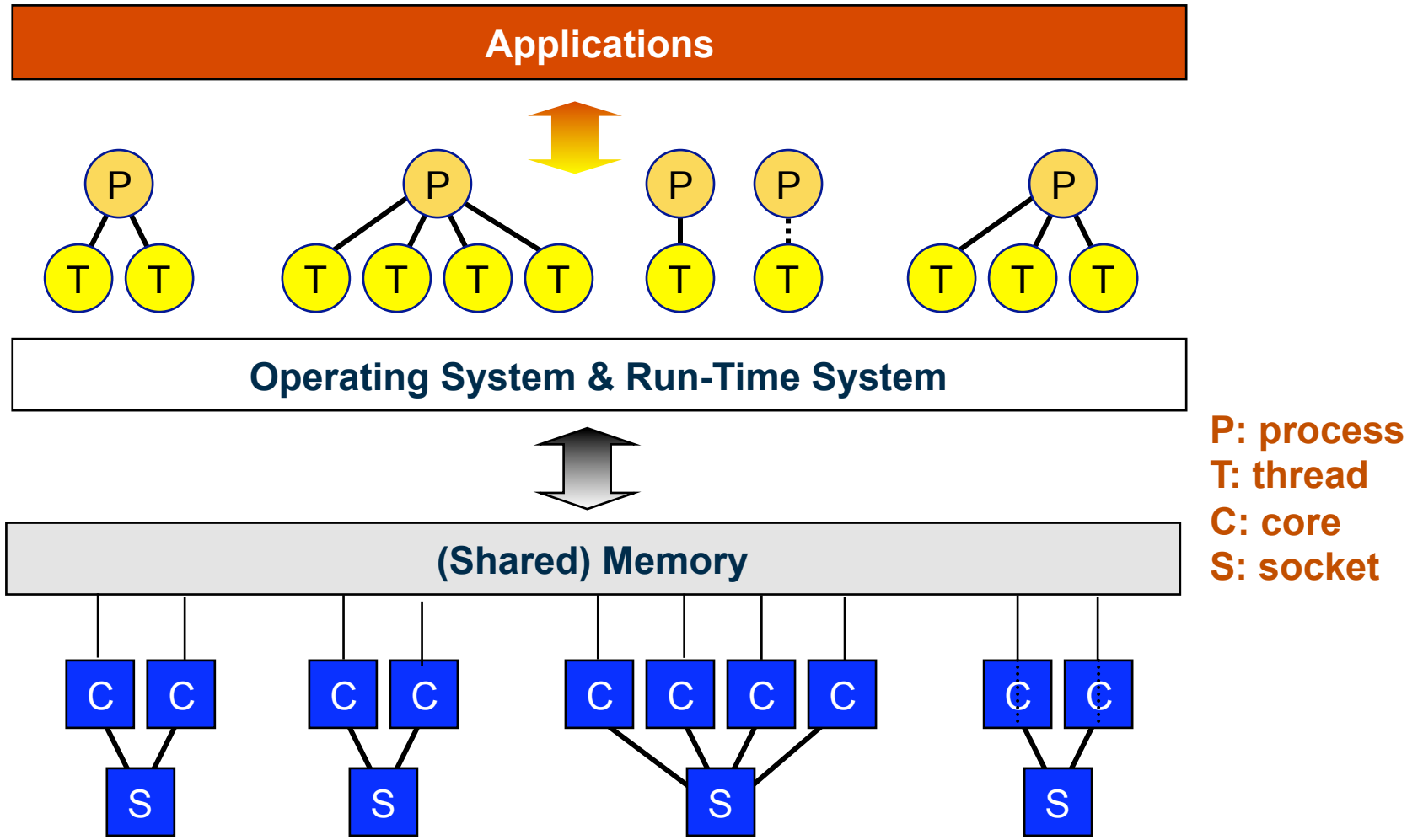
Software level

- Process: an instance of a computer program that is being executed (sequentially). It contains the program code and its current activity: its own "address space" with all the program code and data, its own file descriptors with the operating system permission, its own heap and its own stack.
- SW Thread: a process can *fork* in different threads of execution. These threads run in the same address space, share the same program code, the operating system resources as the process they belong to. Each thread gets its own stack.

Hardware level

- Core: unity for executing a software process or thread: execution logic, cache storage, register files, instruction counter (IC)
- HW Thread: addition of a set of register files plus IC

Parallel Environments



Schematic overview

Examples of multi-cores

- HW-Threads x Cores x Sockets = "Slots" available
  - » CELL Processor: 9 x 1 x 1 = 9
  - » Dual-socket Intel quad-core i7: 2 x 4 x 2 = 16
  - » Quad-socket Intel Dunnington server: 1 x 6 x 4 = 24
  - » 16-socket IBM dual-core Power6: 2 x 2 x 16 = 64
  - » Tesla Nvidia GPU: 1 x 240 x 1 = 240
  - » Quad-socket Sun Niagara (T2+): 8 x 8 x 4 = 256
  - » Radeon ATI/AMD GPU: 1 x 1600 x 1 = 1600
- In future we expect an increase on the number of slots: Thousands!!!
  - » Are we ready to write parallel code for those massive (many-cores) parallel architectures?

(Parallel) Software Engineering

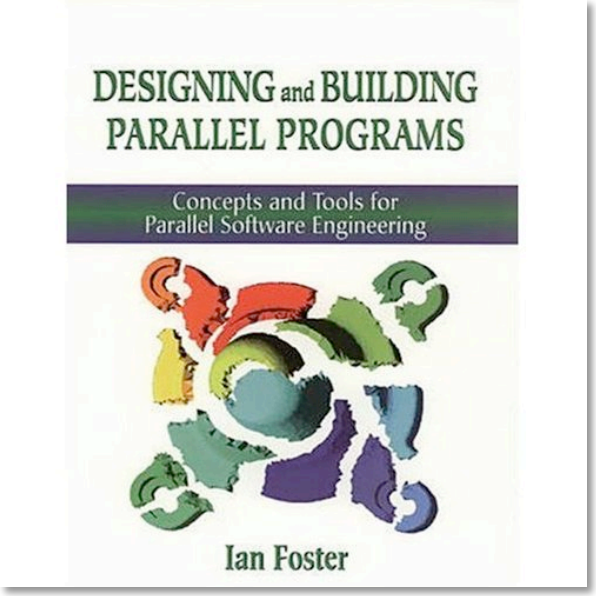
Engineering Parallel software follows the "usual" software development process with one difference: **Think Parallel!**

- **Analyze, Find & Design**
  - Analyze problem, Finding and designing parallelism
- **Specify & Implement**
  - How will you express the parallelism (in detail)?
- **Check correctness**
  - How will you determine if the parallelism is right or wrong?
- **Check performance**
  - How will you determine if the parallelism improves over sequential performance?

Foster's Design Methodology

Four Steps:

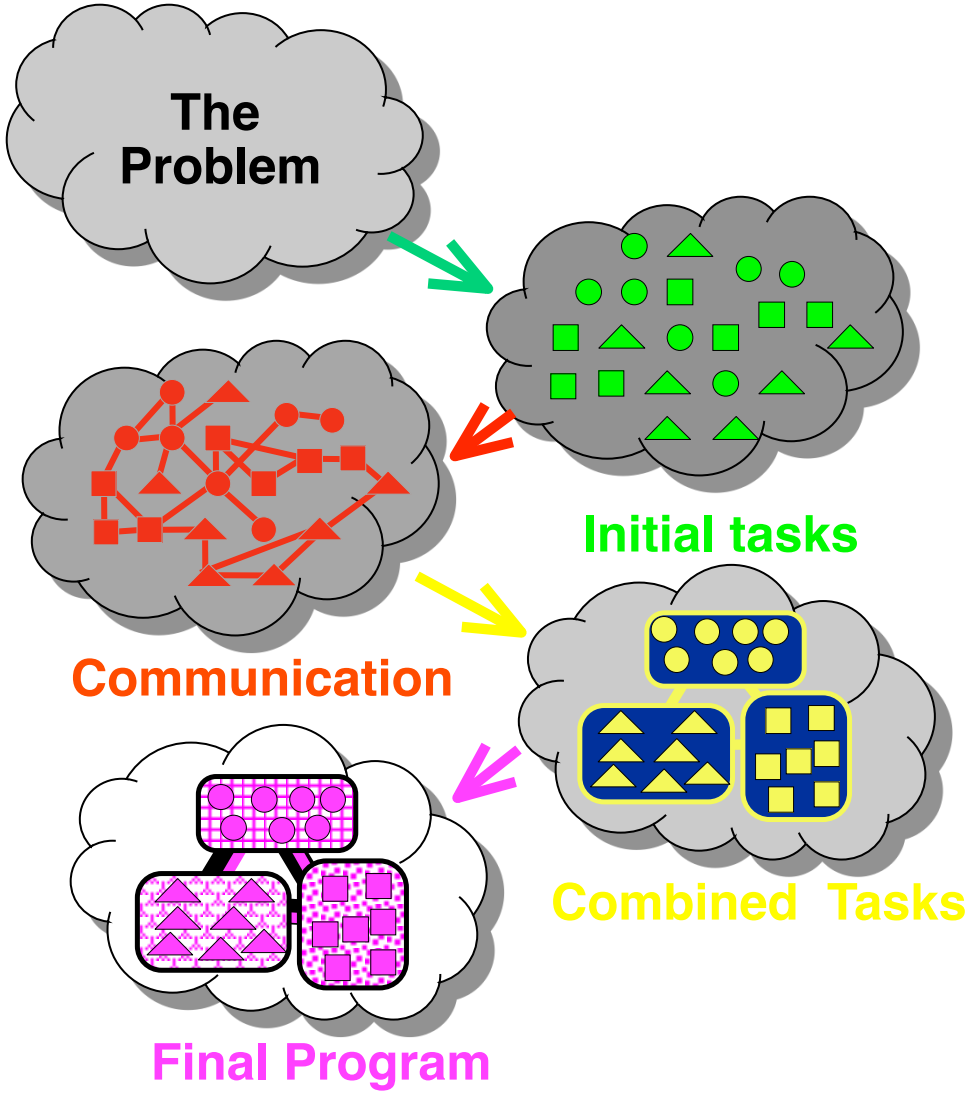
- Partitioning
  - » Dividing computation and data
- Communication
  - » Sharing data between computations
- Agglomeration
  - » Grouping tasks to improve performance
- Mapping
  - » Assigning tasks to processors/threads



From "*Designing and Building Parallel Programs*" by Ian Foster

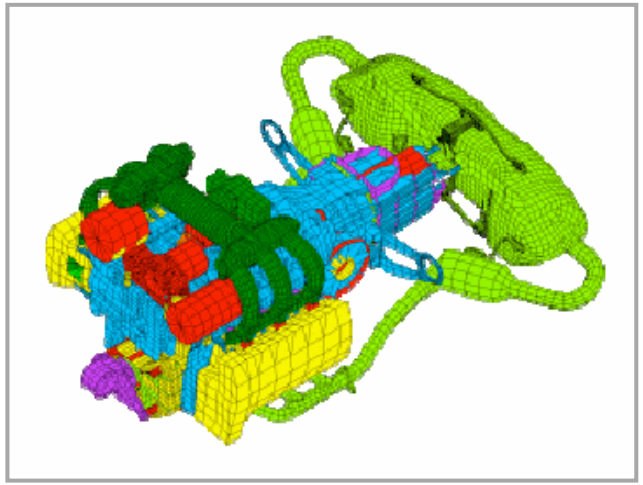
Designing Threaded Programs

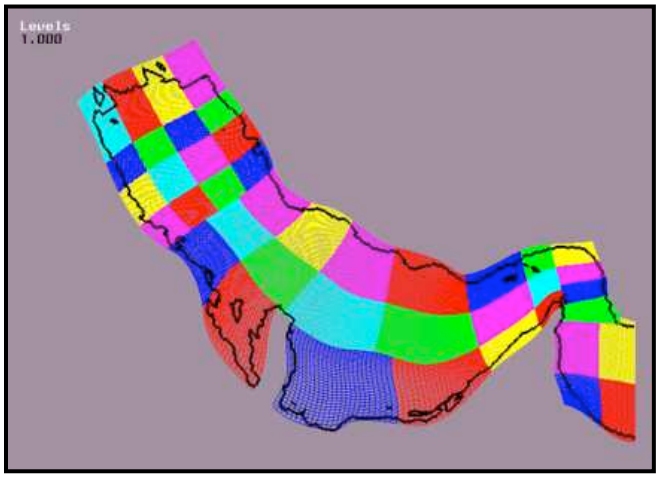
- **Partition**
  - » Divide problem into tasks
- **Communicate**
  - » Determine amount and pattern of communication
- **Agglomerate**
  - » Combine tasks
- **Map**
  - » Assign agglomerated tasks to created threads



Domain (Data) Decomposition

- Exploit large datasets whose elements can be computed independently
  - » Divide data and associated computation amongst threads
  - » Focus on largest or most frequently accessed data structures
  - » Data parallelism: same operations(s) applied to all





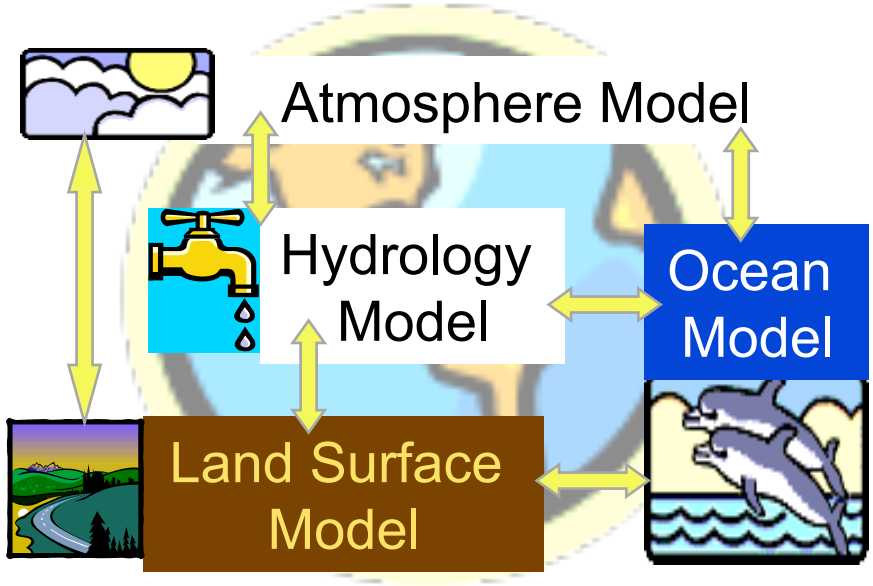
Functional Decomposition

– Divide computation based on a natural set of independent functions

» Predictable organization and dependencies

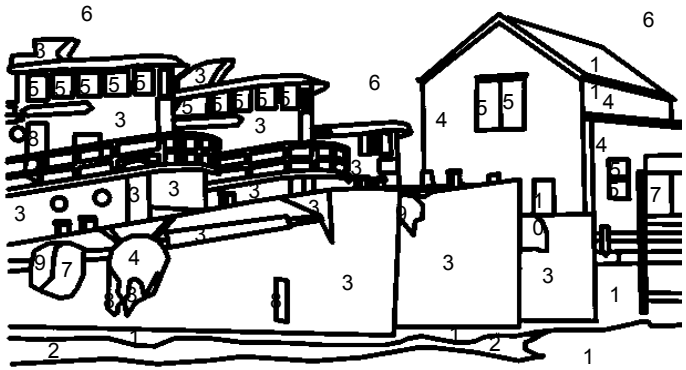
» Assign data for each task as needed

• Conceptually a single data value or transformation is performed repeatedly



Activity (Task) Decomposition

- Divide computation based on a natural set of independent tasks
  - » Non deterministic transformation
  - » Assign data for each task as needed
  - » Little communication
- Example: Paint-by-numbers
  - » Painting a single color is a single task



Parallelization: practical cases

When we want to parallelize

- Reduction of the wall-time: we want to achieve better performance, defined as (results response/execution) times

- Memory footprint: large data sample, so we want to split in different sub-samples

Typical problem suitable for parallelization

- The problem can be broken down into subparts (embarrassing parallelism):
  - » Each subpart is independent of the others
  - » No communication is required, except to split up the problem and combine the final results
  - » Ex: Monte-Carlo simulations

- Regular and Synchronous Problems:
  - » Same instruction set (regular algorithm) applied to all data
  - » Synchronous communication (or close to): each processor finishes its task at the same time
  - » Ex: Algebra (matrix-vector products), Fast Fourier transforms

Scalability issue in parallel applications

- Ideal case
  - » our programs would be written in such a way that their performance would scale automatically
  - » Additional hardware, cores/threads or vectors, would automatically be put to good use
  - » Scaling would be as expect:
    - If the number of cores double, scaling (speed-up) would be 2x (or maybe 1.99x), but certainly not 1.05x
- Real case
  - » Much more complicated situation…

Speed-up (Amdahl's Law)

– Definition:

$S \rightarrow$ speed-up

$N \rightarrow$ number of parallel processes

$T_1 \rightarrow$ execution time for sequential algorithm

$T_N \rightarrow$ execution time for parallel algorithm with N processes

$$S(N) = \frac{T_1}{T_N}$$

» Remember to balance the load between the processes. Final time is given by the slowest process!

– Maximum theoretical speed-up: Amdahl's Law

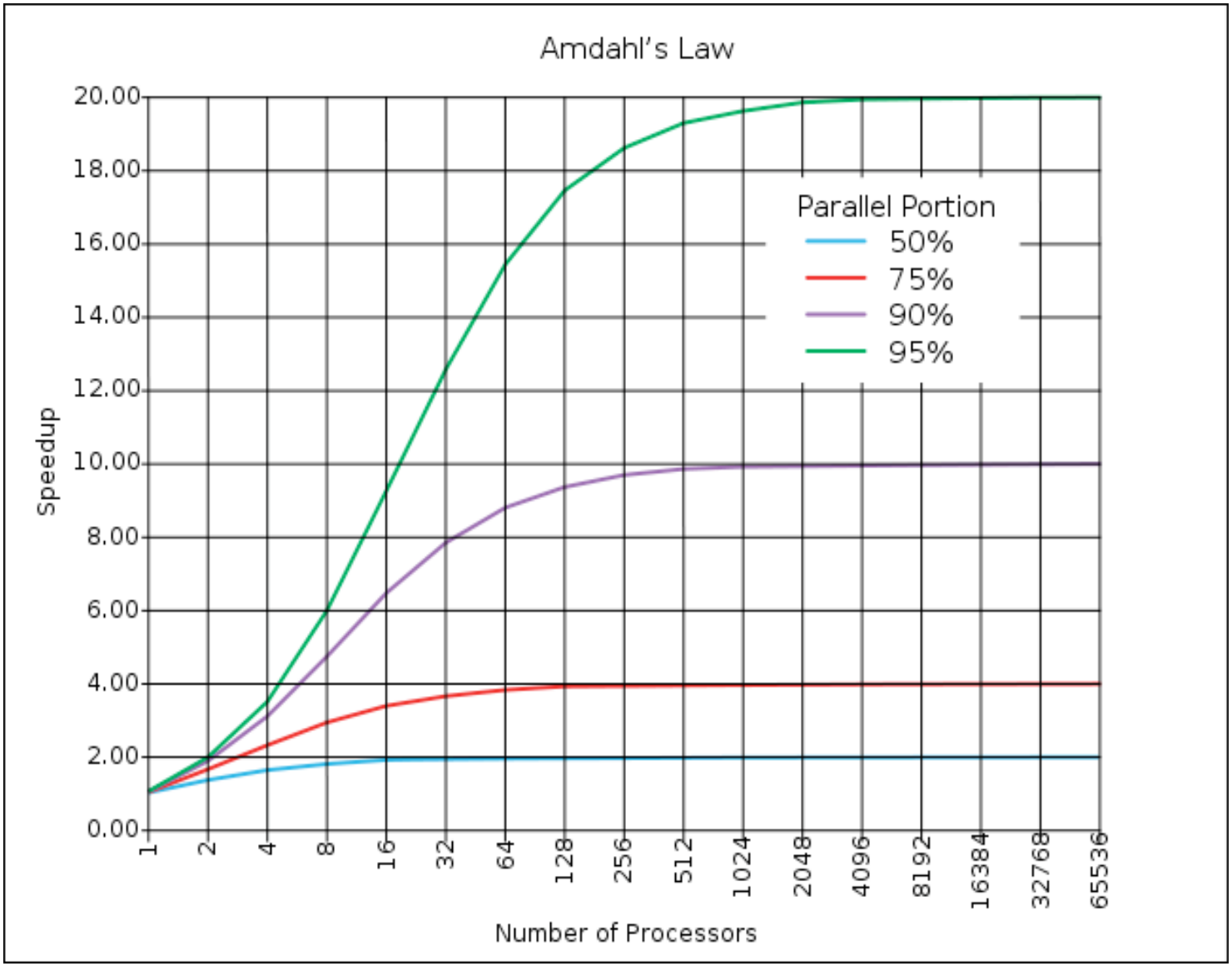
$P \rightarrow$ portion of code which is parallelized

$$S_{max}(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

» Implication: $S_{max}(N \rightarrow \infty) = \frac{1}{(1-P)}$

» Need to find good algorithms to be parallelized!

Amdahl's Law


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Speed-up: Gustafson's Law

– Any sufficiently large problem can be efficiently parallelized

$$S_{max}(N) = 1 + P(N-1)$$

$S \rightarrow$ speed-up

$N \rightarrow$ number of parallel processes

$P \rightarrow$ portion of code which is parallelized

– Amdahl's law VS Gustafson's law

» Amdahl's law is based on fixed workload or fixed problem size. It implies that the sequential part of a program does not change with respect to machine size (i.e, the number of processors). However the parallel part is evenly distributed by N processors

» Gustafson's law removes the fixed problem size on the parallel processors: instead, he proposed a fixed time concept which leads to scaled speedup for larger problem sizes

Amdahl's law VS Gustafson's law: A Driving Metaphor

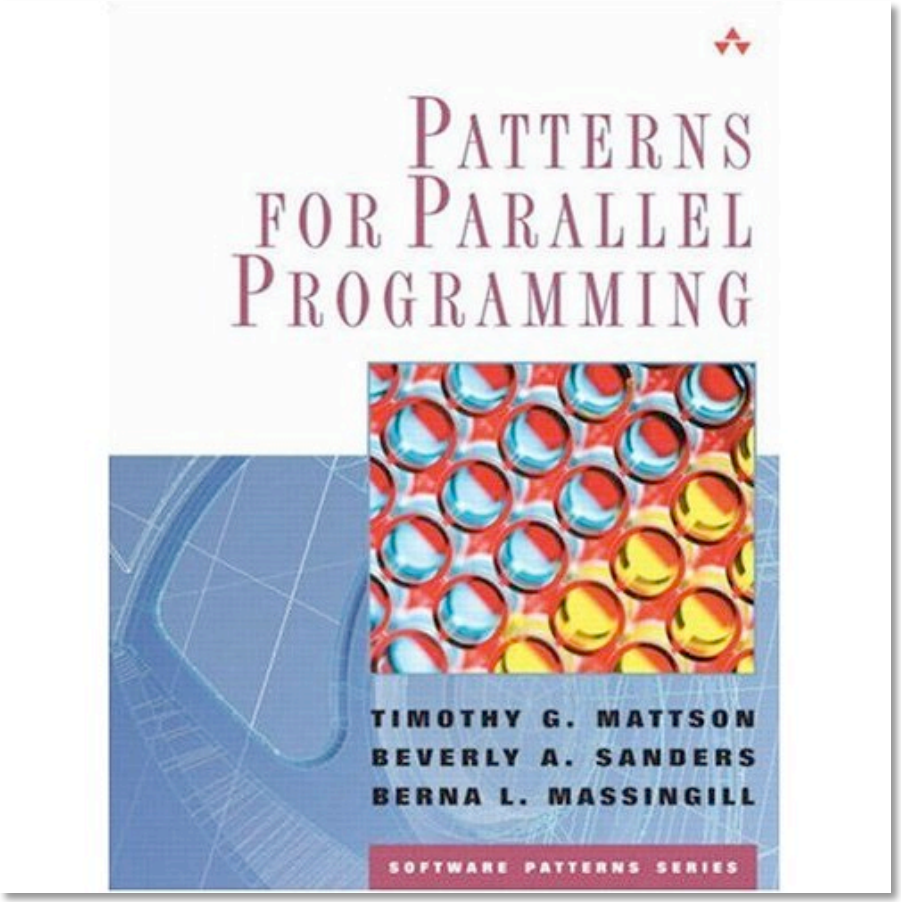
- Amdahl's Law approximately suggests:
  - » Suppose a car is traveling between two cities 60 miles apart (fixed problem size), and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average (speed-up) before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph.
- Gustafson's Law approximately states:
  - » Suppose a car has already been traveling for some time at less than 90 mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph (speed-up), no matter how long or how slowly it has already traveled. For example, if the car spent one hour at 30 mph, it could achieve this by driving at 120 mph for two additional hours, or at 150 mph for an hour, and so on (fixed time concept).

Source wikipedia: http://en.wikipedia.org/wiki/Gustafson%27s_law

Parallelization: how-to

Patterns for Parallel Programming

- In order to create complex software it is necessary to compose programming patters
- Examples:
  - » Pipes and filters
  - » Layered systems
  - » Agents and Repository
  - » Event-Based Systems
  - » Puppeteer
  - » Map/Reduce
- No time to describe them here but you can look at the book…



Parallelization in the code languages

- Automatic parallelization of a sequential program by a compiler is the holy grail of parallel computing
  - » automatic parallelization has had only limited success so far…
- Parallelization must be explicitly declared in a program (or at the best partially implicit, in which a programmer gives the compiler directives for parallelization)
  - » Some languages define parallelization as own instructions
    - High Performance Fortran
    - Chapel (by Cray)
    - X10 (by IBM)
    - C++1x (the new C++ standard)
  - » In most cases parallelization relays on external libraries
    - Native: pthreads/Windows threads
    - OpenMP (www.openmp.org)
    - Intel Threading Building Blocks (TBB)
    - OpenCL (www.khronos.org/opencl)
    - CUDA (by Nvidia, for GPU programming)

Parallelization: examples

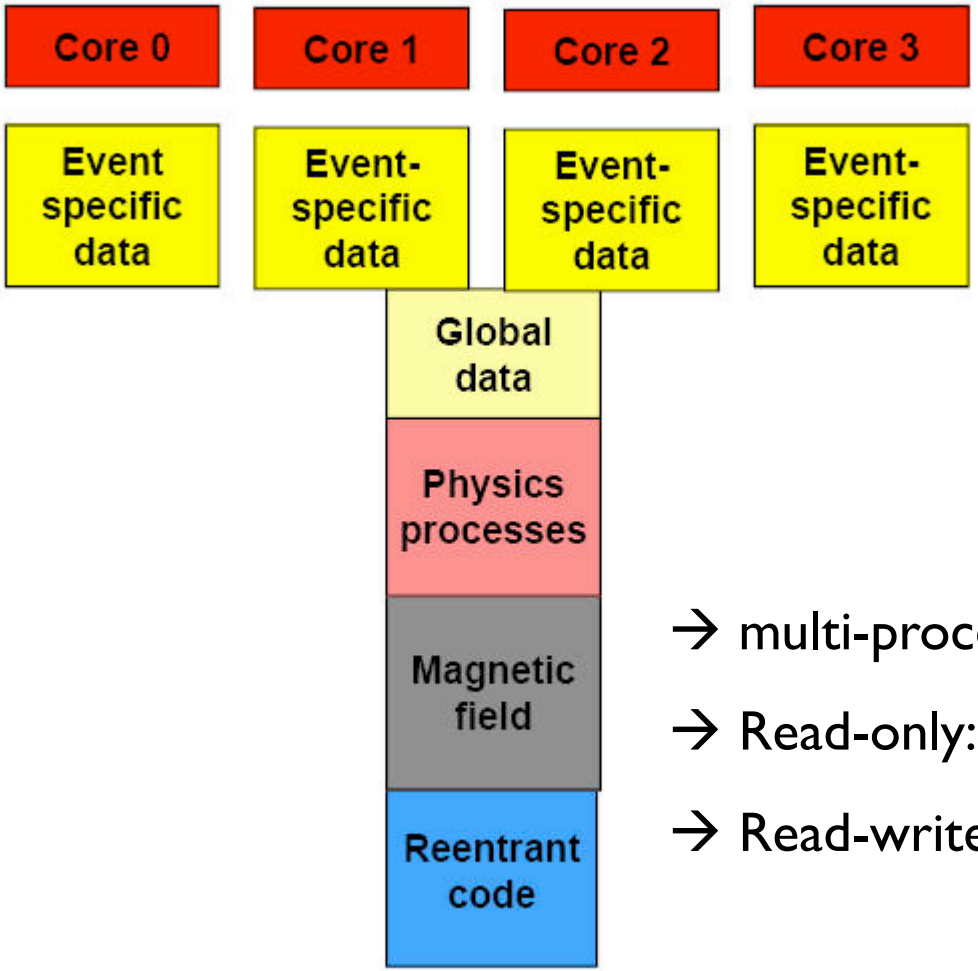
Parallelization in High Energy Physics

- Event-level parallelism mostly used
  - » Compute one event after the other in a single process
  - » Advantage: large jobs can be split into N efficient processes, each responsible for process M events
    - Built-in scalability
  - » Disadvantage: memory must be made available to each process
    - With 2 – 4 GB per process, with a dual-socket server with Quad-core processors we need 16 –32 GB (or more)
    - Memory is expensive (power and cost!) and the capacity does not scale as the number of cores
- Algorithm parallelization
  - » Prototypes using posix-thread, OpenMP and parallel gcclib
  - » Effort to provide basic thread-safe/multi-thread library components
- See examples of applications in the backup slides

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

Conclusions

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
  - » Parallelism beyond event level
  - » Fine grain parallelism
  - » Parallel I/O
- But, Amdahl docet, algorithm concept have to change to take advantages on parallelism: think parallel, write parallel!

Credits & References

– Special thanks to Vincenzo Innocente (CERN/PH) and Sverre Jarp (CERN/Openlab)
  » I took most of the slides from their presentations at ESC09 school: http://web.infn.it/esc09/

– Many references:
  » Google and wikipedia
  » Books and links that I suggested during the presentations

– Further readings (just as a starting point):
  » "Intel Threading Building Blocks: Outfitting C++ for Multi-core Processors Parallelism", J. Reinders, O'Reilly, first edition, 2007
  » "Principles of Concurrent and Distributed Programming", M. Ben-Ari, 2nd edition, Addison Wesley, 2006
  » "The Software Optimization Cookbook", R.Gerber, A.J.C. Bik, K.B. Smith and X. Tian, Intel Press, 2nd edition, 2006

Q & A

CERN openlab

Backup slides

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, Intel Ct, OpenCL

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

Exploit Copy on Write (COW)

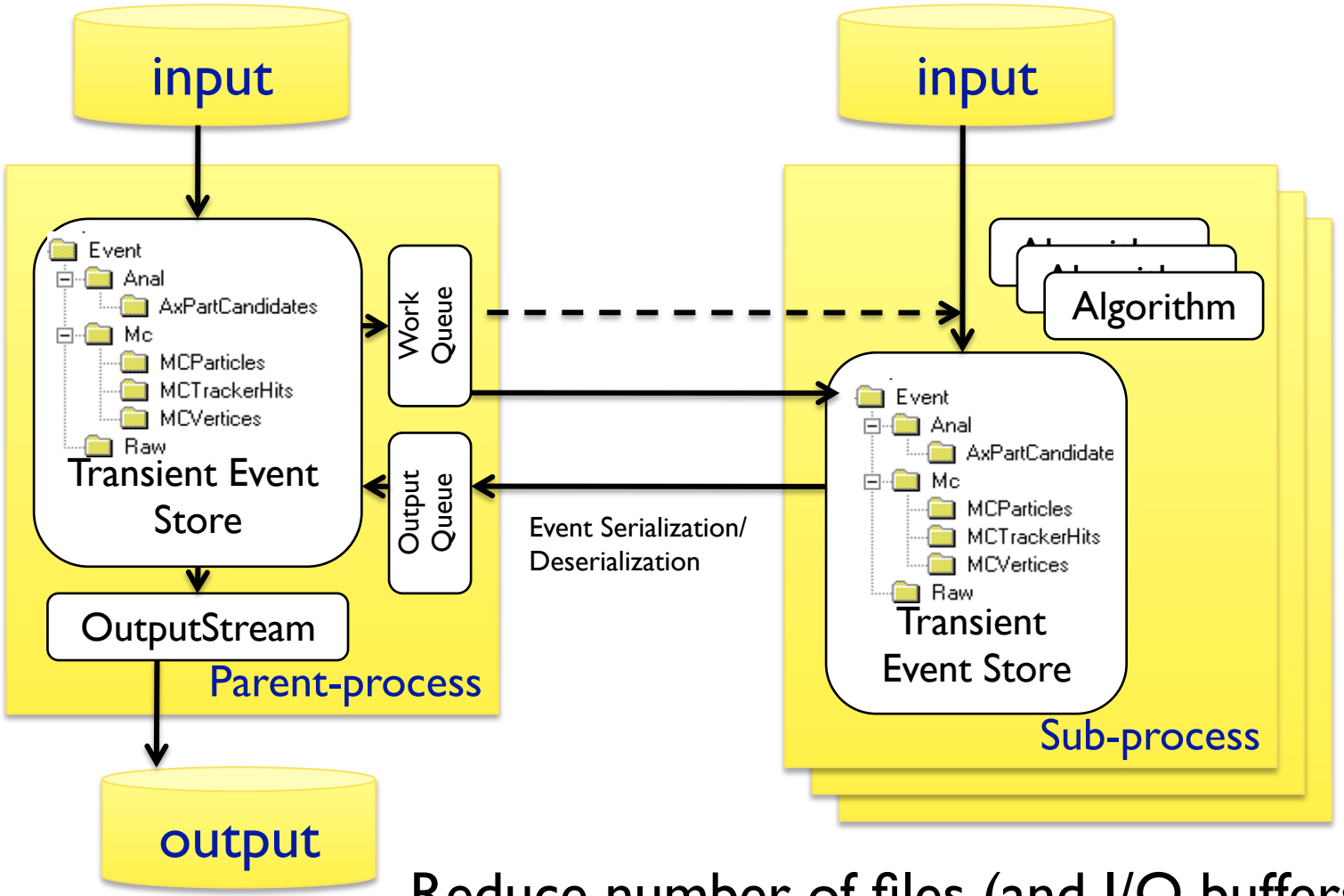
- Modern OS share read-only pages among processes dynamically
  - » A memory page is copied and made private to a process only when modified
- Prototype in Atlas and LHCb
  - » Encouraging results as memory sharing is concerned (50% shared)
  - » Concerns about I/O (need to merge output from multiple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

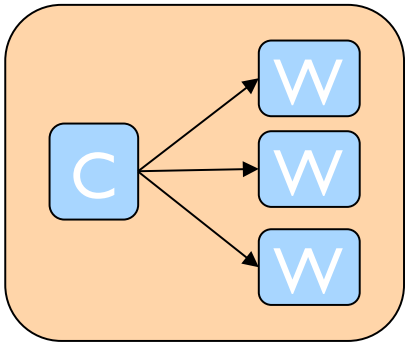
Handling Event Input/Output



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

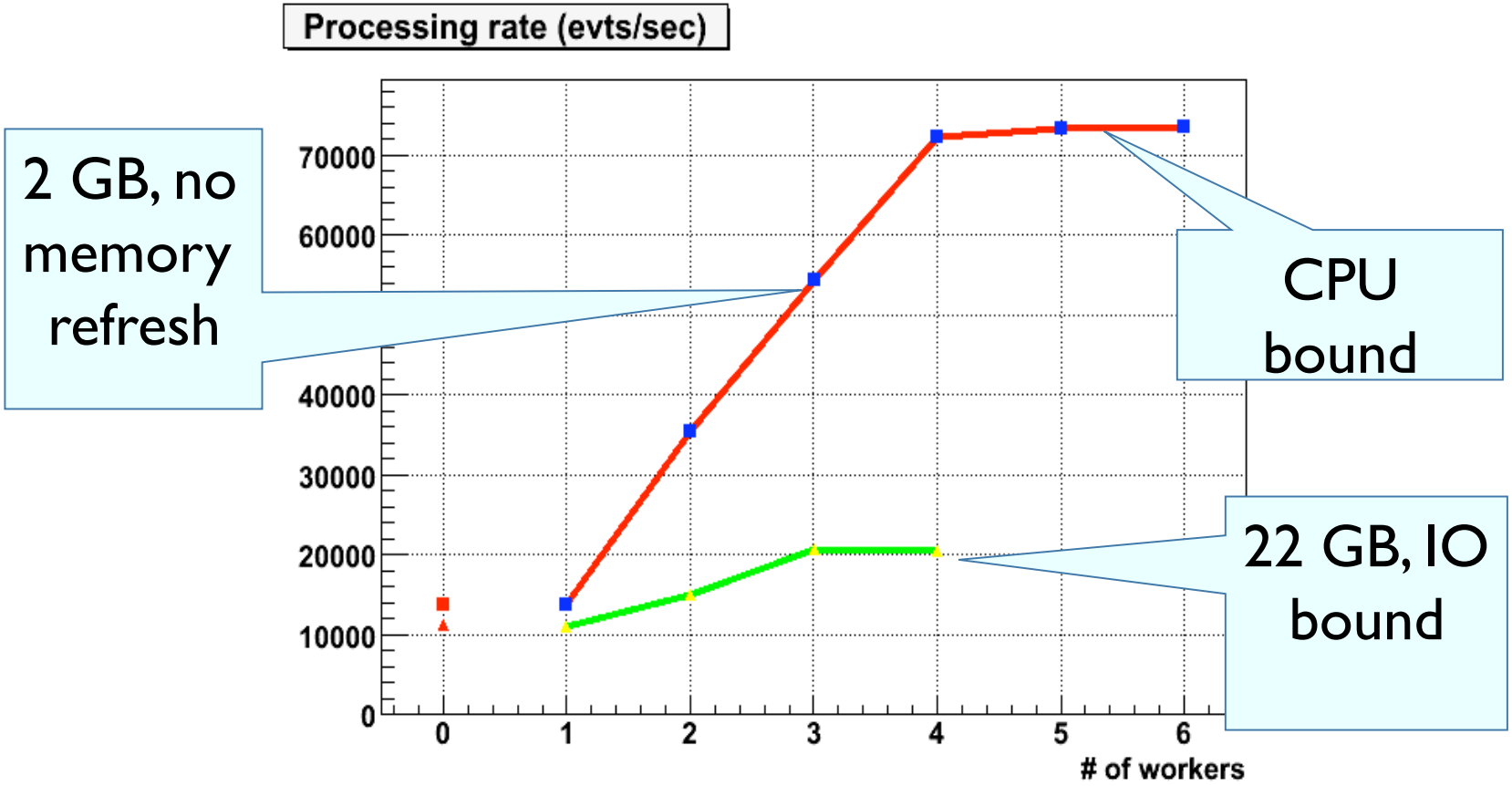
PROOF Lite

- 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 (4 core box)

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



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

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

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

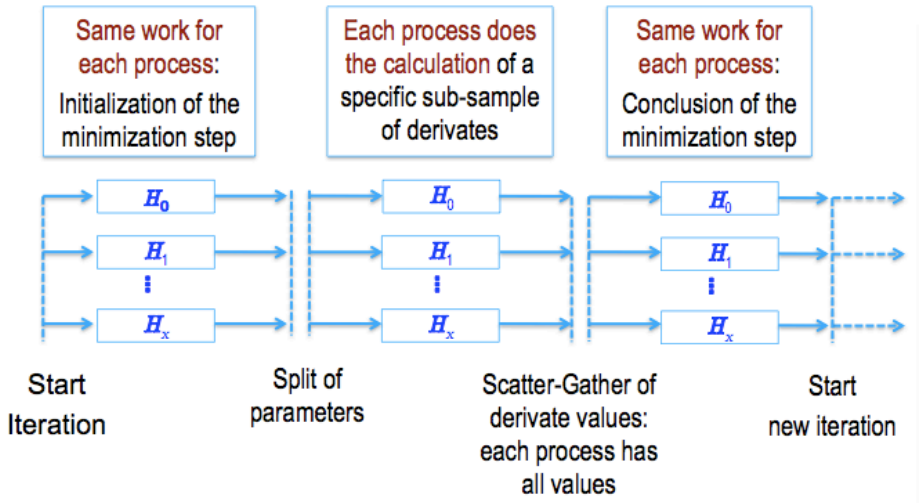
$$NLL = \ln\left(\sum_{j=1}^{s} n_j\right) - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j^i\right)$$

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

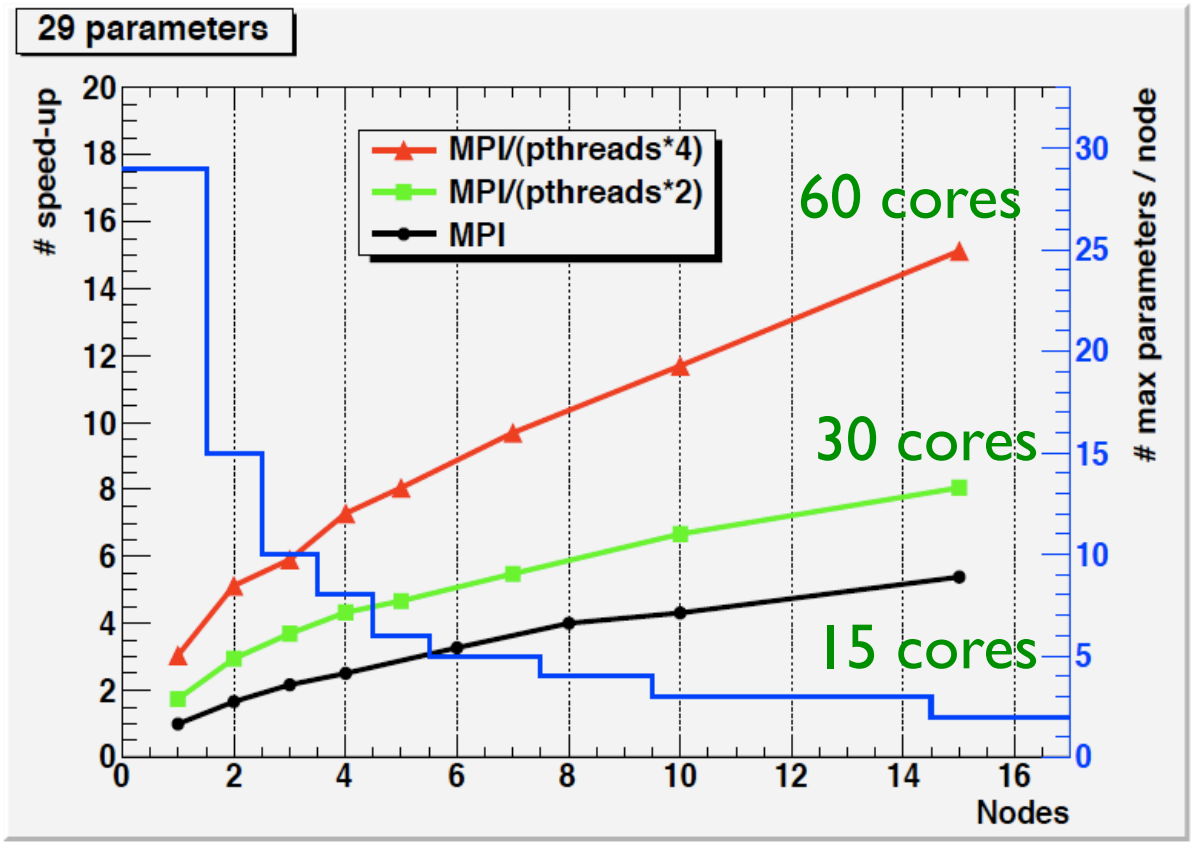
– Two strategies for the parallelization of the gradient and NLL calculation:

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

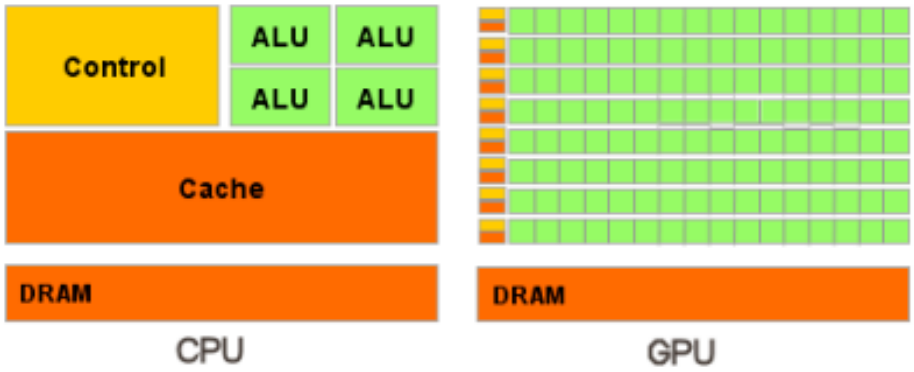
– Waiting time for fit to converge down from several days to a night (Babar examples)
  » iteration on results back to a human timeframe!



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

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. Some example in HEP (see M. Al-Turany's talk at CHEP09 on GPU for event reconstruction at Panda experiment)
  - » 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 and Intel Larrabee architecture (standard x86)