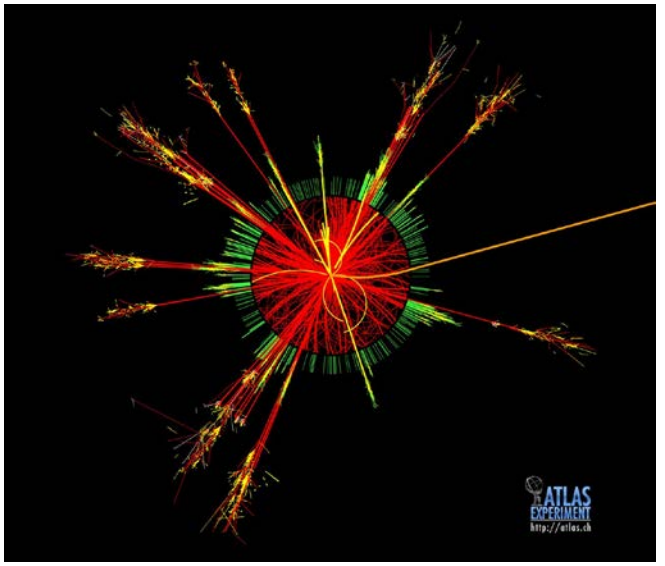
CERN openlab / Intel Workshop:
Parallelism, Compilers and Performance
25, 26, 27 March 2013

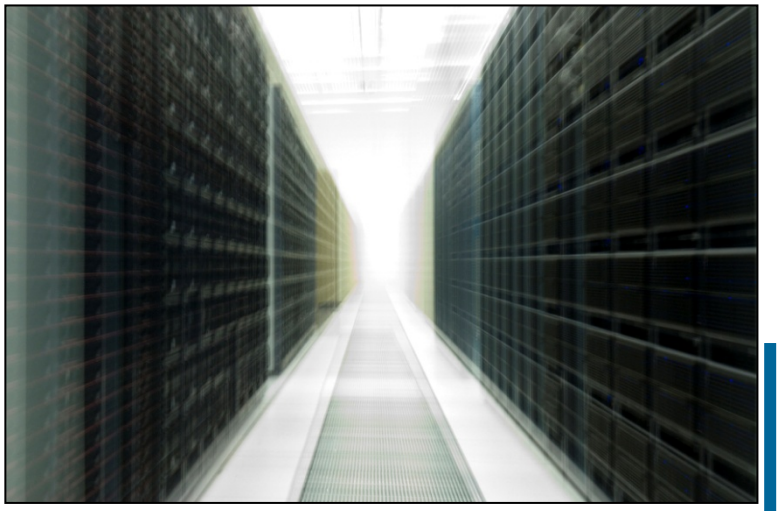
Scalability in Software and Hardware

"The 7 dimensions of performance"



Sverre Jarp
CERN
openlab
CTO

IT Dept., CERN



Contents

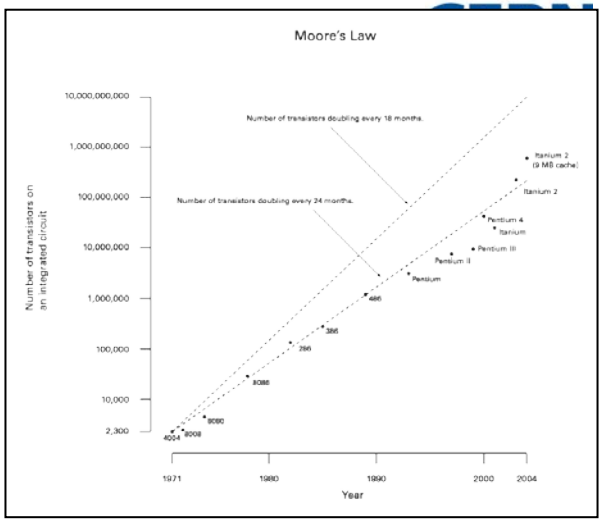
- **Why worry about performance?**
- **Review of fundamental architectural principles**
- **Addressing performance "dimensions"**
- **Scaling "inside a core" (first 3 dimensions)**
  - Architectural overview
  - Causes of execution delays
  - Important performance metrics
- **Scaling "across cores"**
  - Next set of dimensions
- **Software:**
  - Programming paradigms
  - Achieving better memory footprints
  - Achieving more parallelism; Concurrency
  - C++ parallelization support
  - Example of parallelization: Track fitting and others
- **Conclusions**

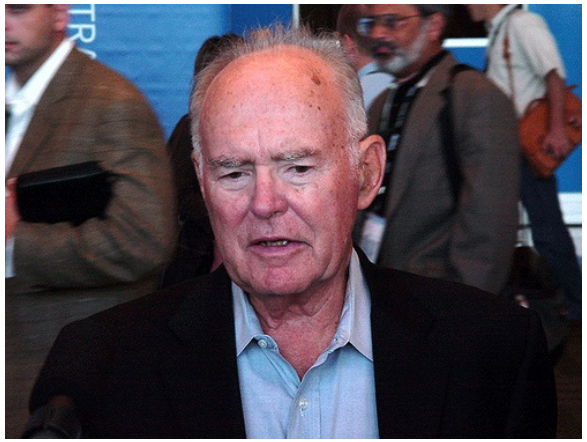
Why worry about performance?

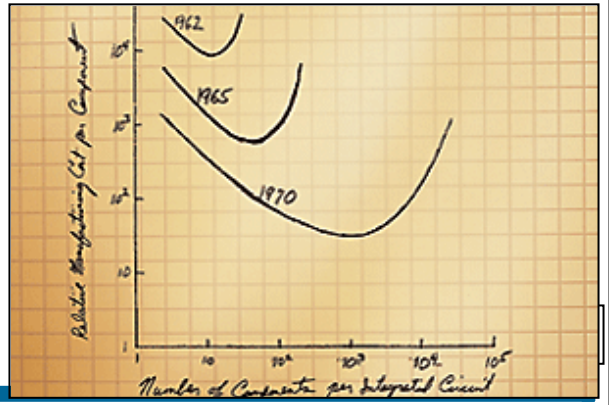
- **My arguments:**
  - The “easy ride” disappeared: The frequency scaling we enjoyed in the past does not exist any longer.
    It stopped 10 years ago!
    - ..and, as a “by-product”, the CPU architecture is becoming (much) more complicated
  - Performance per watt: There are important thermal issues associated with large scale computing
    - Even when 1W processors exist!
  - Performance per €: There are important cost issues associated with large scale computing
    - Even when using “commodity equipment”

Moore's law

- **We continue to double the number of transistors every other year**
- **The consequences:**
  - CPUs
    - Single core → Multicore → Manycore
    - Hardware vector support
    - Hardware threading
  - GPUs
    - Huge number of floating-point units
- **Today, we commonly acquire chips with 1'000'000'000 transistors!**
  - Intel/AMD server chips and high-end GPU devices:
    - Much more!







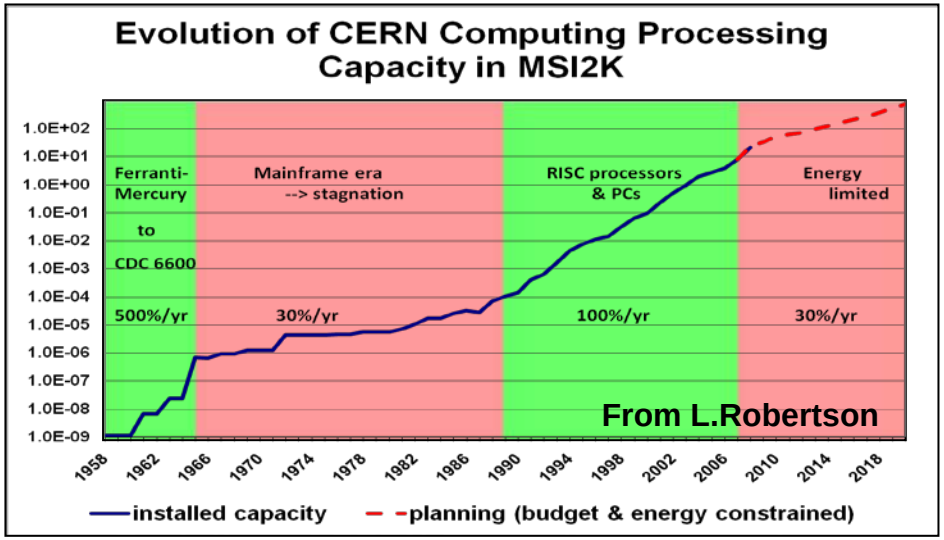
Real consequence of Moore's law

- **We are being "snowed under" by transistors:**
  - More (and more complex) execution units
    - Hundreds of new instructions
  - Longer hardware vectors
  - More and more cores
  - More hardware threading
- **In order to profit we need to "think parallel"**
  - Data parallelism
  - Task parallelism

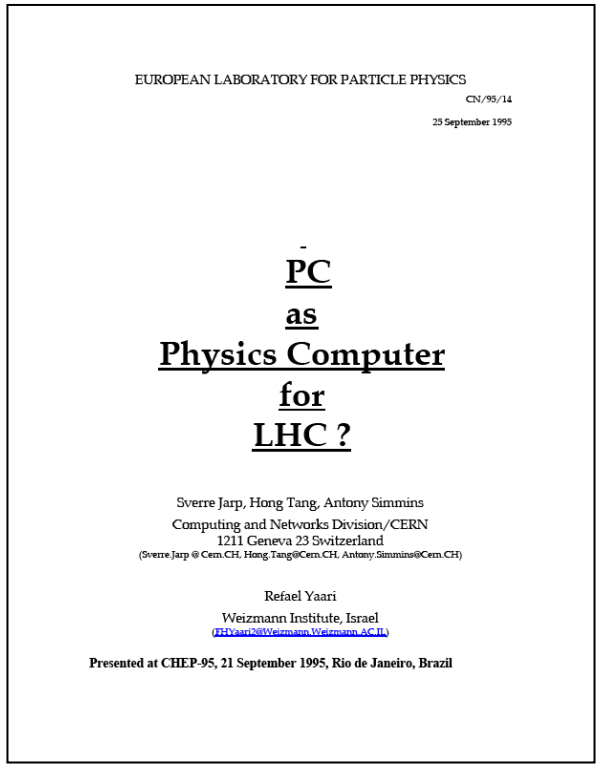
"Data Oriented Design"

Evolution of CERN's computing capacity

- **During the LEP era (1989 – 2000):**
  - Doubling of computing capacity every year
  - Initiated with the move from mainframes to RISC systems, and then to PCs



- **The PC has been with us for more than 15 years!**
  - At CHEP-95 I made the first recommendation to move to PCs
    - After a set of encouraging benchmark results



EUROPEAN LABORATORY FOR PARTICLE PHYSICS

CN/95/14

25 September 1995

PC
as
Physics Computer
for
LHC ?

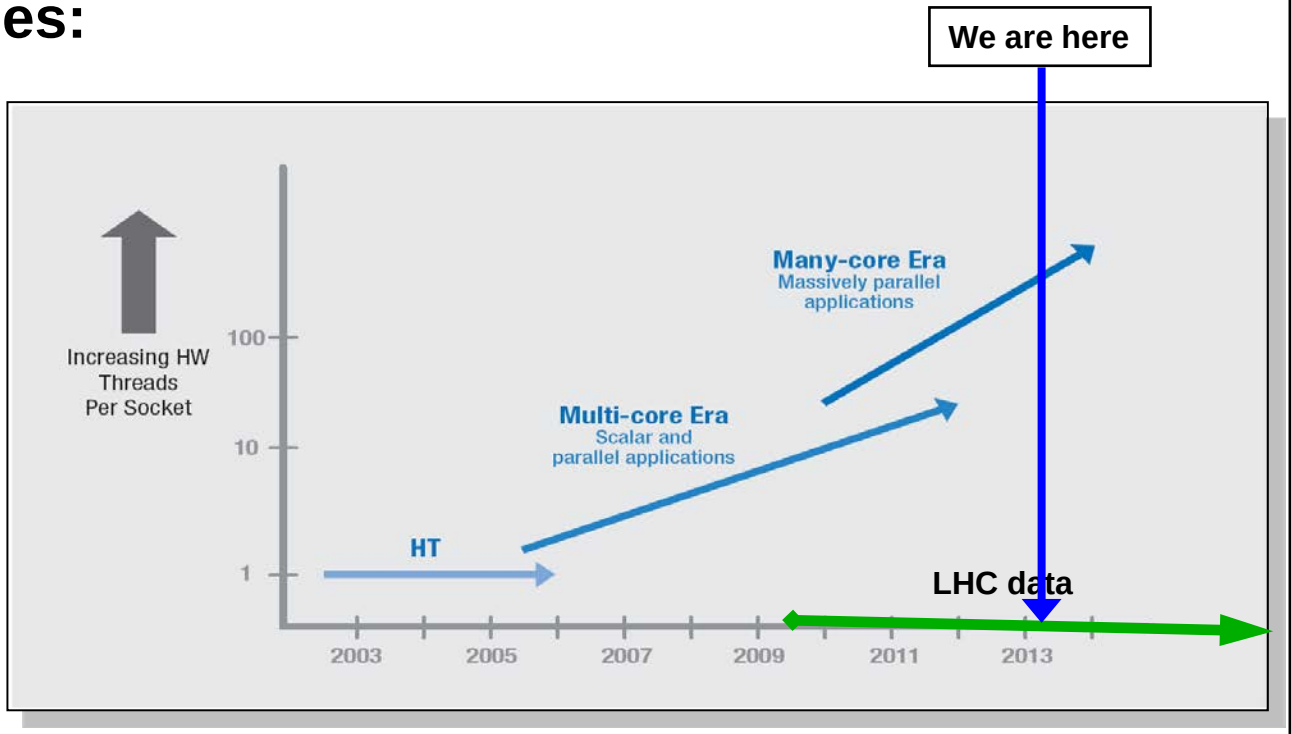
Sverre Jarp, Hong Tang, Antony Simmins
Computing and Networks Division/CERN
1211 Geneva 23 Switzerland
(Sverre.Jarp @ Cern.CH, Hong.Tang@Cern.CH, Antony.Simmins@Cern.CH)

Refael Yaari
Weizmann Institute, Israel
(FHYaari2@Weizmann.Weizmann.AC.IL)

Presented at CHEP-95, 21 September 1995, Rio de Janeiro, Brazil

“Intel platform 2015” (and beyond)

- **Today’s silicon processes:**
  - 32, 28, 22 nm
- **Being introduced:**
  - 14 nm (2013/14)
- **In research:**
  - 10 nm (2015/16)
  - 7 nm (2017/18)
  - 5 nm (2019/20)
    - Source: Intel

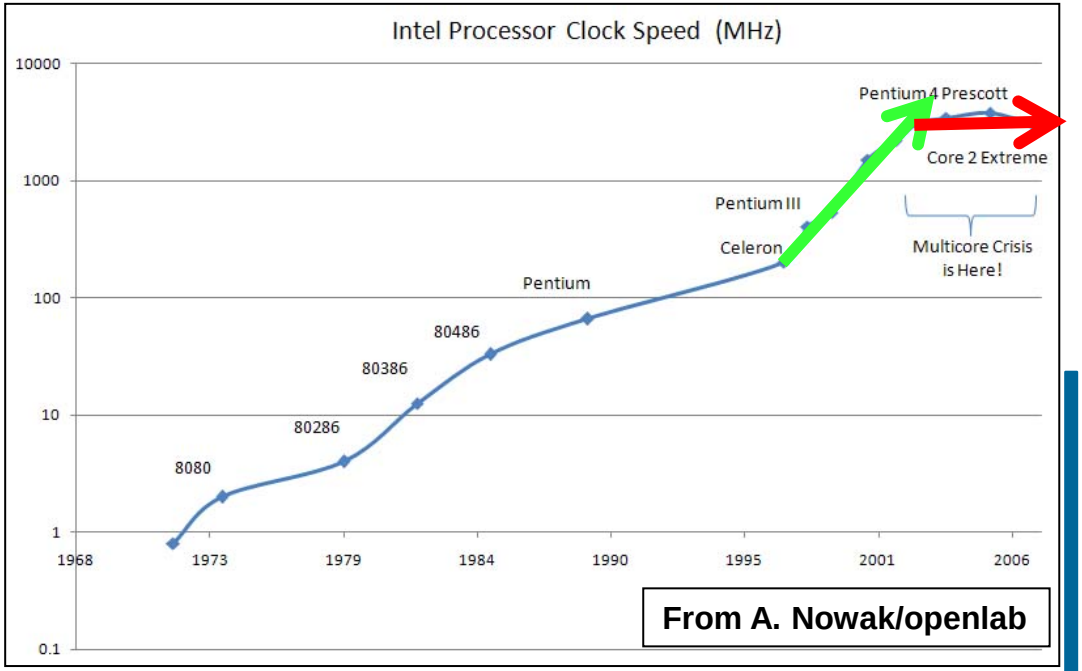


S. Borkar et al. (Intel), "Platform 2015: Intel Platform Evolution for the Next Decade", 2005.

- **Each generation will push the core count:**
  - We are inside the many-core era (whether we like it or not) !

Frequency scaling

- **The 7 “fat” years of frequency scaling in HEP**
  - The Pentium Pro in 1996: 150 MHz
  - The Pentium 4 in 2003: 3.8 GHz (~25x)
- **But, this was 10 years ago!**
- **Since then**
  - Core 2 systems:
    - ~3 GHz
    - Multi-core
- **Recent CERN purchase:**
  - Intel Xeon E5-2630L
    - “only” 2.00 GHz



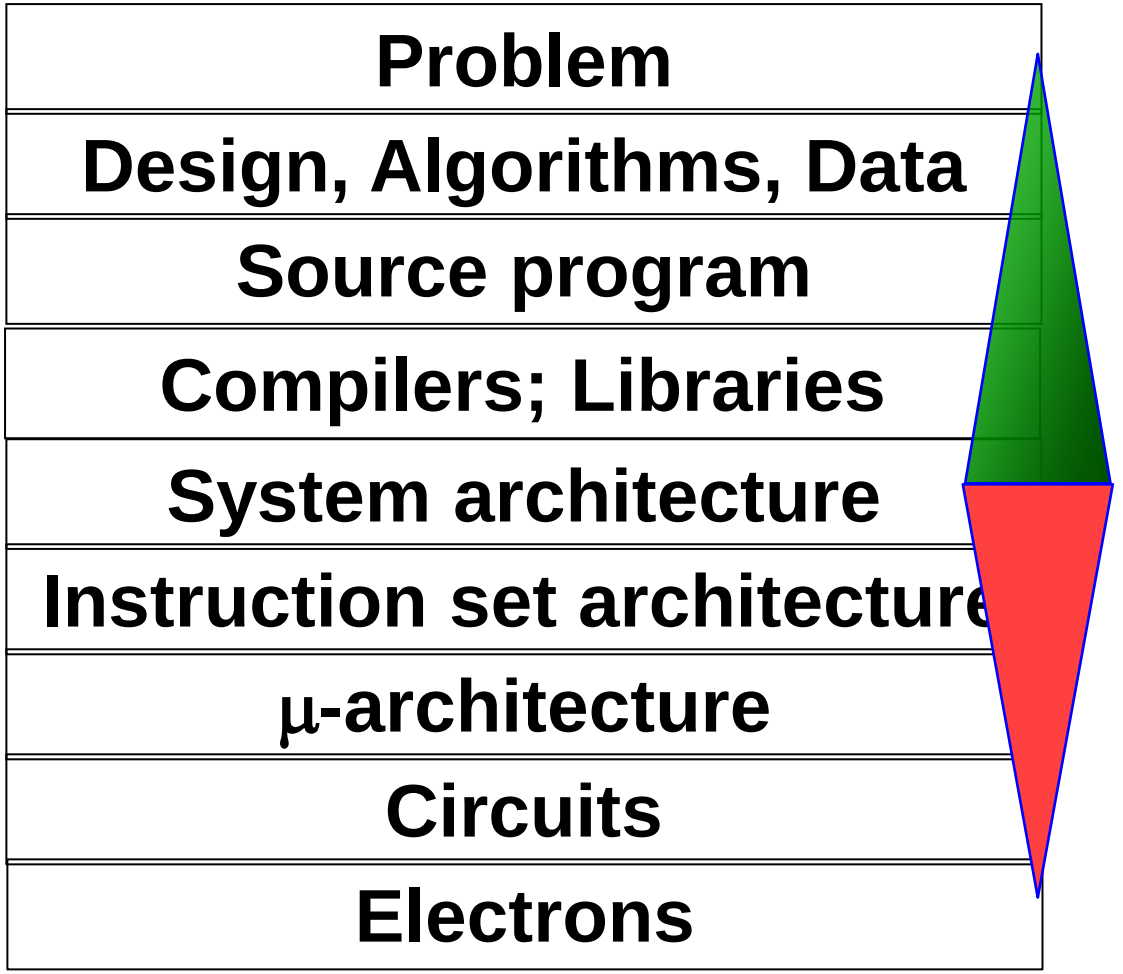
The holy grail: Forward scalability

- **Not only should a program be written in such a way that it extracts maximum performance from today's hardware**
- **On future processors, performance should scale automatically**
  - In the worst case, one would have to recompile or relink
- **Additional CPU/GPU hardware, be it cores/threads or vectors, would automatically be put to good use**
- **Scaling would be as expected:**
  - If the number of cores (or the vector size) doubled:
    - Scaling would be close to 2x, but certainly not just a few percent
- **We cannot afford to "rewrite" our software for every hardware change!**

Performance: A complicated story!

- **We start with a concrete, real-life problem to solve**
  - For instance, simulate the passage of elementary particles through matter
- **We write programs in high level languages**
  - C++, JAVA, Python, etc.
- **A compiler (or an interpreter) transforms the high-level code to machine-level code**
- **We link in external libraries**
- **A sophisticated processor with a complex architecture and even more complex micro-architecture executes the code**
- **In most cases, we have little clue as to the efficiency of this transformation process**

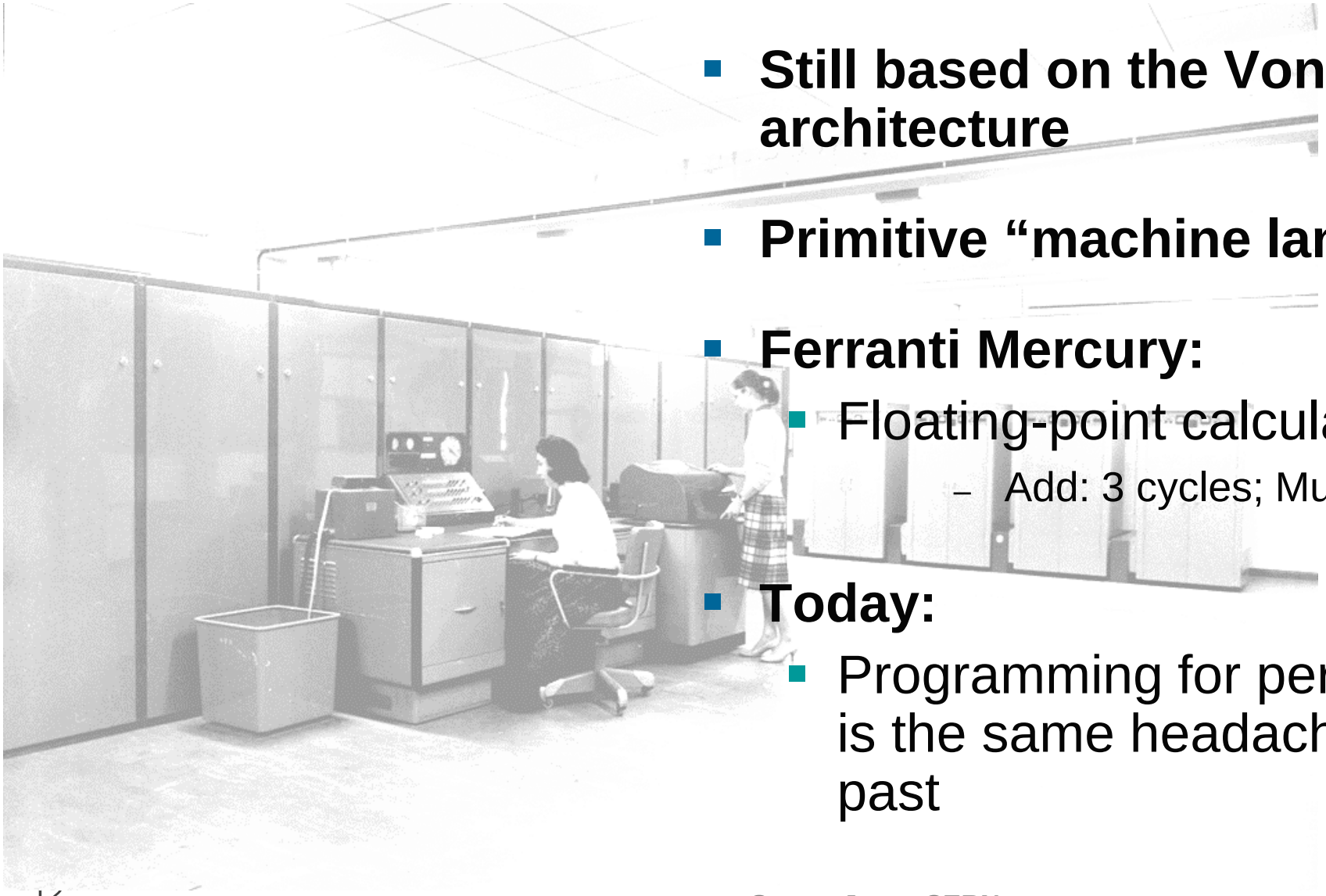
A Complicated Story (in 9 layers!)



- **We must avoid being fenced into a single layer!**

Adapted from Y.Patt, U-Austin

Archaic CPUs



- **As “stupid” as 50 years ago**
- **Still based on the Von Neumann architecture**
- **Primitive “machine language”**
- **Ferranti Mercury:**
  - Floating-point calculations
    - Add: 3 cycles; Multiply: 5 cycles
- **Today:**
  - Programming for performance is the same headache as in the past

And the language is ancient, too!

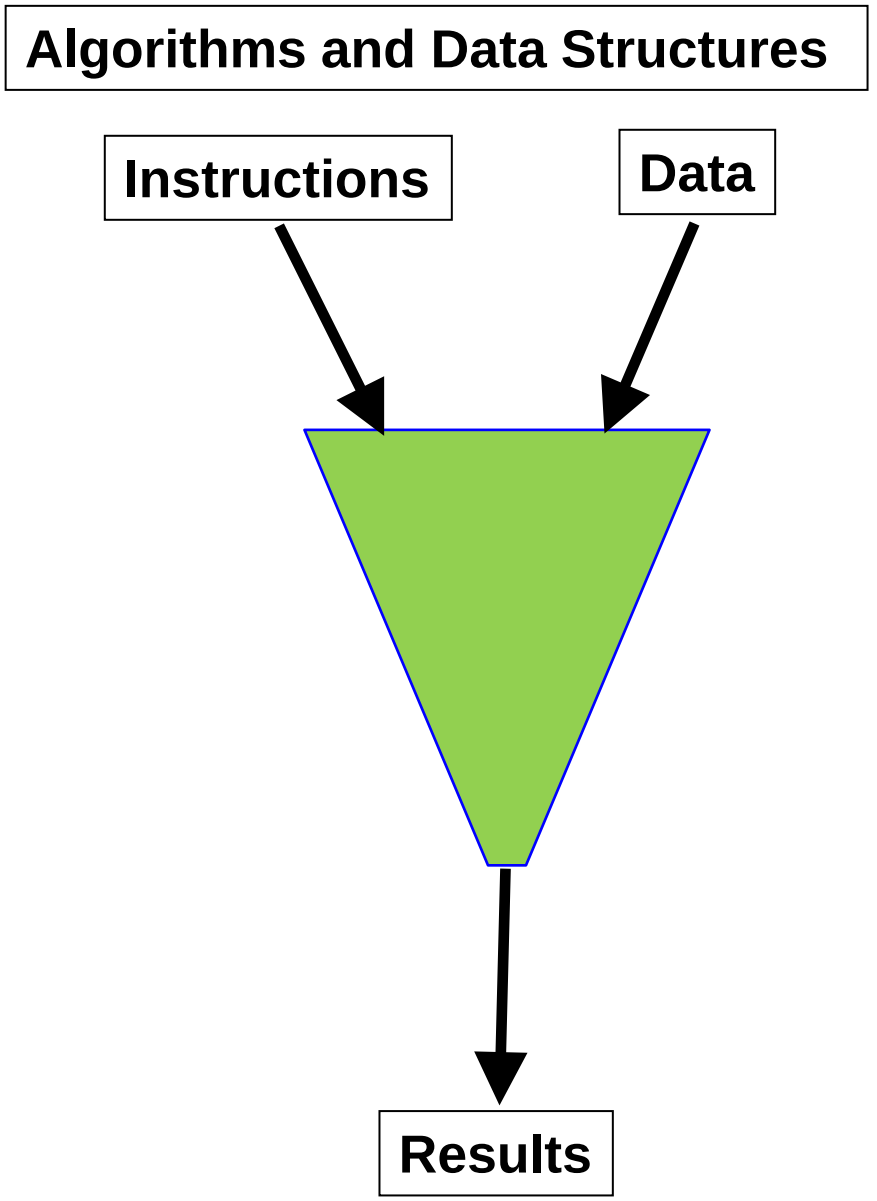
- **Assembly/machine code!**

```
..B1.31:                        # Preds ..B1.31 ..B1.30          # Infreq
        movsd     (%rsp), %xmm3                                    #94.17
        lea       (%rbx,%rbx,2), %rcx                              #94.36
        movsd     (%rsi,%rcx,8), %xmm2                             #94.40
        incl      %eax                                             #93.42
        movsd     8(%rsi,%rcx,8), %xmm0                            #94.40
        cmpl      %edx, %eax                                       #93.39
        mulsd     %xmm2, %xmm2                                     #94.40
        mulsd     %xmm0, %xmm0                                     #94.40
        movsd     16(%rsi,%rcx,8), %xmm1                           #94.40
        addsd     %xmm0, %xmm2                                     #94.40
        mulsd     %xmm1, %xmm1                                     #94.40
        movl      %eax, %ebx                                       #93.42
        addsd     %xmm1, %xmm2                                     #94.40
        sqrtsd    %xmm2, %xmm2                                     #94.40
        addsd     %xmm2, %xmm3                                     #94.17
        movsd     %xmm3, (%rsp)                                    #94.17
        jb        ..B1.31       # Prob 82%                       #93.39
```

But, let's start with the basics!

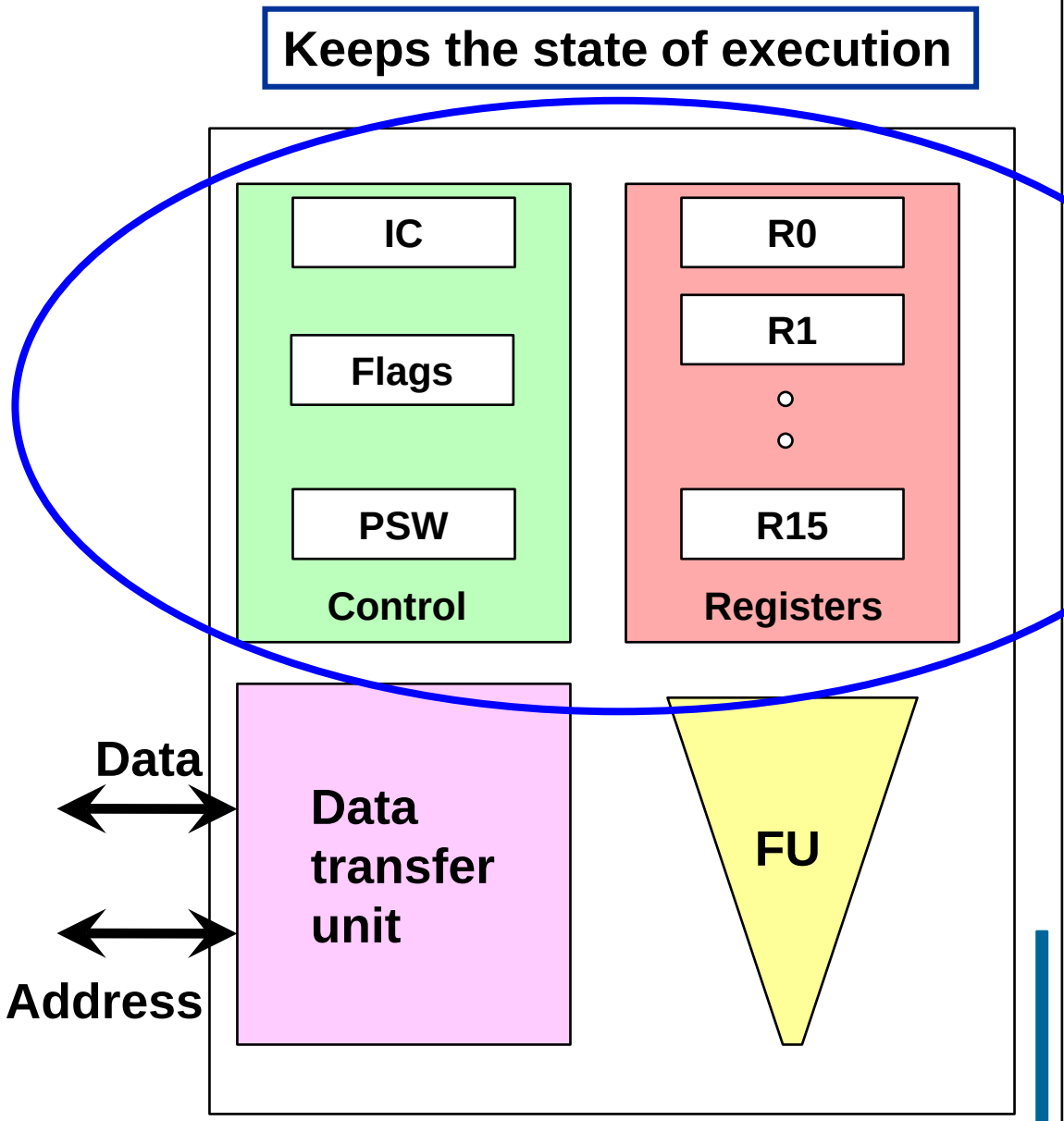
Von Neumann architecture

- **From Wikipedia:**
  - The von Neumann architecture is a computer design model that uses a processing unit and a single separate storage structure to hold both instructions and data.
- **It can be viewed as an entity into which one streams instructions and data in order to produce results**
- **The goal is to produce results as fast as possible**



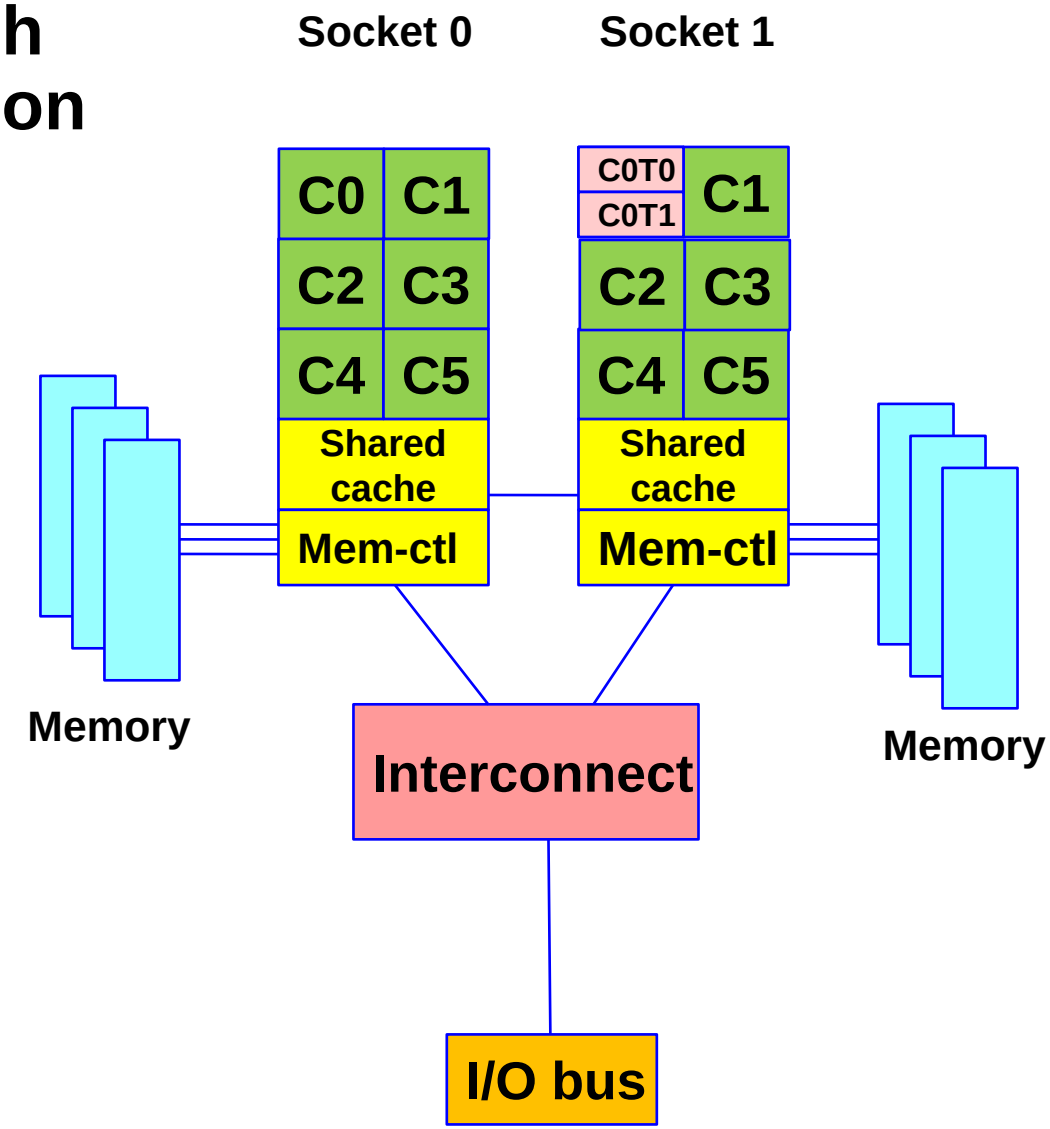
Simple processor layout

- **A simple processor with four key components:**
  - Control Logic
    - Instruction Counter
    - Program Status Word
  - Register File
  - Functional Unit
  - Data Transfer Unit
    - Data bus
    - Address bus



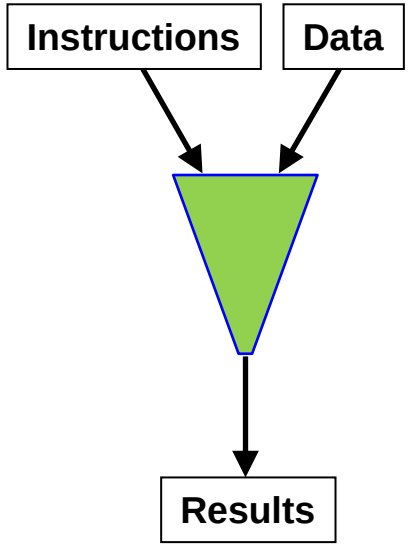
Simple server diagram

- **Multiple components which interact during the execution of a program:**
  - Processors/cores
    - w/private caches
  - Shared caches
    - Instructions (I-cache)
    - Data (D-cache)
  - Memory controllers
  - Memory (non-uniform)
  - I/O subsystem
    - Network attachment
    - Disk subsystem



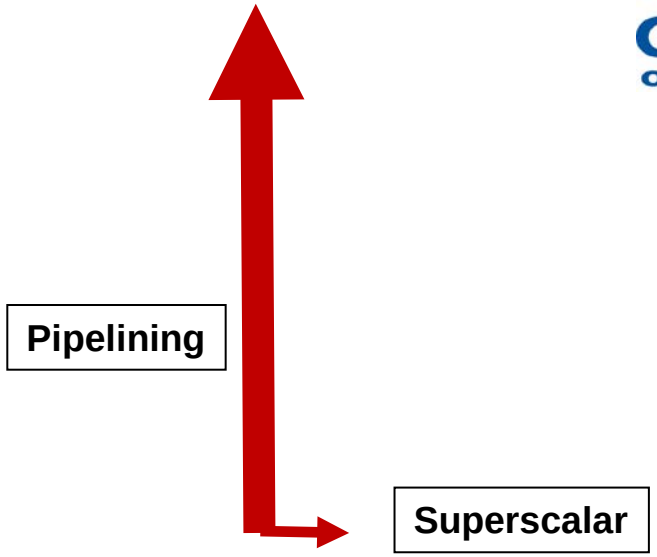
Initial premise

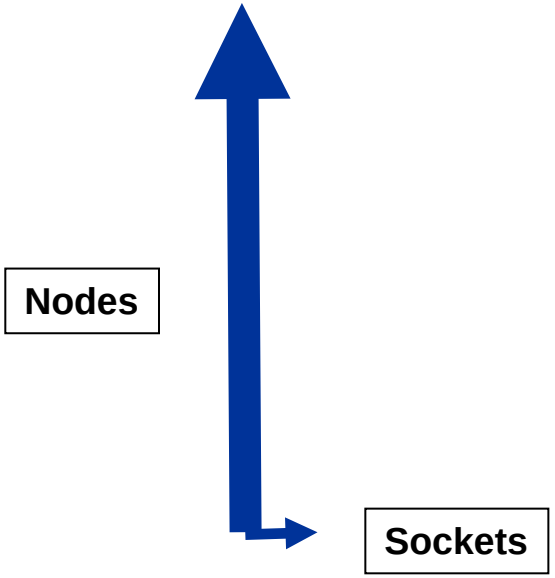
- **We want the process to complete in the shortest possible time**
  - Our compute job (a process) will require the execution of a given number of (machine-level) instructions
    - Dictated by the algorithms inside (and the compiler)
  - This time corresponds to a given number of machine cycles
- **Simple example:**
  - A program consists of 10^{10} instructions
  - We measure an execution time of 6 seconds on a processor running at 2.0 GHz
  - We can now compute a key value:
    - Cycles per Instruction (CPI)
    - Our result: $(6 * 2.0 * 10^9) / 10^{10} = 1.2$



In the days of the Pentium

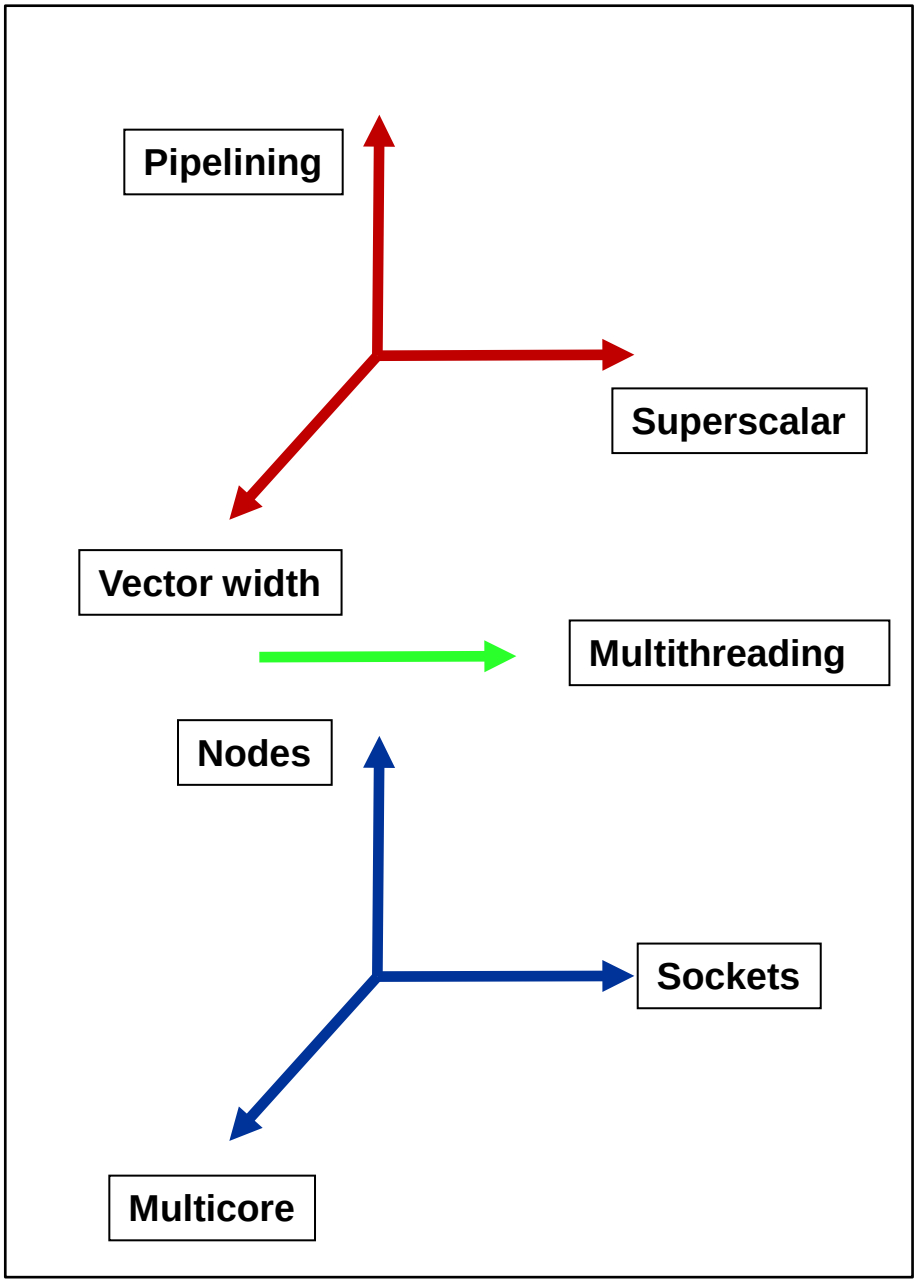
- **Life was really simple:**
  - Basically two dimensions
    - The frequency of the pipeline
    - The number of boxes
  - The semiconductor industry increased the frequency
  - We acquired the right number of (single-socket) boxes





Now: Seven dimensions of performance

- **First three dimensions:**
  - Hardware vectors/SIMD
  - Pipelining
  - Superscalar
- **Next dimension is a "pseudo" dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes



SIMD = Single Instruction Multiple Data

Seven multiplicative dimensions:

- **First three dimensions:**
  - Hardware vectors/SIMD
  - Pipelining
  - Superscalar

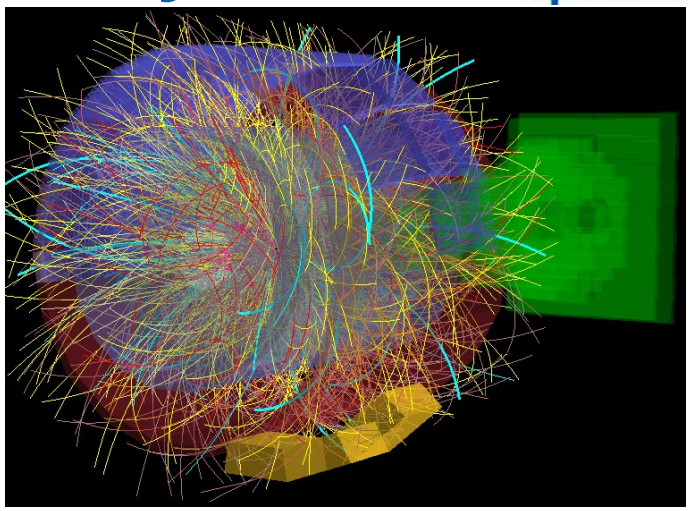
Data parallelism (Vectors/Matrices)

- **Next dimension is a "pseudo" dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes

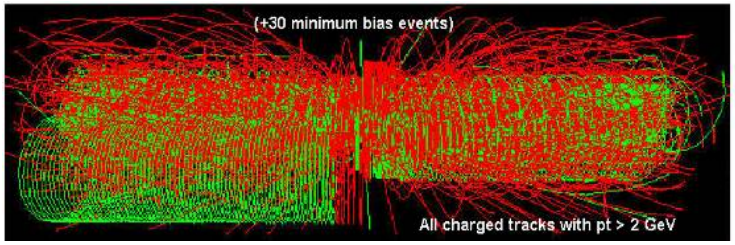
Task parallelism (Events/Tracks)

Task/process parallelism

Concurrency in High Energy Physics



- **We are “blessed” with lots of it:**
  - Entire events
  - Particles, hits, tracks and vertices
  - Physics processes
  - I/O streams (ROOT trees, branches)
  - Buffer handling (also data compaction, etc.)
  - Fitting variables
  - Partial sums, partial histograms
  - and many others …..



- **Usable for both data and task parallelism!**
- **But, fine-grained parallelism is not well exposed in today’s software frameworks**

Autoparallelization/Autovectorization

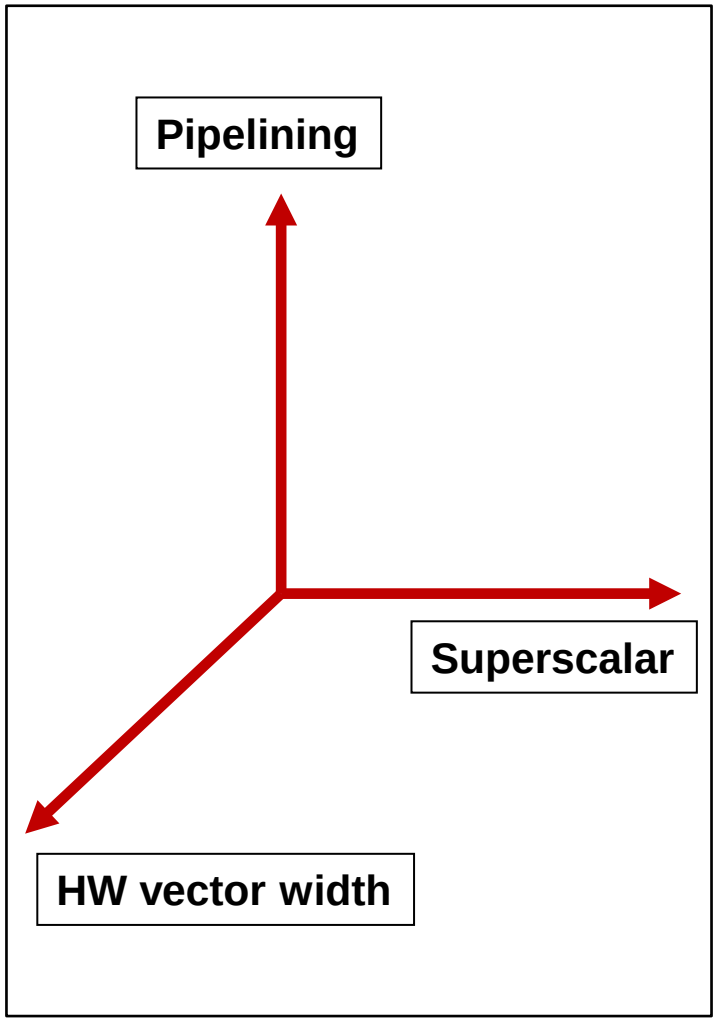
- **Would it not be wonderful if the compilers could do all the (vectorization/parallelisation) work automatically?**
- **GNU compiler (4.3.0 or later):**
  - Autovectorization: YES, but needs “-ftree-vectorize” or “-O3”
    - “-ftree-vectorizer-verbose=[0-7]” for reports
  - OpenMP support available
  - Autoparallelization support in preparation
- **Intel compiler (10.1 or later):**
  - Autovectorization: YES, included in “-O”
    - “-vec-reportN” for reports
  - Autoparallelization: YES with “-parallel”
    - “-par-reportN” for reports

Use “-guide” for both scenarios to get advice [“Guided Auto Parallelisation” or “GAP”]

Autovectorization is beginning to look serious in recent compiler versions!

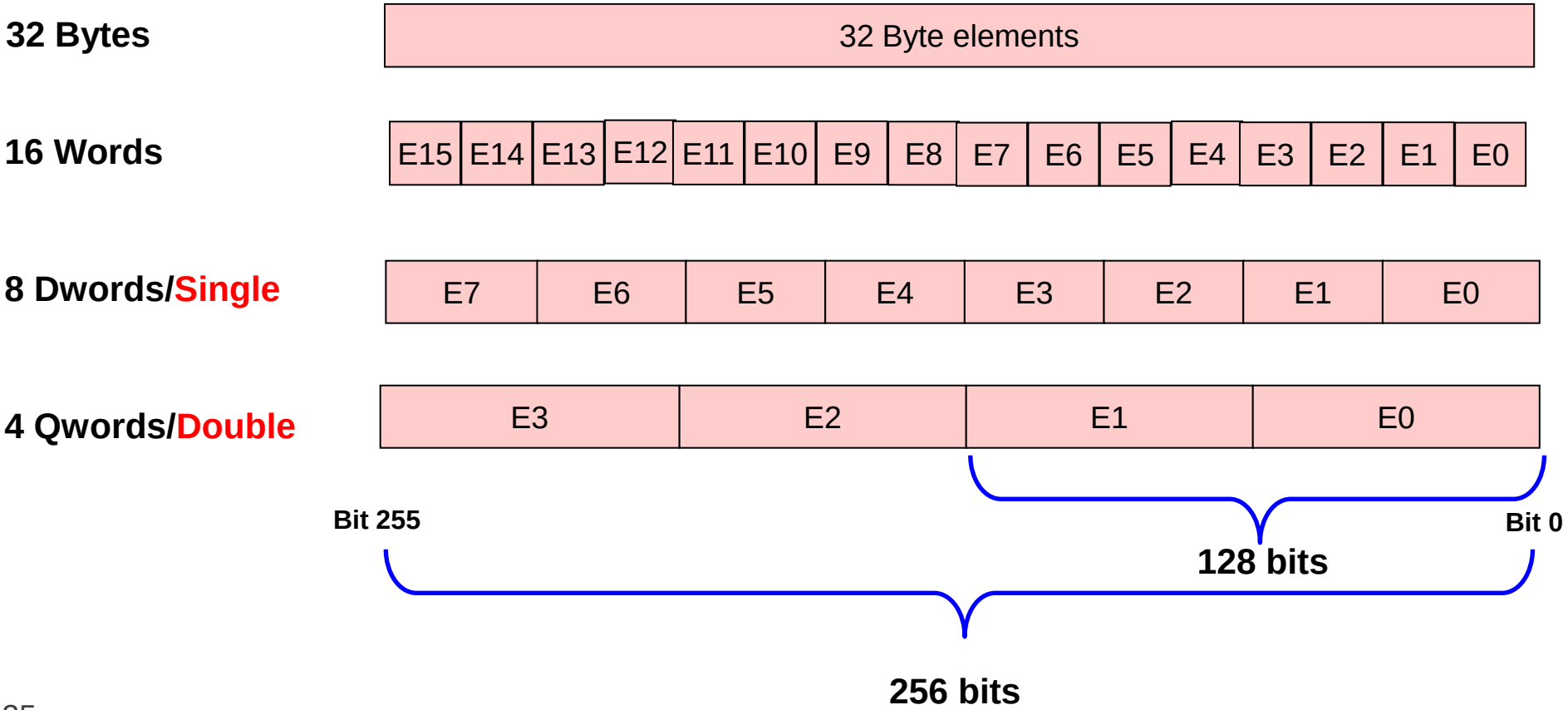
Part 1: Opportunities for scaling performance inside a core

- **Here are the first three dimensions**
- **The resources:**
  - HW vectors: Fill the computational width
  - Pipelining: Fill the stages
  - Superscalar: Fill the ports
- **Best approach: Data Oriented Design**
- **In HEP today, we probably extract less than 10% of peak execution capability!**



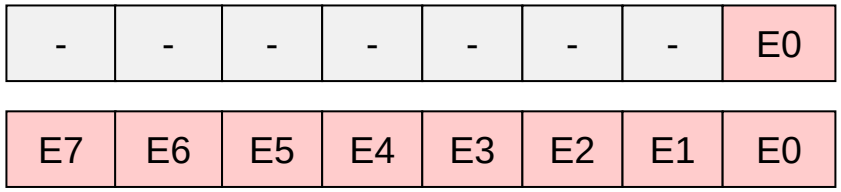
First topic: Vector registers

- **Until recently, Steaming SIMD Extensions (SSE):**
  - 16 “XMM” registers with 128 bits each (in 64-bit mode)
- **New (as of 2011): Advanced Vector eXtensions (AVX):**
  - 16 “YMM” registers with 256 bits each

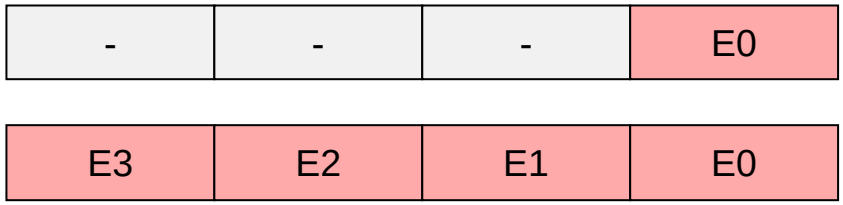


Four floating-point data flavours

- **Single precision**
  - Scalar single (SS)
  - Packed single (PS)



- **Double precision**
  - Scalar Double (SD)
  - Packed Double (PD)



- **Note:**
  - Scalar mode (with AVX) means using only:
    - 1/8 of the width (single precision)
    - 1/4 of the width (double precision)
  - Even longer vectors are coming!
    - Definitely 512 bits (and maybe even 1024 bits one day)

Scalable programming inside a core

- **Easiest way to fill the execution capabilities is to program software vectors**
- **But, which ones?**
  - Standard C arrays
    - Intel has added C Extended Array Notation (CEAN) to their 12.0 compiler
      - Also coming in gcc 4.9
  - STL vectors
  - TBB vectors (thread-safe)
  - Intrinsics
  - etc.

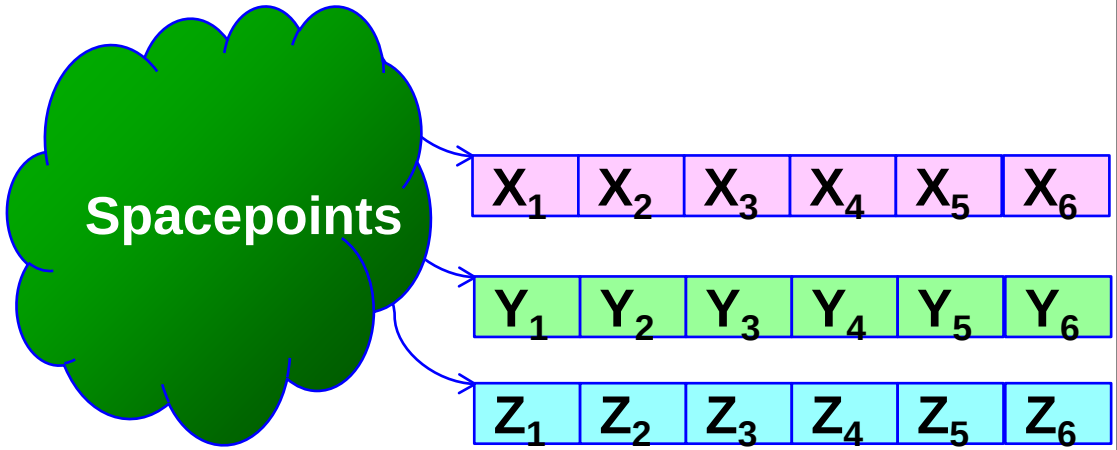
```
float  u[100], v[100];

for (int i = 0; i<50; ++i) u[i] = 0.0;

for (i = 0; i<50; ++i) u[i] = sin(v[i]);

for (int i = 0; i<50; ++i) u[i] = v[i*2+1];
```

CEAN example:

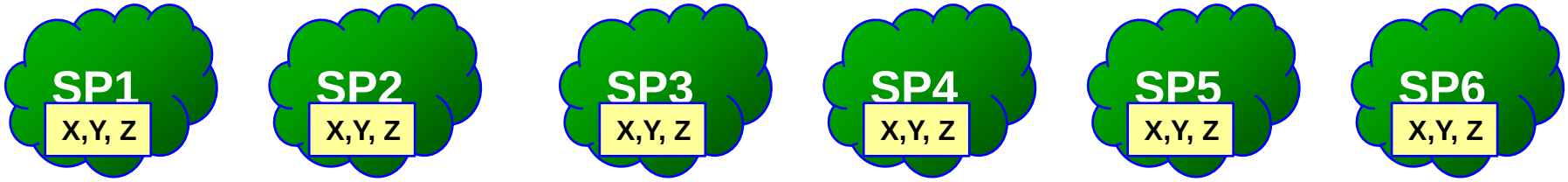
```
A[i:n] = 2.5 * B[j:n];
```

SoA versus AoS

- **In general, compilers and hardware prefer the former!**
- **Structure of Arrays (SoA):**

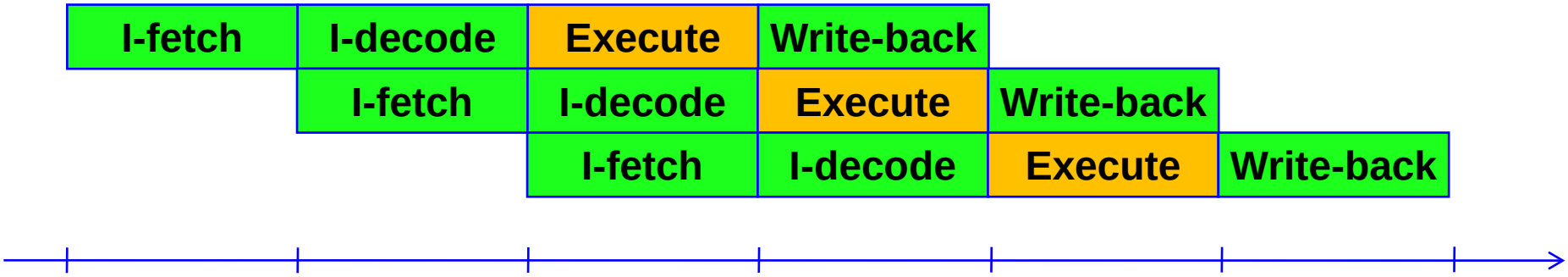


- **Array of Structures (AoS):**

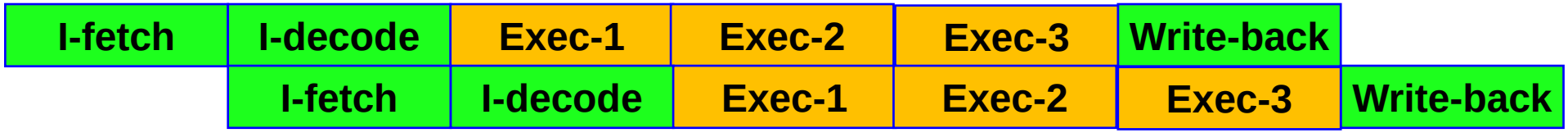


Second topic: Instruction pipelining

- **Instructions are broken up into stages.**
  - With a one-cycle execution latency (simplified):



  - With a three-cycle execution latency (FADD, etc.):



Real-life latencies

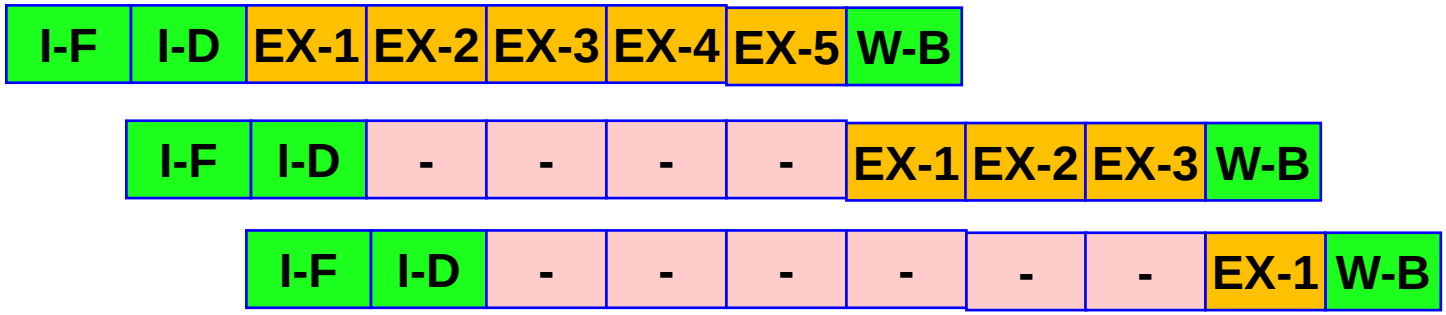
- **Most integer/logic instructions have a one-cycle execution latency:**
  - For example:
    - ADD, AND, SHL (shift left), ROR (rotate right)
  - Amongst the exceptions:
    - IMUL (integer multiply): 3
    - IDIV (integer divide): 13 – 23
- **Floating-point latencies are typically multi-cycle**
  - FADD (3), FMUL (5)
    - Same for both x87 and SIMD double-precision variants
  - Exception: FABS (absolute value): 1
  - Many-cycle: FDIV (20), FSQRT (27)
  - Other math functions: even more

Latencies in the Core micro-architecture (Intel Manual No. 248966-026 or later). AMD processor latencies are similar.

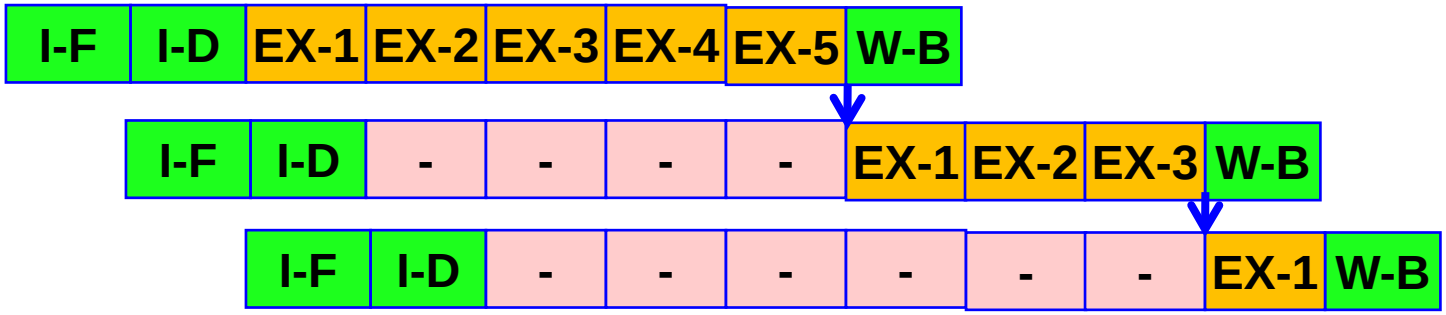
Latencies and serial code (1)

- **In serial programs, we typically pay the penalty of a multi-cycle latency during execution:**
  - In this example:
    - Statement 2 cannot be started before statement 1 has finished
    - Statement 3 cannot be started before statement 2 has finished

```
double a, b, c, d, e, f;

b = 2.0; c = 3.0; e = 4.0;

a = b * c;  // Statement 1

d = a + e;  // Statement 2

f = fabs(d);   // Statement 3
```



Latencies and serial code (2)



- **Observations:**
  - Even if the processor can fetch and decode a new instruction every cycle, it must wait for the previous result to be made available
    - Fortunately, the result takes a ‘bypass’, so that the write-back stage does not cause even further delays
  - The result: CPI is equal to 3
    - 9 execution cycles are needed for 3 instructions!
- **A good way to hide latency is to [get the compiler to] unroll (vector) loops !**

Mini-example of real-life scalar, serial code

- **Suffers long latencies:**

High level C++ code →

```
if (abs(point[0] - origin[0]) > xhalfsz) return FALSE;
```

Machine instructions →

```
movsd 16(%rsi), %xmm0
subsd 48(%rdi), %xmm0   // load & subtract
andpd _2il0floatpacket.1(%rip), %xmm0 // and with a mask
comisd 24(%rdi), %xmm0 // load and compare
jbe ..B5.3      # Prob 43%  // jump if FALSE
```

Same instructions laid out according to latencies on the Core 2 processor →

NB: Out-of-order scheduling not taken into account.

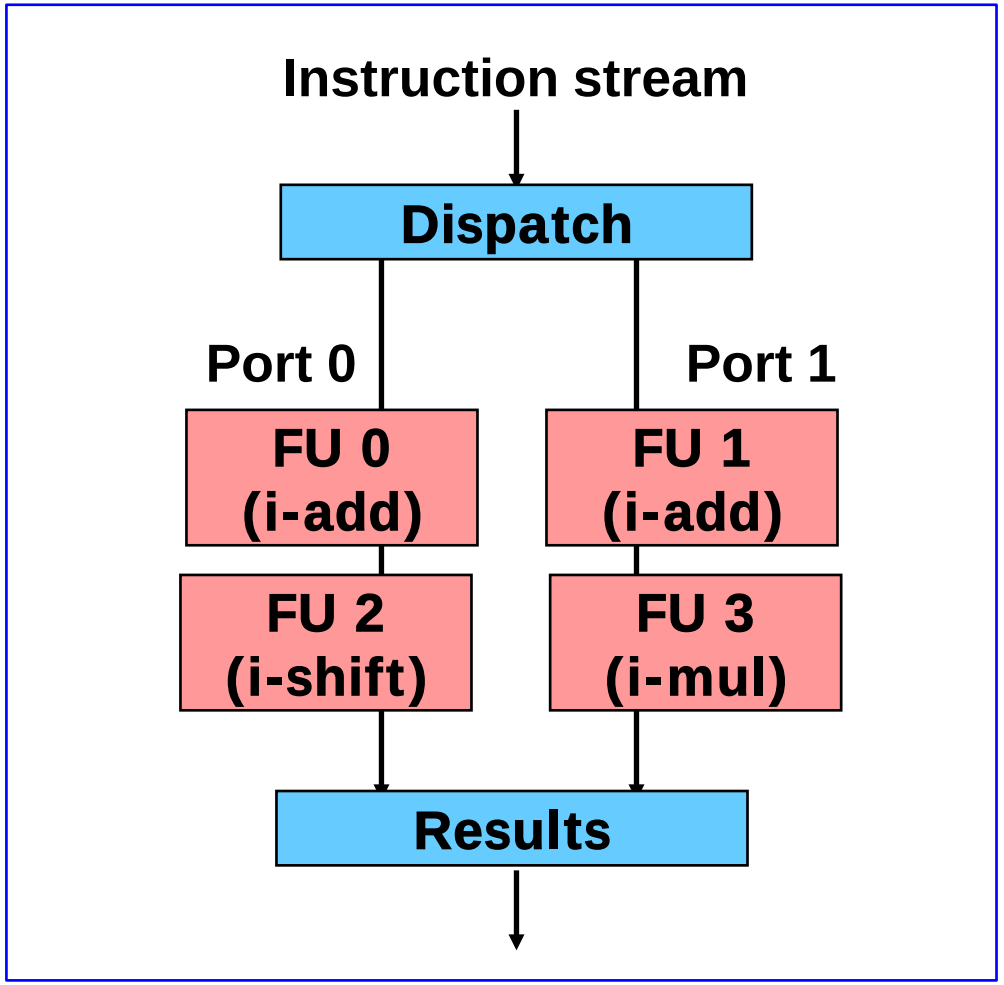
Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1			load point[0]			
2			load origin[0]			
3						
4						
5						
6		subsd	load float-packet			
7						
8			load xhalfsz			
9						
10	andpd					
11						
12	comisd					
13						jbe

Out-of-order (OOO) scheduling

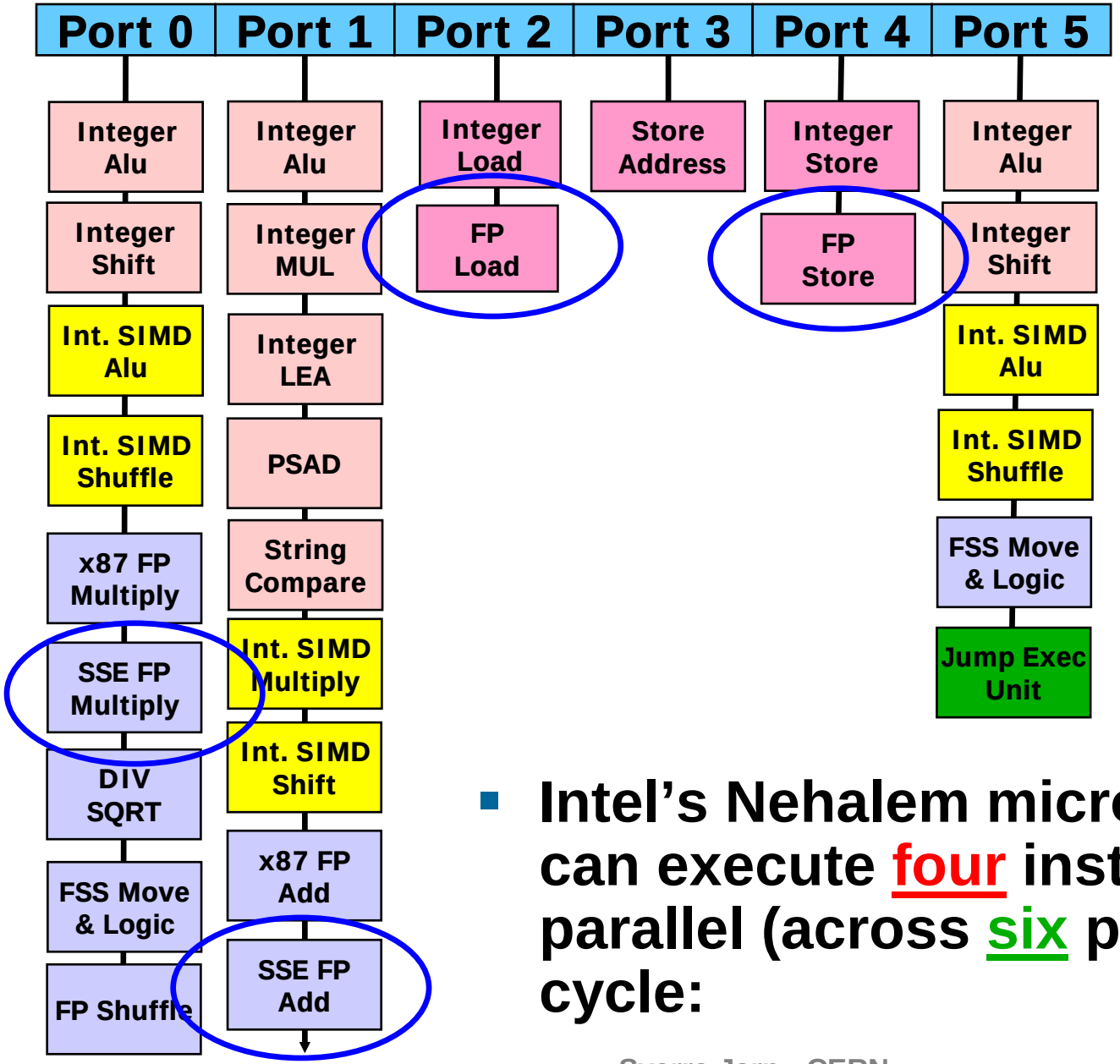
- **Modern x86 processors use OOO scheduling**
  - This means that they will speculatively execute instructions ahead of time (inside a “window” of ~150 instructions)
  - In certain cases the results of such executed instructions must be discarded
- **At the end, there is a difference between “executed instructions” and “retired instructions”**
  - One typical reason for this is mispredicted branches
- **Potential problem with OOO:**
  - A lot of extra energy is needed!

Third topic: Superscalar architecture

- **In this simplified design, instructions are decoded in sequence, but dispatched to two ports controlling Functional Units.**
  - The decoder and dispatcher must be able to handle two instructions per cycle
  - The FUs can have identical or different execution capabilities



Recent superscalar architecture



Alu = Arithmetic, Logical Unit
FSS = FP/SIMD/SSE2

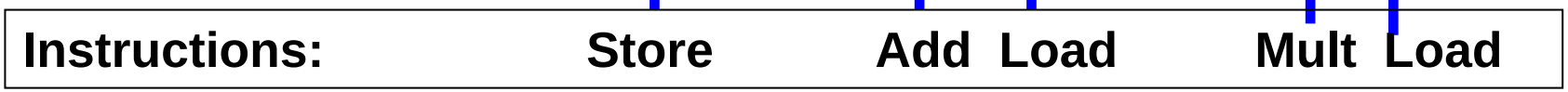
Issue ports in the Core micro-architecture (from Intel Manual No. 248966-026)

- **Intel's Nehalem microarchitecture can execute four instructions in parallel (across six ports) in each cycle:**

Matrix multiply example

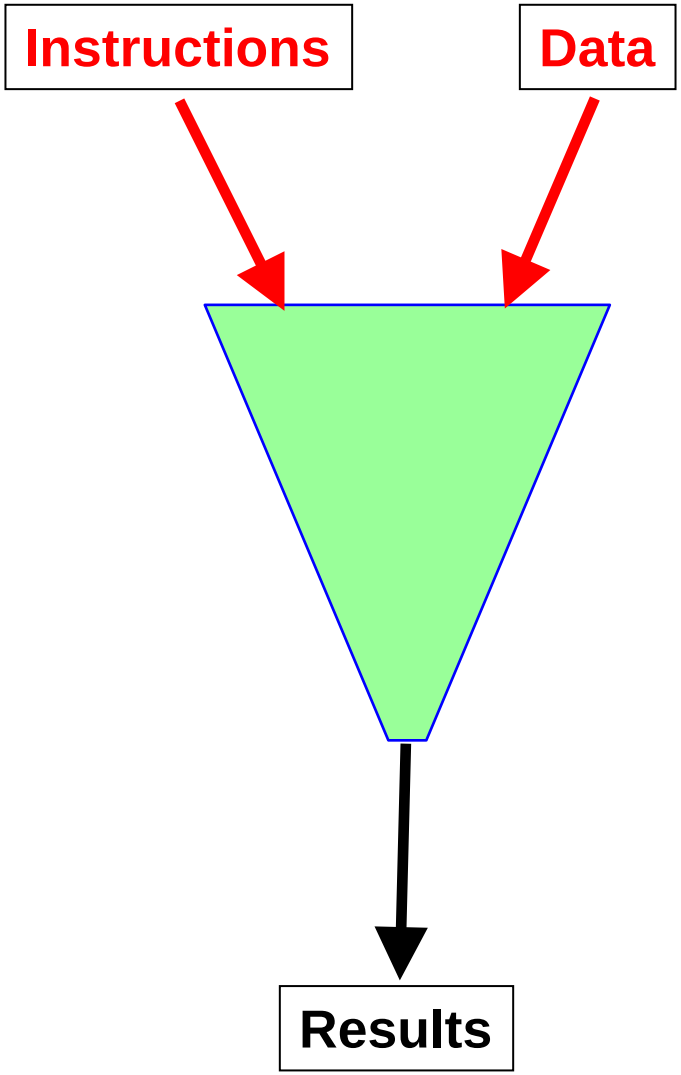
- **For a given algorithm, we can understand exactly which functional execution units are needed**
  - For instance, in the innermost loop of matrix multiplication

```
for ( int i = 0; i < N; ++i ) {
        for ( int j = 0; j < N; ++j ) {
                for ( int k = 0; k < N; ++k ) {
                        c[ i * N + j ]  +=  a[ i * N + k ]  *  b[ k * N + j ];
                }
        }
}
```



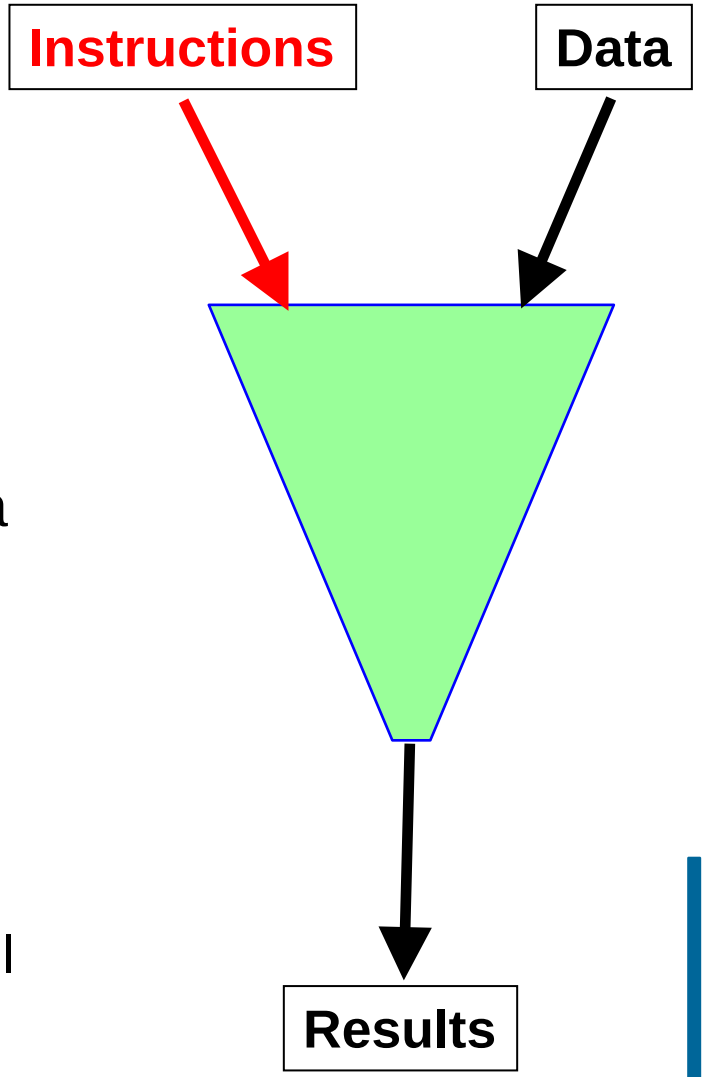
Possible causes of execution delays (1)

- **We already stated that the aim is to keep instructions and data flowing, so that results are generated optimally**
- **First issue:**
  - Instructions and/or data stop flowing
    - Instructions are not found in the I-cache
    - Data is not found in the D-cache
  - Before execution can continue, instructions and data must be fetched from a lower level of the memory hierarchy



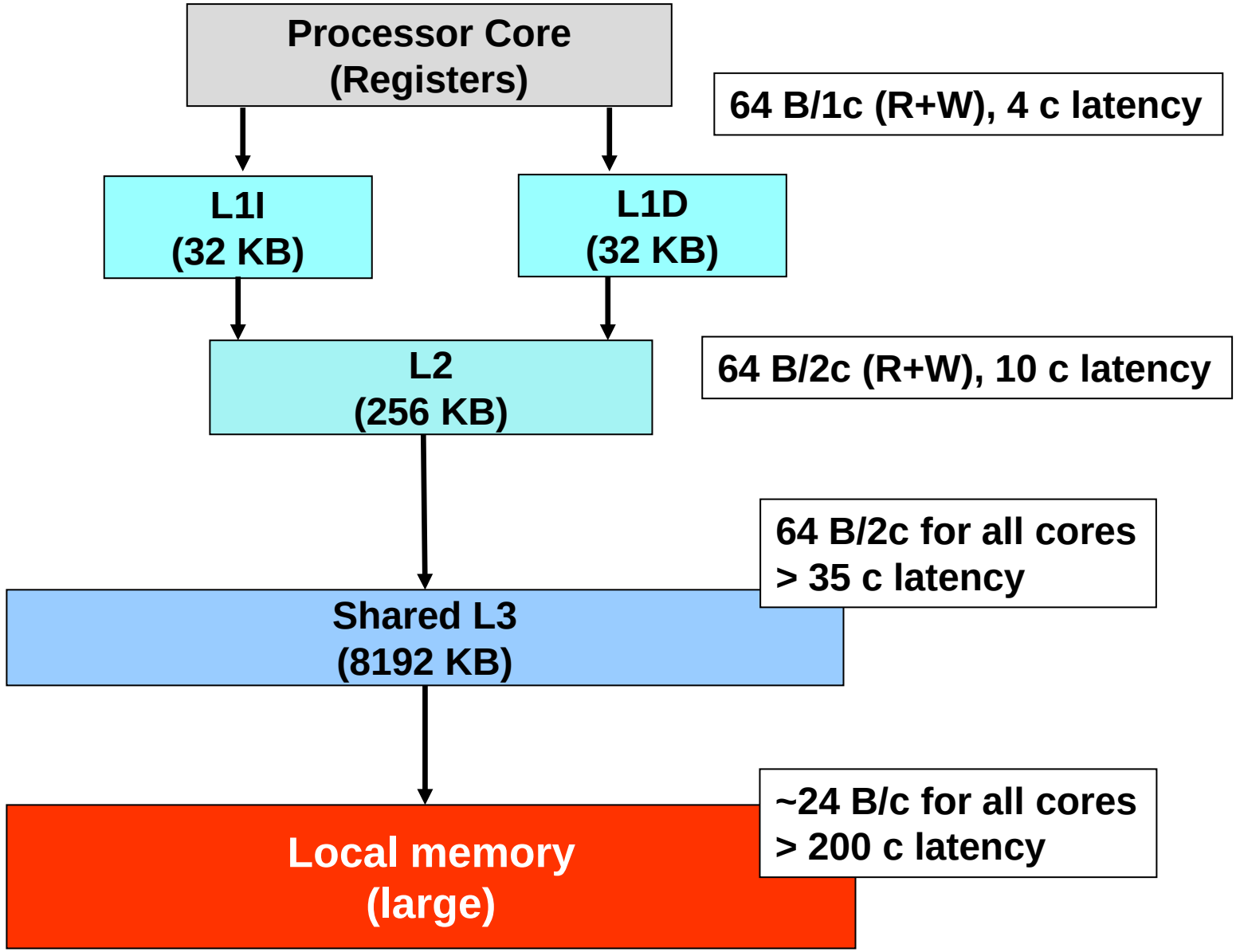
Other causes of execution delays (2)

- **Second issue:**
  - Instructions are not ready in time for execution (Front-end stalls)
    - Typically caused by branching
    - If the branch is mispredicted, we suffer a stall (cycles add up, but no work gets done)
    - We typically find that 10% of all instructions are branch instructions
      - Or even more
        - And, unavoidingly, some of them will be mispredicted



Memory Hierarchy

- **From CPU to main memory on a Nehalem processor**
  - With multicore, memory bandwidth is shared between cores in the same processor (socket)



c = cycle

Cache lines (1)

- **When a data element or an instruction is requested by the processor, a cache line is ALWAYS moved (as the minimum quantity) to Level-1**

- **A cache line is a contiguous section of memory, typically 64B in size (8 * double)**
  - A 32KB level-1 cache holds 512 (64B) lines
- **When cache lines have to be moved come from memory**
  - Latency is long (>200 cycles, as already mentioned)
    - It is even longer if the memory is remote
  - Memory controller stays busy (~8 cycles)

Cache lines (2)

- **Space locality is vital**
  - When only one element (4B or 8B) element is used inside the cache line:
    - A lot of bandwidth is wasted!

- **Multidimensional C arrays should be accessed with the last index changing fastest:**

```
for (i = 0; i < rows; ++i)
        for (j = 0; j < columns; ++j)
                mymatrix [i] [j]   += increment;
```

- **Pointer chasing (in linked lists) can easily lead to “cache thrashing” (too much memory traffic)**

Programming the memory hierarchy is an art in itself.

Inside-the-core: HEP and vectors

- **Too little common ground!**
  - Practically all attempts in the past failed.
    - w/CRAY, CYBER 205, IBM 3090-Vector Facility, etc.
    - Interesting reading: Dekeyser J 1987 “Vectorization of the GEANT3 geometrical routines for a Cyber 205” Nuclear Instruments and Methods in Physics Research Section A, Volume 264, Issue 2-3, p. 291-296
- **From time to time, we see a good vector example**
  - For example: Track Fitting code from ALICE trigger
    - → Explained in the examples
- **Interesting development from ALICE (Matthias Kretz):**
  - Vc (Vector Classes) being implemented into ROOT v.6
    - http://compeng.uni-frankfurt.de/index.php?id=vc
- **Hopefully, there will be renewed efforts to use vectors efficiently**

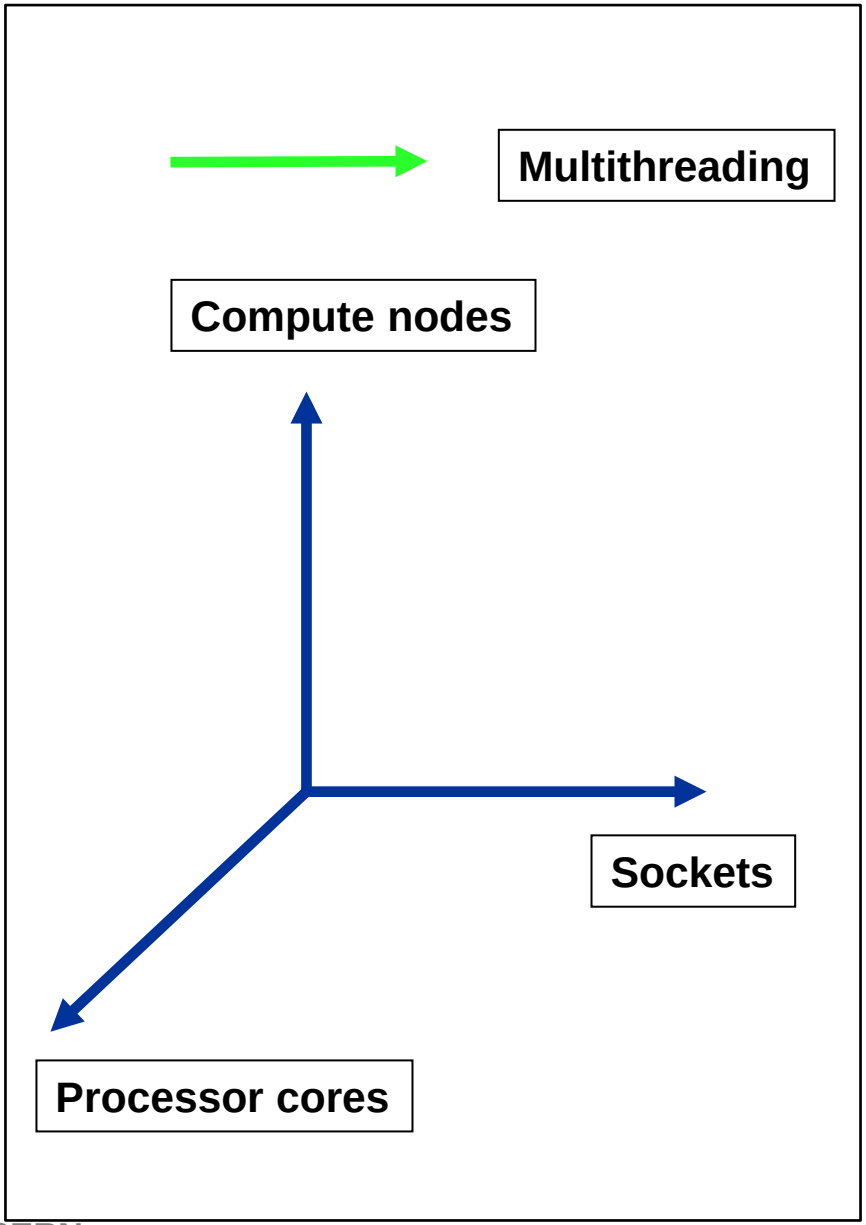
Important performance measurements
(that can tell you if things go wrong)

- **Related to what we have discussed:**
  - The total cycle count (C)
  - The total instruction count (I)
  - Derived value: CPI
  - Resource Stall count: Cycles when no execution occurred
  - Total number of executed branch instructions
  - Total number of mispredicted branches

- **Plus:**
  - Total number of cache accesses
  - Total number of (last-level) cache misses
  - The total number (and the type) of computational SSE/AVX instructions
  - The total number of SSE/AVX instructions

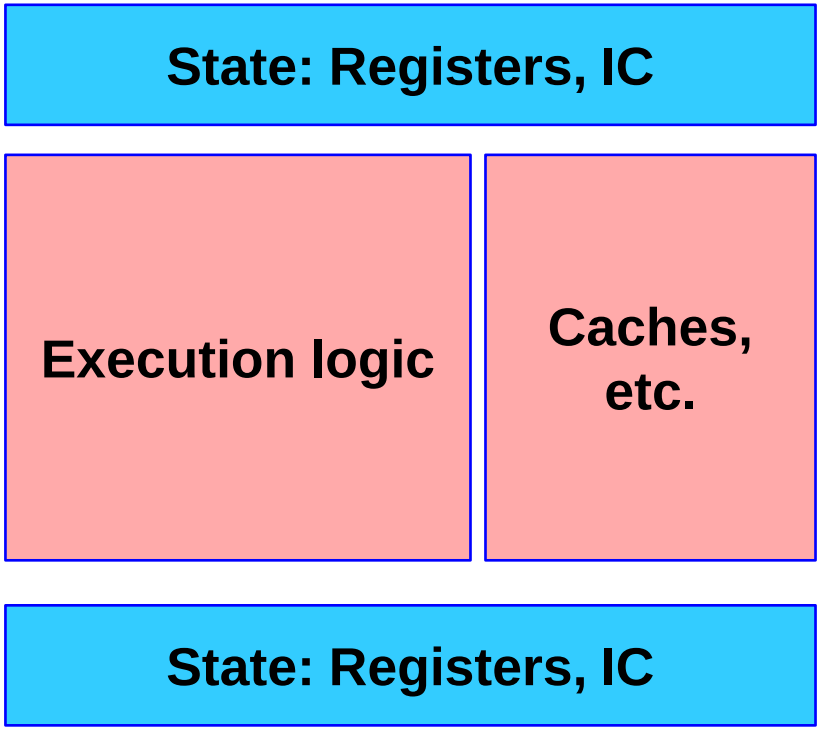
Part 2: Parallel execution across hw-threads and cores

- **Next dimension is a “pseudo” dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes
- **Multiple nodes will not be discussed here**
  - Our focus is scalability inside a node



Definition of a hardware core/thread

- **Core**
  - A complete ensemble of execution logic, and cache storage as well as register files plus instruction counter (IC) for executing a software process or thread
- **Hardware thread**
  - Addition of a set of register files plus IC



The sharing of the execution logic can be coarse-grained or fine-grained.

Please refer to slide 18

Definition of a software process and thread

- **Process (OS process):**
  - An instance of a computer program that is being executed (sequentially). It typically runs as a program with its private set of operating system resources, i.e. in its own “address space” with all the program code and data, its own file descriptors with the operating system permissions, its own heap and its own stack.

- **Thread:**
  - A process may have multiple 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.

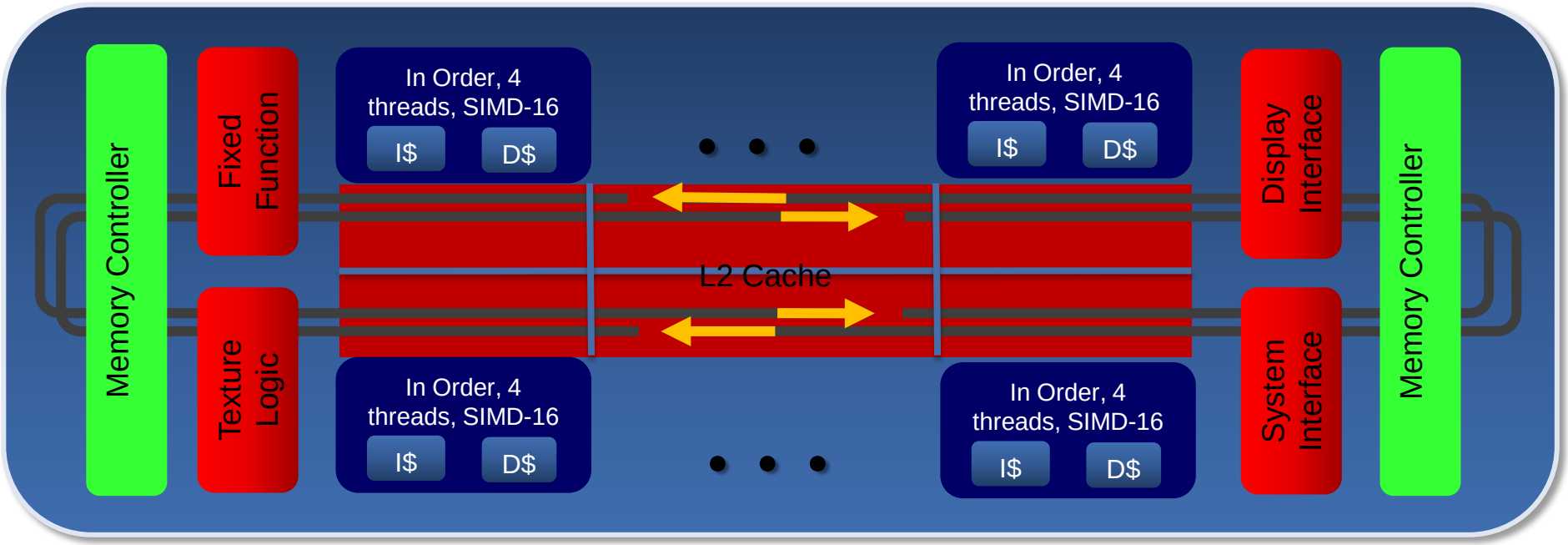
Adapted from Wikipedia

The move to many-core systems

- **Examples of "CPU slots": Sockets * Cores * HW-threads**
  - Basically what you observe in "cat /proc/cpuinfo"
  - Conservative:
    - Dual-socket AMD six-core (Istanbul): 2 * 6 * 1 = 12
    - Dual-socket Intel six-core (Westmere-EP): 2 * 6 * 2 = 24
  - More aggressive:
    - Quad-socket AMD Interlagos (16-core) 4 * 16 * 1 = 64
    - Quad-socket Westmere-EX "octo-core": 4 * 10 * 2 = 80
- **In the near future: Hundreds of CPU slots !**
  - Quad-socket Oracle/Sun Niagara (T3) processors w/16 cores and 8 threads (each): 4 * 16 * 8 = 512
- **And, by the time new software is ready: Thousands !!**

Accelerators (1): Intel Xeon Phi

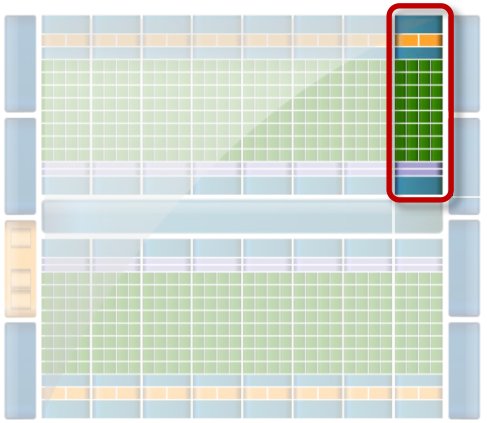
- **Intel Many Integrated Cores (MIC):**
  - Announced at ISC10, available 2 ½ years later
  - Based on the x86 architecture, 22nm
  - Many-core (up to 62 cores) + 4-way multithreaded + 512-bit vector unit
  - Limited memory: 8 Gigabytes

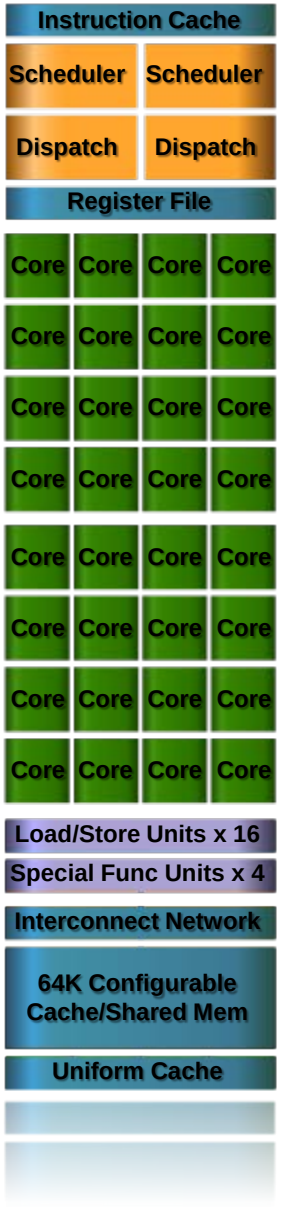


Accelerators (2): Nvidia Fermi GPU

- **Streaming Multiprocessing (SM) Architecture**

- 32 “CUDA cores” per SM (512 total)
- Peak single precision floating point performance (at 1.15 GHz”:
  - Above 1 Tflops
- Double-precision: ~300 Gigaflops
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)
- A few Gigabytes of main memory





Considerable interest in the HEP on-line community

Adapted from Nvidia

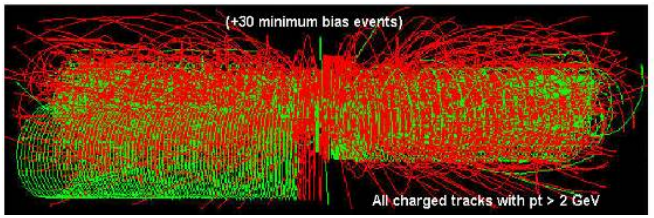
Accelerators (3): Nvidia Kepler GPU

- **Made available in 4Q2012**
  - GK110 GPU
  - 3x DP performance:
    - 1 Teraflops
  - Innovative design:
    - **SMX** (streaming multiprocessors)
    - **Dynamic parallelism** for spawning new threads
    - **Hyper-Q** enables multiple CPU cores to utilise CUDA cores

Adapted from Nvidia

HEP programming paradigm

- **Event-level parallelism has been used for decades**
- **And, we should not lose this advantage:**
  - Large jobs can be split into N efficient “chunks”, each responsible for processing M events
  - Has been our “forward scalability”



- **Disadvantage with current approach:**
  - Memory must be made available to each process
    - A dual-socket server with six-core processors needs 24 – 36 GB (or more)
    - Today, SMT is often switched off in the BIOS (!)
- **We must not let memory limitations decide our ability to compute efficiently!**

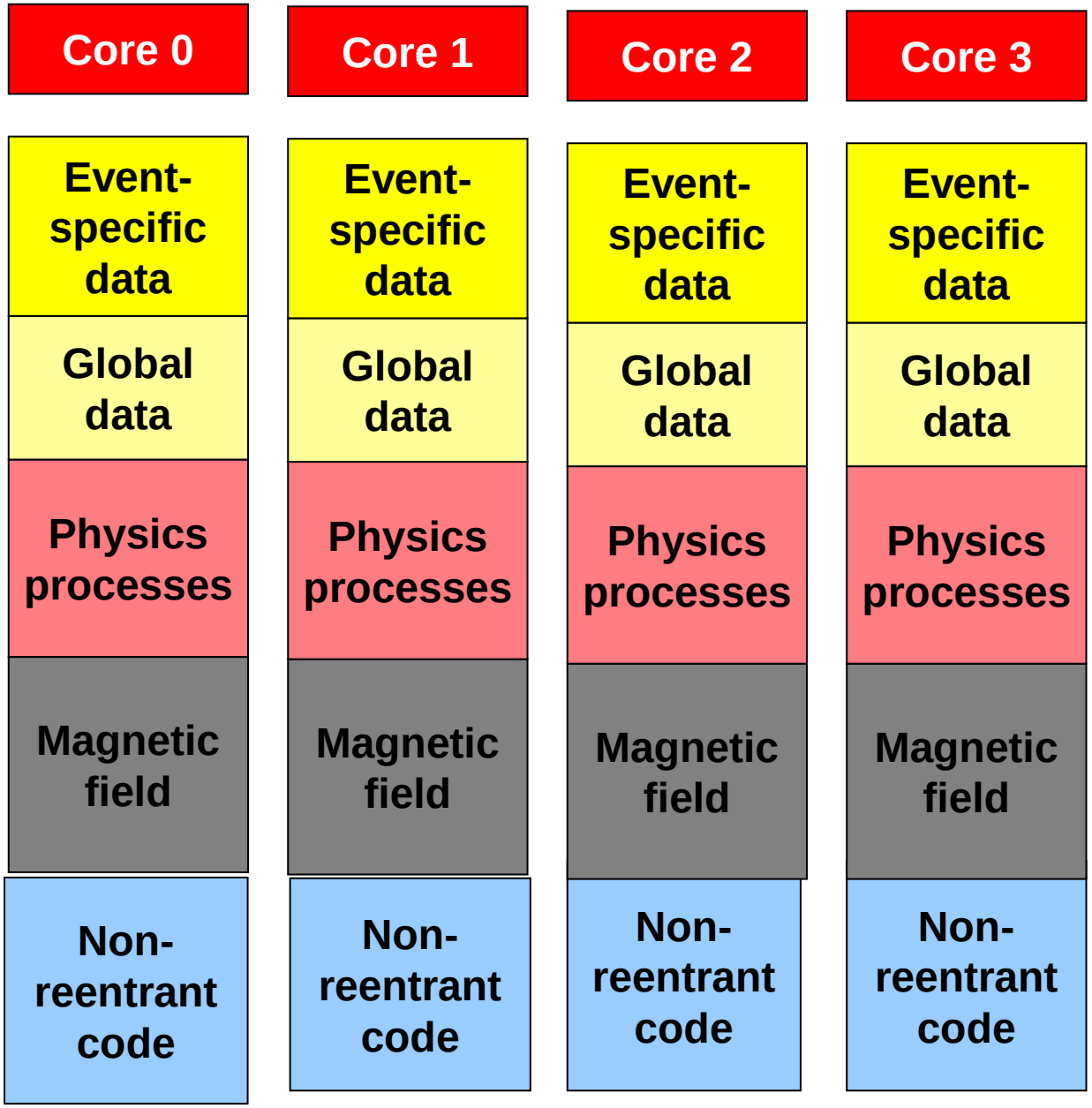
What are the options?

- **There is currently a discussion in the community about the best way forward (in a many-core world):**

1) Stay with event-level parallelism (and entirely independent processes)
   - Assume that the necessary memory remains affordable
   - Or rely on tools, such as KSM, to help share pages
2) Rely on forking:
   - Start the first process; Fork N others
   - Rely on the OS to do “copy on write”, in case pages are modified
3) Move to a fully multi-threaded paradigm
   - Still using coarse-grained (event-level) parallelism
     - But, watch out for increased complexity

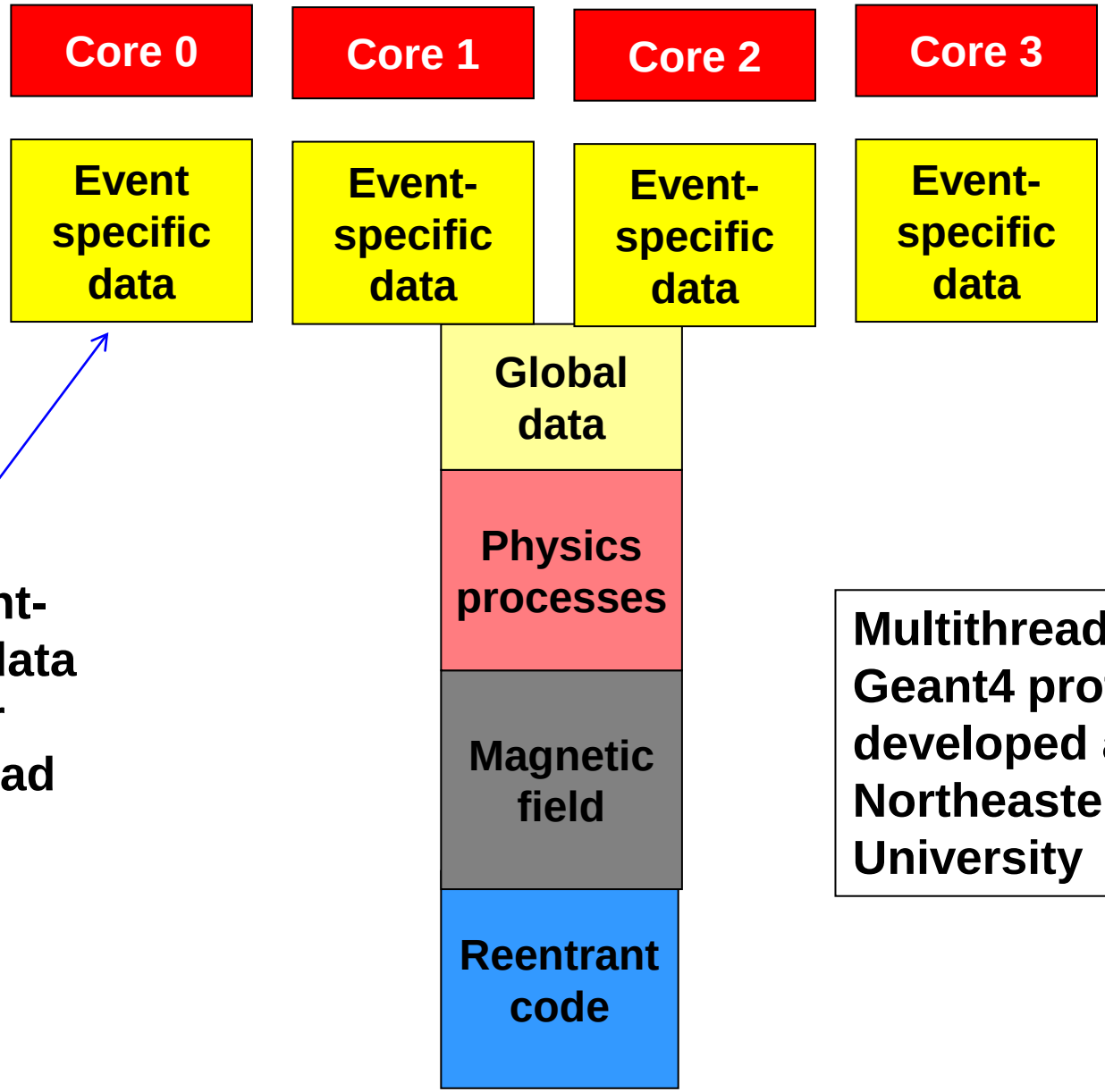
Towards an efficient memory footprint

- **Rather than a wasteful approach:**



Towards an efficient memory footprint (2)

- **A much more elegant memory usage:**



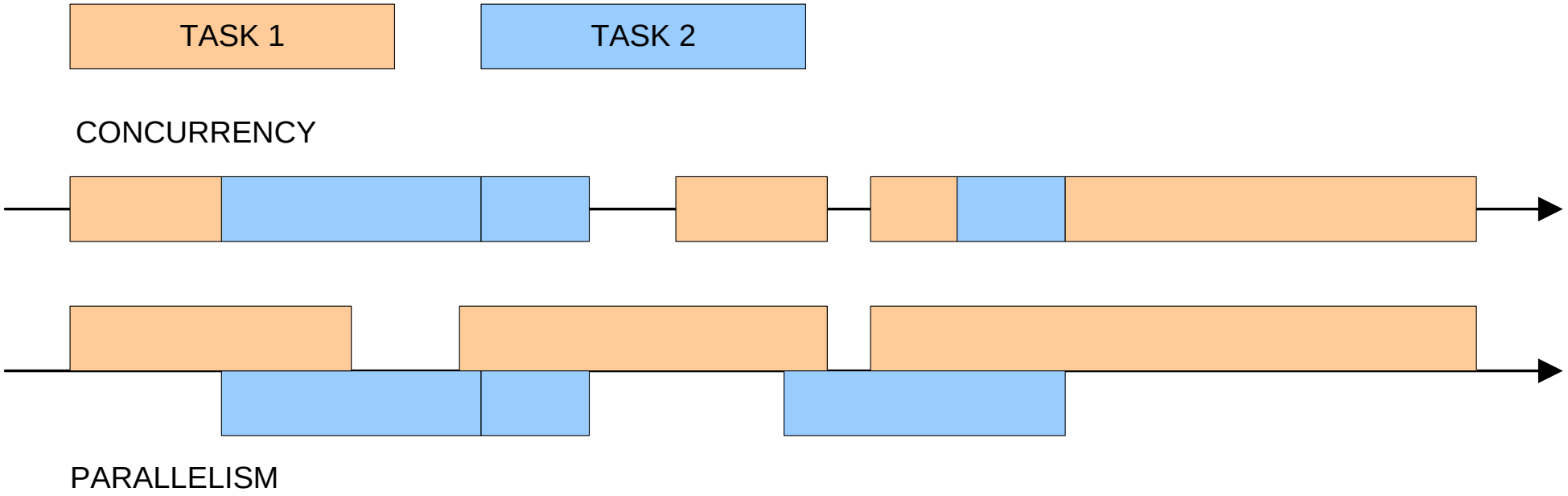
Only event-specific data added for each thread

Multithreaded Geant4 prototype developed at Northeastern University

Let’s briefly introduce parallelism

Definition of concurrency/parallelism

- **Concurrent programming:**
  - Expression of a total algorithmic problem in logically independent parts (independent control flows)
- **Parallel execution**
  - Independent parts of a program execute simultaneously



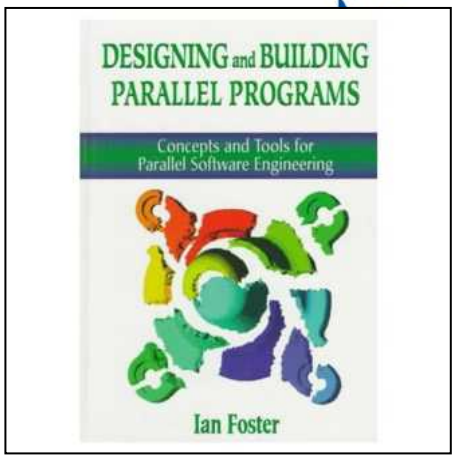
From Concurrency to Parallel Execution

- **Multiple steps must be kept in mind:**
  - Concurrency
  - Decomposition
  - Communication
  - Synchronization
  - Mapping
  - Execution

- **Keeping Amdahl's law for max speedup in mind**

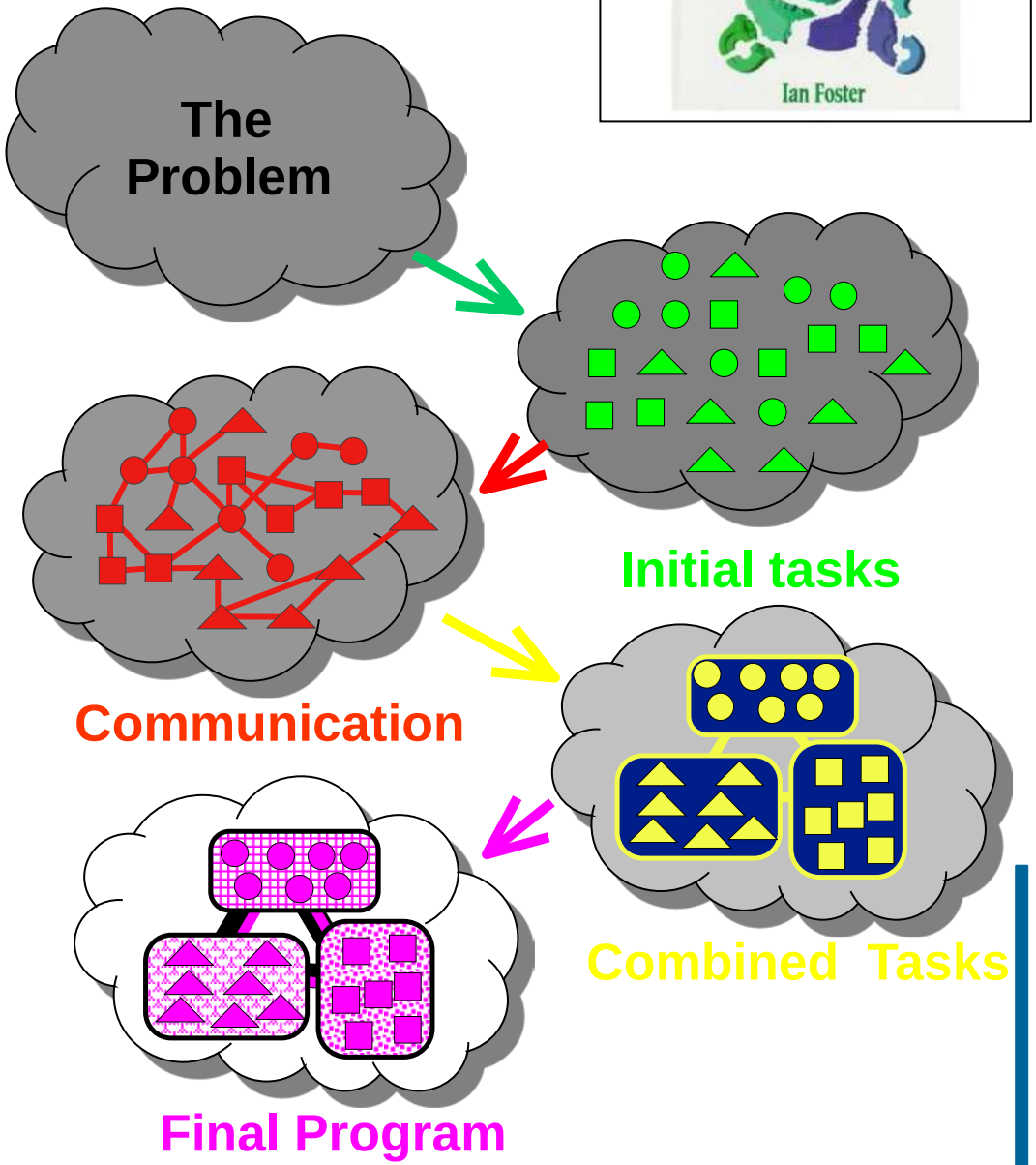
$$S_p^{\max}(n) = \frac{1}{1 - p + \frac{p}{n}}$$

where:
p (parallel portion)
s (serial portion)
p + s = 1.0



Designing Threaded Programs

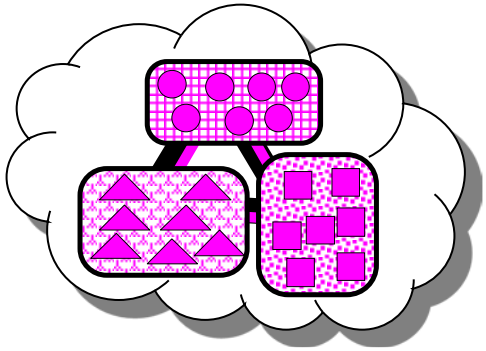
- **Partition**
  - Divide problem into tasks
- **Communicate**
  - Determine amount and pattern of communication
- **Agglomerate**
  - Combine tasks
- **Map**
  - Assign agglomerated tasks to created threads



More on decomposition

- **Divide the total work into smaller parts,**
  - Which can be executed concurrently

- **Some techniques:**
  - Data decomposition
    - Partition the data domain
  - Functional decomposition
    - Split according to program functions
  - Task decomposition
    - Split according to “logical” tasks
  - Recursive decomposition
    - Divide-and-conquer strategy
  - Exploratory decomposition
    - Search for a configuration space for a solution
      - Not guaranteed to reduce amount of work



Recommendations

(based on observations in openlab)

A proposal for “agile” software:

1) **Create compute-intensive kernels**
2) **Seek out parallelism at all levels**
   a. Events, tracks, vertices, etc.
   b. Perform “chunk” processing (removing event separation)
3) **Build forward scalability**
4) **Aim for an efficient memory footprint**
5) **Optimise data layout for locality of reference**
6) **Program in performance-oriented C++**
7) **Combine broad Programming Talents**
8) **Use best-of-breed Tools**

Performance guidance

- **Take the whole program and its execution behaviour into account**
  - Get yourself a global overview as soon as possible
    - Via early prototypes
    - Influence early the design and definitely the implementation
- **Foster clear split:**
  - Prepare to compute
  - Do the heavy computation
    - In kernels, where you go after all the available parallelism
  - Post-processing

- **Often, a single kernel is not sufficient**
  - A sequence of kernels may be needed

Performance recommendations

- **Control memory usage (both in a multi-core and an accelerator environment):**
  - Optimize malloc/free
  - Optimize the cache hierarchy
    - Prefer SoA to minimise cache misses
  - Forking is good; it may cut memory consumption in half
  - Don’t be afraid of threading; it may perform miracles !
  - Be aware of NUMA (Non-Uniform Memory Access): The “new” blessing (or curse?)

C++ parallelization support

- **Large selection of tools (inside the compiler or as additions):**
  - Native: pthreads/Windows threads
  - New C++ standard: std::thread
  - OpenMP
  - Intel Threading Building Blocks (TBB)
  - Thread wrapper classes
  - MPI (from multiple providers), etc.
  - CUDA (from Nvidia) ← Not exactly C++, but…

We must also keep a close eye on OpenCL and OpenACC

Performance-oriented C++

- **C++ for performance**
  - Use light-weight C++ constructs
  - Minimize virtual functions
  - Inline whenever important
  - Optimize the use of math functions
    - SQRT, DIV
    - LOG, EXP, POW
    - SIN, COS, ATAN2

Use vector libraries whenever possible

Learn to inspect the compiler-generated assembly, especially of kernels

Performance tools

- **Surround yourself with good tools:**
  - Compilers (not just one!)
  - Libraries
  - Profilers
  - Debuggers
  - Thread checkers
  - Thread profilers

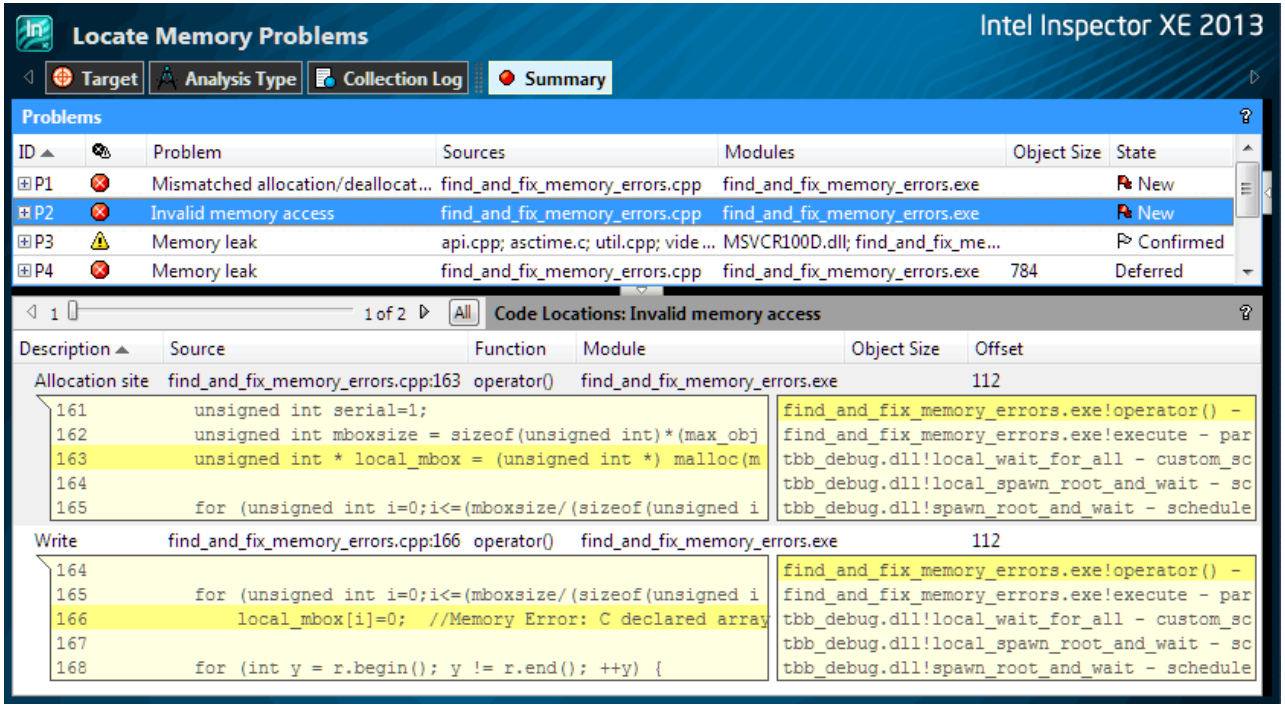
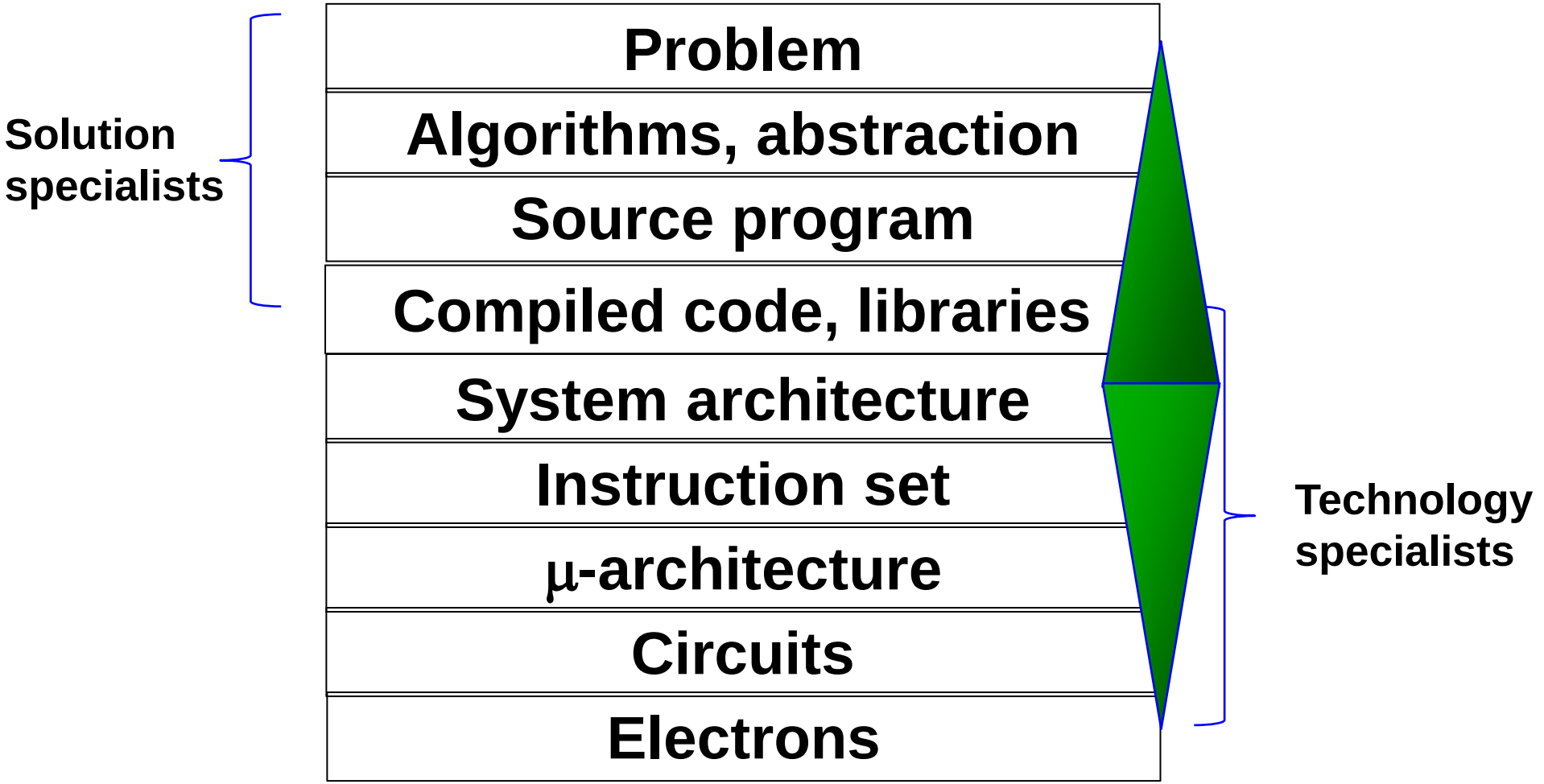


Image: software.intel.com

Broad Programming Talent

- **In order to cover as many layers as possible**



HEP examples

HEP and Symmetric Multi-Threading

- **Because we have "thin" instruction streams, we ought to profit from SMT, provided the memory issue is under control**
  - It would seem that we could easily tolerate up to 4 hardware threads!

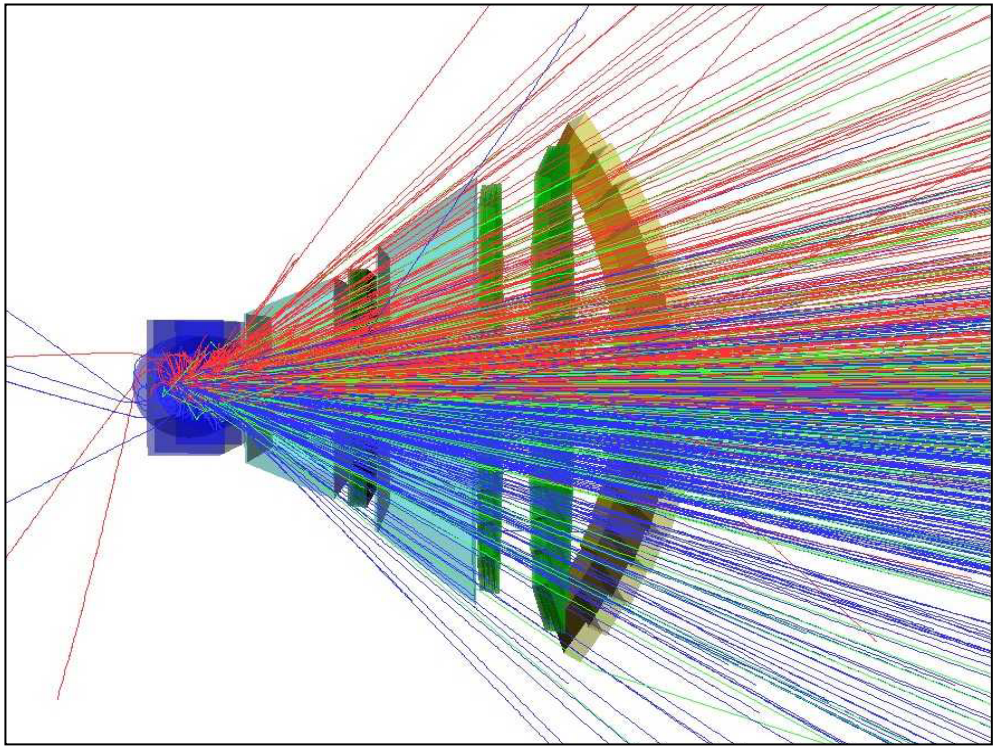
Unfortunately, on Xeon E5-2680 (EP Sandy Bridge), we currently get ~23% from the second hardware thread ! Will it change with Haswell ?

Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1			load point[0]			
2			load origin[0]			
3						
4						
5						
6		subsd	load float-packet			
7						
8			load xhalfsz			
9						
10	andpd					
11						
12	comisd					
13						jbe

Examples of parallelism: CBM/ALICE track fitting

- **Extracted from the High Level Trigger (HLT) Code**
  - Originally ported to IBM's Cell processor
- **Tracing particles in a magnetic field**
  - Embarrassingly parallel code
- **Re-optimization on x86-64 systems**
  - Using vectors instead of scalars

I.Kisel/GSI: "Fast SIMDized Kalman filter based track fit"
http://www-linux.gsi.de/~ikisel/17_CPC_178_2008.pdf



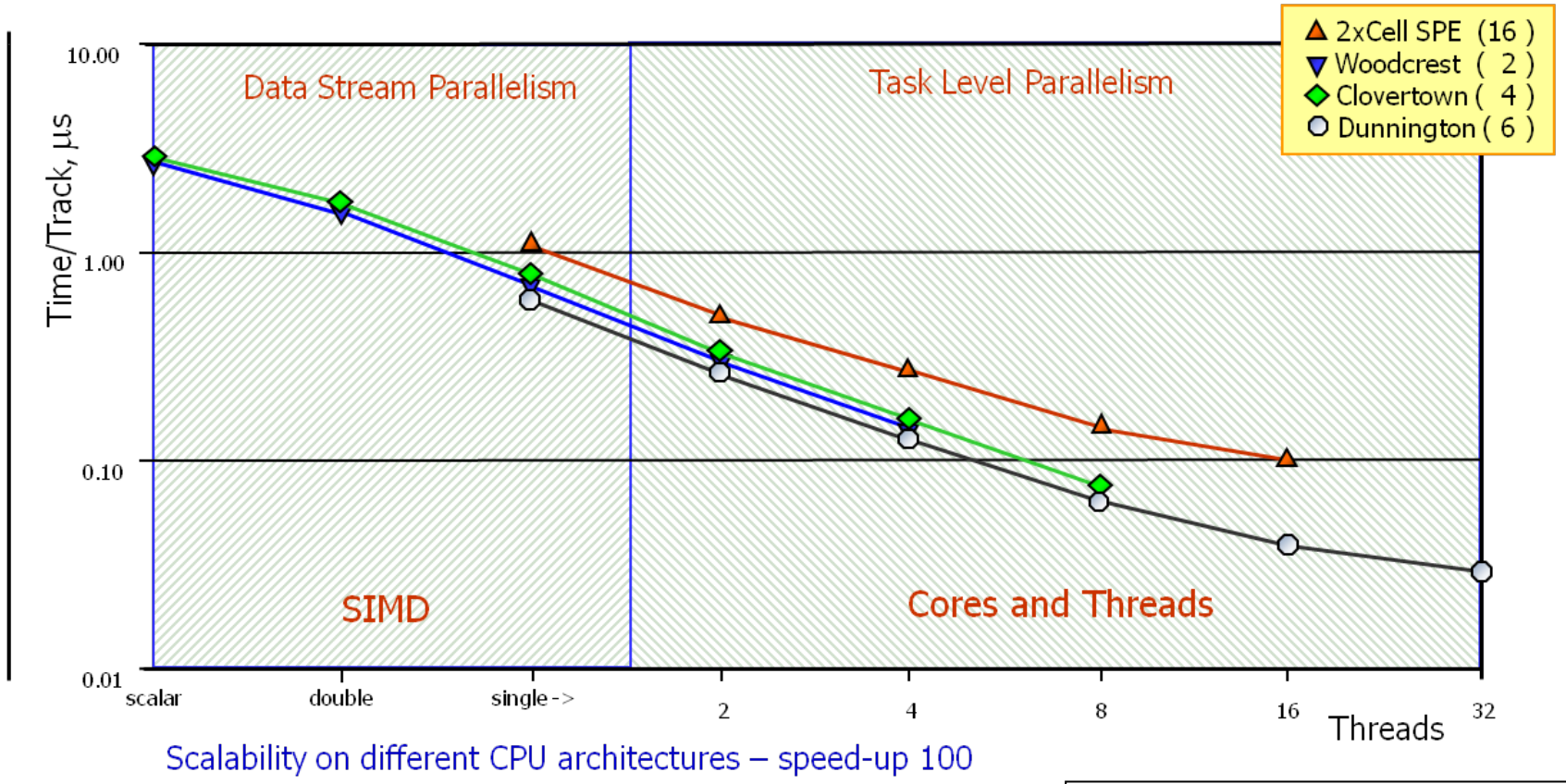
"Compressed Baryonic Matter"

CBM/ALICE track fitting

- **Details of the re-optimization on x86-64:**
  - Part 1: use SSE vectors instead of scalars
    - Operator overloading allows seamless change of data types
    - Intrinsics (from Intel/GNU header file): Map directly to instructions:
      - __mm_add_ps corresponds directly to ADDPS, the instruction that operates on **four** packed, single-precision FP numbers
        - 128 bits in total
    - Classes
      - P4_F32vec4 – packed single class with overloaded operators
        - F32vec4 operator +(const F32vec4 &a, const F32vec4 &b) { return _mm_add_ps(a,b); }
    - Result: 4x speed increase from x87 scalar to packed SSE (single precision)

Examples of parallelism: CBM track fitting

- **Re-optimization on x86-64 systems**
  - Step 1: Data parallelism using SIMD instructions
  - **Step 2**: use TBB (or OpenMP) to scale across cores

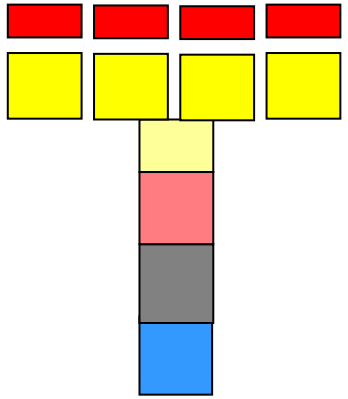


Scalability on different CPU architectures – speed-up 100

From H.Bjerke/CERN openlab, I.Kisel/GSI

Examples of parallelism: GEANT4

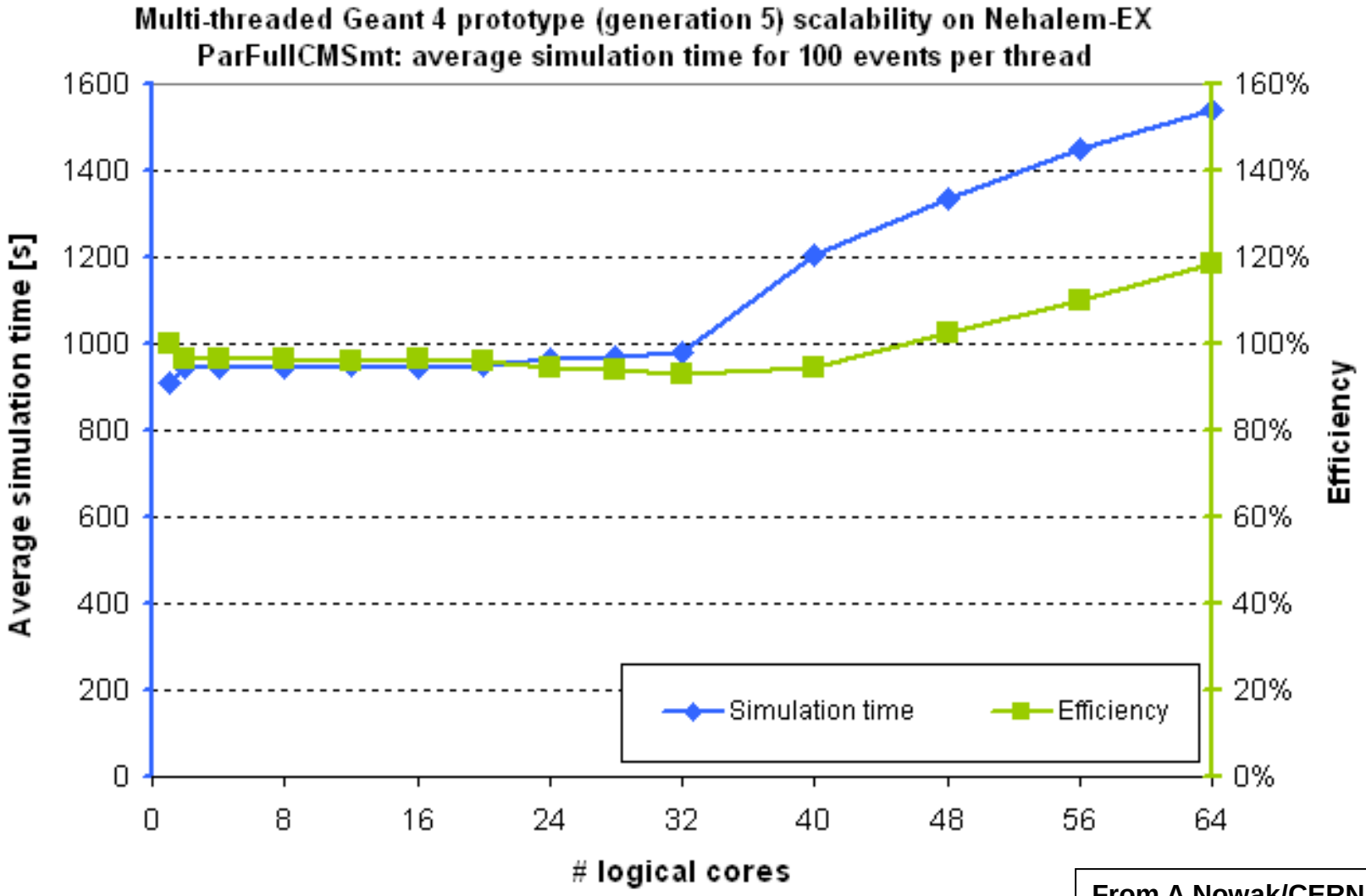
- **Initially; ParGeant4 (Gene Cooperman/NEU)**
  - implemented event-level parallelism to simulate separate events across remote nodes.
- **New prototype re-implements thread-safe event-level parallelism inside a multi-core node**
  - Done by NEU PhD student Xin Dong:
    - Using FullCMS and TestEM examples
  - Required change of lots of existing classes (10% of 1 MLOC):
    - Especially *global*, "*extrn*", and *static* declarations
    - Preprocessor used for automating the work.
  - Major reimplementation:
    - Now in separate branch in the G4 source tree



- **Additional memory: Only 25 MB/thread (!)**

Multithreaded GEANT4 benchmark

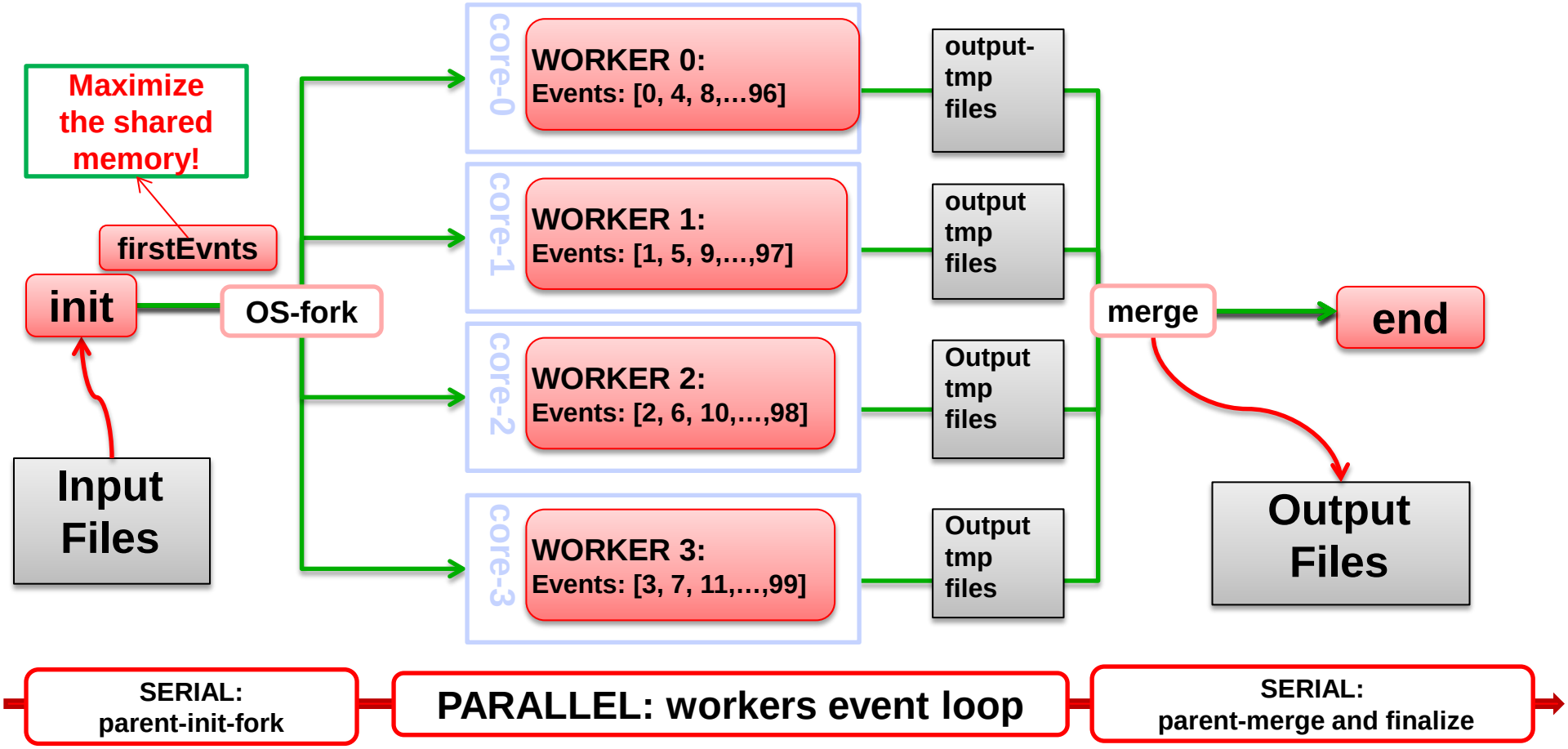
- **Excellent "weak" scaling on 32 (real) cores**
  - With a 4-socket server



From A.Nowak/CERN openlab

ATHENA-MP: event-level parallelism

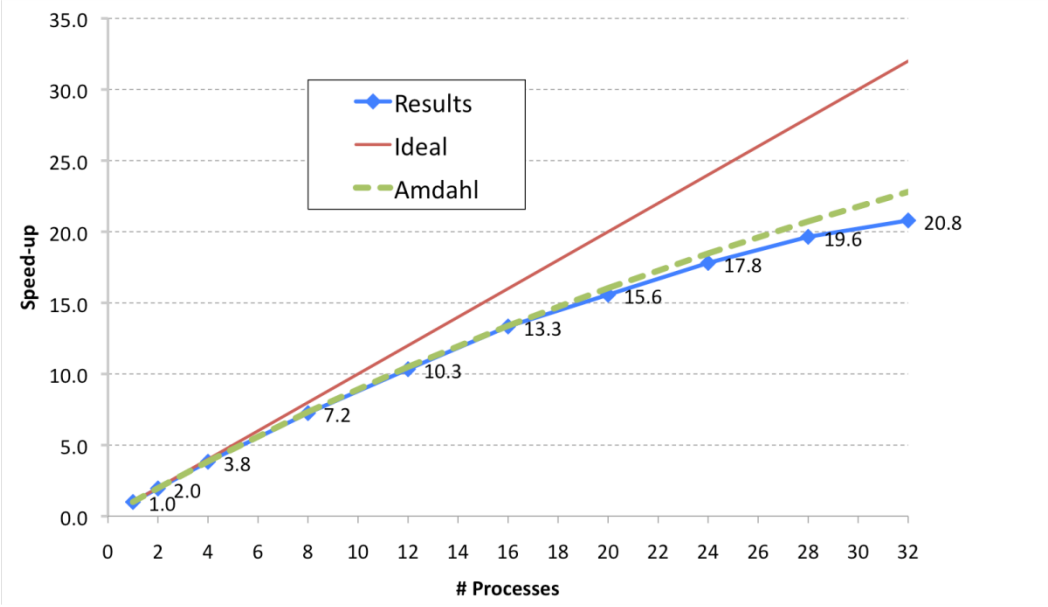
- **Based on forking.**
  - Reduces the memory footprint by 30 – 50%



From M.Tatarkhanov/ATLAS

Example: ROOT minimization and fitting

- **Minuit parallelization is independent of user code**
- **Log-likelihood parallelization (splitting the sum) is quite efficient**
- **Example on a 32-core server:**



Recent paper:
Comparison of Software Technologies for Vectorization and Parallelization
(CERN openlab, 2012)

- **In principle, we can have combinations of:**
  - vectorization (using SSE or AVX)
  - parallelization via multi-threading in a multi-core CPU
  - multiple process in a distributed computing environment

If you think that all of this is “crazy”

- **Please read:**
  - “Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor”
    - J.Kurzak, W.Alvaro, J.Dongarra
    - Parallel Computing 35 (2009) 138–150

In this paper, single-precision matrix multiplication kernels are presented implementing the C = C – A x B^T operation and the C = C – A x B operation for matrices of size 64x64 elements. For the latter case, the performance of 25.55 Gflop/s is reported, or 99.80% of the peak, using as little as 5.9 kB of storage for code and auxiliary data structures.

Concluding remarks

- **The aim of these lectures was to help understand:**
  - Changes in modern computer architecture
  - Impact on our programming methodologies
- **Keeping in mind that there is not always a straight path to reach (all of) the available performance by our programming community.**
- **In most HEP programming domains event-level processing will (continue to) dominate**
  - Provided we get the memory requirements under control
- **Will you be ready for 100+ cores and long vectors?**
  - Are you thinking “parallel, parallel, parallel” ?
- **It helps to know the seven hardware dimensions and how appropriate software constructs can assist you !**

Further reading:

- **“Designing and Building Parallel Programs”, I. Foster, Addison-Wesley, 1995**
- **“Foundations of Multithreaded, Parallel and Distributed Programming”, G.R. Andrews, Addison-Wesley, 1999**
- **“Computer Architecture: A Quantitative Approach”, J. Hennessy and D. Patterson, 3rd ed., Morgan Kaufmann (MK), 2002**
- **“Patterns for Parallel Programming”, T.G. Mattson, Addison Wesley, 2004**
- **“Principles of Concurrent and Distributed Programming”, M. Ben-Ari, 2nd edition, Addison Wesley, 2006**
- **“The Software Vectorization Handbook”, A.J.C. Bik, Intel Press, 2006**
- **“The Software Optimization Cookbook”, R. Gerber, A.J.C. Bik, K.B. Smith and X. Tian; Intel Press, 2nd edition, 2006**
- **“Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism”, J. Reinders, O’Reilly, 1st edition, 2007**
- **“Inside the Machine”, J. Stokes, Ars Technica Library, 2007**
- **“Structured Parallel Programming; Patterns for Efficient Computation”, M. McCool et al, MK, 2012**

Thank you!