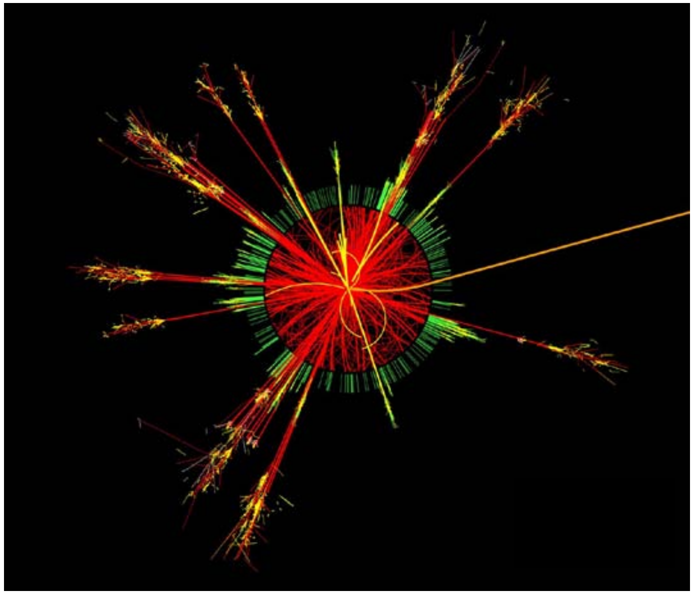
Efficient use of modern CPU architectures

“The 7 dimensions of performance”

Sverre Jarp
CERN
openlab
CTO



Bertinoro, 22 November 2010

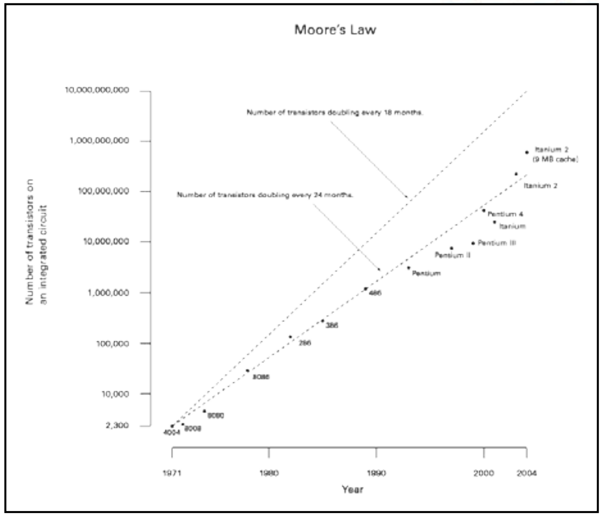
Contents

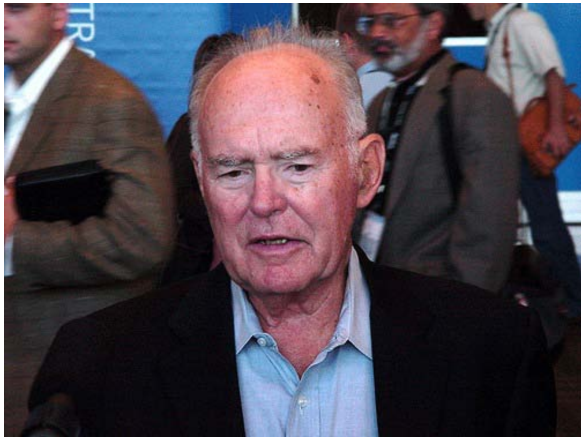
- **The driving force: Moore’s law**
- **Review of fundamental architectural principles**
- **Addressing performance “dimensions”**
- **Scaling “inside-a-core”:**
  - First 3 dimensions
  - Causes of execution delays
  - Performance monitoring
- **Scaling “across-cores”**
  - Next set of dimensions
  - Parallel programming paradigms
  - Achieving better memory footprints
  - C++ parallelization support
  - Example of parallelization: Track fitting and others
- **Conclusions**

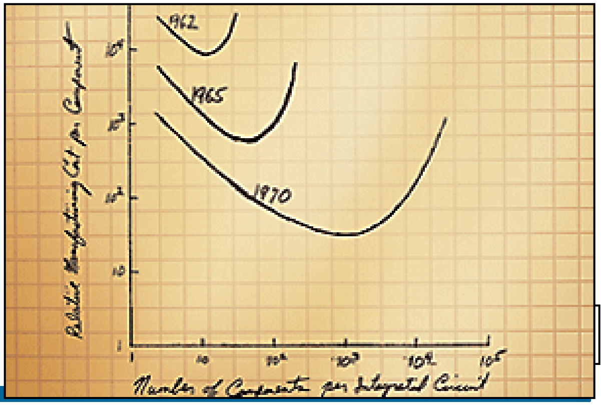
Moore’s law

- **We continue to double the number of transistors every other year**
  - The consequences
  - CPUs
    - Single core → Multicore → Manycore
    - Vectors
    - Hardware threading
  - GPUs
    - Huge number of FMA units

- **Today we commonly acquire chips with 1’000’000’000 transistors!**





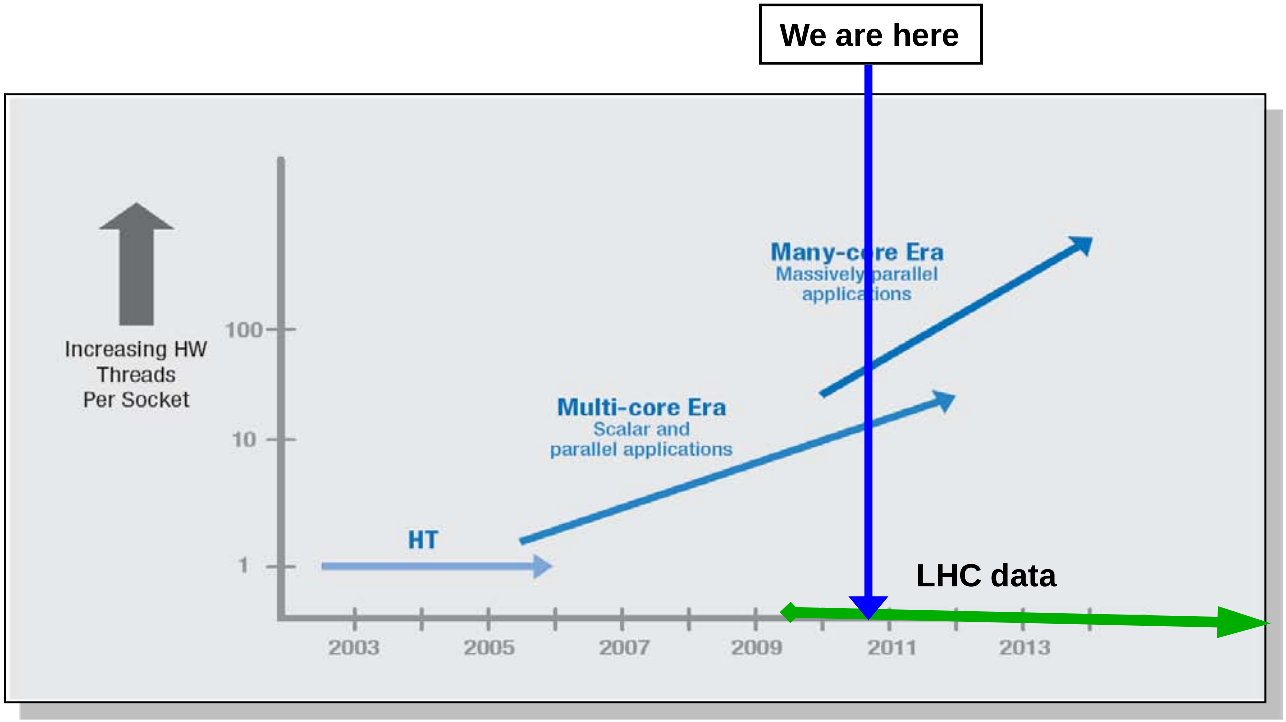

Relative Manufacturing Cost per Component
1962
1965
1970
10^5
10^3
10^2
10
1
10
10^2
10^3
10^4
10^5
Number of Components per Integrated Circuit

Real consequence of Moore’s law

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

“Intel platform 2015” (and beyond)

- **Today’s silicon processes:**
  - 45 nm
  - 32 nm
- **On the roadmap:**
  - 22 nm (2011/12)
  - 16 nm (2013/14)
- **In research:**
  - 11 nm (2015/16)
  - 8 nm (2017/18)
    - Source: Bill Camp/Intel HPC



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

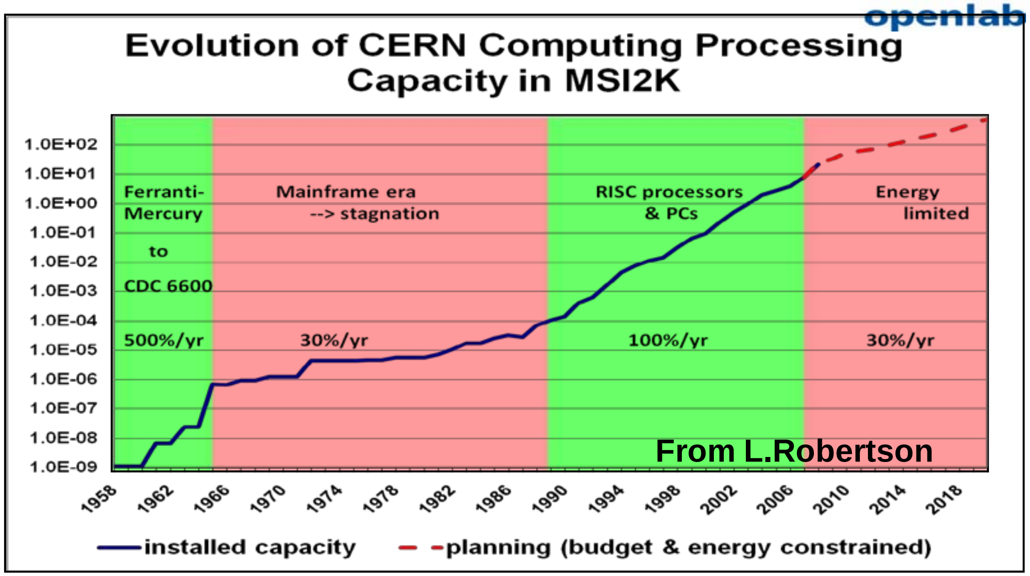
- **Each generation will push the core count:**
  - We are entering the many-core era (whether we like it or not) !

The holy grail: Forward scalability

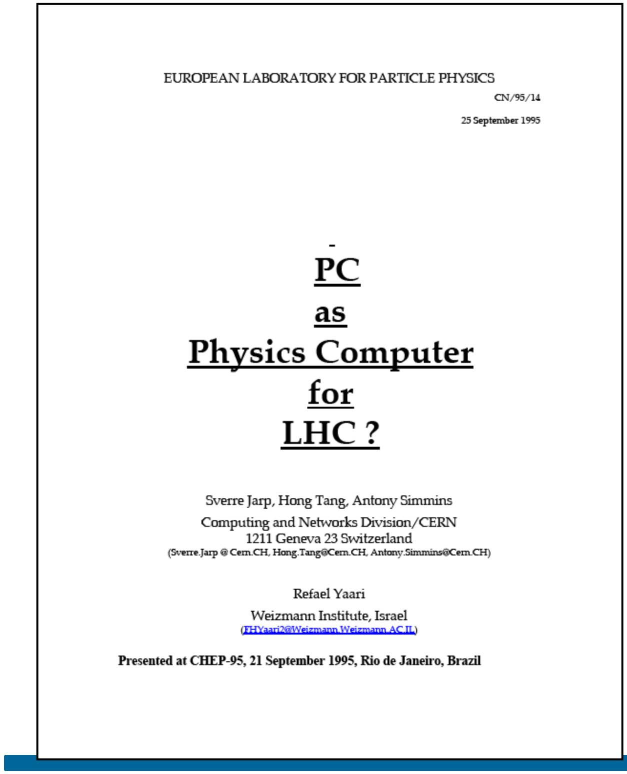
- **Not only should a program be written in such a way that it extracts maximum performance from today's hardware**
- **On future processors, performance should scale automatically**
  - In the worst case, one would have to recompile or relink
- **Additional CPU/GPU hardware, be it cores/threads or vectors, 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!**

Evolution of CERN’s computing capacity

- **During the LEP era (1989 – 2000):**
  - Doubling of total computing capacity every year
  - Initiated with the move from mainframes to RISC systems
- **The PC has been with us for 15 years!**
  - At CHEP-95 I made the first recommendation to move to PCs
    - After a set of encouraging benchmark results



Evolution of CERN Computing Processing Capacity in MSI2K



EUROPEAN LABORATORY FOR PARTICLE PHYSICS

CN/95/14

25 September 1995

-

PC

as

Physics Computer

for

LHC ?

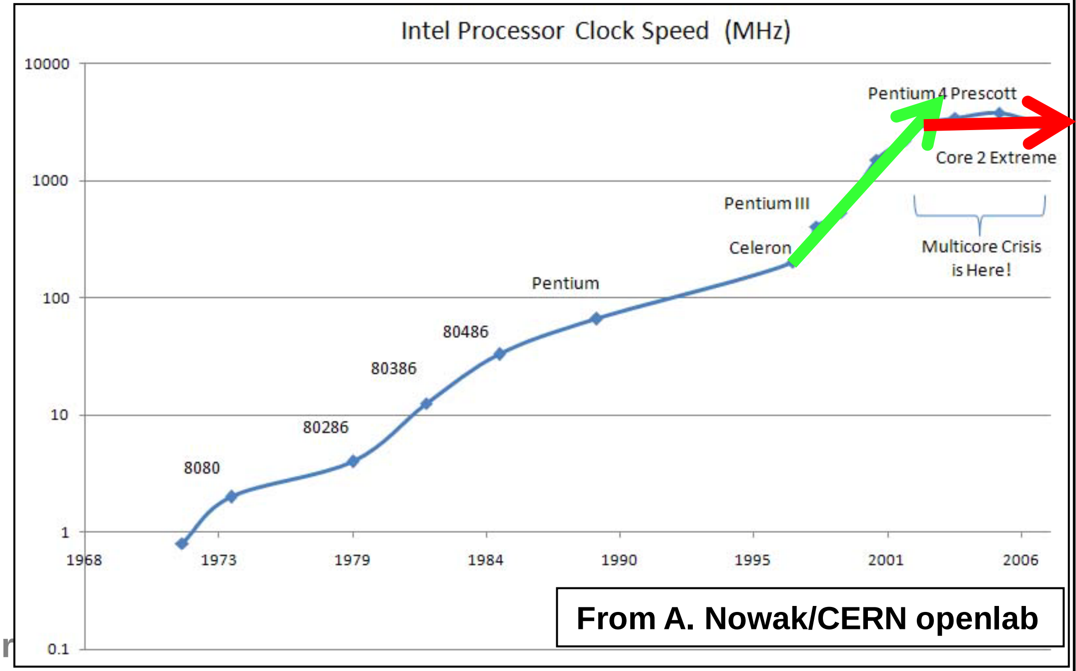
Sverre Jarp, Hong Tang, Antony Simmins
Computing and Networks Division/CERN
1211 Geneva 23 Switzerland
(Sverre.Jarp @ Cern.CH, Hong.Tang@Cern.CH, Antony.Simmins@Cern.CH)

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

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

Frequency scaling

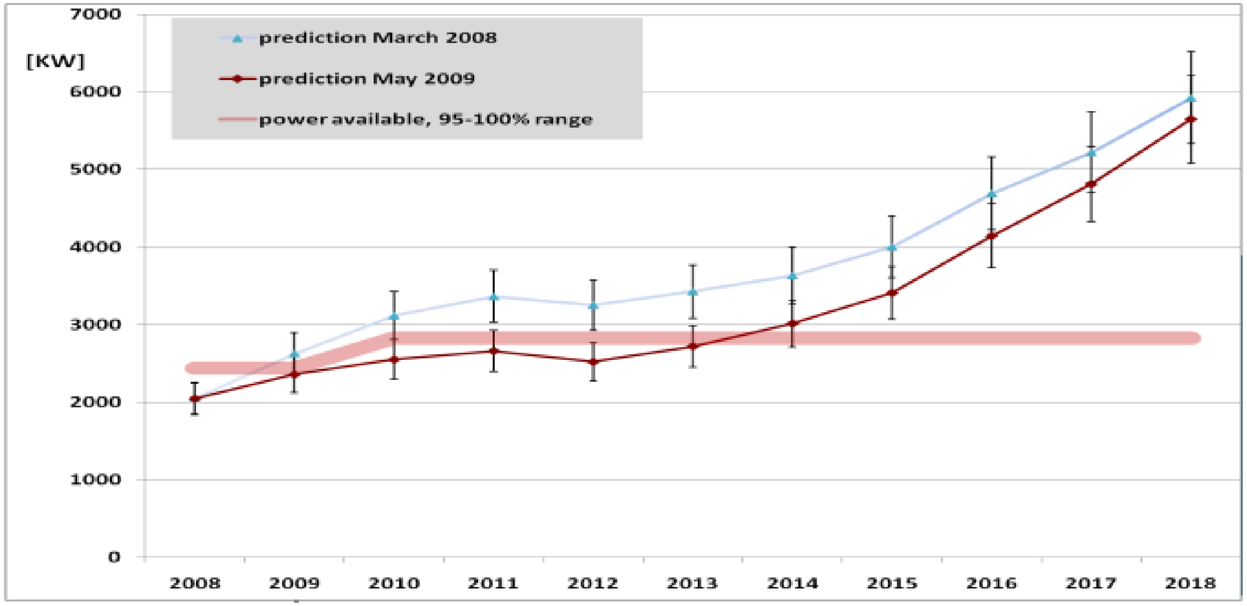
- **The 7 “fat” years of <u>easy</u> frequency scaling in HEP**
  - The Pentium Pro in 1996: 150 MHz
  - The Pentium 4 in 2003: 3.8 GHz (~25x)
- **Since then**
  - Core 2 systems:
    - ~3 GHz
    - Multi-core
- **Recent CERN purchase:**
  - Intel L5640 CPUs
    - 2.26 GHz



From A. Nowak/CERN openlab

The Power Wall

- **For example, the CERN Computer Centre can supply 2.9 MW of electric power**
  - Plus 2.3 MW to remove the corresponding heat!
- **Spread over a complex infrastructure:**
  - CPU servers; Disk servers
  - Tape servers + robotic equipment
  - Database servers
  - Infrastructure.
  - Network



- **We are hovering around the limit!**

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.

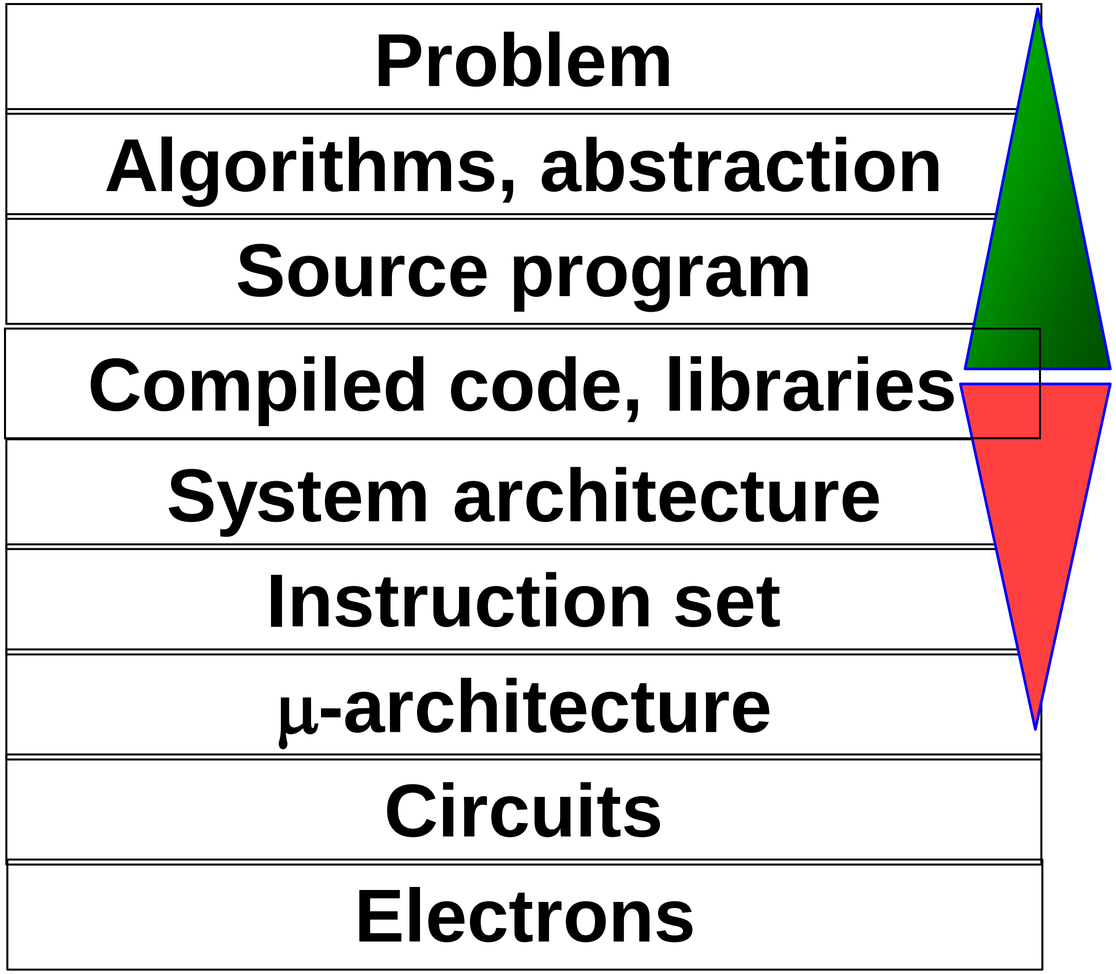
- **A compiler (or an interpreter) <u>transforms</u> the high-level code to machine-level code**

- **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 layers!)



Problem

Algorithms, abstraction

Source program

Compiled code, libraries

System architecture

Instruction set

μ-architecture

Circuits

Electrons

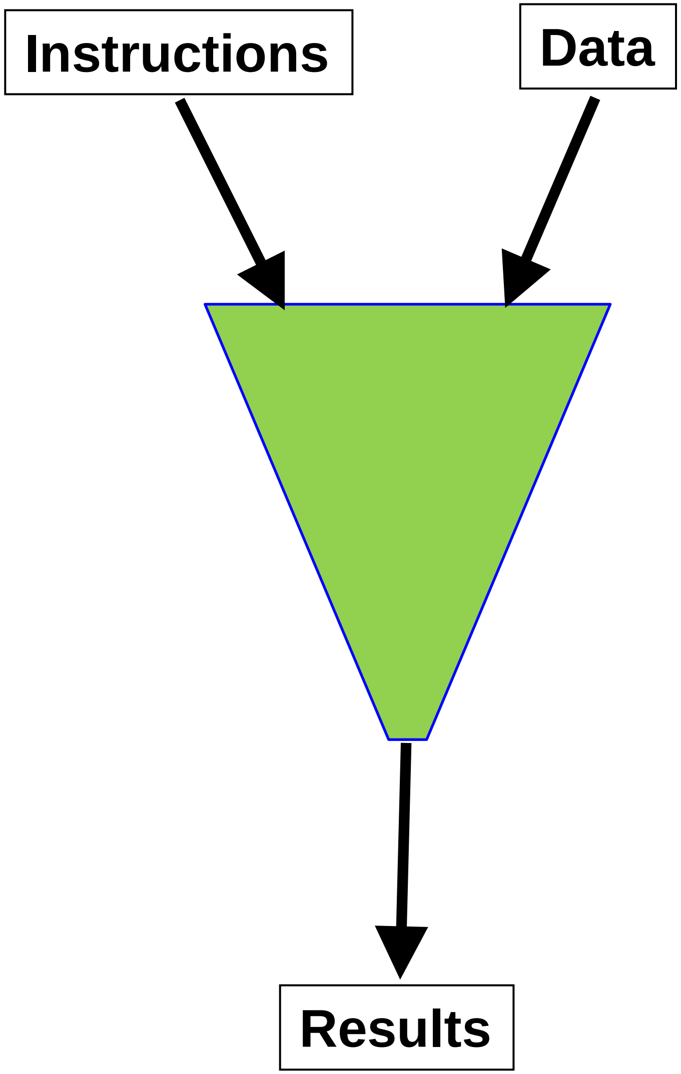
- **We must avoid being fenced into a single layer!**

Adapted from Y.Patt, U-Austin

Let’s start with the basics!

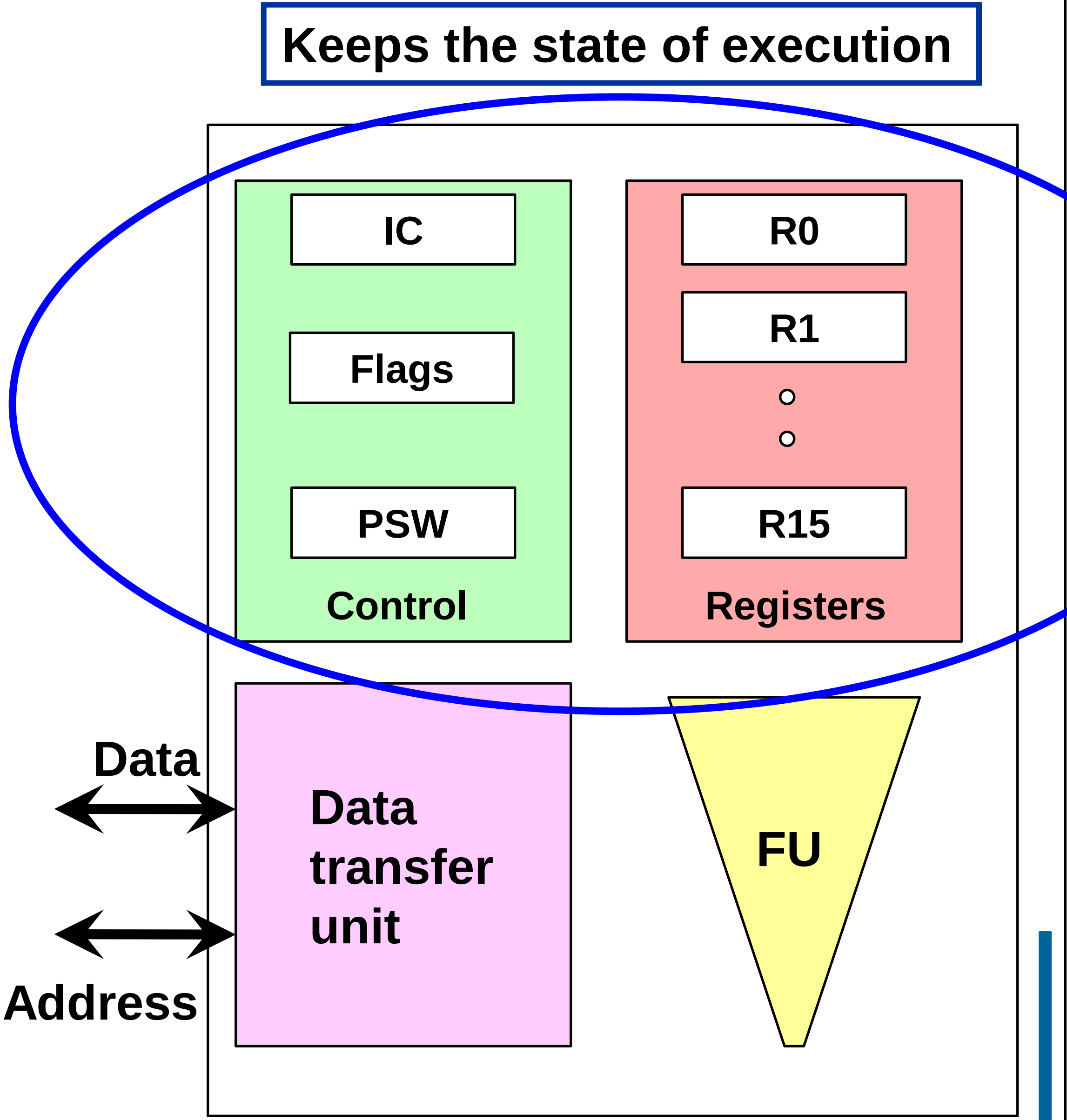
Von Neumann architecture

- **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**
- **Our goal is to produce results as fast as possible**



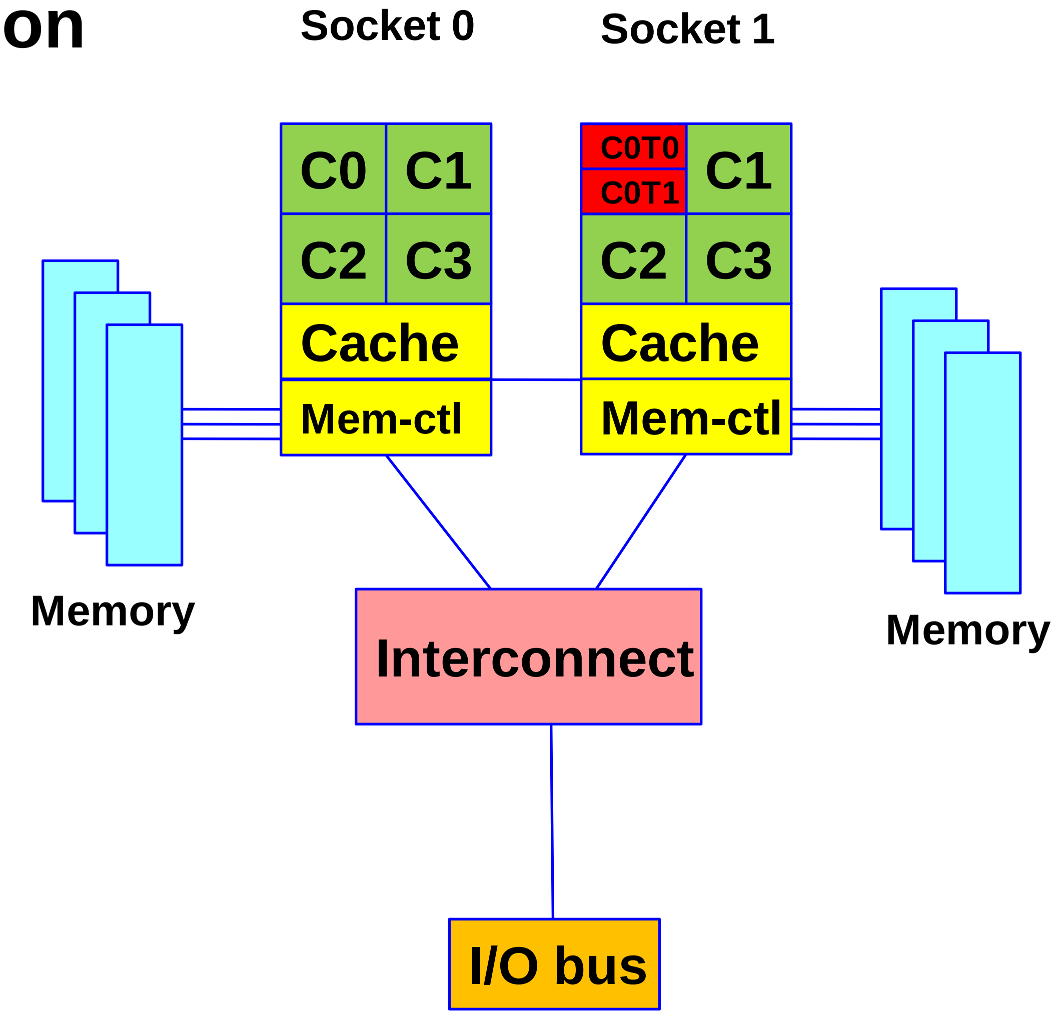
Simple processor layout

- **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



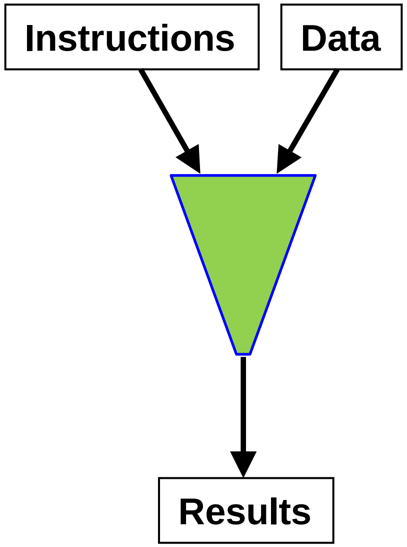
Simple server diagram

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



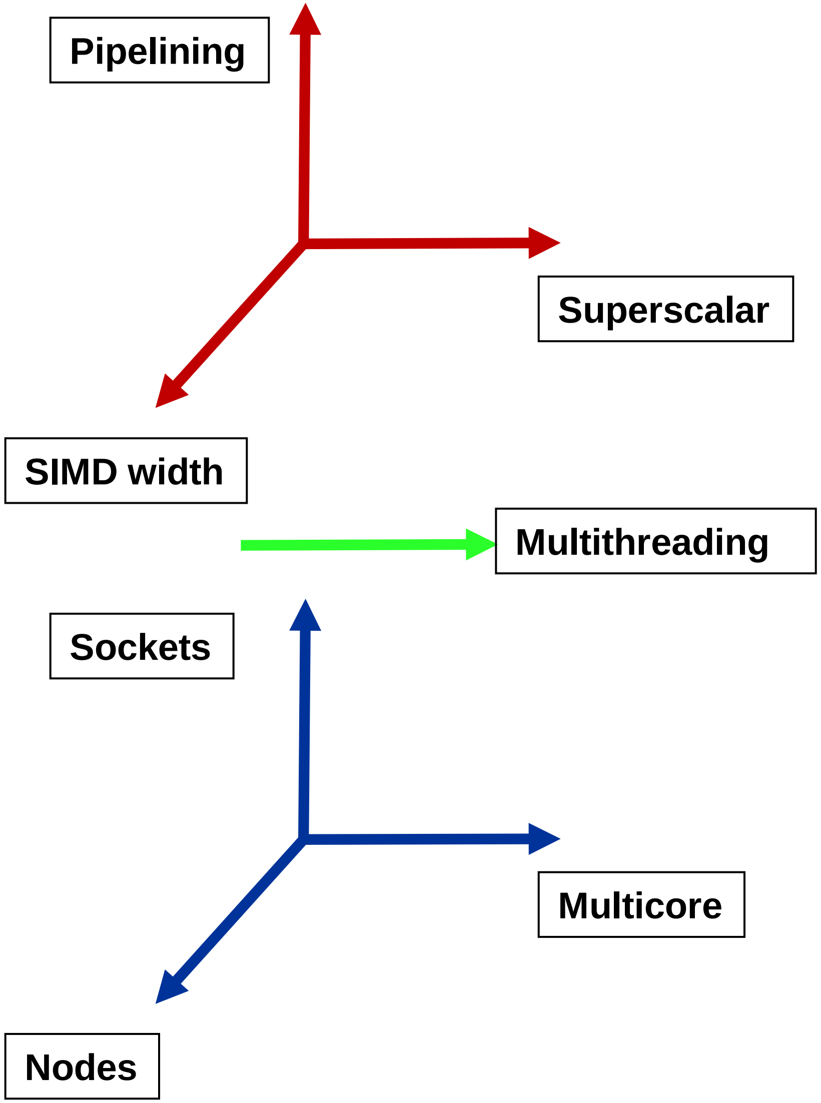
Initial premise

- **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: $(6 * 2.0 * 10^9) / 10^{10} = 1.2$



Seven dimensions of performance

- **First three dimensions:**
  - Superscalar
  - Pipelining
  - Computational width/SIMD
- **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:**
  - Superscalar
  - Pipelining
  - Computational width/SIMD

Data parallelism (Vectors/Scalars)

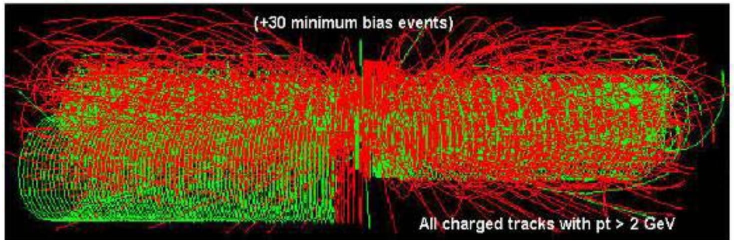
- **Next dimension is a “pseudo” dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes

Task parallelism (Events/Tracks)

Task/process parallelism

Concurrency in HEP

- **We are “blessed” with lots of it:**
  - Entire events
  - Particles, 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?**
- **Intel compiler (10.1 or later):**
  - Autovectorization: YES, included in “-O”
    - “-vec-reportN” for reports
  - Autoparallelization: YES with “-parallel”
    - “-par-reportN” for reports
- **GNU compiler (4.3.0 or later):**
  - Autovectorization: YES, but needs “-ftree-vectorize”
    - “-ftree-vectorizer-verbose=[0-7]” for reports
  - Autoparallelization support in preparation
    - OpenMP support available

In addition, both compilers support intrinsics:
“higher-level assembly instructions” for explicit vectorization

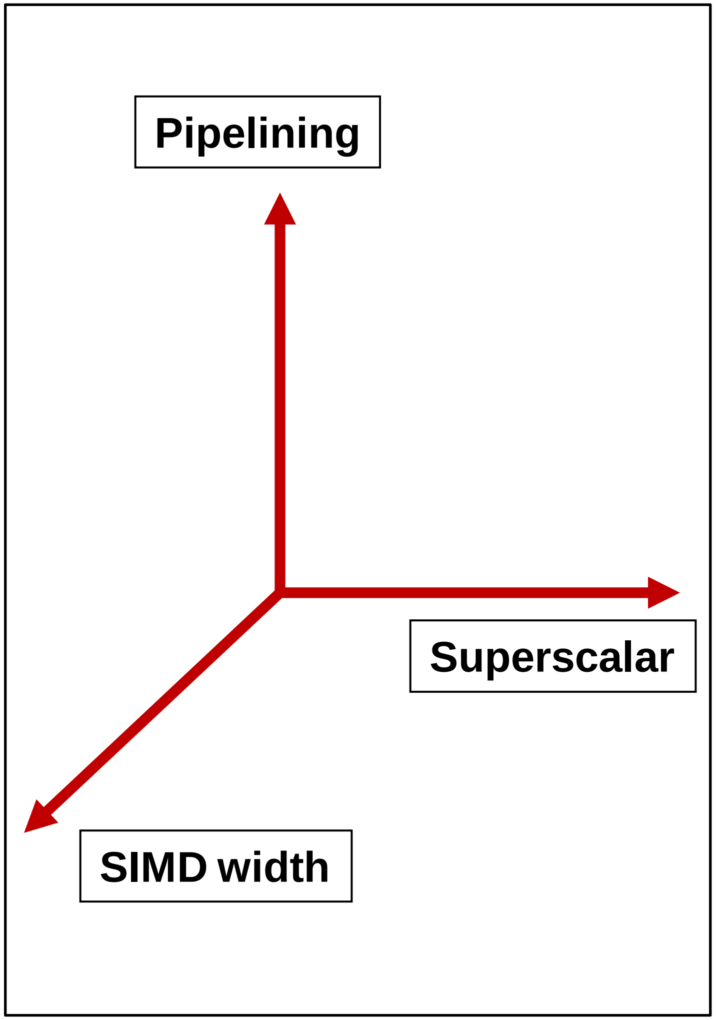
Part 1: Opportunities for scaling performance inside a core

- **Here are the first three dimensions**

- **The resources:**
  - Superscalar: Fill the ports
  - Pipelined: Fill the stages
  - SIMD: Fill the computational width

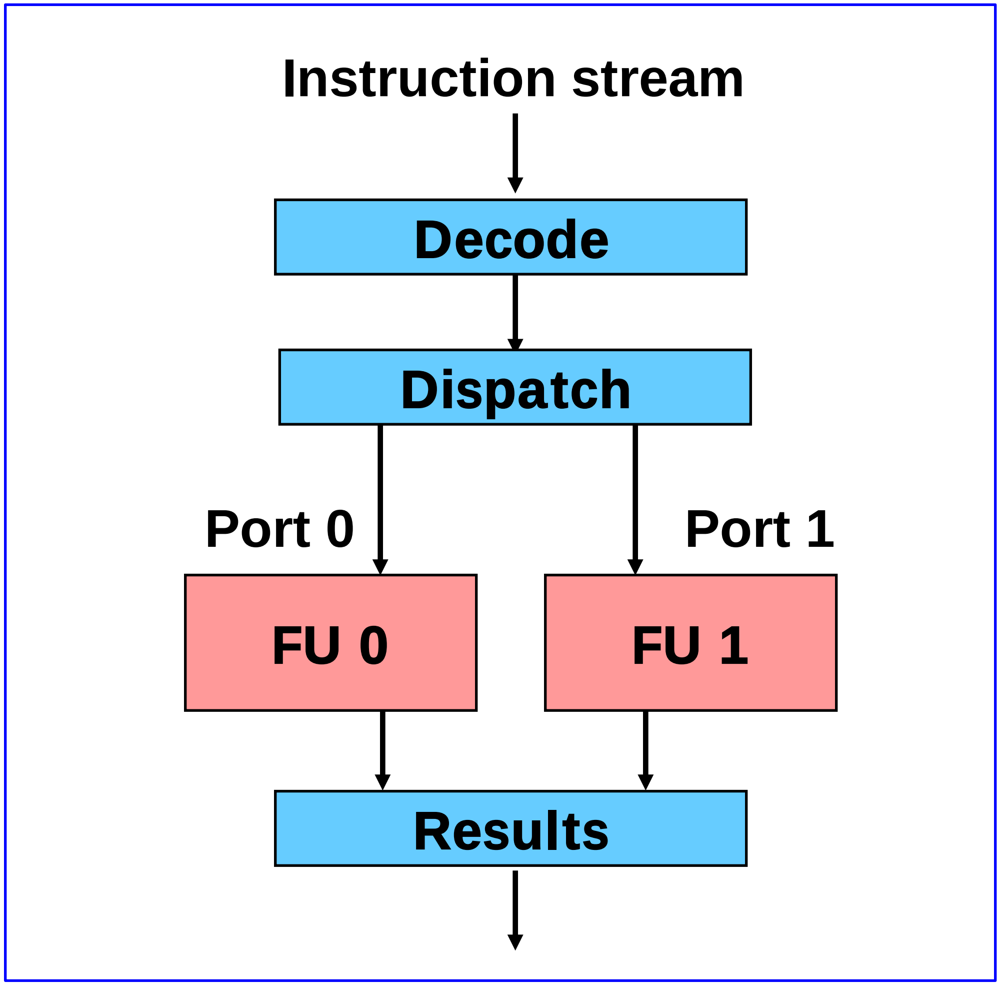
- **Best approach: data parallelism**

- **In HEP, we probably extract only 10-15% of peak execution capability!**



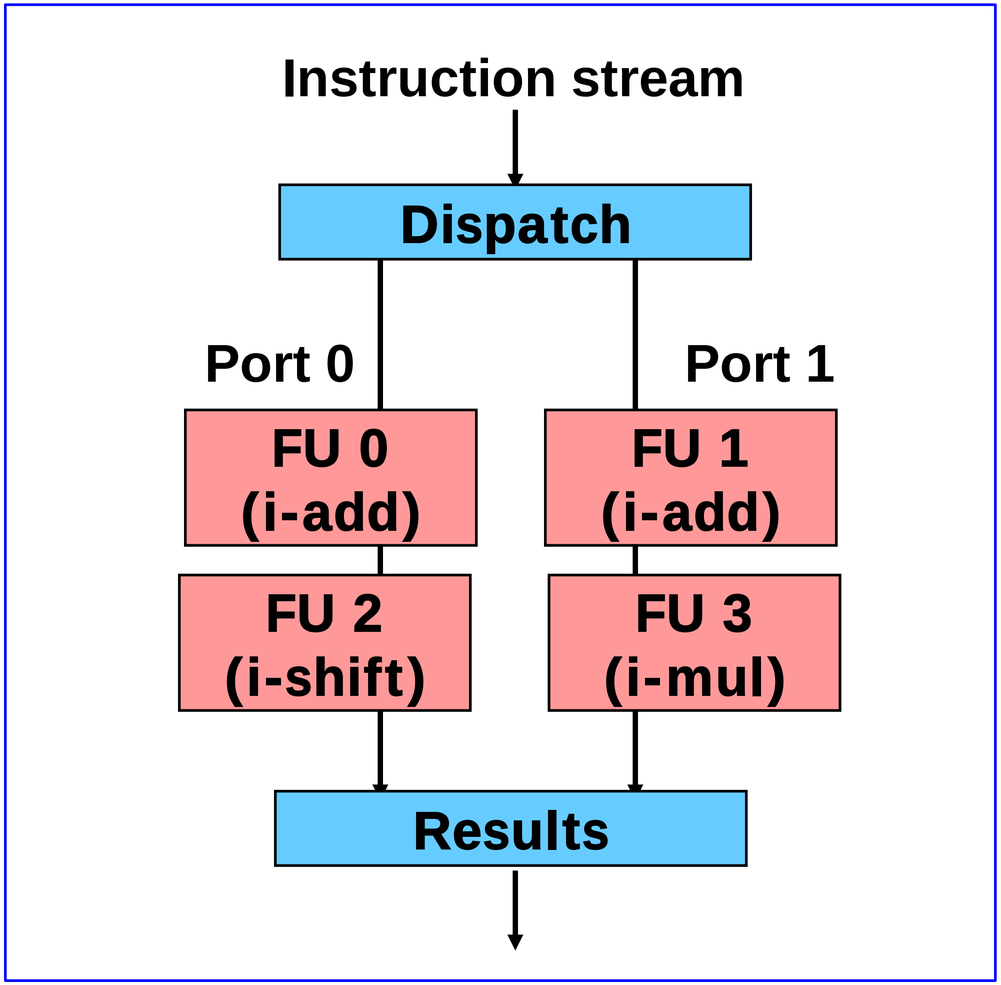
First: Superscalar architecture

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



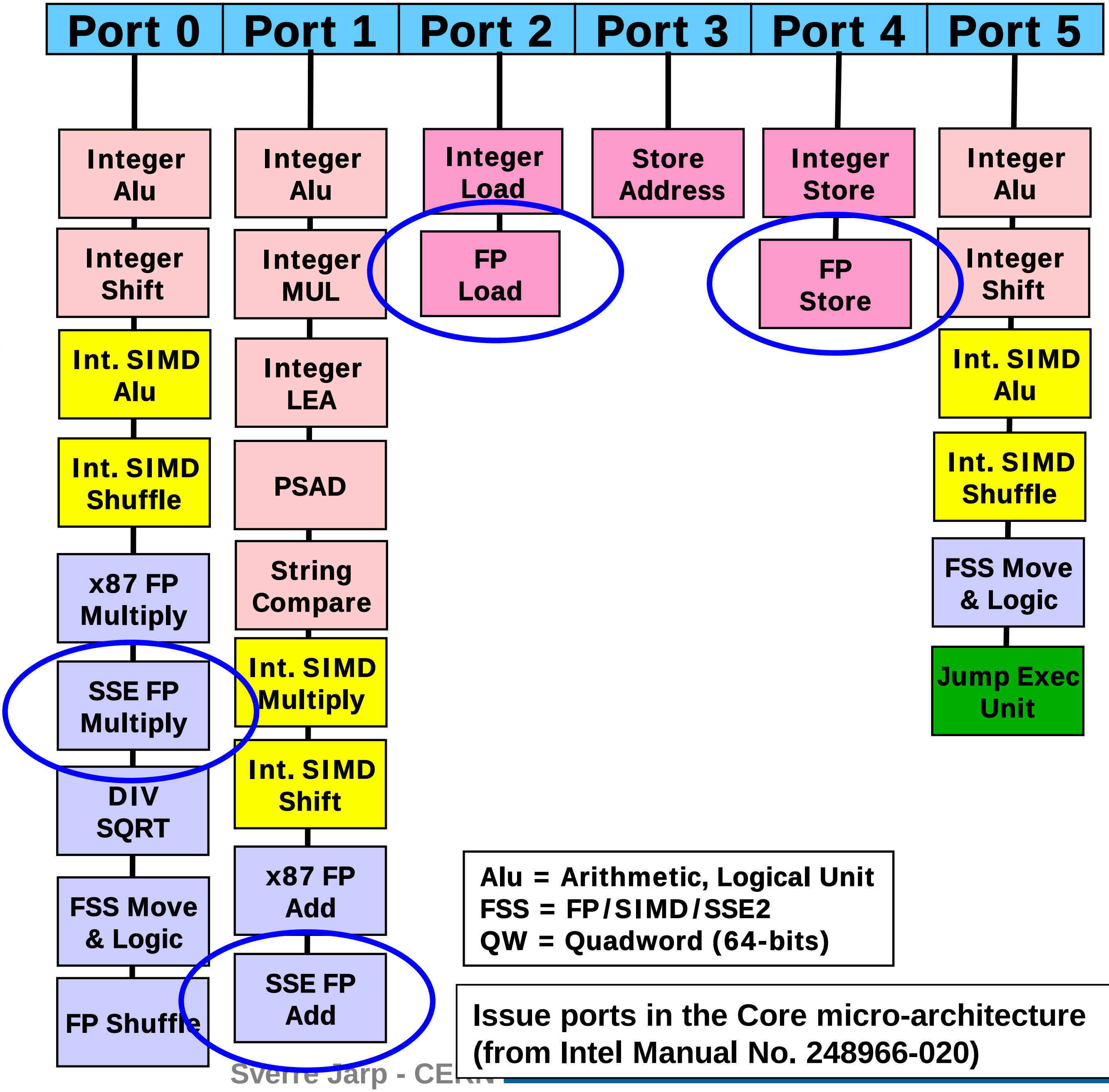
Enhanced superscalar architecture

- **A more realistic architecture will have multiple FUs hanging off the same port**
  - An instruction can be dispatched to either matching execution unit on a given port, but not to both units on the same port in a given cycle



Today's superscalar architecture

- **For instance, Intel's Nehalem microarchitecture can dispatch/execute/ retire four instructions in parallel (across six ports) in each cycle:**



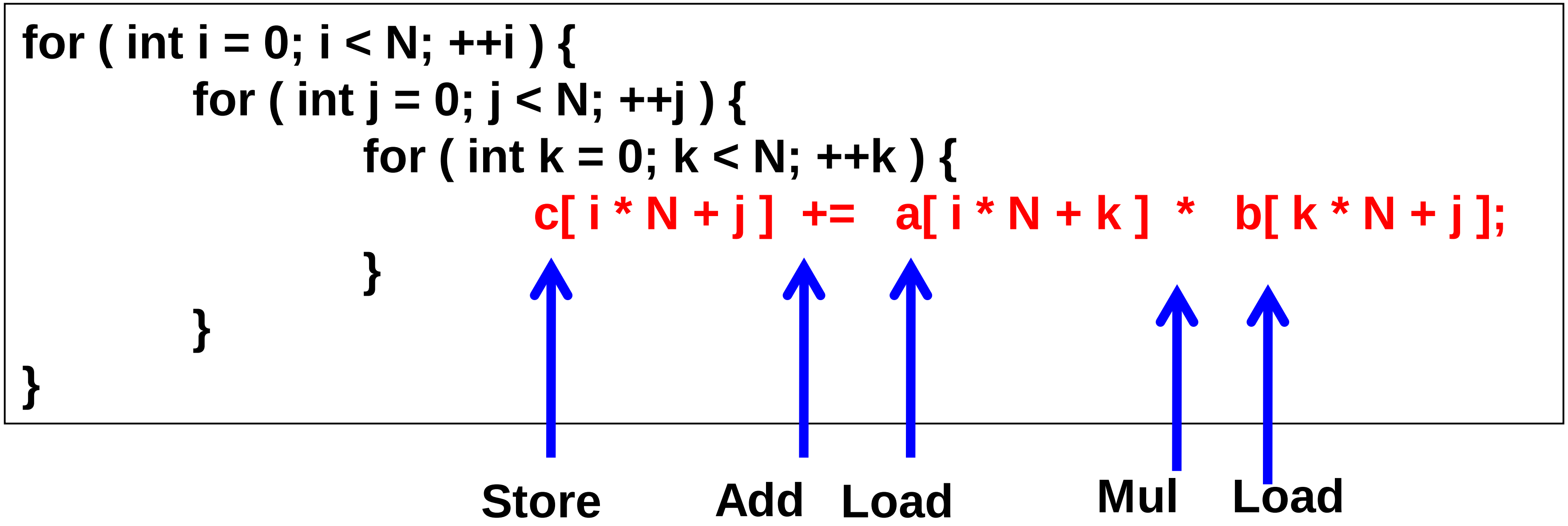
Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
Integer Alu	Integer Alu	Integer Load	Store Address	Integer Store	Integer Alu
Integer Shift	Integer MUL	FP Load		FP Store	Integer Shift
Int. SIMD Alu	Integer LEA				Int. SIMD Alu
Int. SIMD Shuffle	PSAD				Int. SIMD Shuffle
x87 FP Multiply	String Compare				FSS Move & Logic
SSE FP Multiply	Int. SIMD Multiply				Jump Exec Unit
DIV SQRT	Int. SIMD Shift				
FSS Move & Logic	x87 FP Add				
FP Shuffle	SSE FP Add				

Alu = Arithmetic, Logical Unit
FSS = FP/SIMD/SSE2
QW = Quadword (64-bits)

Issue ports in the Core micro-architecture (from Intel Manual No. 248966-020)

Mulmul example

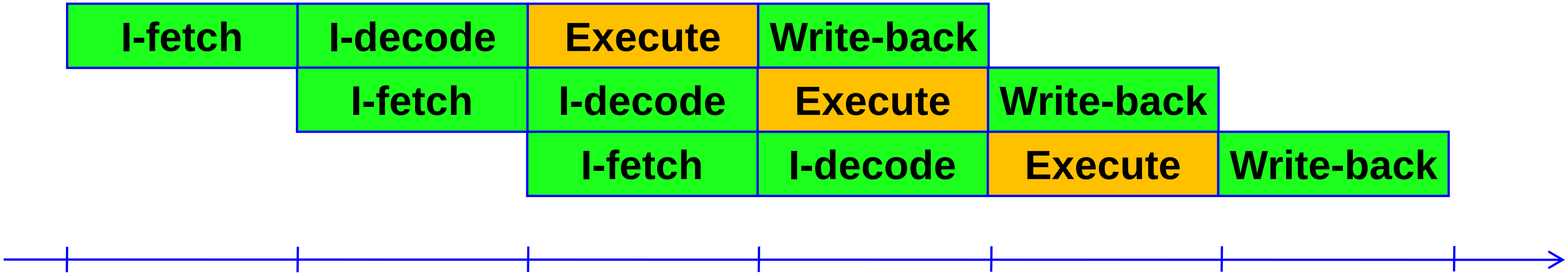
- **For a given algorithm, we can understand exactly which functional execution units are needed**
  - For instance, in the innermost loop of matrix multiplication



```
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 ];
                }
        }
}
```

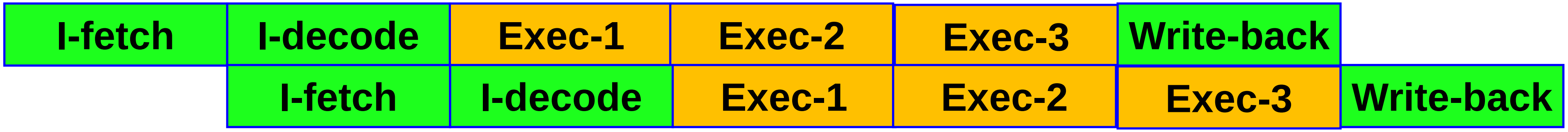
Next topic: Instruction pipelining

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



I-fetch | I-decode | Execute | Write-back
I-fetch | I-decode | Execute | Write-back
I-fetch | I-decode | Execute | Write-back

  - With a three-cycle execution latency:



I-fetch | I-decode | Exec-1 | Exec-2 | Exec-3 | Write-back
I-fetch | I-decode | Exec-1 | Exec-2 | Exec-3 | Write-back

Real-life latencies

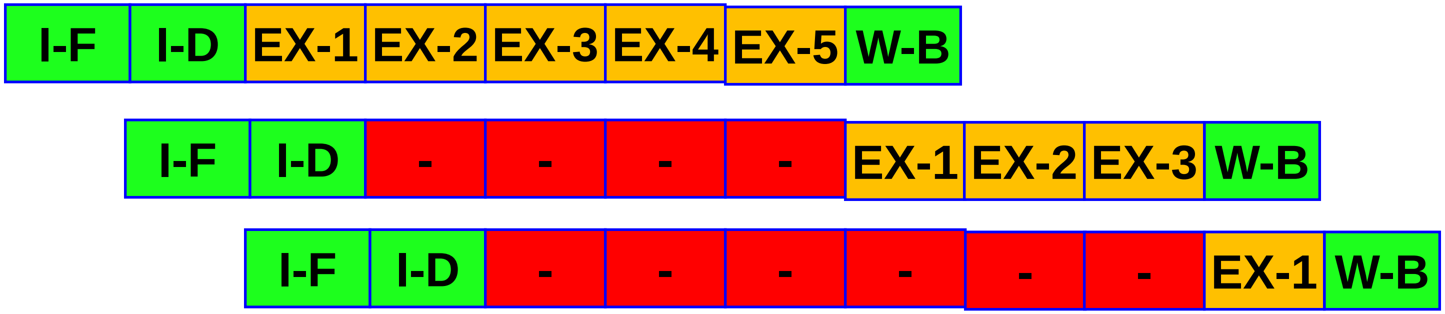
- **Most integer/logic instructions have a one-cycle execution latency:**
  - For example:
    - ADD, AND, SHL (shift left), ROR (rotate right)
  - Amongst the exceptions:
    - IMUL (integer multiply): 3
    - IDIV (integer divide): 13 – 23
- **Floating-point latencies are typically multi-cycle**
 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

Latencies in the Core micro-architecture (Intel Manual No. 248966-020 or later). AMD processor latencies are similar.

Latencies and serial code (1)

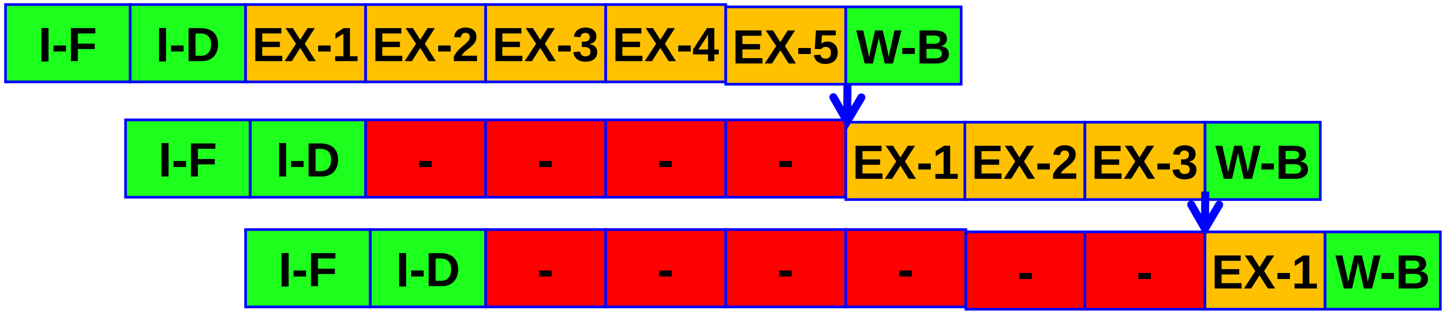
- **In serial programs, we typically pay the penalty of a multi-cycle latency during execution:**
  - In this example:
    - Statement 2 cannot be started before statement 1 has finished
    - Statement 3 cannot be started before statement 2 has finished

```
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
```



I-F	I-D	EX-1	EX-2	EX-3	EX-4	EX-5	W-B				
	I-F	I-D	-	-	-	-	EX-1	EX-2	EX-3	W-B	
		I-F	I-D	-	-	-	-	-	-	EX-1	W-B

Latencies and serial code (2)



I-F	I-D	EX-1	EX-2	EX-3	EX-4	EX-5	W-B				
	I-F	I-D	-	-	-	-	EX-1	EX-2	EX-3	W-B	
		I-F	I-D	-	-	-	-	-	-	EX-1	W-B

- **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 here:
    - 9 execution cycles are needed for three instructions!
      - CPI is equal to 3

Mini-example of real-life 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
```

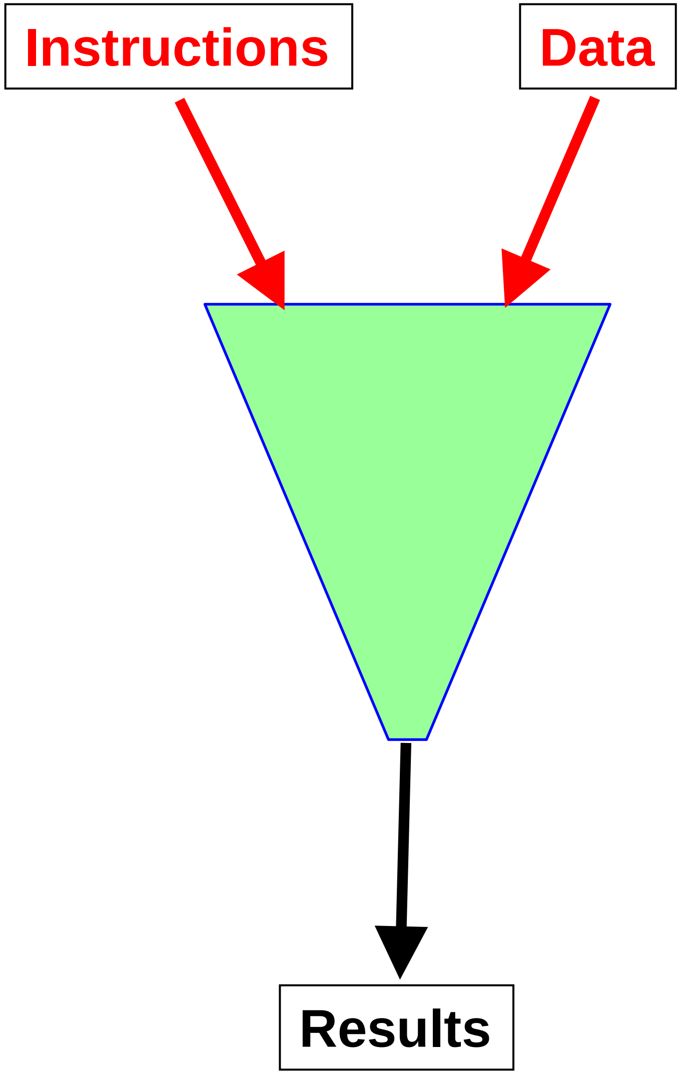
Same instructions laid out according to latencies on the Core 2 processor →

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

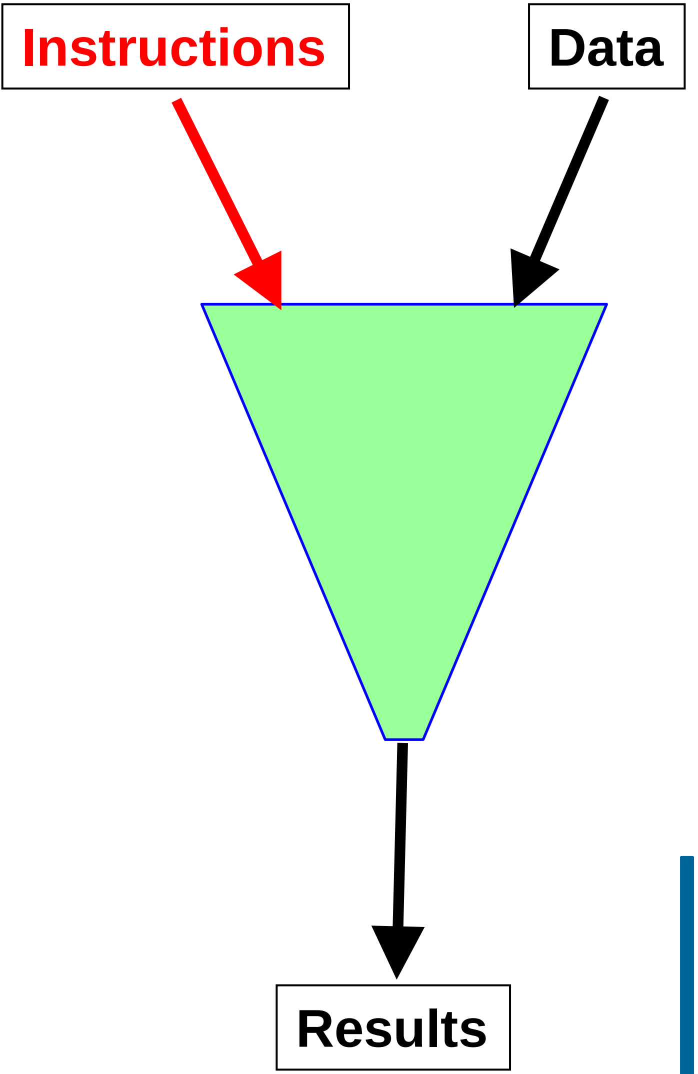
Other causes of execution delays (1)

- **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



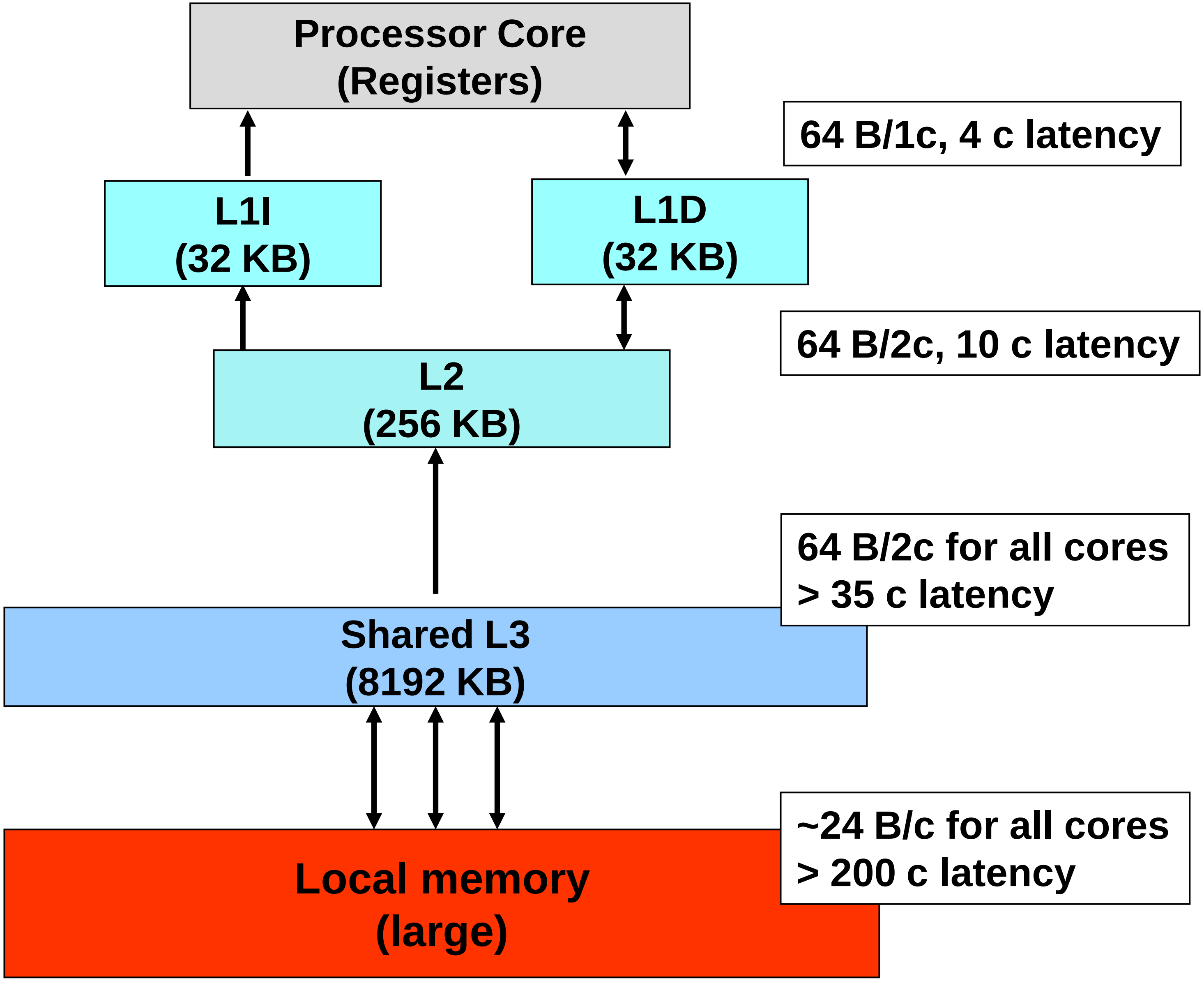
Other causes of execution delays (2)

- **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



Memory Hierarchy

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



Processor Core (Registers)

L1I (32 KB)

L1D (32 KB)

L2 (256 KB)

Shared L3 (8192 KB)

Local memory (large)

64 B/1c, 4 c latency

64 B/2c, 10 c latency

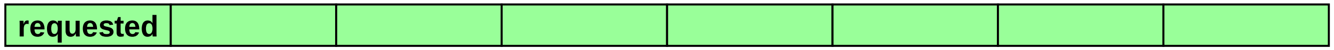
64 B/2c for all cores
> 35 c latency

~24 B/c for all cores
> 200 c latency

c = cycle

Cache lines (1)

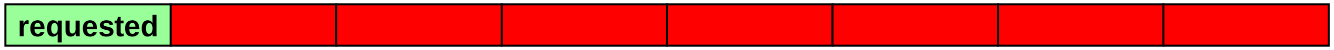
- **When a data element or an instruction is requested by the processor, a cache line is ALWAYS moved (as the minimum quantity) to Level-1**


requested

- **Cache lines are typically 64B (8 * double)**
  - A 32KB level-1 cache holds 512 (64B) lines
- **When cache lines have to be moved come from memory**
  - Latency is long (>150 cycles, as already mentioned)
    - It is even longer if the memory is remote
  - Memory controller stays busy (~8-10 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 arrays should be accessed with the last index changing fastest:**

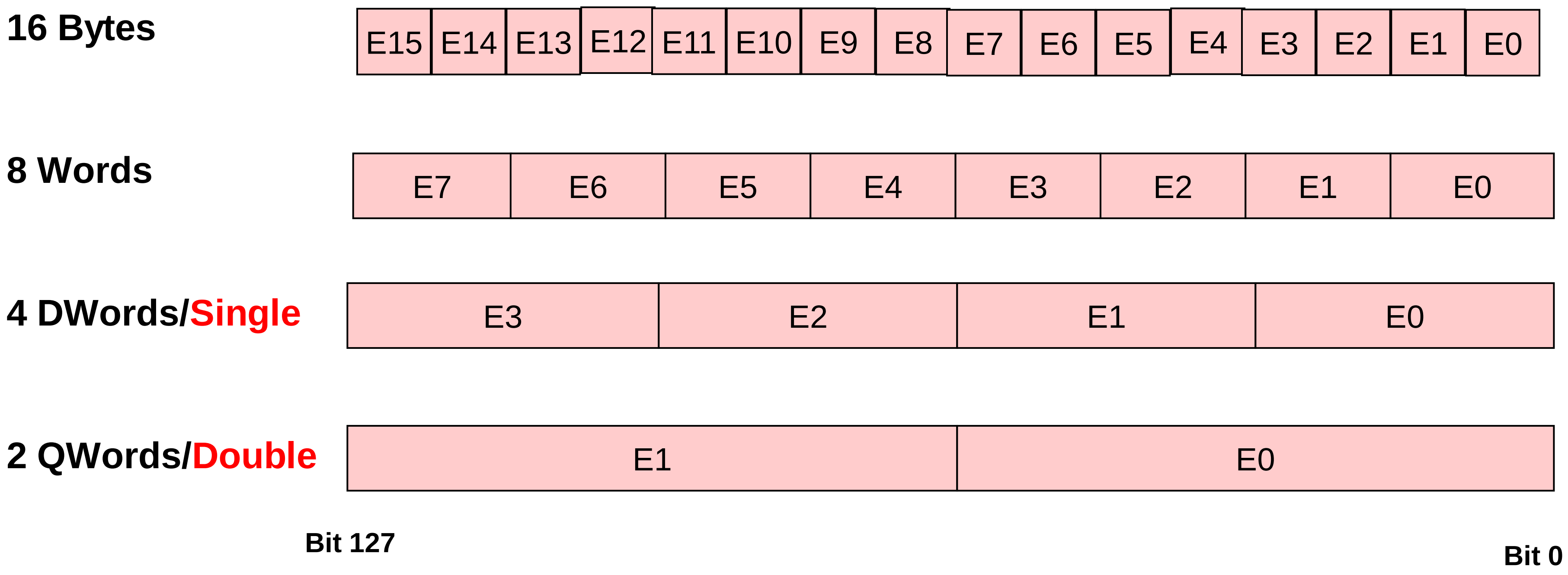
```
for (i = 0; i < rows; ++i)
          for (j = 0; j < columns; ++j)
                    mymatrix [i] [j]   += increment;
```

- **Pointer chasing (in linked lists) can easily lead to cache thrashing**

Programming the memory hierarchy is an art in itself.

Third topic: Registers for SSE

- **16 “XMM” registers with 128 bits each in 64-bit mode**



16 Bytes
E15 E14 E13 E12 E11 E10 E9 E8 E7 E6 E5 E4 E3 E2 E1 E0

8 Words
E7 E6 E5 E4 E3 E2 E1 E0

4 DWords/Single
E3 E2 E1 E0

2 QWords/Double
E1 E0

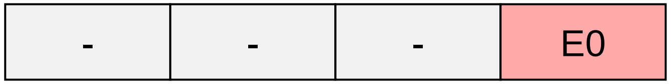
Bit 127

Bit 0

SSE = Streaming SIMD extensions

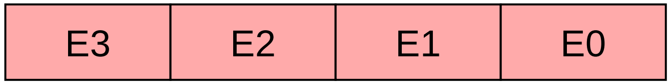
Four floating-point data flavours

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



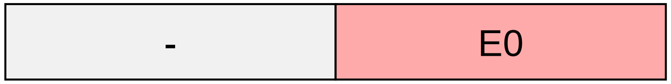
-	-	-	E0

  - Packed single (PS)



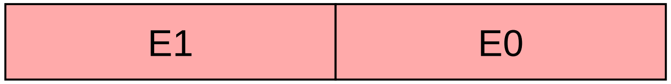
E3	E2	E1	E0

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



-	E0

  - Packed Double (PD)



E1	E0

- **Note:**
  - 1) Today, “scalar” means running at ½ or ¼ of the peak speed
  - 2) Intel and AMD have announced Advanced Vector eXtensions (AVX) with 256-bit registers (available next year !)
    - “scalar” will mean 1/4 or 1/8 of peak!
  - 3) even longer vectors are coming!

Scalable programming for a single core

- **Easiest way to fill the execution capabilities is to use vectorization**
  - Either, vector syntax, à la Fortran-90
  - Or, loop syntax which the compiler can “vectorize” automatically
  - Or, explicit *intrinsics*
    - See CBM example later.

```
REAL U(100), V(100)

U = 0.0

U = SIN(V)

U(1:50) = V(2:100:2)
```

```
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];
```

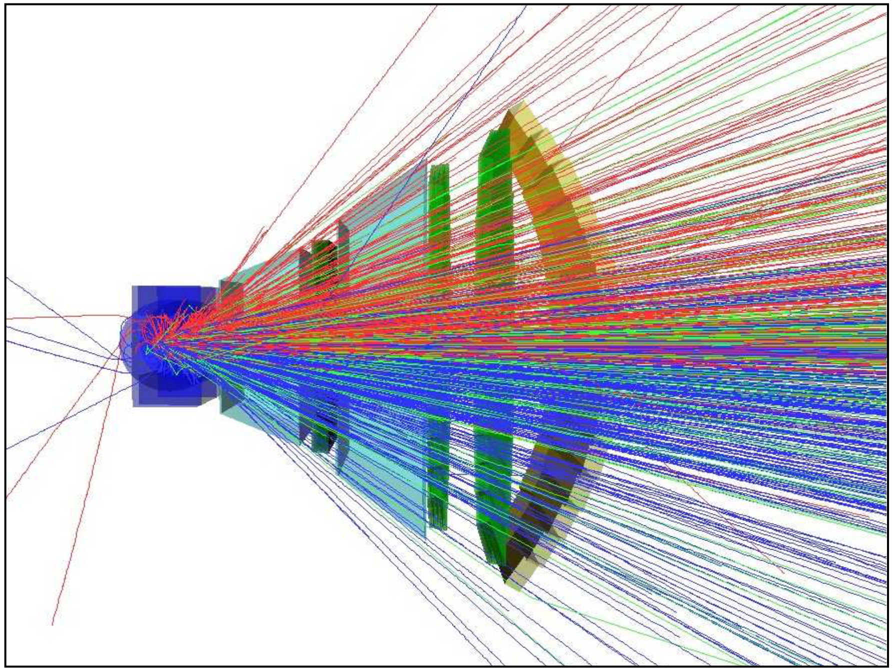
HEP and vectors

- **Too little common ground**
  - And, practically all attempts in the past failed!
    - w/Cyber-205, CRAY, IBM 3090-Vector Facility, etc.
- **From time to time, we see a good vector example**
  - For example: Track Fitting code from ALICE trigger
    - → See the next slide
- **Interesting development from ALICE (Matthias Kretz):**
  - Vc (Vector Classes)
    - http://www.kip.uni-heidelberg.de/~mkretz/Vc/
- **Other examples: Use of STL vectors; small matrices**

Examples of parallelism: CBM/ALICE track fitting

- **Extracted from their 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

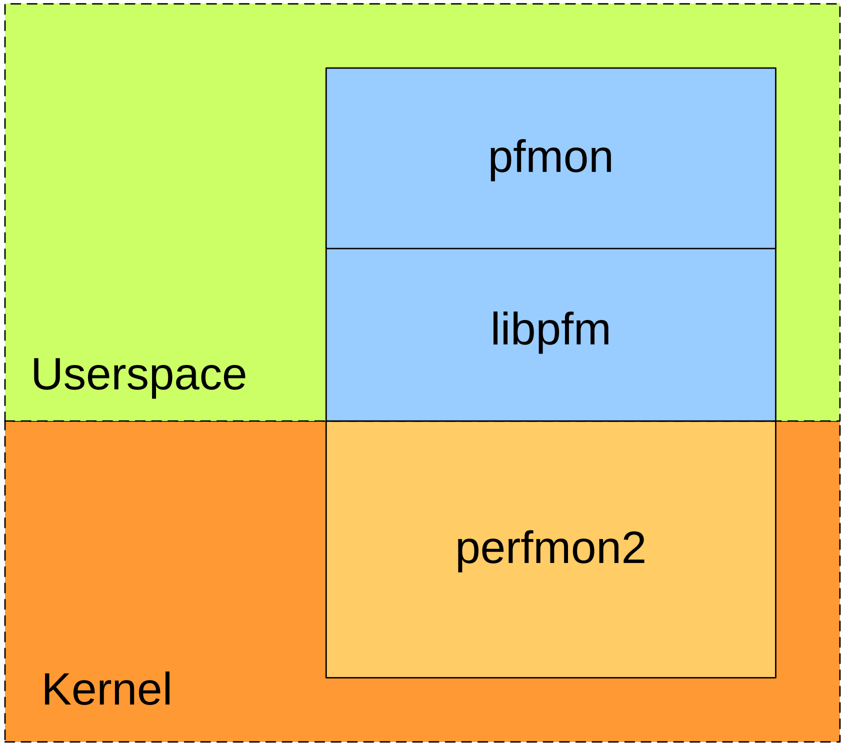
- **Re-optimization on x86-64 systems**
  - First: use SSE vectors instead of scalars
    - Operator overloading allows seamless change of data types
    - Intrinsics (from Intel/GNU header file): Map directly to instructions:
      - __mm_add_ps corresponds directly to ADDPS, the instruction that operates 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)

Performance monitoring in hardware

- **Most modern CPUs are able to provide real-time statistics concerning executed instructions..**
  - Via a Performance Monitoring Unit (PMU)
- **The PMU is observing your application in real-time!**
  - And everything else that uses the CPU
- **Limited number of counters (sentries) available**
  - But they are quite versatile
- **Recorded occurences are called events**
- **On the Core i7 (Nehalem):**
  - 4 universal counter: #0, #1, #2, #3
  - 3 specialised counters: #16, #17, #18
  - Eight “uncore” counters: #20 - #27

Pfmon overview

- **Console-based interface to libpfm/perfmon2**
- **Provides convenient access to performance counters**
- **Wide range of functionality**
  - Counting events
  - Sampling in regular intervals
  - Flat profile
  - System-wide mode
  - Triggers
  - Different data read-out “plug-in” modules available



Events

- **Many events in the CPU can be monitored**
  - A comprehensive list is dependent on the CPU and can be extracted from the manufacturers’ manuals or from relevant tools
- **On some CPUs (i.e. Intel Core), some events have bit-masks which limit their range**
  - “unit masks” or “umasks”
    - Example: instructions retired: all / loads only / stores only
- **In pfmon:**
  - Getting a list of supported events: *pfmon –l*
  - Getting information about an event: *pfmon –i eventname*

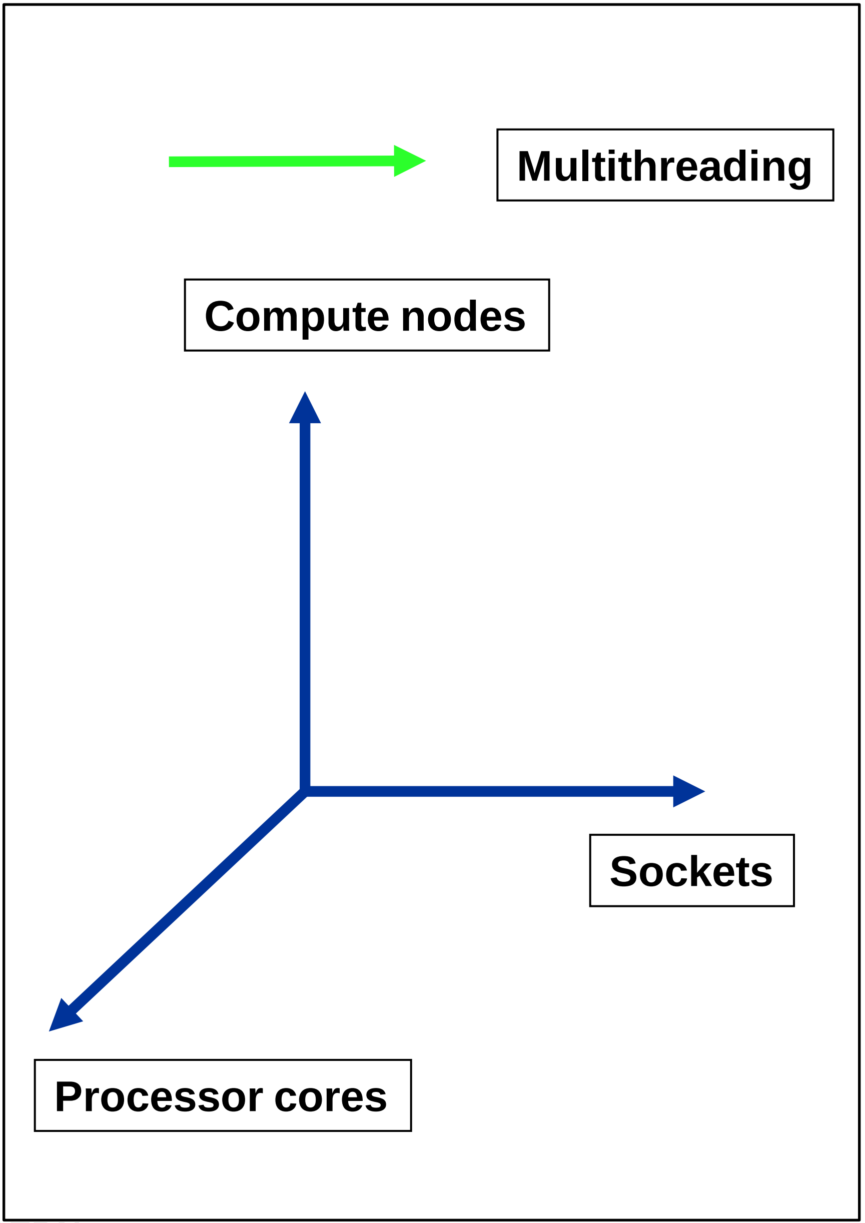
Important performance counters
(that can tell you if things go wrong)

- **Related to what we have discussed:**
  - The total cycle count (C)
  - The total instruction count (I)
  - Derived value: CPI
  - Bubble/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 instructions
  - The total number of SSE instructions

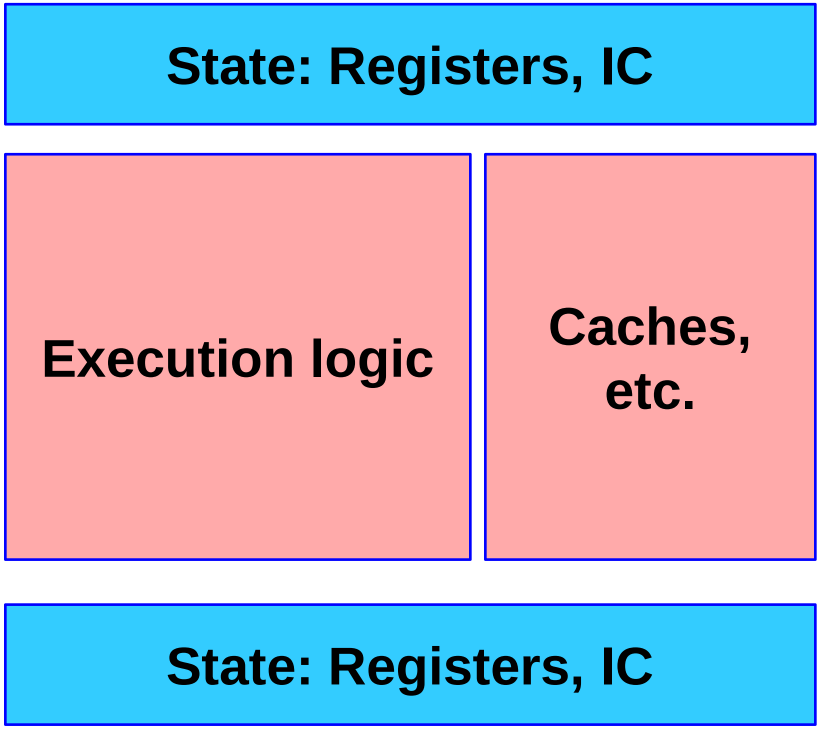
Part 2: Parallel execution across hw-threads and cores

- **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



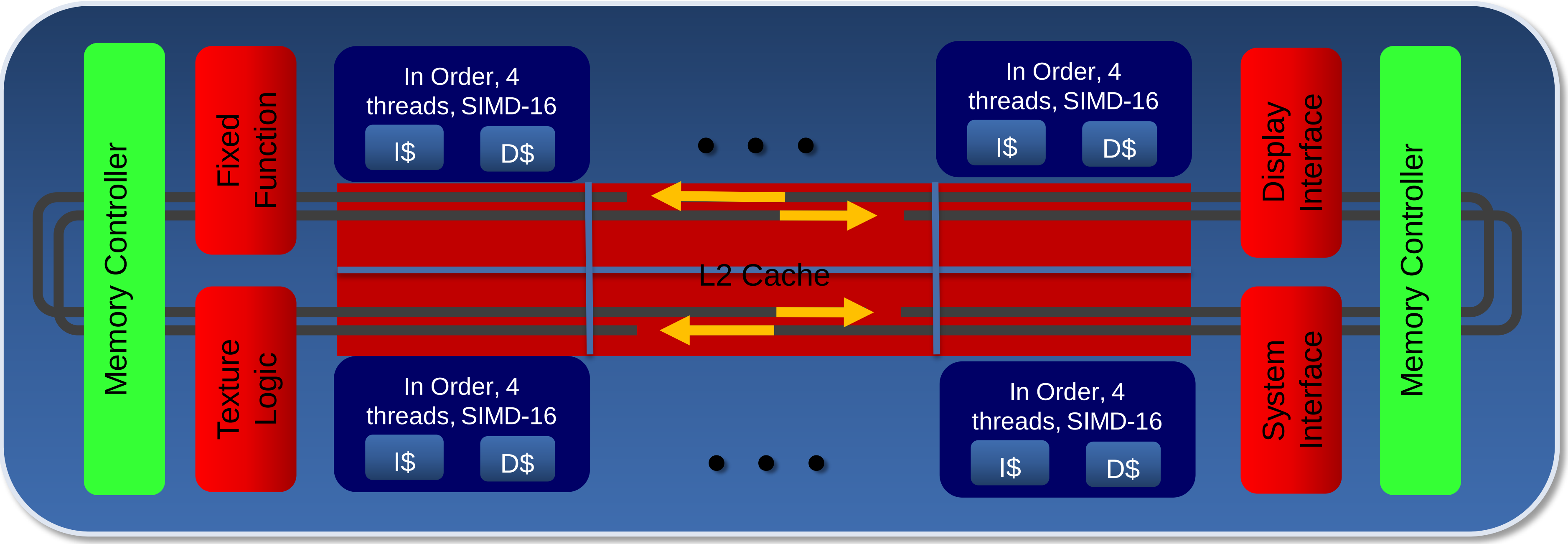
The sharing of the execution logic can be coarse-grained or fine-grained.

The move to many-core systems

- **Examples of “dispatch 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): 2 * 6 * 2 = 24
  - Aggressive:
    - Quad-socket AMD Magny-Cours (12-core) 4 * 12 * 1 = 48
    - Quad-socket Nehalem-EX “octo-core”: 4 * 8 * 2 = 64
- **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 MIC

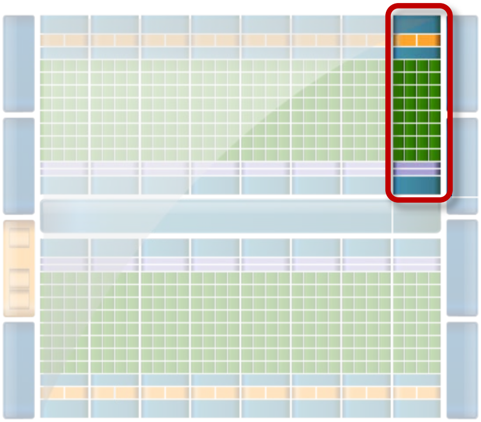
- **Many Integrated Core architecture:**
  - Announced at ISC10 (June 2010)
  - Based on the x86 architecture, 22nm (in 2012?)
  - Many-core (> 50 cores) + 4-way multithreaded + 512-bit vector unit
  - Limited memory: A few Gigabytes



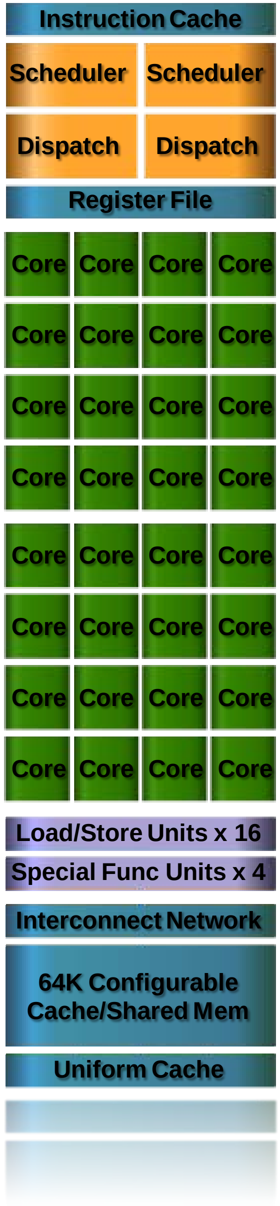
Accelerators (2): Nvidia Fermi GPU

- **Streaming Multiprocessing (SM) Architecture**

- **32 “CUDA cores” per SM (512 total)**
- **Peak single precision floating point performance (at 1.15 GHz”:**
  - **Above 1 Tflop**
- **Double-precision: 50%**
- **Dual Thread Scheduler**
- **64 KB of RAM for shared memory and L1 cache (configurable)**
- **A few Gigabytes of main memory**



Lots of interest in the HEP on-line community



Adapted from Nvidia

Definition of a software process and thread

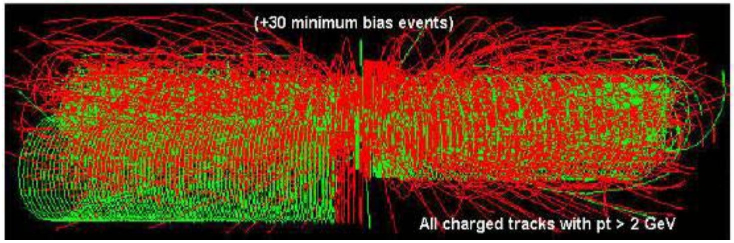
- **Process (OS process):**
  - An instance of a computer program that is being executed (sequentially). It typically runs as a program with its private set of operating system resources, i.e. in its own "address space" with all the program code and data, its own file descriptors with the operating system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

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!**

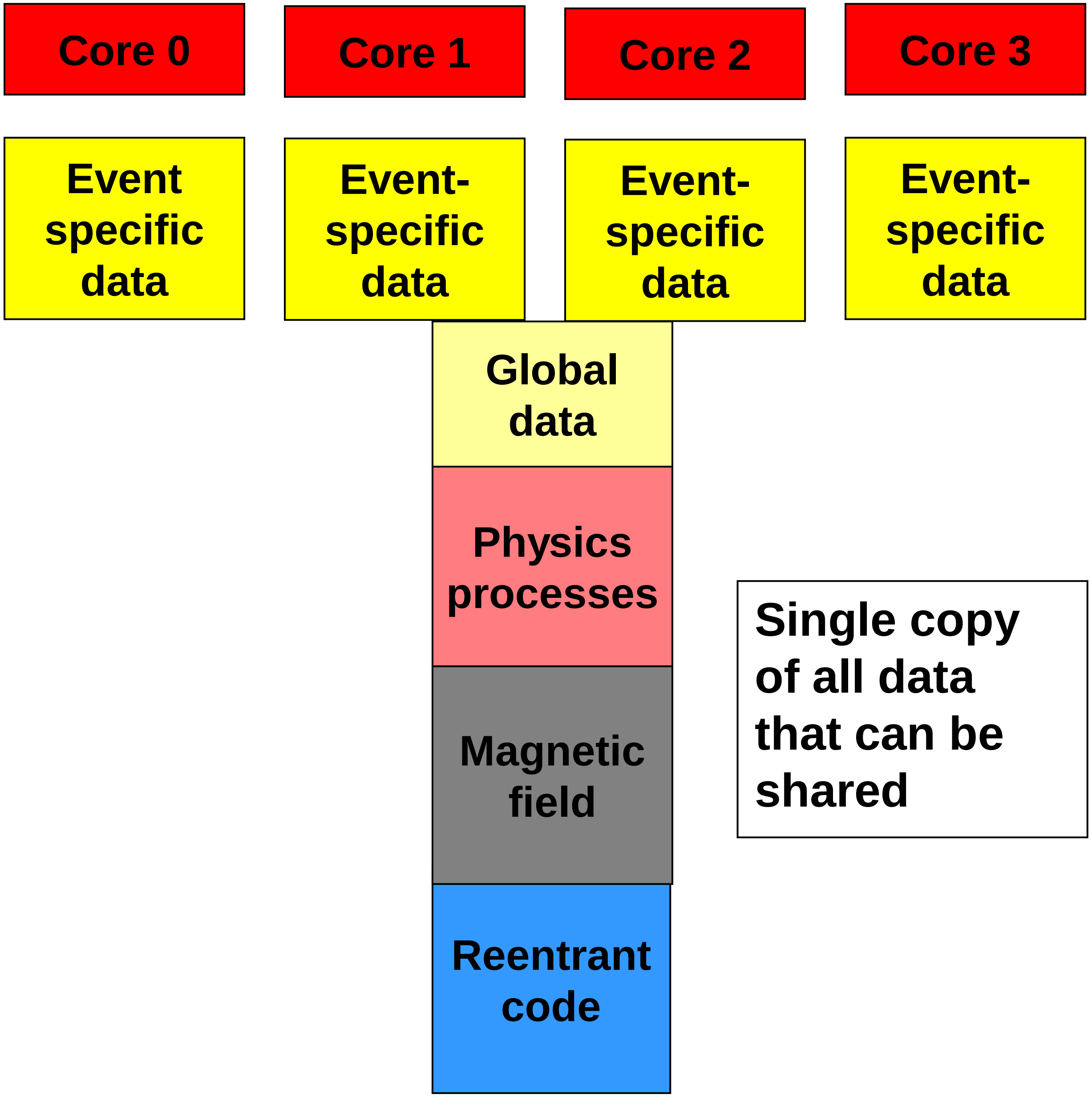
What are the multi-core options?

- **There is currently a discussion in the community about the best way forward:**

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; Run through the first "event"
   - Fork N other processes
   - 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

Achieving efficient memory footprint

- As follows:



Core 0

Core 1

Core 2

Core 3

Event specific data

Event-specific data

Event-specific data

Event-specific data

Global data

Physics processes

Magnetic field

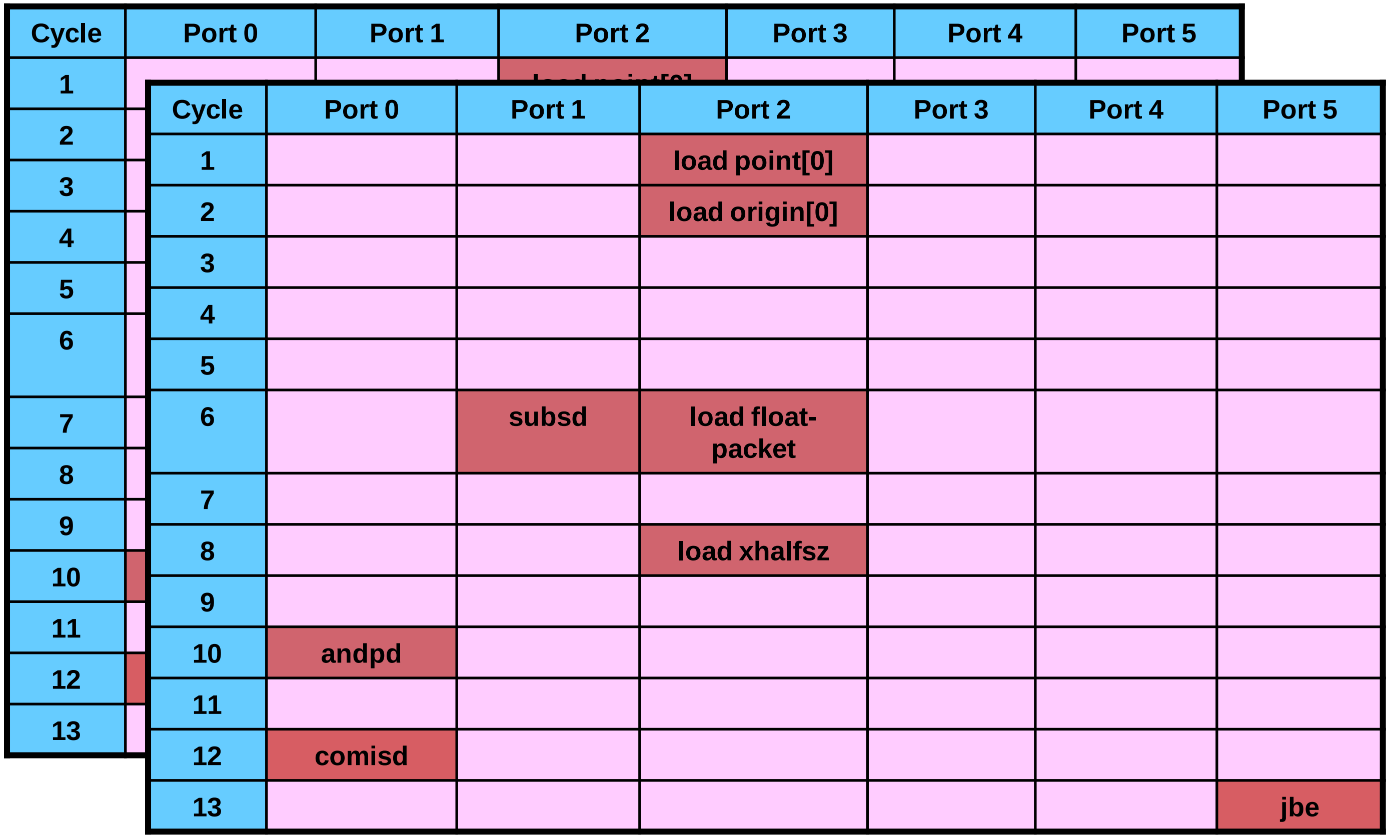
Reentrant code

Single copy of all data that can be shared

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 5600, we currently get max 25% from the second hardware thread !



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

Let’s look more closely at parallelism

From Concurrency to Parallel Execution

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

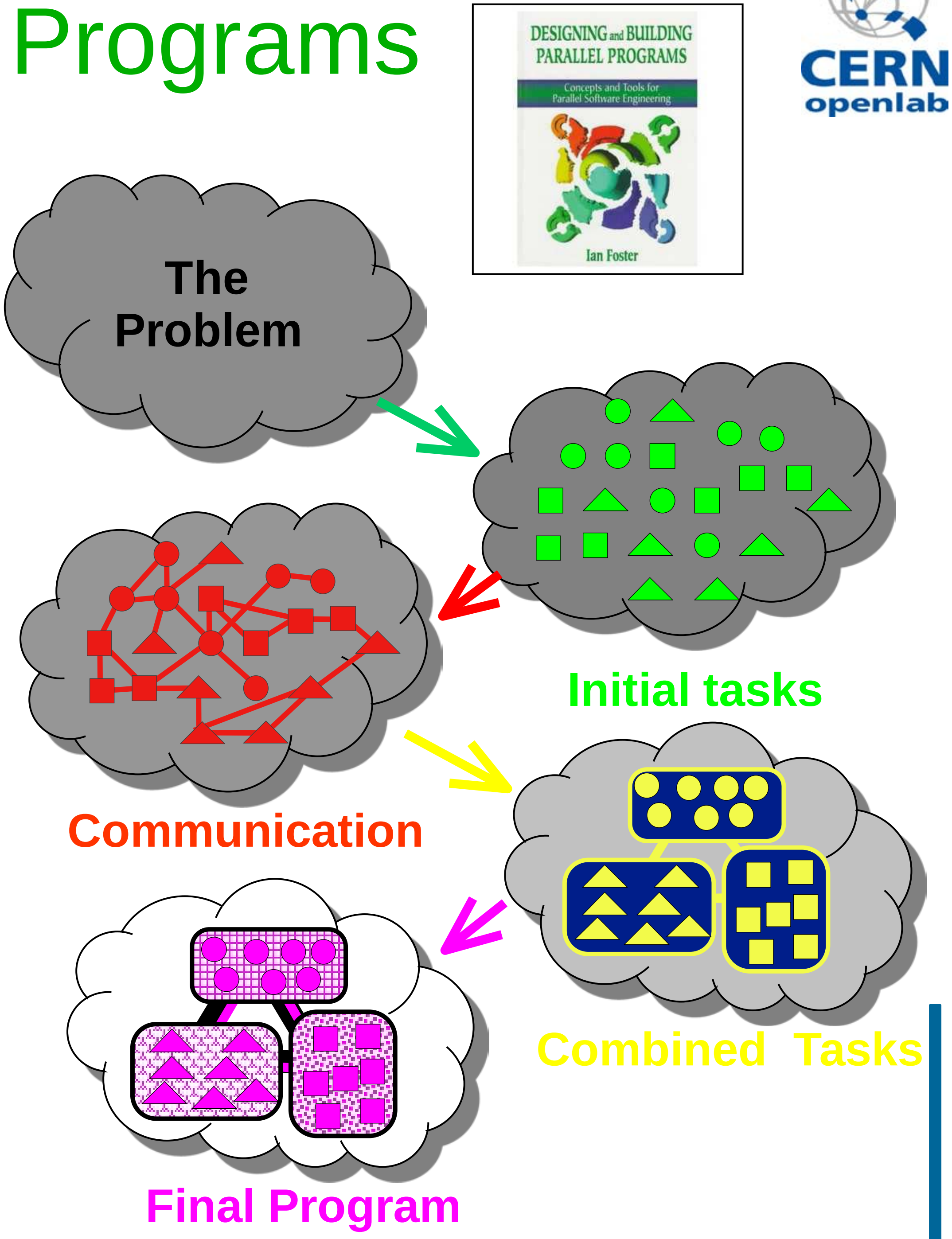
- **Keeping Amdahl's law for max speedup in mind**

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

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


DESIGNING and BUILDING PARALLEL PROGRAMS
Concepts and Tools for Parallel Software Engineering
Ian Foster
The Problem
Initial tasks
Communication
Combined Tasks
Final Program

More on decomposition

- **Divide the total work into smaller parts,**
  - Which can be executed concurrently
- **Some techniques:**
  - Data decomposition
    - Partition the data domain
  - Task/functional decomposition
    - Split according to “logical” tasks/functions
  - Recursive decomposition
    - Divide-and-conquer strategy
  - Exploratory decomposition
    - Search for a configuration space for a solution
      - Not guaranteed to reduce amount of work

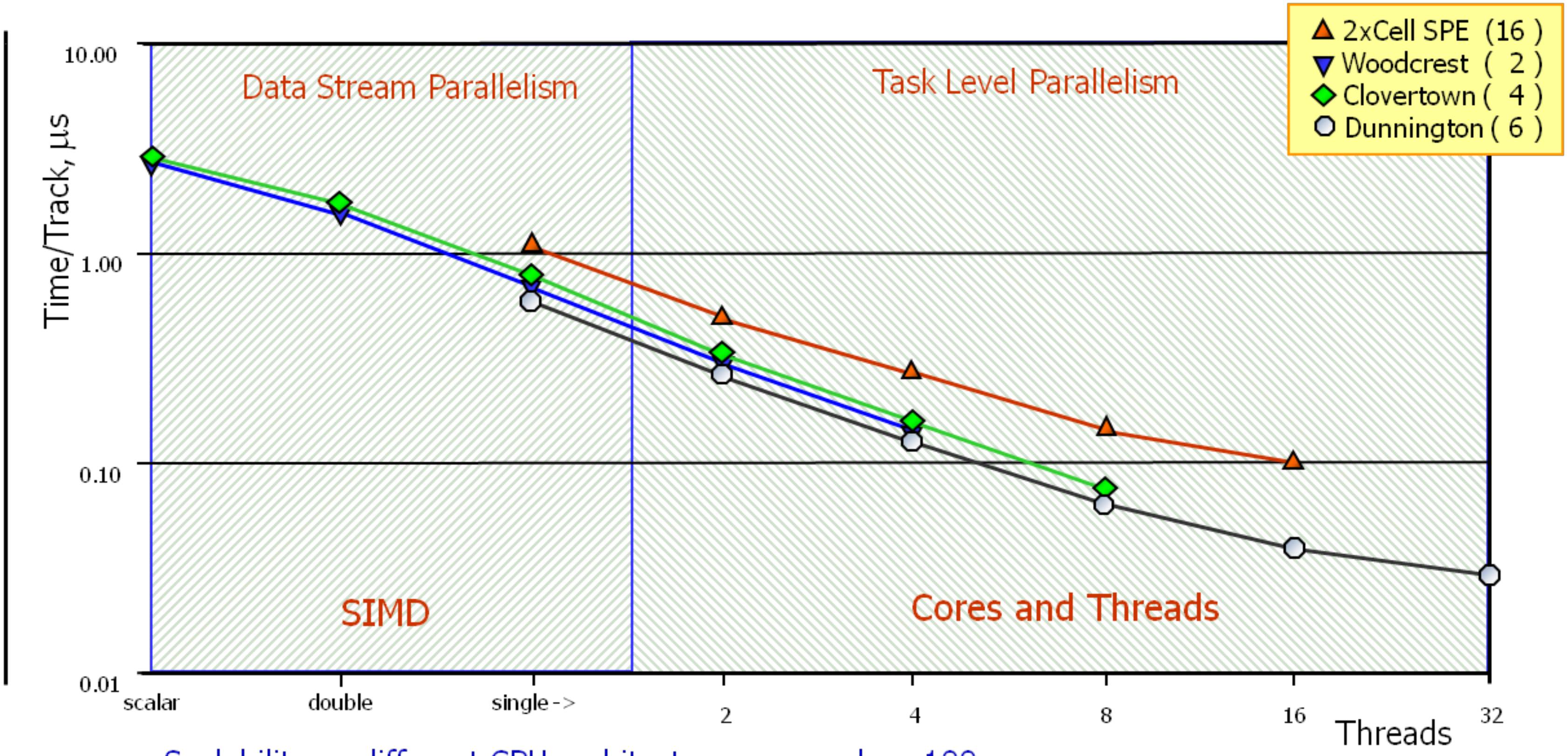
C++ parallelization support

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

We must also keep a close eye on OpenCL (www.khronos.org/opencl)

Examples of parallelism: CBM/ALICE track fitting

- **Re-optimization on x86-64 systems**
  - Part1: Data parallelism using SIMD instructions
  - Part 2: use TBB (or OpenMP) to scale across cores



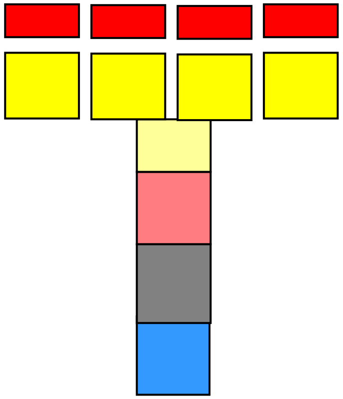
Scalability on different CPU architectures – speed-up 100

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:
    - Physics tables, geometry, stepping, etc.

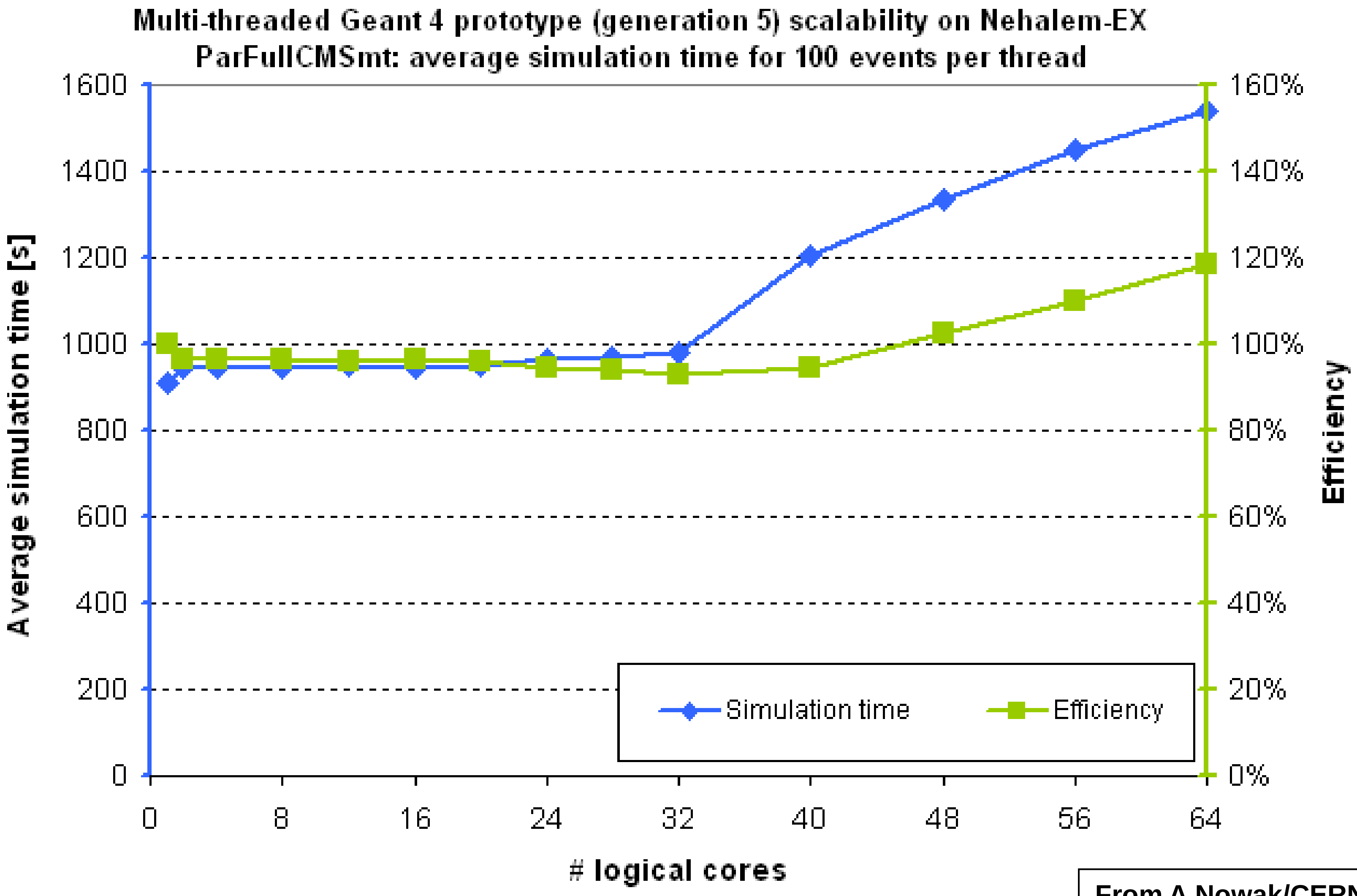


- **Additional memory: Only 25 MB/thread (!)**

Dong, Cooperman, Apostolakis: "Multithreaded Geant4: Semi-Automatic Transformation into Scalable Thread-Parallel Software", Europar 2010

Multithreaded GEANT4 benchmark

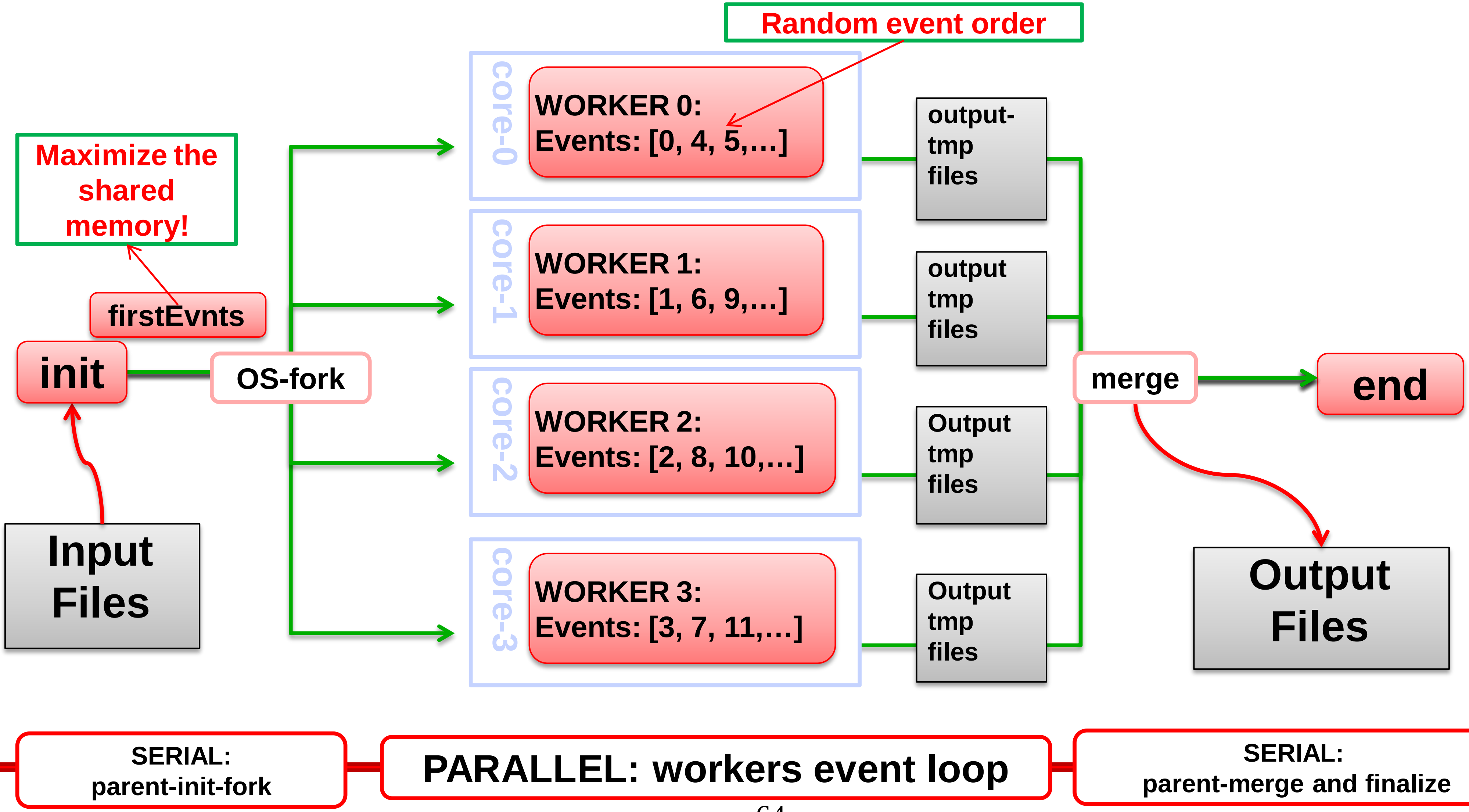
- **Excellent scaling on 32 (real) cores**
  - With a 4-socket server



From A.Nowak/CERN openlab

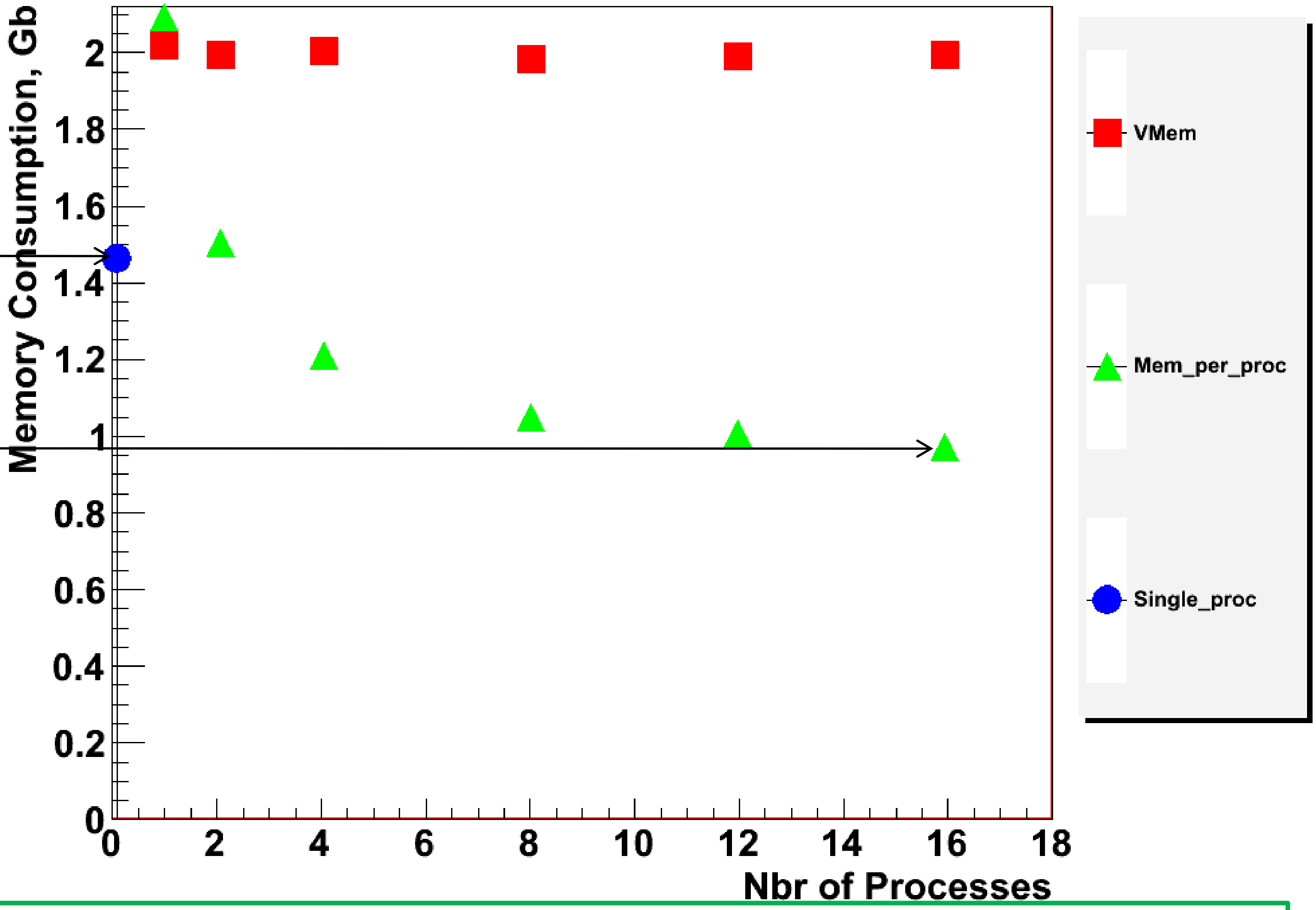
AthenaMP: event level parallelism

```
$> Athena.py  --nprocs=4  -c  EvtMax=100  Jobo.py
```



From: Mous TATARKHANOV/May 2010

Memory footprint of AthenaMP



From
~1.5 GB

To
~1.0 GB

AthenaMP ~0.5 GB physical memory saved per process

From: Mous TATARKHANOV/May 2010

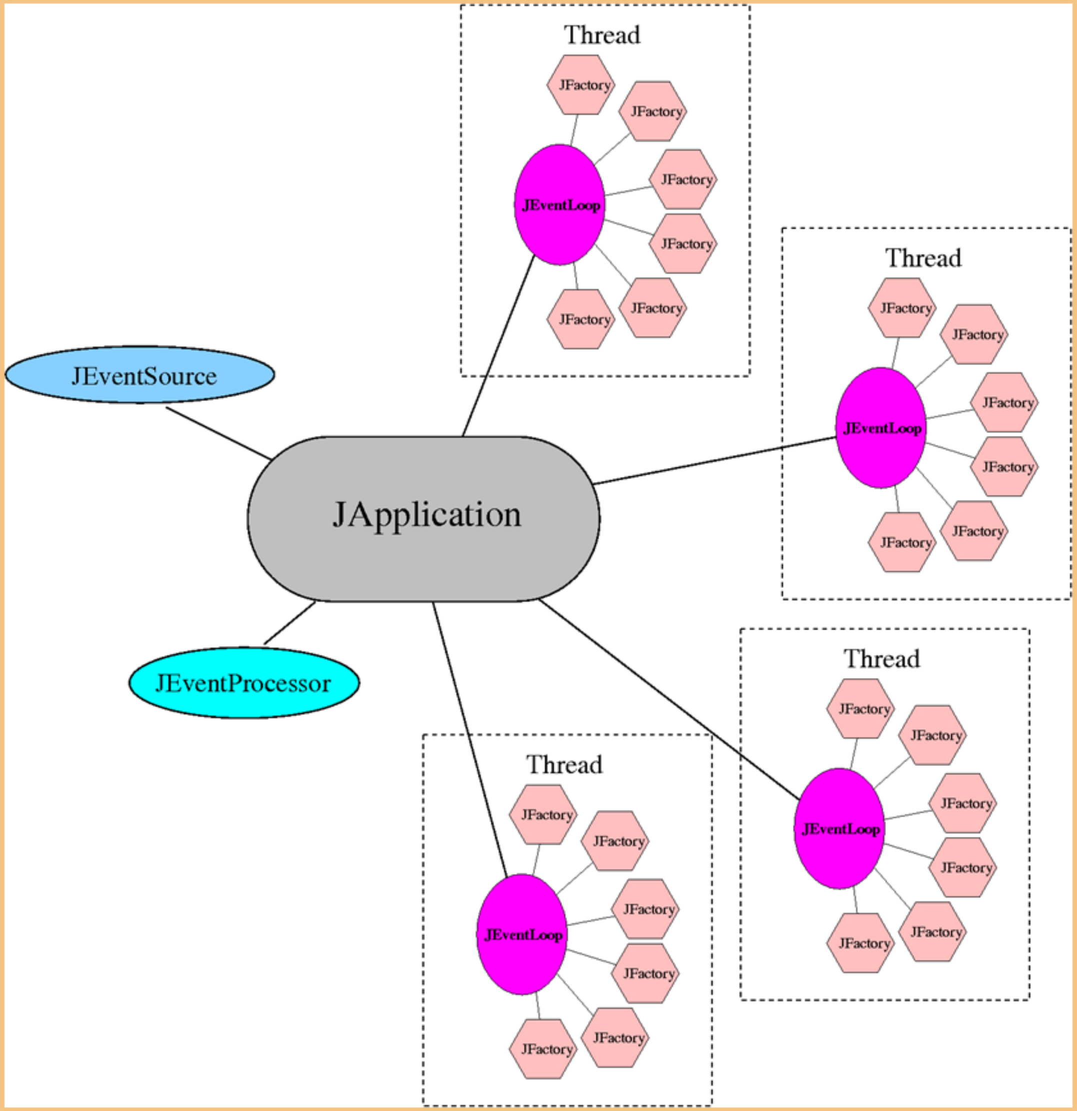
Examples of thread parallelism: JANA

- Each thread in JANA is composed of its own event processing loop and a complete set of factories

- **Reconstruction** of a given event is done entirely inside of a single thread

- No mutex locking is required by authors of reconstruction code

- Threads work asynchronously to maximize rates at the expense of not maintaining the event order on output

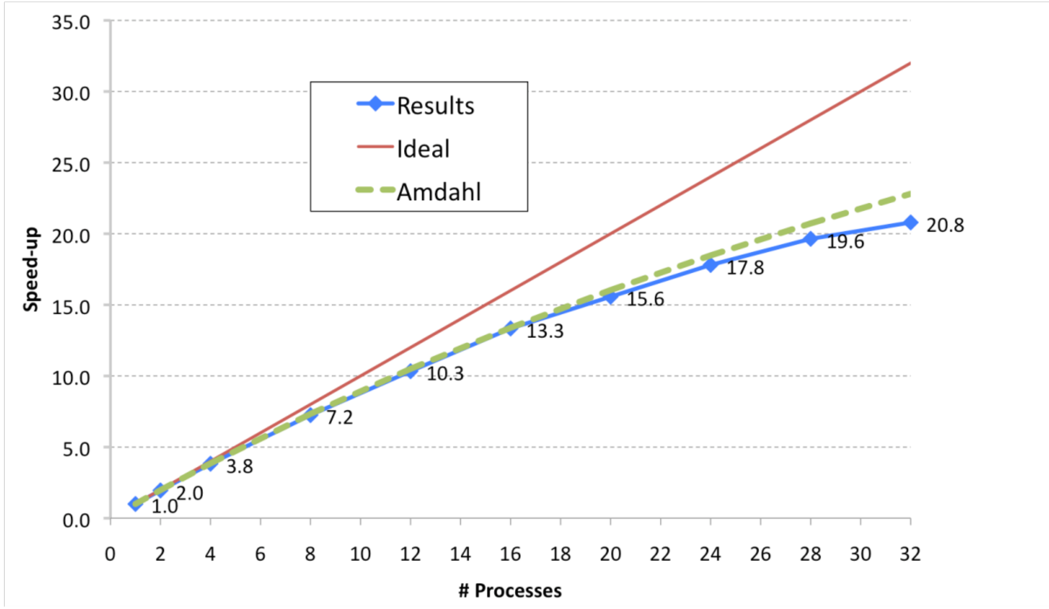


Developed by D.Lawrence/JLAB

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:**

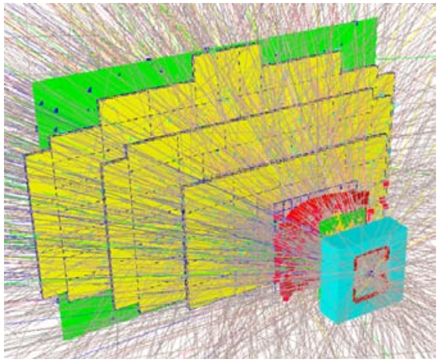


complex BaBar fitting provided by A. Lazzaro and parallelized using MPI

- **In principle, we can have combination of:**
  - parallelization via multi-threading in a multi-core CPU
  - multiple processes in a distributed computing environment

CUDA in the PANDA experiment

- **Track propagation in the PANDA experiment**
  - Runge-Kutta propagator from Geant3
    - All tracks propagated in parallel



  - Preliminary results from Feb.2010
    - M. Al Turany (ACAT2010)

#tracks	CPU (single core)	GPU emul. (on CPU)	Tesla C1060
100	210	160	5
1000	210	177	1.9

Time in microseconds/track

Recommendations

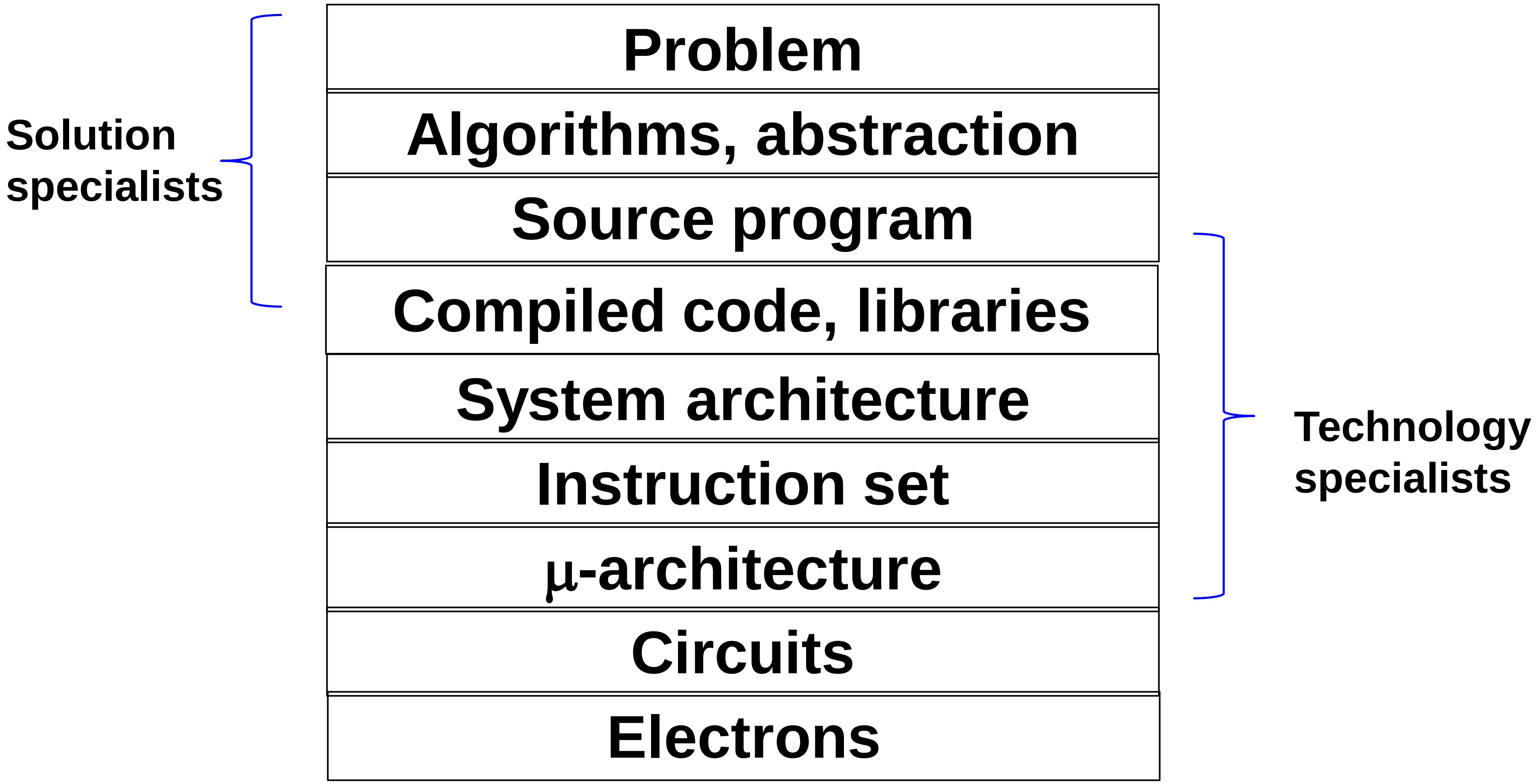
(based on observations in openlab)

Shortlist

1) Broad Programming Talent
2) Holistic View with a clear split: Prepare to compute – Compute
3) Controlled Memory Usage
4) C++ for Performance
5) Best-of-breed Tools

Broad Programming Talent

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



Solution specialists

Problem

Algorithms, abstraction

Source program

Compiled code, libraries

System architecture

Instruction set

μ-architecture

Circuits

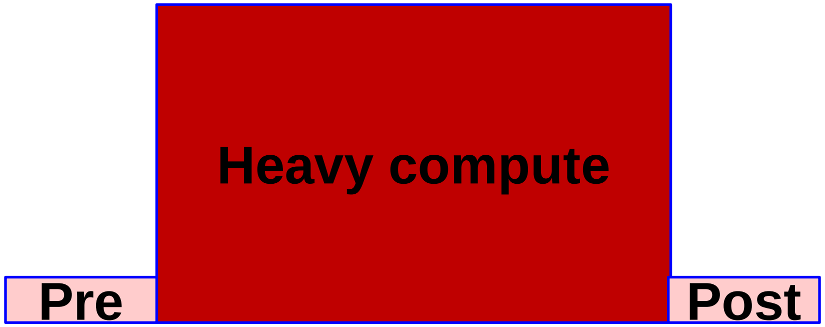
Electrons

Technology specialists

Adapted from Y.Patt, U-Austin

Performance guidance (cont'd)

- **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
    - Where you go after the available parallelism
  - Post-processing



- **Consider exploiting the entire server**
  - Using affinity scheduling

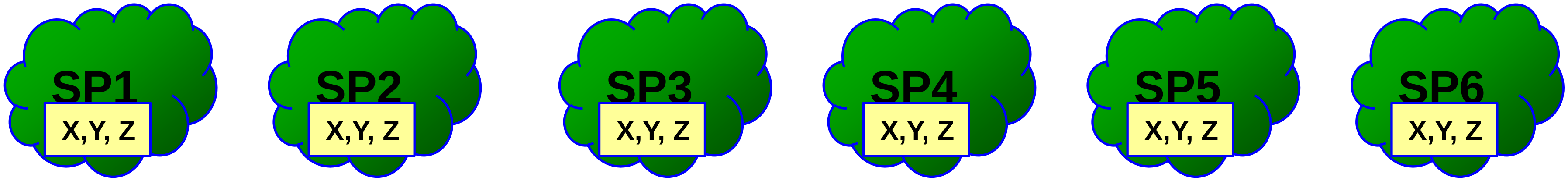
Performance guidance (cont’d)

- **Control memory usage (both in a multi-core and an accelerator environment)**
  - Optimize malloc/free
  - Forking is good; it may cut memory consumption in half
  - Don’t be afraid of threading; it may perform miracles !
  - Optimize the cache hierarchy
  - NUMA: The “new” blessing (or curse?)
- **C++ for performance**
  - Use light-weight C++ constructs
  - Prefer SoA over AoS
  - Minimize virtual functions
  - Inline whenever important
  - Optimize the use of math functions
    - SQRT, DIV; LOG, EXP, POW; ATAN2, SIN, COS

Organization of data: AoS vs SoA

- **In general, compilers and hardware prefer the latter!**

- **Arrays of Structures:**



SP1 X,Y, Z

SP2 X,Y, Z

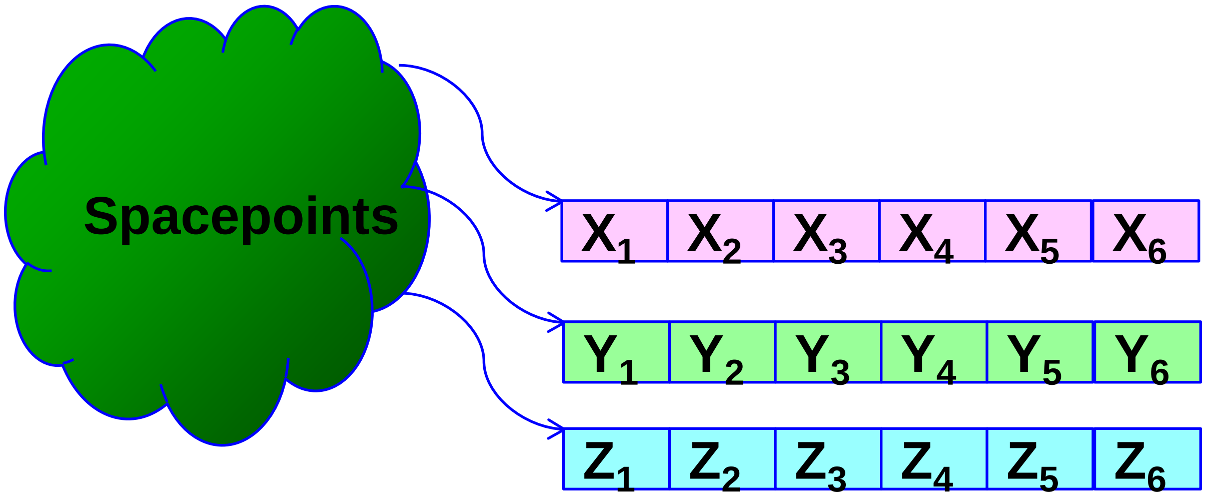
SP3 X,Y, Z

SP4 X,Y, Z

SP5 X,Y, Z

SP6 X,Y, Z

- **Structure of Arrays:**



Spacepoints

X_1	X_2	X_3	X_4	X_5	X_6
Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
Z_1	Z_2	Z_3	Z_4	Z_5	Z_6

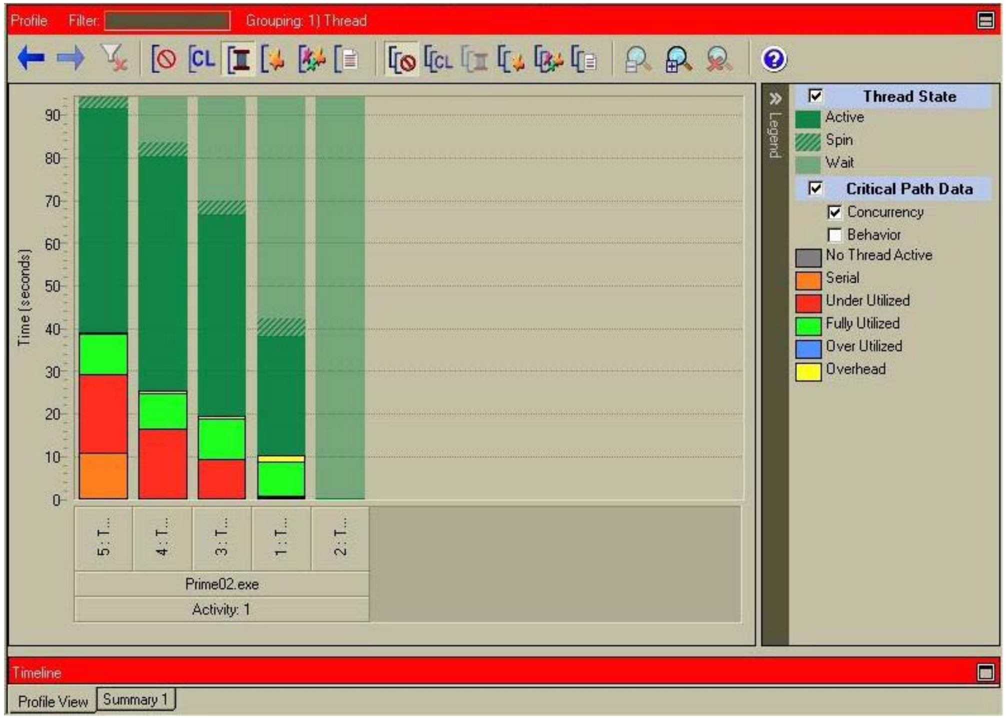
C++ parallelization support

- **Large selection of tools (inside the compiler or as additions):**
  - Native: pthreads/Windows threads
  - Forthcoming C++ standard: std::thread
  - OpenMP
  - Intel Array Building Blocks (beta version from Intel; integrating RapidMind)
  - Intel Threading Building Blocks (TBB)
  - TOP-C (from NE University)
  - MPI (from multiple providers), etc.
  - . . .

We must also keep a close eye on OpenCL (www.khronos.org/opencl)

Performance guidance (cont'd)

- **Surround yourself with good tools:**
  - Compilers
  - Libraries
  - Profilers
  - Debuggers
  - Thread checkers
  - Thread profilers



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?**
- **It helps to know the seven hardware dimensions and how appropriate software constructs can help!**

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, 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**

Thank you!

BACKUP I

Derived performance events

- **Too much information available?**
- **Low level and fine grained events can be combined to produce ratios (so called "derived events")**
- **Extensive information: Intel Manual 248966-020**
  - Intel Manual 248966-020 "Intel 64 and IA-32 Architectures Optimization Reference Manual"
  - AMD CPU-specific manuals, i.e. #32559 "BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors"

Basic modes

- **Counting**
  - Example: How many instructions did my application execute?
  - Example: How many times did my application have to stop and wait for data from the memory?
- **Sampling**
  - Reporting results in “regular” intervals
  - Example: every 100’000 cycles record the number of SSE operations since the last sample
- **Profiling**
  - Example: how many cycles are spent in which function?
  - Example: how many cache misses occur in which function?
  - Example: which code address is the one most frequently visited? (looking for hotspots)

Enabling different modes

- **Different modes are triggered by the presence of certain command line switches**
- **Counting**

  default mode

- **Sampling**

  ```
  --smpl-module=compact
  ```

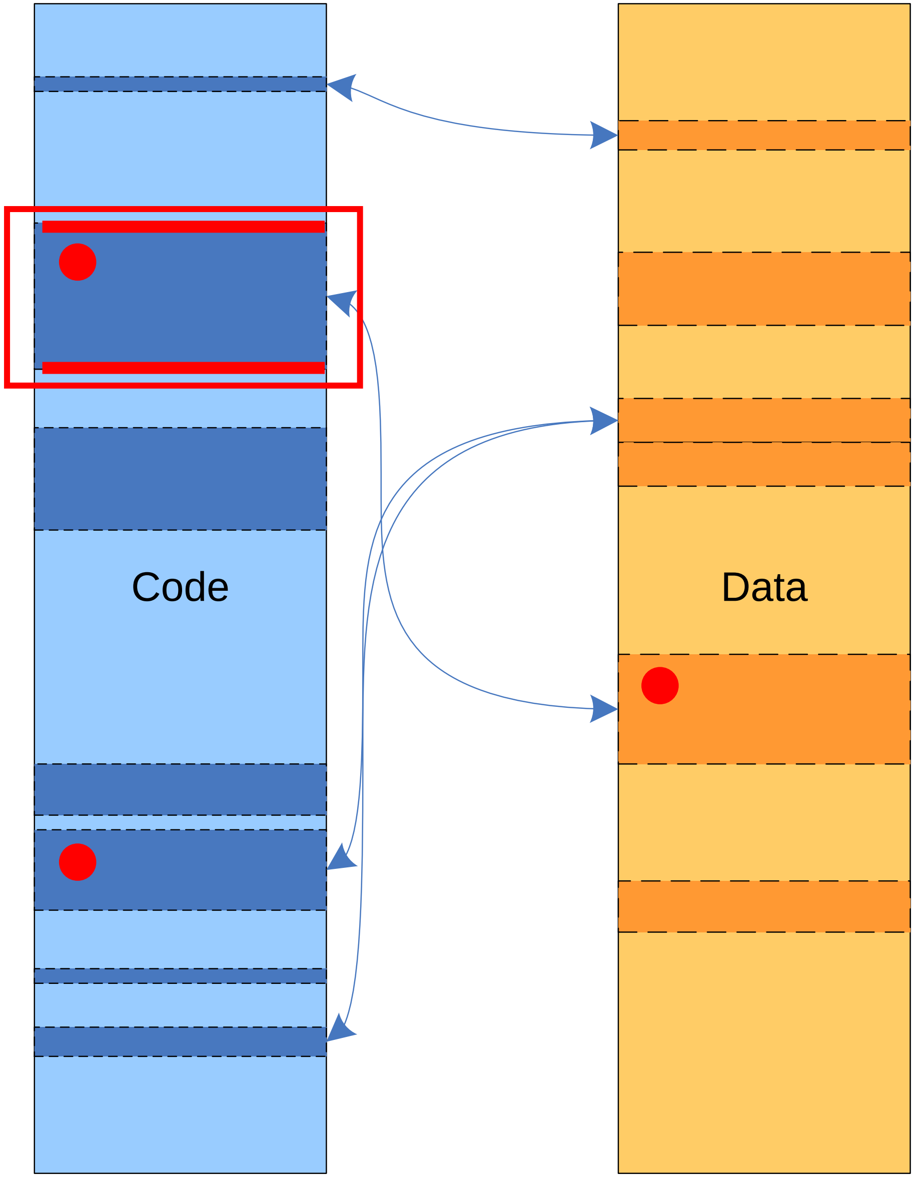
- **Profiling**

  ```
  --long-smpl-period=NUM
  ```

Triggers

- **Automatically start or stop monitoring**
- **Trigger types:**
  - Code
  - Data
- **A symbol name…**
  - i.e. “foobar”
- **…or an address**
  - i.e. 0x8103b91e

! **Limitation: symbol names are available only within the first binary**



Perfmon 2 resources

- **Resources:**
  - http://cern.ch/openlab
  - http://sf.net/projects/perfmon2
  - http://perfmon2.sourceforge.net (documentation)
  - http://perfmon2.sourceforge.net/pfmon_usersguide.html
  - http://www.intel.com (manuals)
  - http://cern.ch/andrzej.nowak (gpfmon)

- **Intel Software Products:**
  - VTune, Thread checker, Thread Profiler: http://intel.com/software
  - PTU: http://softwarecommunity.intel.com/articles/eng/1437.htm

- **HP Caliper**
  - http://h21007.www2.hp.com/portal/site/dspp

BACKUP – basic pfmon options

- **Event specification with umasks**

```
–e INST_RETIRED:STORES:LOADS
```

- **Follow all execution splits**

```
––follow-all
```

- **System wide mode**

```
––system-wide
```

- **Displaying the header**

```
––with-header
```

- **Aggregating results**

```
––aggregate-results
```

Pfmon output formatting

- **EU counter format (`--eu-c`)**

  1.567.123 instead of 1567123

- **US counter format (`--us-c`)**

  1,567,123 instead of 1567123

- **Hex counter format (`--hex-c`)**

  0xdeadbeef instead of 3735928559

- **Show execution time (`--show-time`)**

  real 0h00m00.252s user 0h00m00.000s sys 0h00m00.000s

- **Suppress monitored command output (`--no-cmd-output`)**

Advanced pfmon options

- **Specifying triggers**

```
–-trigger-code-start-address=...
–-trigger-code-stop-address=...
–-trigger-data-start-address=...
–-trigger-data-start-address=...
```

- **Multiplexing**

```
–e EVENT1,EVENT2,… -e EVENTa,EVENTb,… --switch-timeout=NUM
```

Pfmon sampling/profiling options

- **Specifying sampling periods (the unit is reference event occurrences)**

```
--long-smpl-period=NUM
--short-smpl-period=NUM
```

- **Resetting counters back to zero when sampling**

```
--reset-non-smpl-periods
```

- **Limit the sampling entries buffer (useful!)**

```
--smpl-entries=NUM
```

- **Translating addresses into symbol names**

```
--resolve-addresses
```

- **Show results per function rather than per address**

```
--smpl-per-function
```

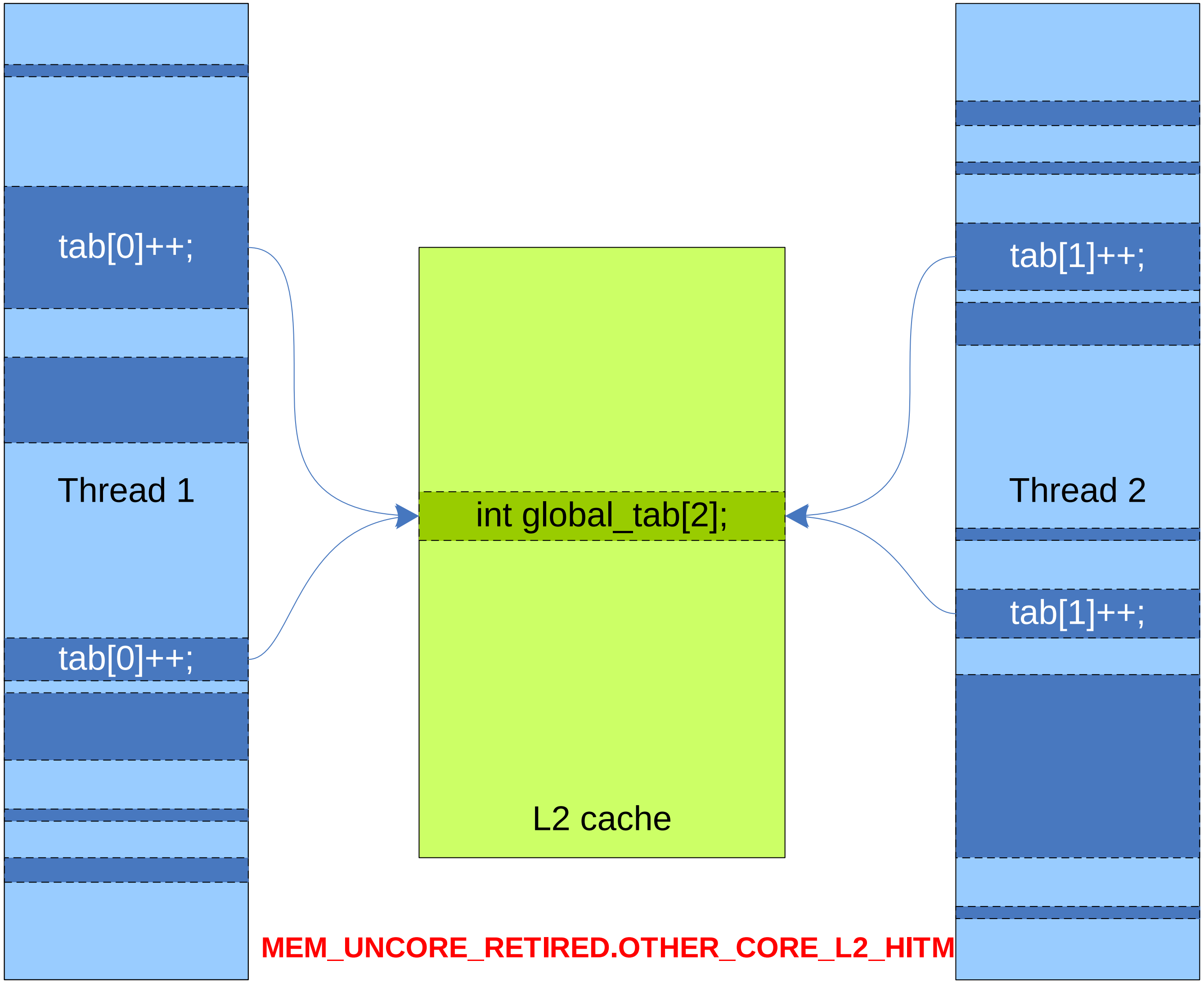
Example profiling results

```
cnt  %self   %cum addr symbol
 80 20.83% 20.83% 0x… do_lookup_x</lib64/ld-2.3.4.so>

 53 13.80% 34.64% 0x… do_page_fault<kernel>
 32  8.33% 42.97% 0x… _init</bin/ls>
 20  5.21% 48.18% 0x… __GI_strlen</lib64/tls/libc-2.3.4.so>
 19  4.95% 53.12% 0x… _int_malloc</lib64/tls/libc-2.3.4.so>
 18  4.69% 57.81% 0x… strcmp</lib64/ld-2.3.4.so>
 17  4.43% 62.24% 0x… __GI___strcoll_l</lib64/tls/libc-2.3.4.so>
 13  3.39% 65.62% 0x… __GI_memcpy</lib64/tls/libc-2.3.4.so>
```

Example sampling results

```
# description of columns:
#      column  1: entry number
#      column  2: process id
#      column  3: thread id
#      column  4: cpu number
#      column  5: instruction pointer
#      column  6: unique timestamp
#      column  7: overflowed PMD index
#      column  8: event set
#      column  9: initial value of overflowed PMD (sampling period)
#      followed by optional sampled PMD values in command line order

1   2     3   4      5                6          7 8    9     10
0 32442 32442 2 0x3061230d6a 0x0004d5f49c2a8e57 17 0 -26670 0x556
1 32442 32442 2 0x3061292980 0x0004d5f49c2b4851 17 0 -26670 0xd66
2 32442 32442 2 0x3061226363 0x0004d5f49c2c04dc 17 0 -26670 0x1aaa
3 32442 32442 2 0x3061010159 0x0004d5f49c2c39cb 17 0 -26670 0x6942
4 32442 32442 2 0x306126b5f0 0x0004d5f49c2c9a1c 17 0 -26670 0x171c
```

False sharing



MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM

BACKUP-II

Items not covered today

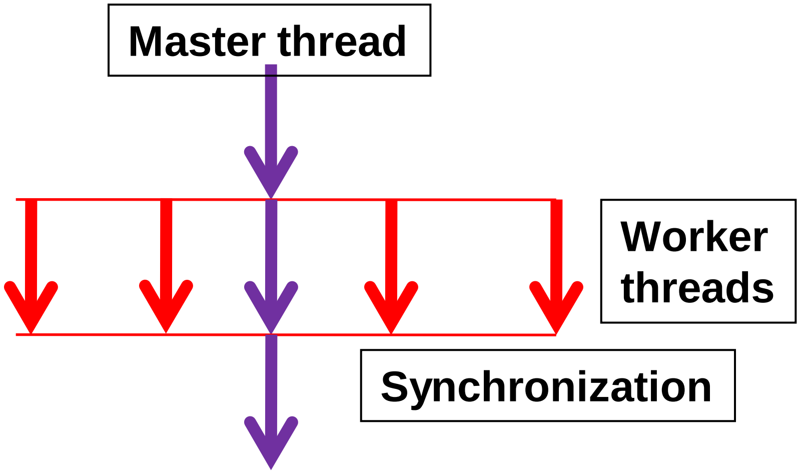
- **Systematic tuning approach**
- **Performance tuning versus correctness**
  - FP accuracy and reproducibility
- **Amdahl’s law (in detail)**
  - Also: Gustafson’s law
- **Parallel programming languages**
- **Detailed compiler “control”**
  - Including regression avoidance

OpenMP overview

- **De-facto standard for writing shared-memory parallel applications in C, C++ or FORTRAN**
- **Consists of:**
  - Compiler directives
  - Run-time routines
  - Environmental variables
- **http://www.openmp.org/**
  - Current version: 3.0
  - Still in active development

```
#pragma omp parallel for \
            shared (n, a, b, c) \
            private(i)
for (i = 0; i < n; i++) c[i] = a[i] + b[i];
```

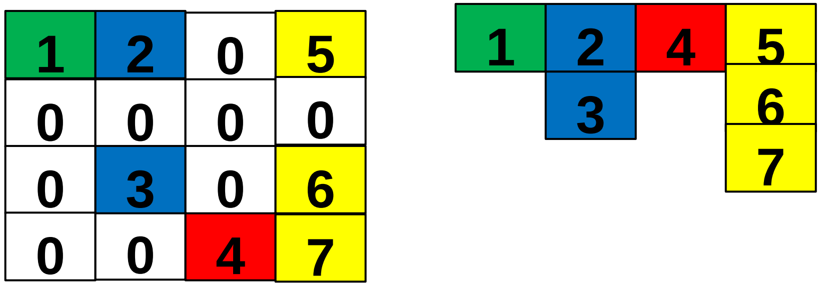
```
gcc –fopenmp –O –oaprog aprog.c
setenv OMP_NUM_THREADS 4
./aprog
```



Array Building Blocks (ArBB)

See: CERN/IT seminar on 11/10/2007 by A.Ghuloum/Intel: Programming Challenges for Manycore Computing

- **Effort by Intel to extend C++ for Throughput Computing**
  - Initially called C_t
- **Features:**
  - Addition of new data types (parallel vectors) & operators
    - NeSL/SASAL-inspired: irregularly nested and sparse/indexed vectors
  - Abstracting away architectural details
    - Vector width/Core count/Memory Model: Virtual Intel Platform
      - Forward-scaling (Future-proof!)
    - Nested data parallelism and deterministic task parallelism
- **Incremental adoption path:**
  - Dedicated Ct-enabled libraries
  - Rewritten “kernels” in Ct
  - Pervasive use of Ct



1	2	0	5
0	0	0	0
0	3	0	6
0	0	4	7

1	2	4	5
	3		6
			7

Intel TBB 2.0 overview

```
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
using namespace tbb;
//
task_scheduler_init init;
  tasks = atoi( argv[1] );
//
 parallel_for(blocked_range<int>(0,
NTracksV, NTracksV / tasks),
ApplyFit(TracksV, vStations, NStations));
```

- **Key features:**
  - Open source extension to C++ (GPL)
  - Task patterns instead of threads
    - Focus on the work, not the workers
  - Designed for scalable performance
    - Automatic scaling to use available resources
  - Components
    - Generic parallel algorithms: parallel_for, parallel_reduce, etc.
    - Low-level synchronisation primitives: atomic, mutex, etc.
    - Concurrent containers: concurrent_vector, concurrent_hash_map, etc.
    - Task scheduler
    - Memory allocation: cache_aligned_allocator
    - Timing

More features in preparation

MPI overview

- **MPI – Message Passing Interface**
  - A language independent communications API
  - Point-to-point message passing and global operations
  - No shared memory concept in MPI-1 (v 1.2)
  - MPI-2 (v. 2.1) introduces numerous enhancements
    - Limited shared memory concept
    - Parallel I/O
    - Dynamic management
    - Remote memory support
  - Numerous implementations exist
    - Including the combination of OpenMP and MPI