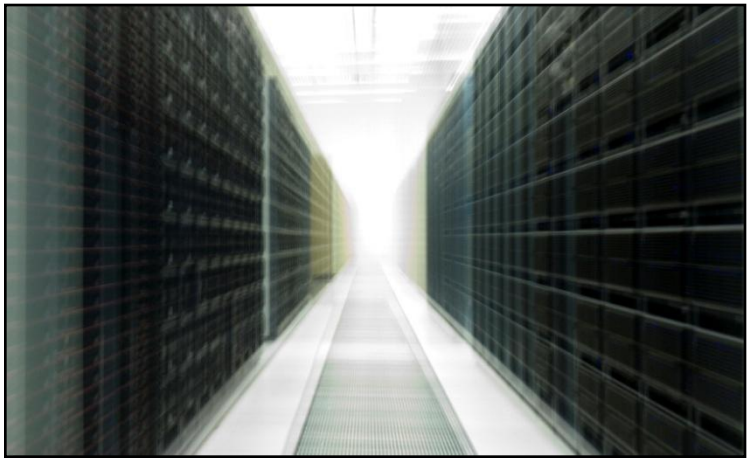
Computer Architecture and Performance Tuning

"Mastering Performance in 7 dimensions"

Sverre Jarp
CERN
openlab
CTO

IT Dept.

CERN



CSC 2009, Göttingen – August 2009

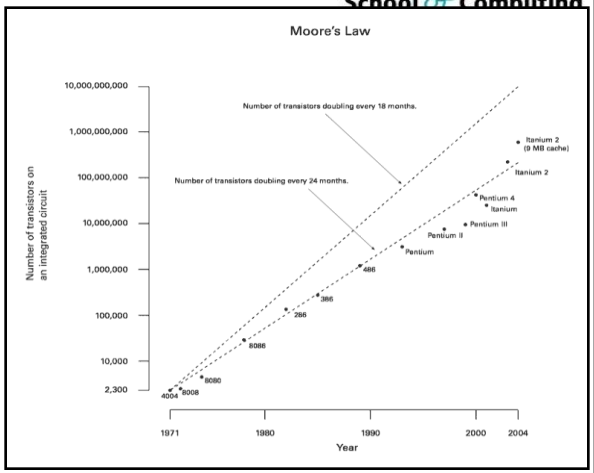
Goal of this lecture series

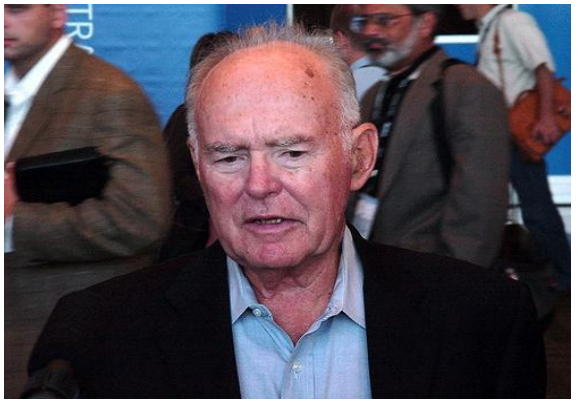
- **Give an understanding of modern computer architectures from a performance point-of-view**
  - Processor, [Cache, Memory subsystem]
  - x86-64 as a de-facto standard
- **Explain (hardware/software) factors that improve or degrade program execution**
  - Help to write well-performing software
- **Teach an approach to detailed performance measurements (3rd lecture)**
  - Highlight the most important events for such measurements

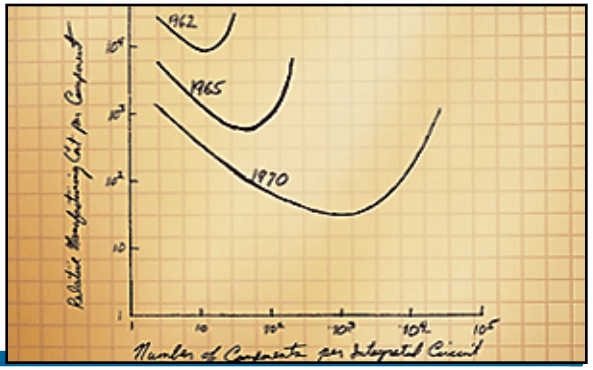
Moore's law

- **We continue to double the number of transistors every other year(*)**
  - Latest consequence
    - Single core → Multicore → Manycore
- **All in all:**
  - An unbelievable "agreement" with all stakeholders
    - Silicon manufacturers
    - System integrators
    - Customers

(*)But, the derivative "law" which stated that the frequency would also double is no longer true!





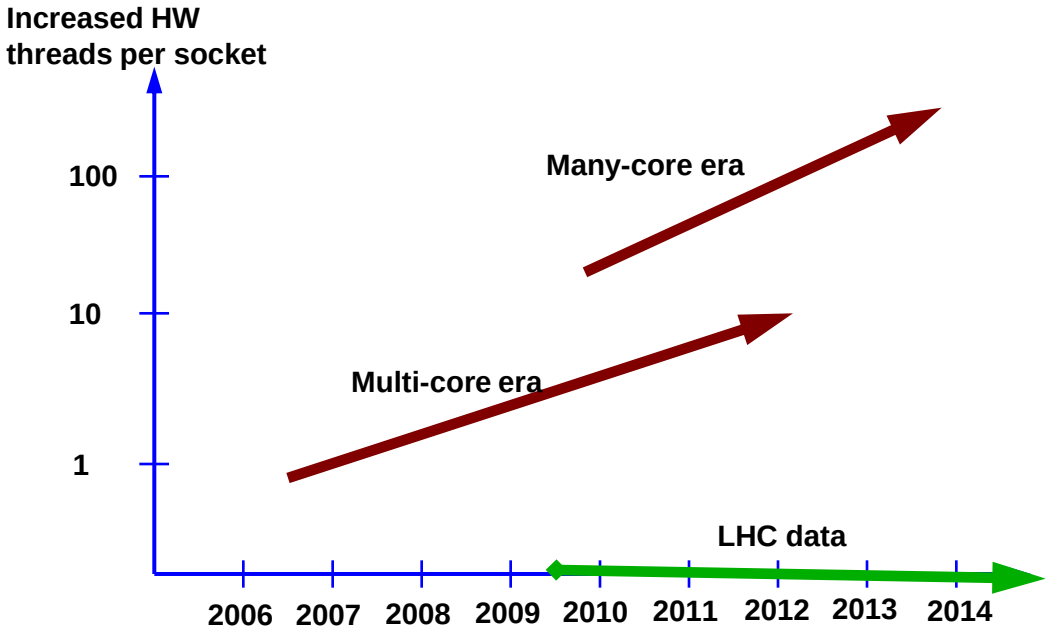


Real consequence of Moore's law

- **We are being "run over" by transistors:**
  - More (and more complex) execution units
  - 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: 45 nm**
- **Already on the roadmap:**
  - 32 nm (2009/10)
  - 22 nm (2011/12)
- **In research:**
  - 16 nm (2013/14)
  - 11 nm (2015/16)
  - 8 nm (2017/18)
    - Source: Bill Camp/Intel HPC

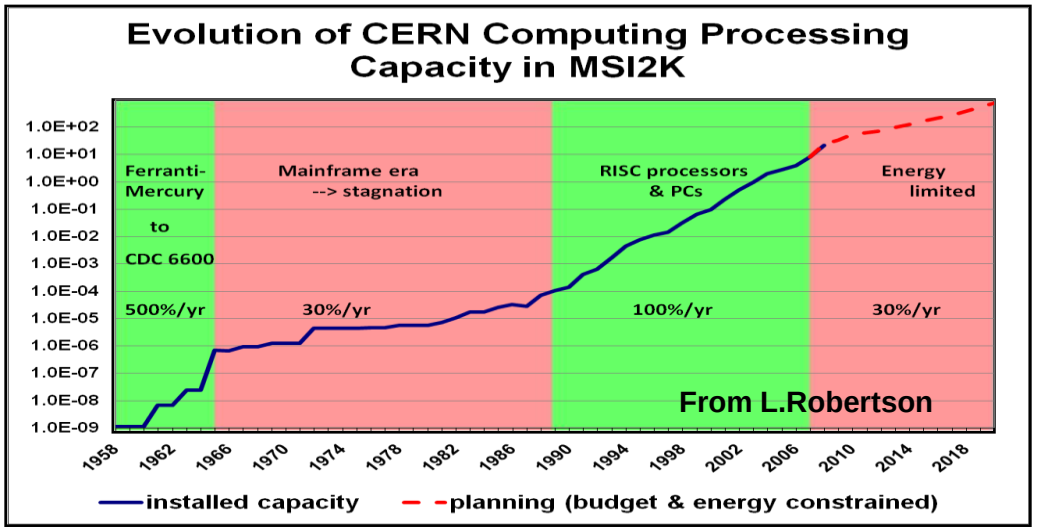


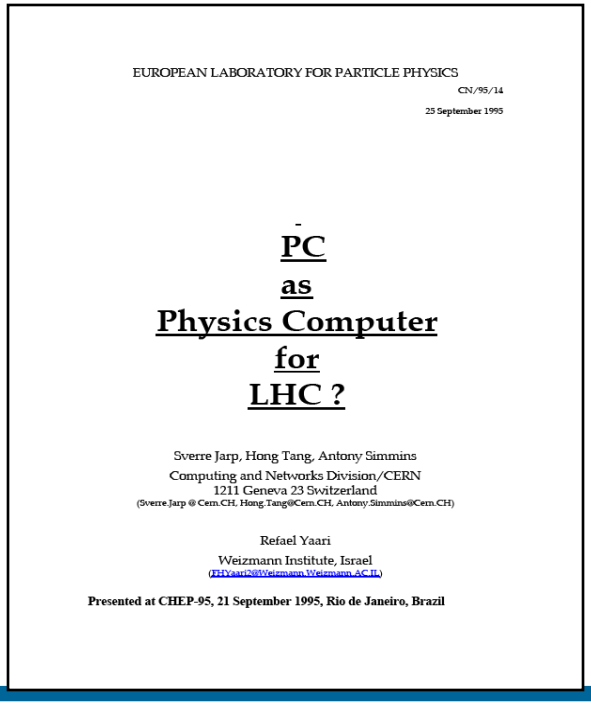
From “Platform 2015: Intel Platform Evolution for the Next Decade” (S.Borkar et al./Intel Corp.)

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

Evolution of CERN's computing capacity

- **During the LEP era (1989 – 2000):**
  - Doubling of compute power every year
  - Initiated with the move from mainframes to RISC systems
- **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 ?

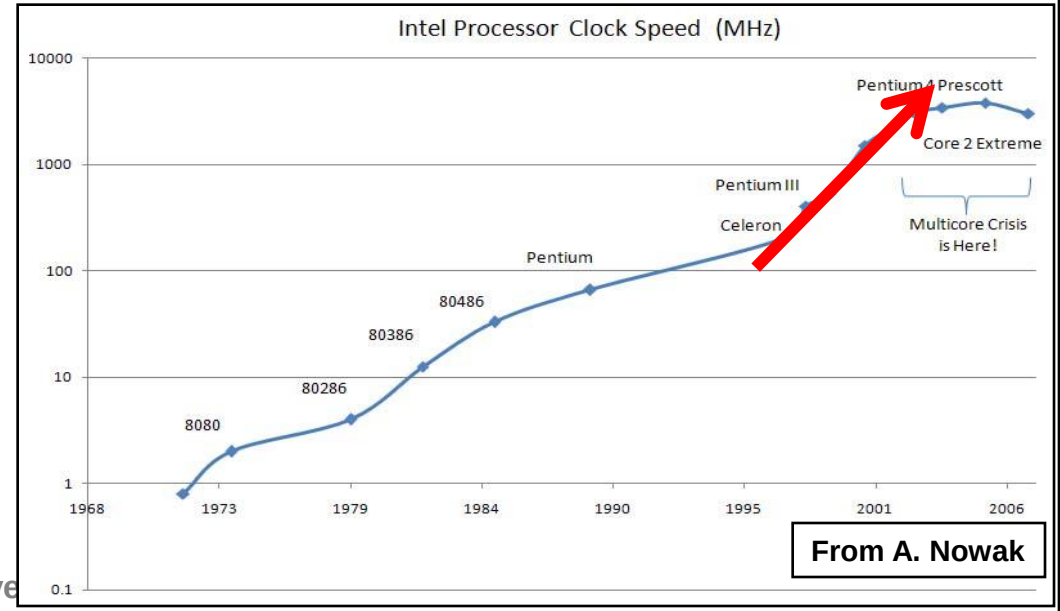
Sverre Jarp, Hong Tang, Antony Simmins
Computing and Networks Division/CERN
1211 Geneva 23 Switzerland
(Sverre.Jarp @ Cern.CH, Hong.Tang@Cern.CH, Antony.Simmins@Cern.CH)

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

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

Frequency scaling

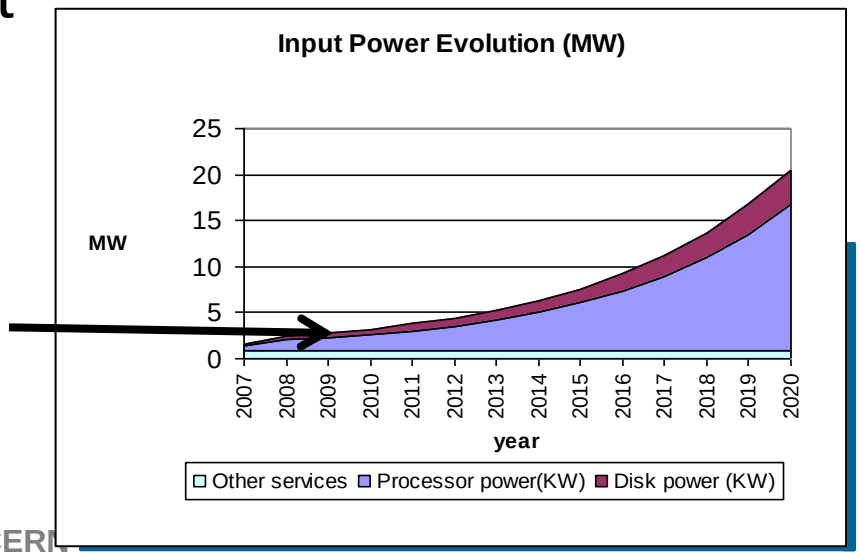
- **The 7 “fat” years of frequency scaling in HEP**
  - From the Pentium Pro in 1996: 150 MHz
  - To the Pentium 4 in 2003: 3.8 GHz (~25x)
- **Since then**
  - Core 2 systems:
    - ~3 GHz
    - Multi-core
- **Recent CERN purchase:**
  - Intel L5520 CPUs
    - 2.26 GHz



CERN School of Computing

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 servers.
  - Network switches and routers
- **This limit will be reached soon!**

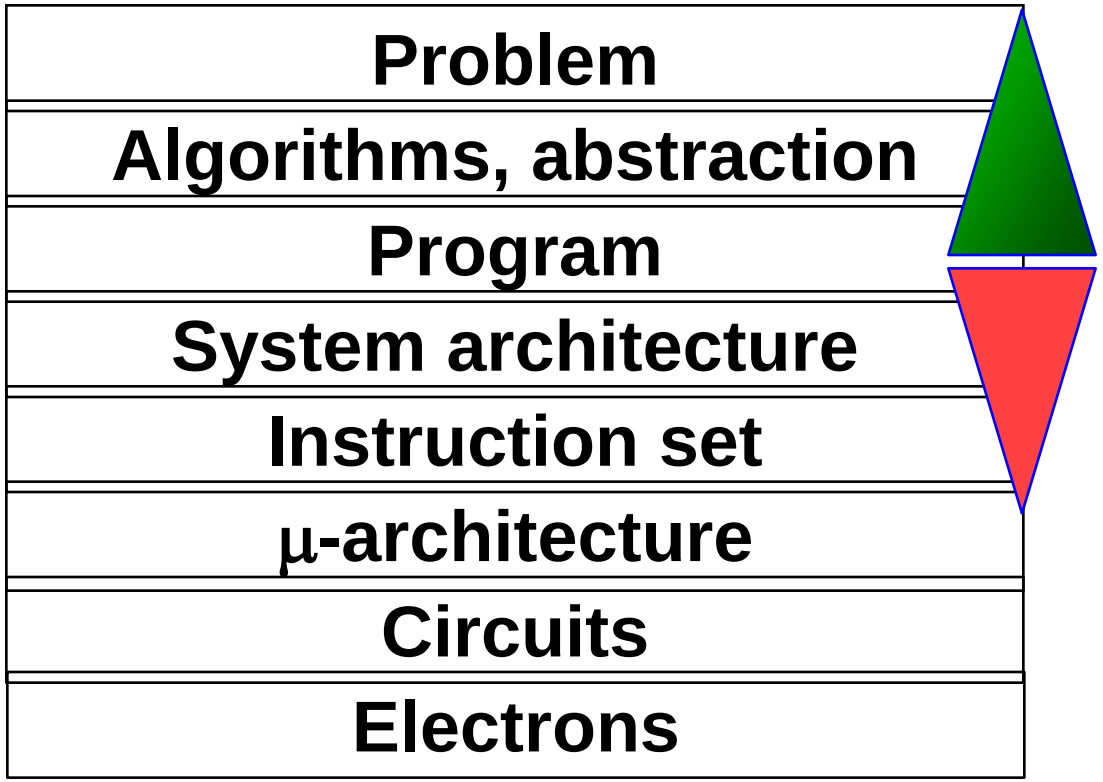


CERN School of Computing

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 layers!)



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

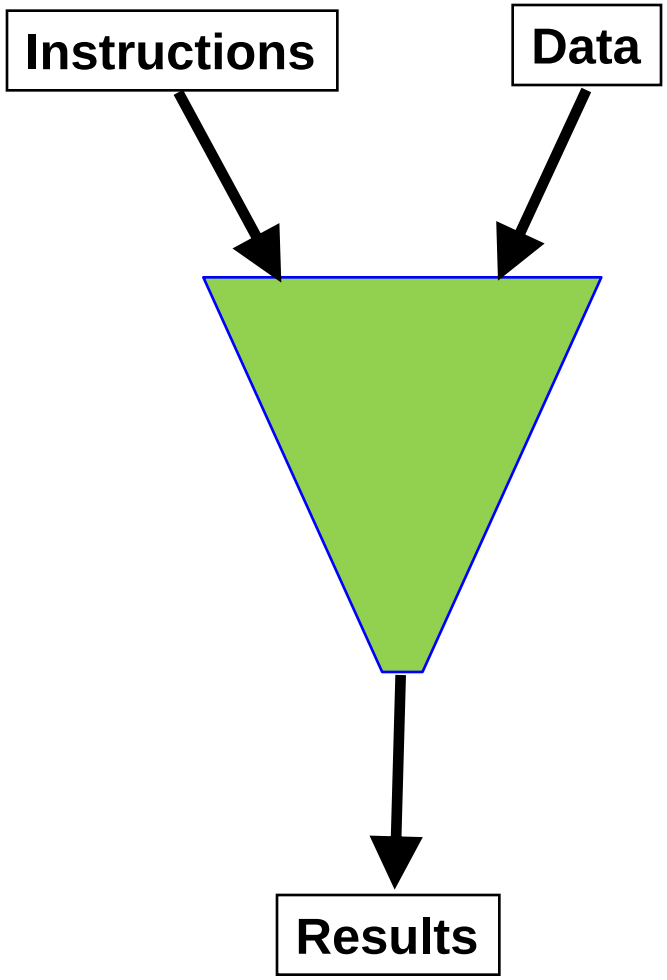
Adapted from Y.Patt, U-Austin

CERN School of Computing

Let’s start with the basics!

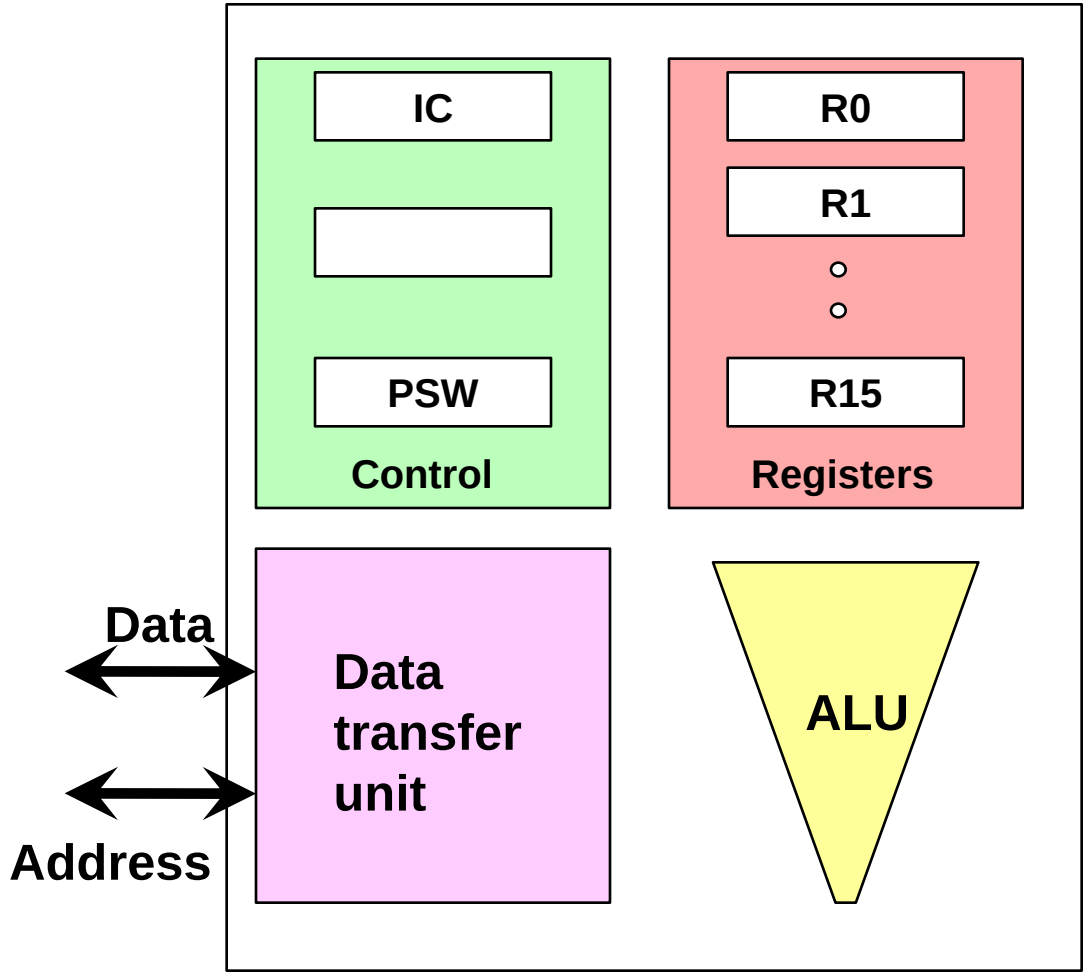
Von Neumann architecture

- **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**



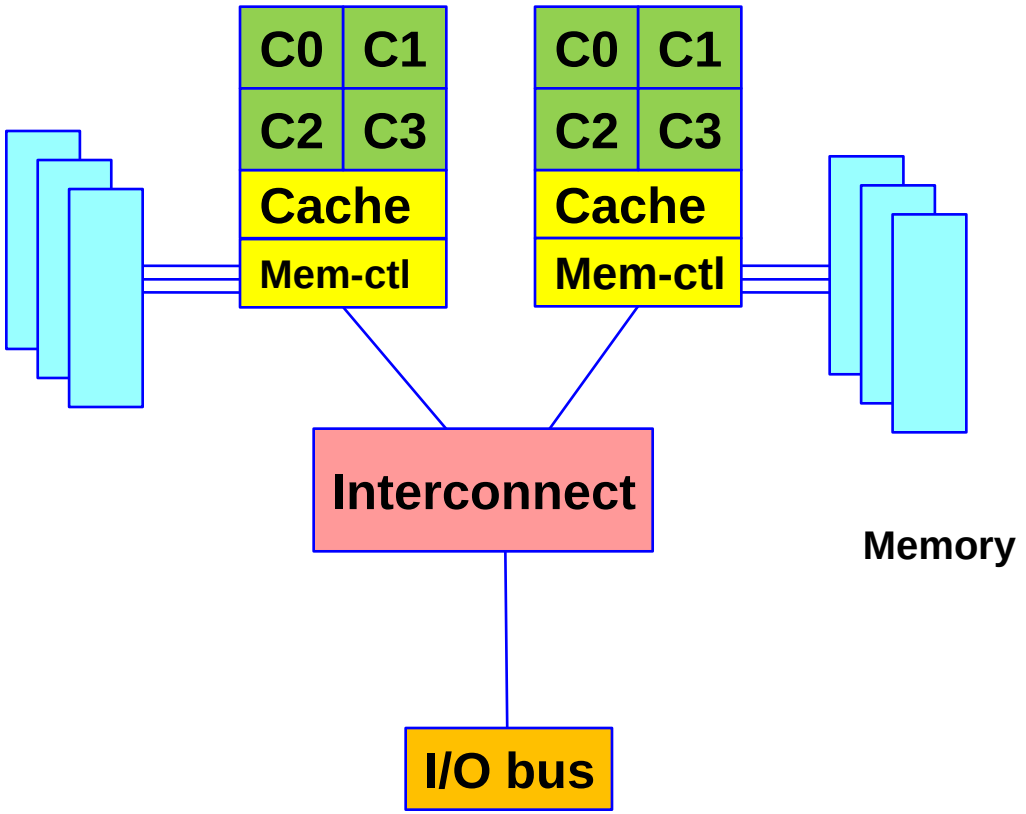
Simple processor layout

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



Simple server diagram

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

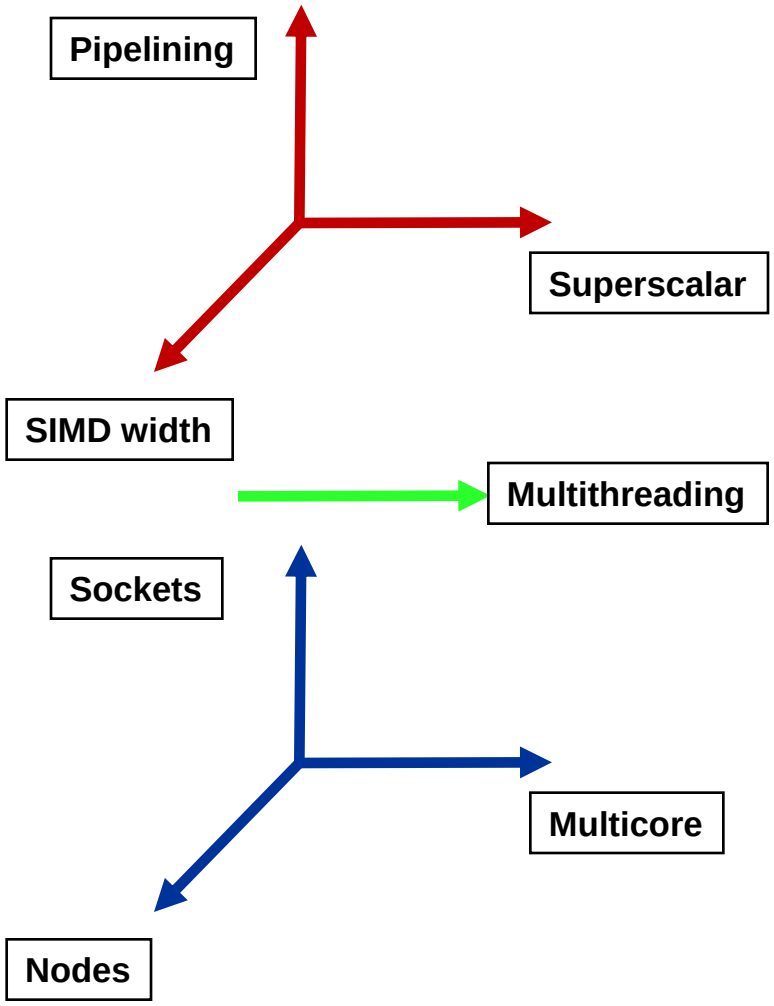


Initial premise

- **To reach completion, a compute job (a process) requires the execution of a given number of (machine-level) instructions**
- **We typically want the process to complete in the shortest possible time**
  - 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 dimensions of performance

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

Data parallelism (Vectors/Scalars)

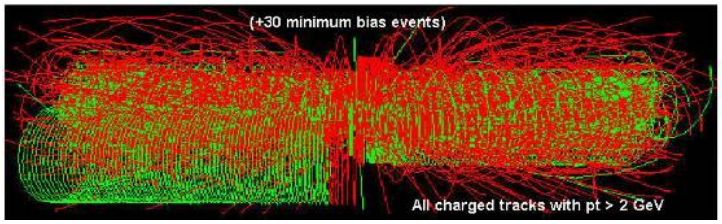
- **Next dimension is a "pseudo" dimension:**
  - Hardware multithreading
- **Last three dimensions:**
  - Multiple cores
  - Multiple sockets
  - Multiple compute nodes

Task parallelism (Events/Tracks)

CERN School of Computing

Concurrency in HEP

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



- **Usable for both data and task parallelism!**

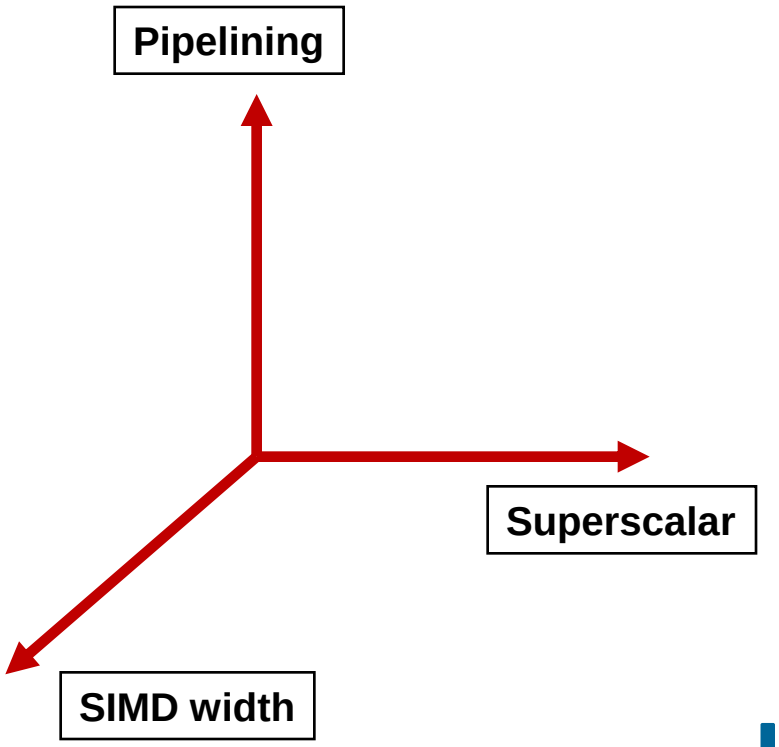
Autoparallelization/Autovectorization

- **Would it not be wonderful if the compilers could do all the (vectorization/parallelisation) work for us automatically?**
- **GNU compiler (4.3.0):**
  - Autovector: YES, but needs “-ftree-vectorize”
  - Intrinsics: YES
  - Autoparallel: YES (as of 4.2.0) with “-fopenmp”
- **Intel compiler (10.1 or later):**
  - Autovector: YES, included in “-O”
  - Intrinsics: YES
  - Autoparallel: YES with “-openmp”

Intrinsics: “higher-level assembly instructions” that the compilers understand

Part 1: Opportunities for scaling performance inside a core

- **Let's look at the first three dimensions**

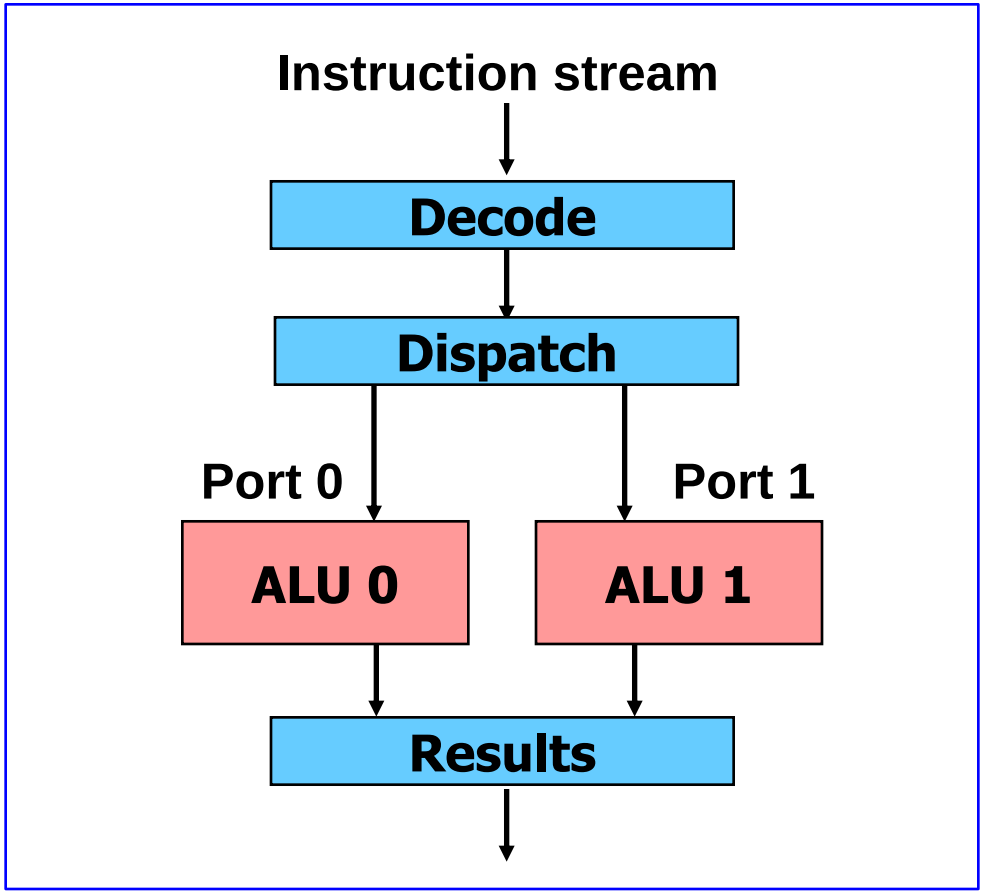


- **The resources:**
  - Superscalar: Fill the ports
  - Pipelined: Fill the stages
  - SIMD: Fill the computational width

- **Best approach: data parallelism**

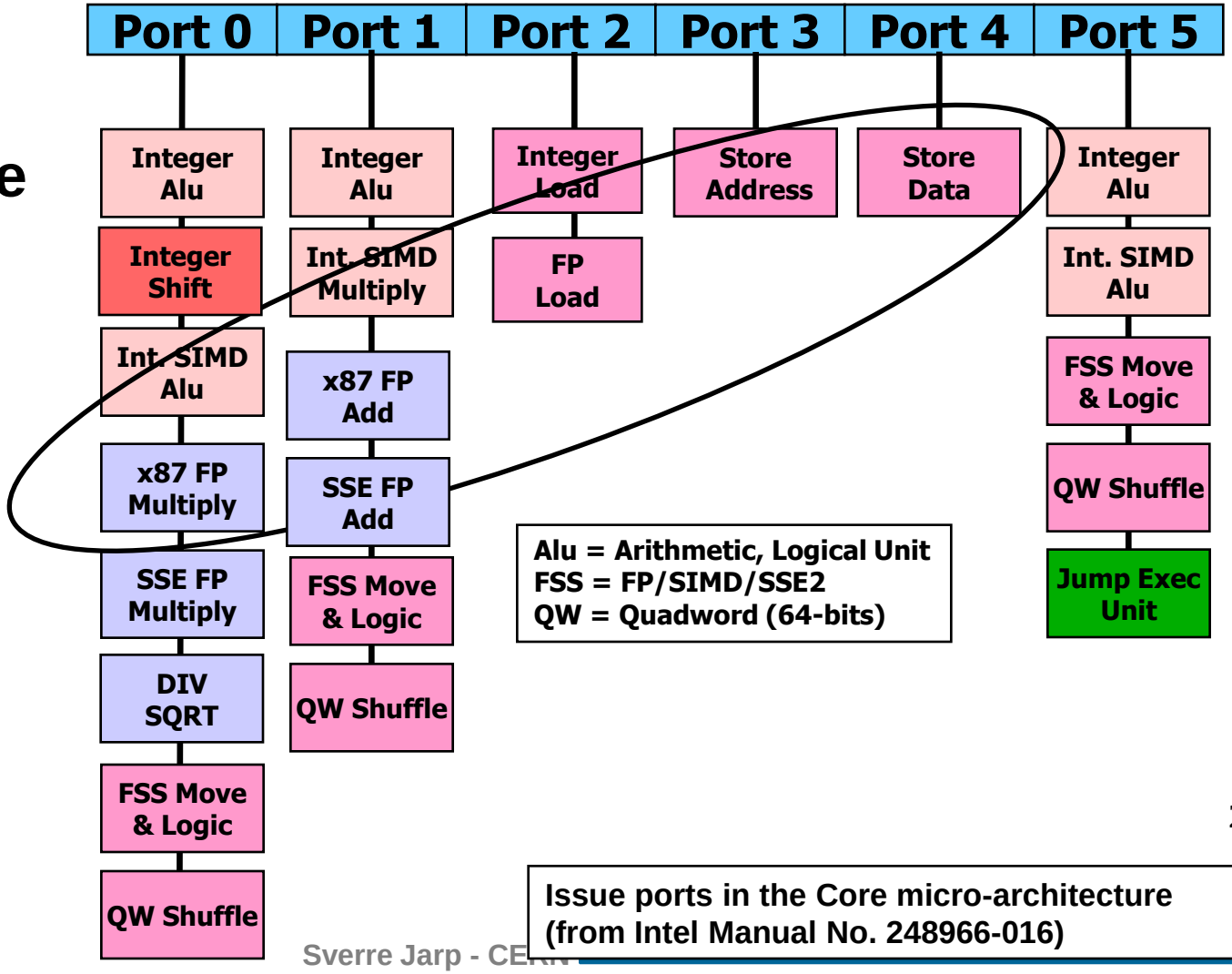
Superscalar architecture

- **In this simplified design, instructions are decoded serially, but dispatched to two ALUs.**
  - The decoder and dispatcher ought to be able to handle two instructions per cycle
  - The ALUs can have identical or different execution capabilities



Core 2 execution ports

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



Issue ports in the Core micro-architecture (from Intel Manual No. 248966-016)

24

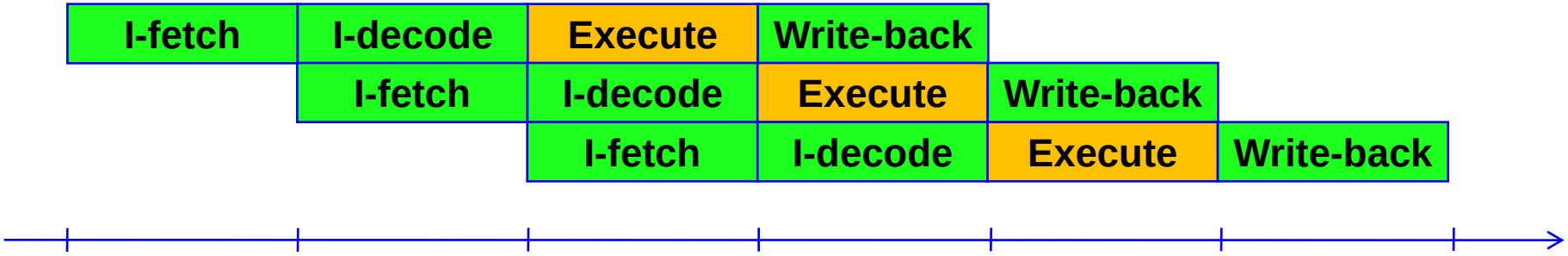
Mulmul example

- **Which execution units are needed ?**
  - in the innermost loop

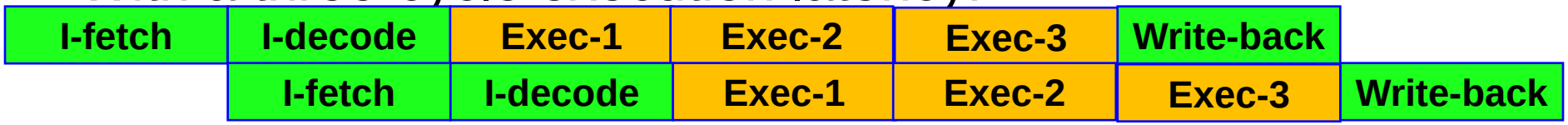
```
for ( int i = 0; i < N; i++ ) {
        for ( int j = 0; j < N; j++ ) {
                for ( int k = 0; k < N; k++ ) {
                        c[ i * N + j ]  +=  a[ i * N + k ]  *  b[ k * N + j ];
                }
        }
}
```

Next topic: Instruction pipelining

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



  - With a three-cycle execution latency:



CERN School of Computing

Real-life latencies

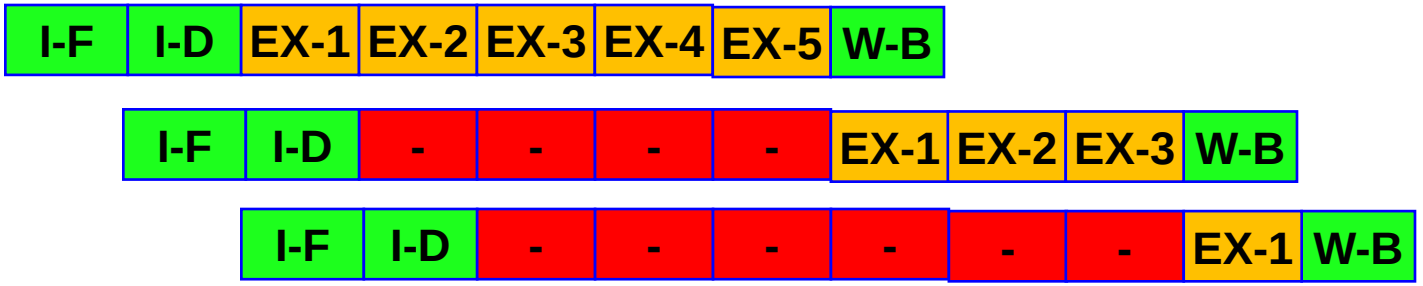
- **Most integer/logic instructions have a one-cycle execution latency:**
  - For example: ADD, AND, SHL (shift left), ROR (rotate right)
  - Amongst the exceptions:
    - IMUL (integer multiply): 3
    - IDIV (integer divide): 13 – 23
- **Floating-point latencies are typically multi-cycle**
  - FADD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

Latencies in the Core micro-architecture (from Intel Manual No. 248966-016)
AMD processor latencies are similar.

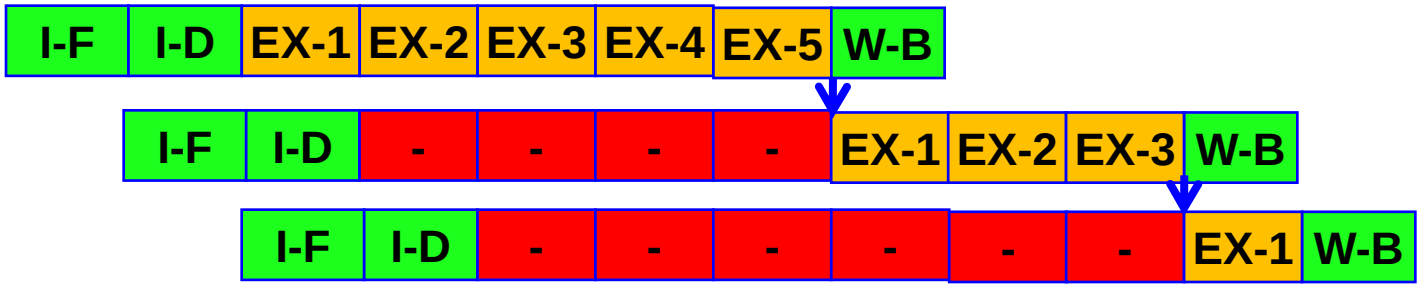
Latencies and serial code (1)

- **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
```



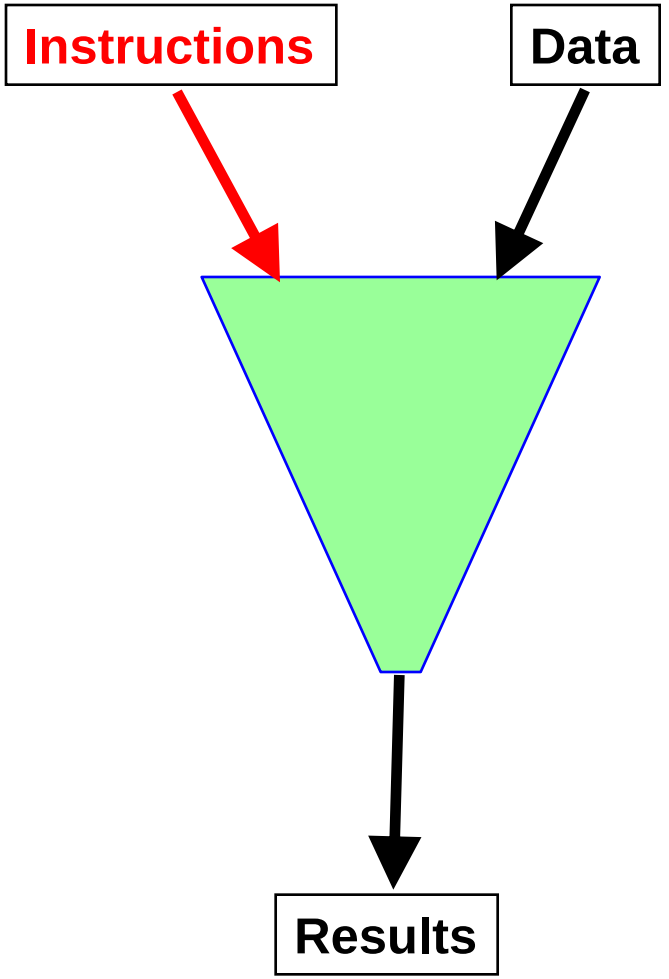
Latencies and serial code (2)



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

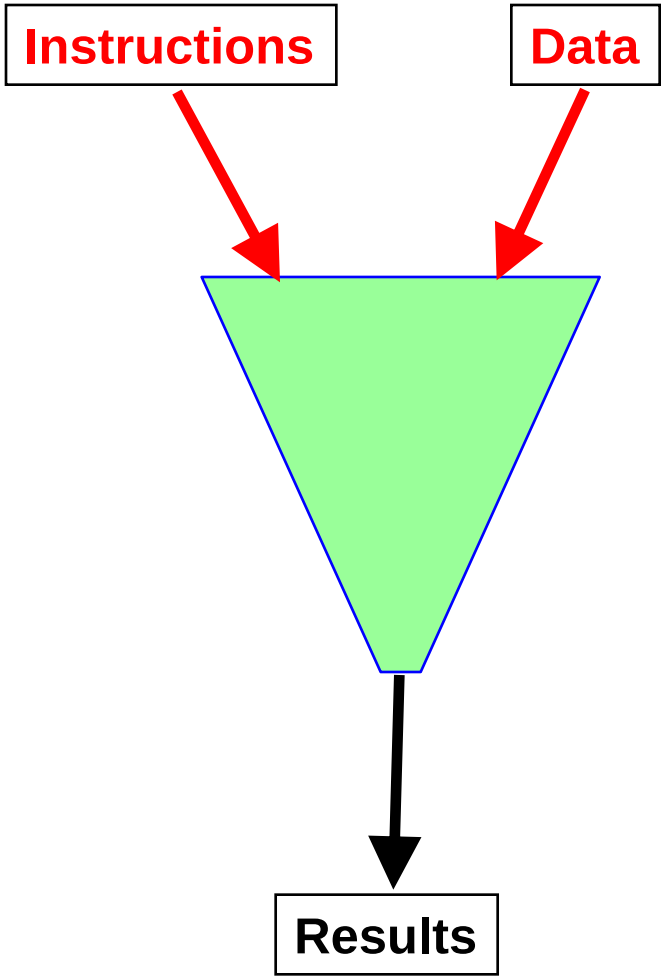
Other causes of execution delays (1)

- **We already stated that the aim is to keep instructions and data flowing, so that results are generated optimally**
- **First issue:**
  - Instructions are unavailable
    - Typically caused by branching
    - There may be a branch instruction in every 10 machine instructions!
      - Or even less
    - If the branch is mispredicted, we suffer a stall (cycles clock up, but no work gets done)



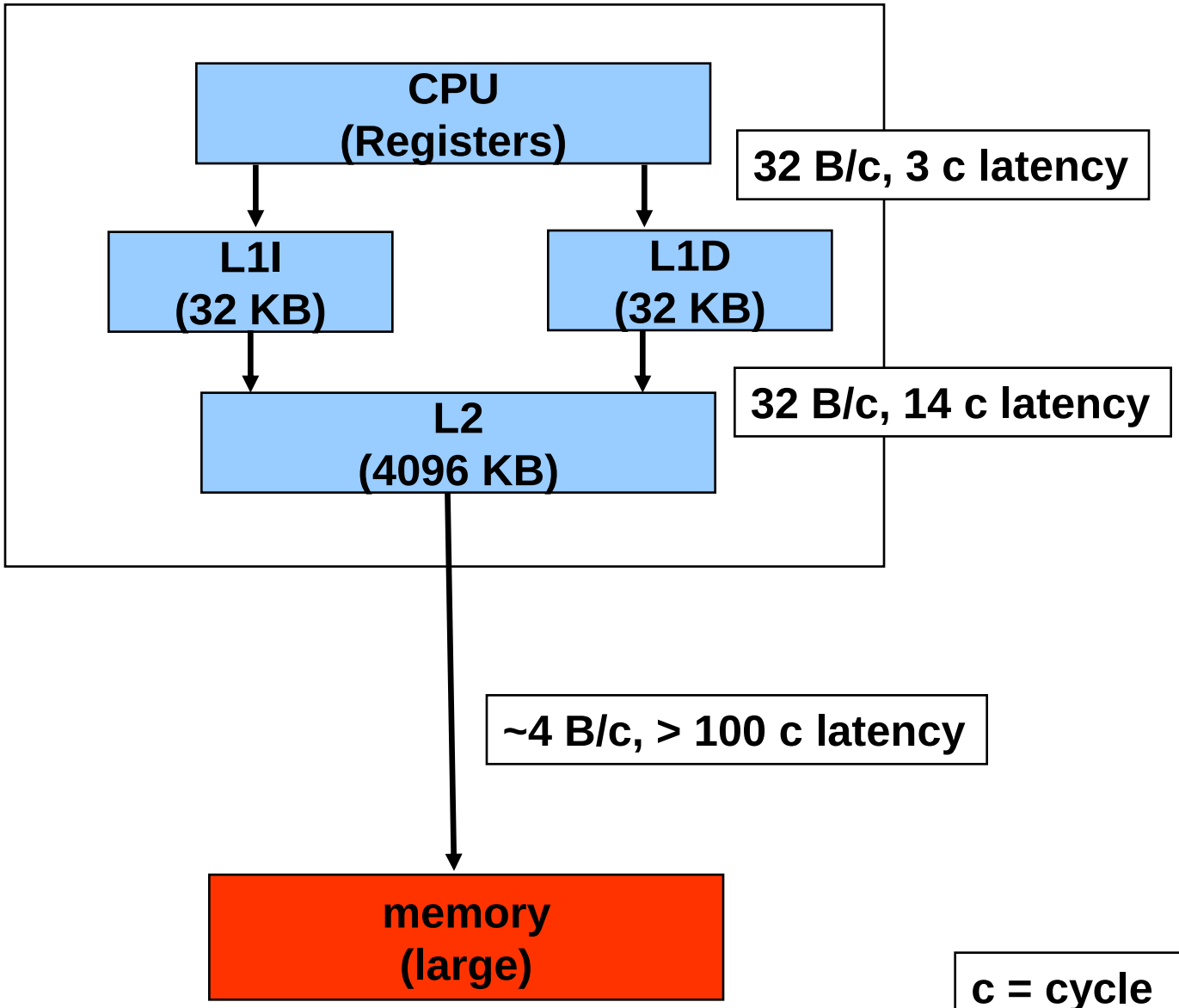
Other causes of execution delays (2)

- **Second 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 further away in the memory hierarchy



Memory Hierarchy

- **From CPU to main memory on a Core 2 uni-processor**
  - With multicore, memory bandwidth is shared between cores on the same bus



Cache lines (1)

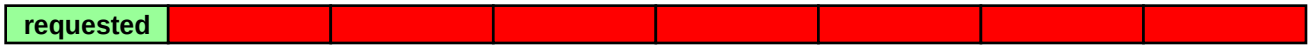
- **When a data element or an instruction is requested by the processor, a cache line is moved (as the minimum quantity) to Level-1**

- **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 (>100 cycles, as already mentioned)
  - Memory bus stays busy (~16 cycles)

CERN School of Computing

Cache lines (2)

- **Space locality is vital**
  - What happens 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:**

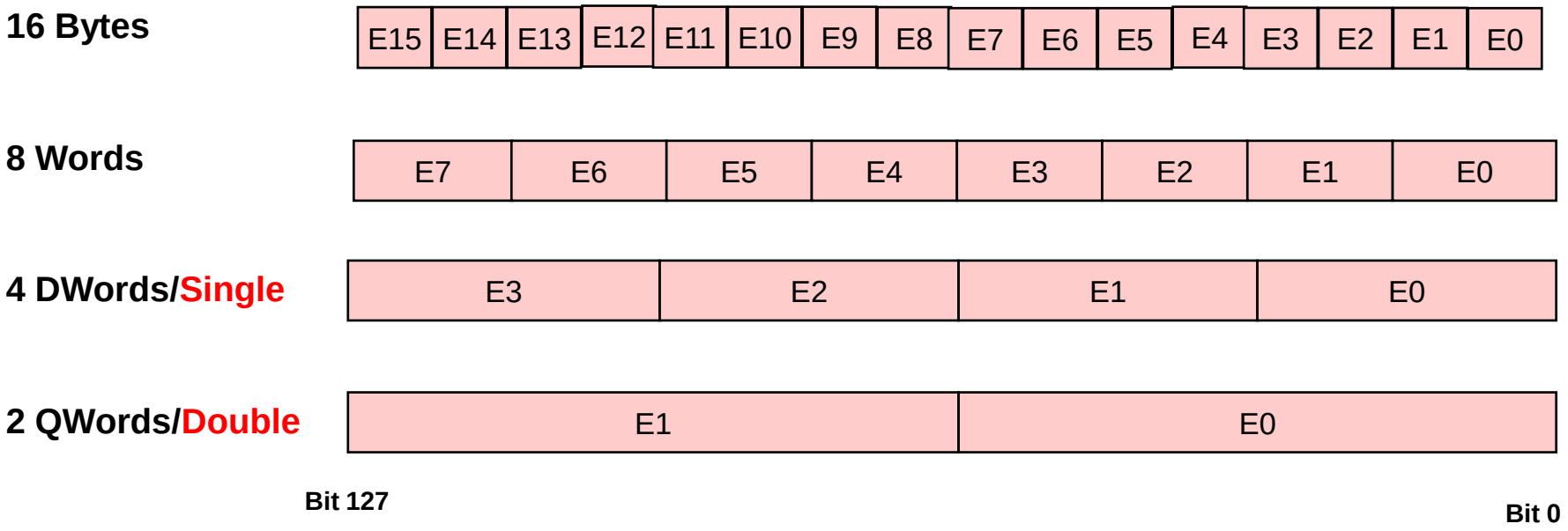
```
for (i = 0; i < rows; i++)
        for (j = 0; j < columns; j++)
                mymatrix [i] [j]   += increment;
```

- **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**

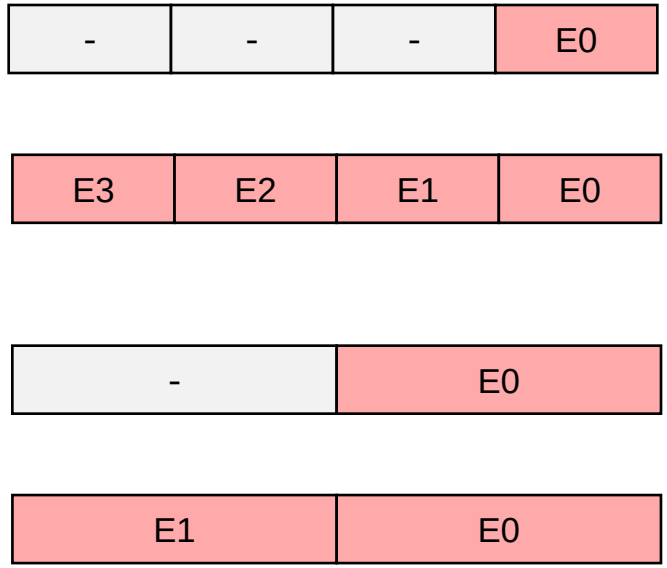


SSE = Streaming SIMD extensions

CERN School of Computing

Four floating-point data flavours

- **Single precision**
  - Scalar single (SS)
  - Packed single (PS)
- **Double precision**
  - Scalar Double (SD)
  - Packed Double (PD)



- **Note:**
  - 1) “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
    - “scalar” will mean 1/4 or 1/8 of peak!

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 CBT 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];
```

CERN School of Computing

HEP and vectors

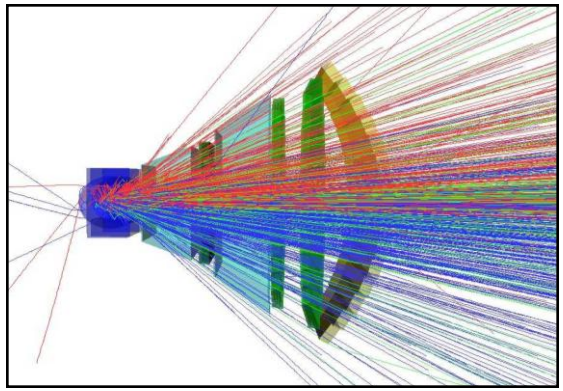
- **Very little common ground**
  - Too little?
  - But all attempts in the past failed!
    - w/CRAY, 3090-VF, etc.
- **From time to time, we stumble across a vector example**
  - My favorite example: Track Fitting code from ALICE trigger
    - See the next slide →
- **Other examples: Use of STL vectors**
- **Note that most compilers (try to) vectorize automatically**

Examples of parallelism: CBM track fitting

CERN School of Computing

I.Kisel/GSI: "Fast SIMDized Kalman filter based track fit"
http://www-linux.gsi.de/~ikisel/reco/CBM/DOC-2007-Mar-127-1.pdf

- **Extracted from CBM's High Level Trigger Code**
  - Originally ported to IBM's Cell processor
- **Tracing particles in a magnetic field**
  - Embarrassingly parallel code
- **Re-optimization on x86-64 systems**
  - Step 1: use SSE vectors instead of scalars
    - Operator overloading allows seamless change of data types, even between primitives (e.g. float) and classes
    - Classes + intrinsics
      - P4_F32vec4 – packed single; operator + = _mm_add_ps
        - F64vec4 operator +(const F64vec4 &a, const F64vec4 &b) { return _mm_add_ps(a,b); }



Important performance measurements

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

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

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
```

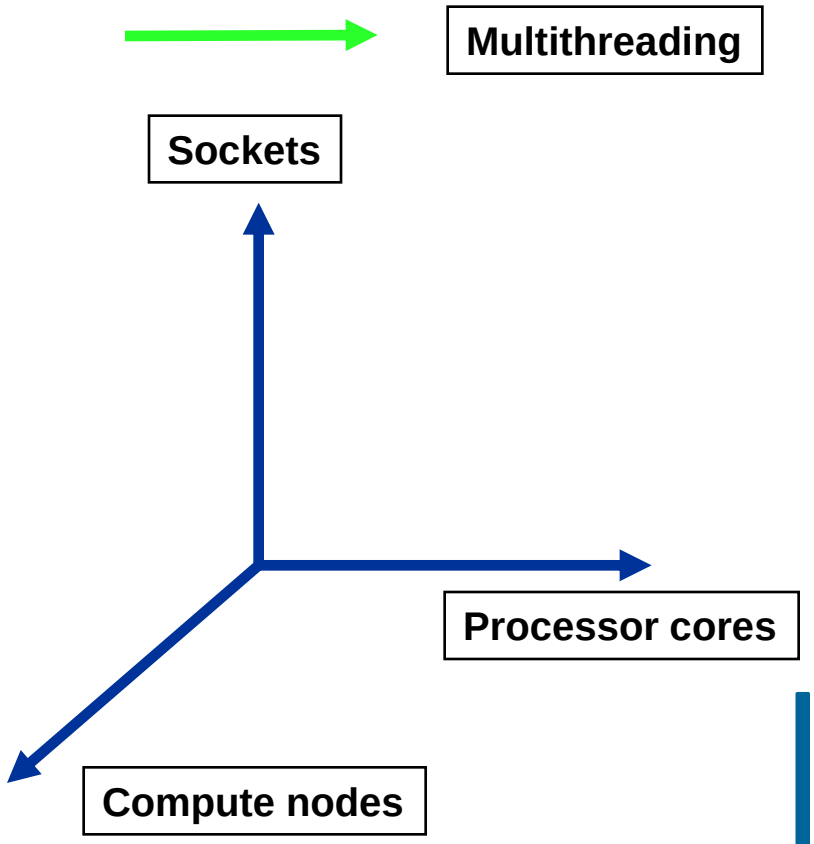
Same instructions laid out according to latencies on the Core 2 processor →

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

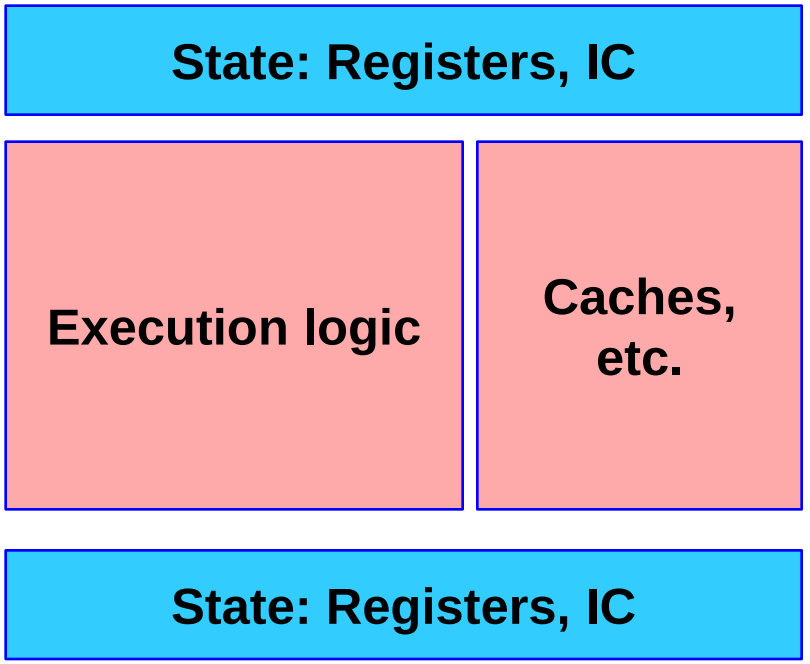
Part 2: Parallel execution across hw-threads and cores

- **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.

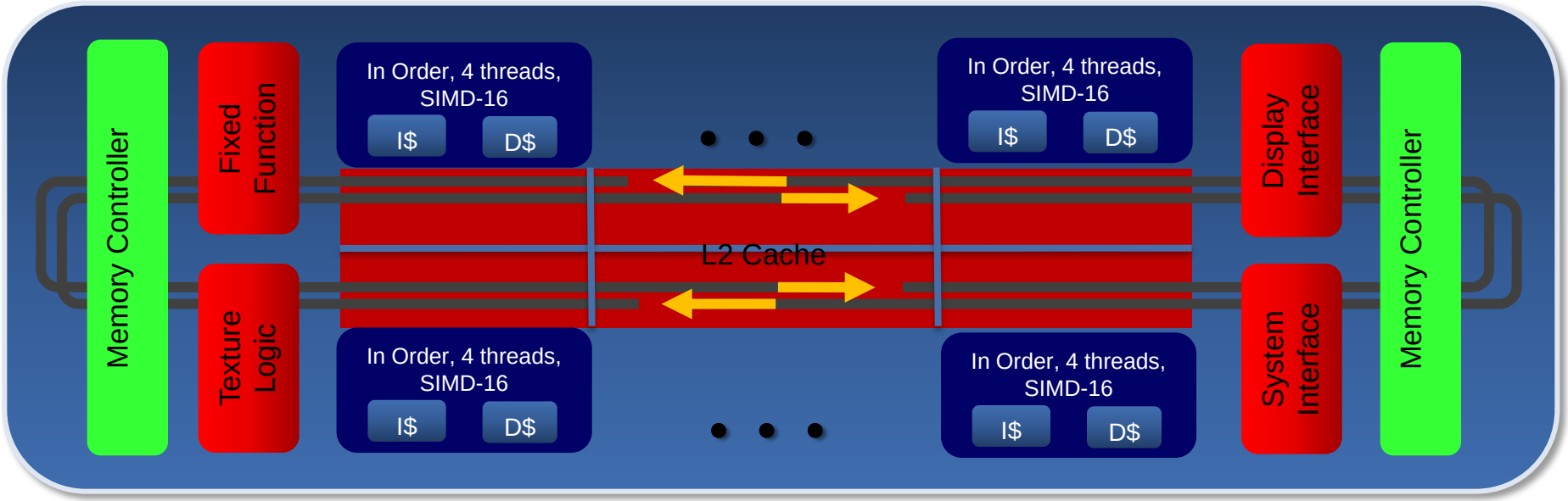
CERN School of Computing

The move to many-core systems

- **Examples of "dispatch slots": Sockets * Cores * HW-threads**
  - Conservative:
    - Dual-socket AMD quad-core (Barcelona): 2 * 4 * 1 = 8
    - Dual-socket Intel quad-core Nehalem: 2 * 4 * 2 = 16
    - Quad-socket Intel Dunnington server: 4 * 6 * 1 = 24
  - Aggressive:
    - Quad-socket Nehalem "octocore": 4 * 8 * 2 = 64
    - Quad-socket Sun Niagara (T2+) processors w/8 cores and 8 threads: 4 * 8 * 8 = 256
- **In the near future: Hundreds of dispatch slots**
- **And, by the time new software is ready: Thousands !!**

Many-core graphics processor

- **Intel's Larrabee:**
  - Already announced at SigGraph 2008!
  - Based on the x86 architecture
  - Many-core + 4-way multithreaded + 512-bit vector unit



- **Not forgetting offerings from NVidia, AMD, IBM, etc.**

CERN School of Computing

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.

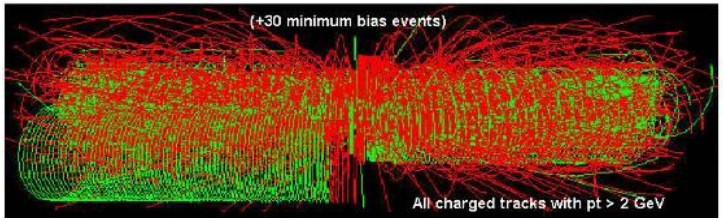
- **Thread:**
  - A process may have multiple threads of execution. These threads run in the same address space, share the same program code, the operating system resources as the process they belong to. Each thread gets its own stack.

Adapted from Wikipedia

CERN School of Computing

HEP programming paradigm

- **Event-level parallelism has been used for decades**
  - Compute one event after the other in a single process
- **Advantage:**
  - Large jobs can be split into N efficient processes, each responsible for processing M events
    - Built-in scalability



- **Disadvantage:**
  - Memory must be made available to each process
    - With 2 – 4 GB per process
    - A dual-socket server with Quad-core processors
      - Needs 16 – 32 GB (or more)

CERN School of Computing

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 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
   - Using coarse-grained (event-level?) parallelism

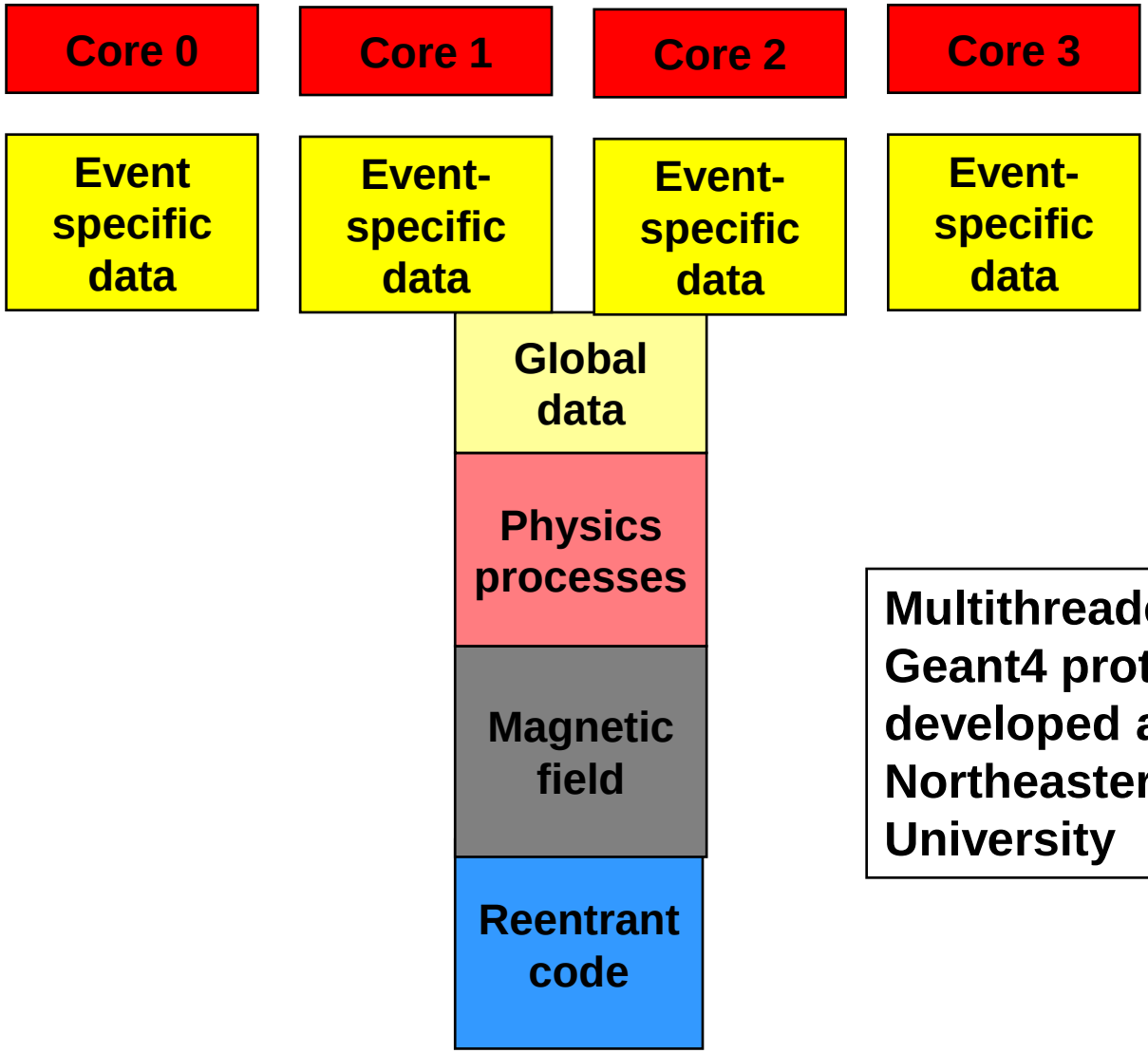
CERN School of Computing

Programming strategies/priorities

- **As I see them:**
  - Get memory usage (per process) under control
    - To allow higher multiprogramming level per server
  - Draw maximum benefit from hardware threading
  - Introduce coarse-grained software multithreading
    - To allow further scaling with large core counts
  - Revisit data parallel constructs at the very base
    - Gain performance inside each core
- **In all cases, use appropriate tools:**
  - To monitor detailed program behaviour
    - Both correctness and performance

Achieving an efficient memory footprint

- As follows:



Multithreaded Geant4 prototype developed at Northeastern University

HEP and Symmetric Multi-Threading

- **Because we have "thin" instruction streams, we could profit from SMT, provided the memory issue is under control**
  - We could easily tolerate 2 – 4 hardware threads!

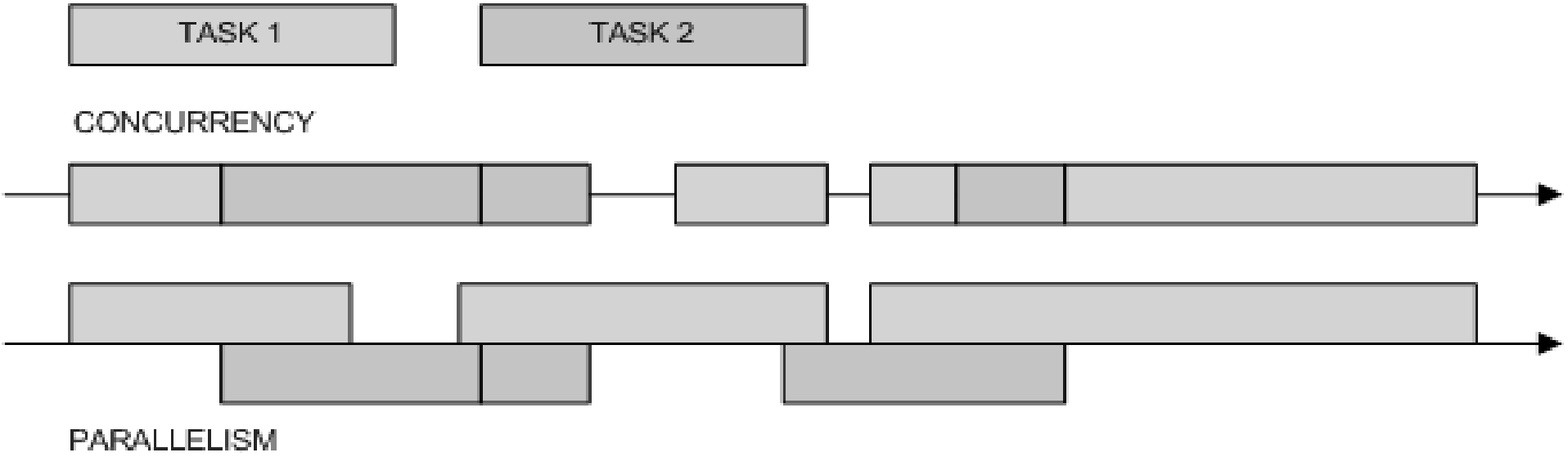
Cycle	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
1			load point[0]			
2			load origin[0]			
3						
4						
5						
6		subsd	load float-packet			
7						
8			load xhalfsz			
9						
10	andpd					
11						
12	comisd					
13						jbe

CERN School of Computing

Let’s look more closely at 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



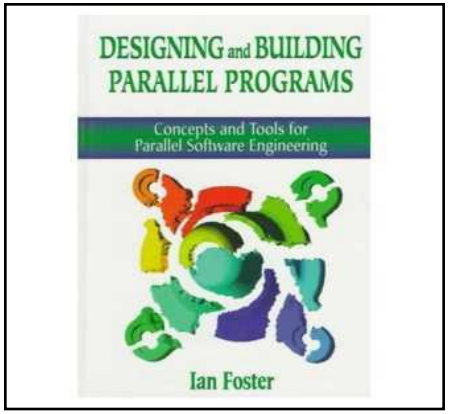
CERN School of Computing

From Concurrency to Parallel Execution

- **Multiple steps must be kept in mind:**
  - Concurrency
  - Decomposition
  - Communication
  - Synchronization
  - Mapping
  - Execution
- **Keeping Amdahl's law in mind**

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

Foster's Design Methodology

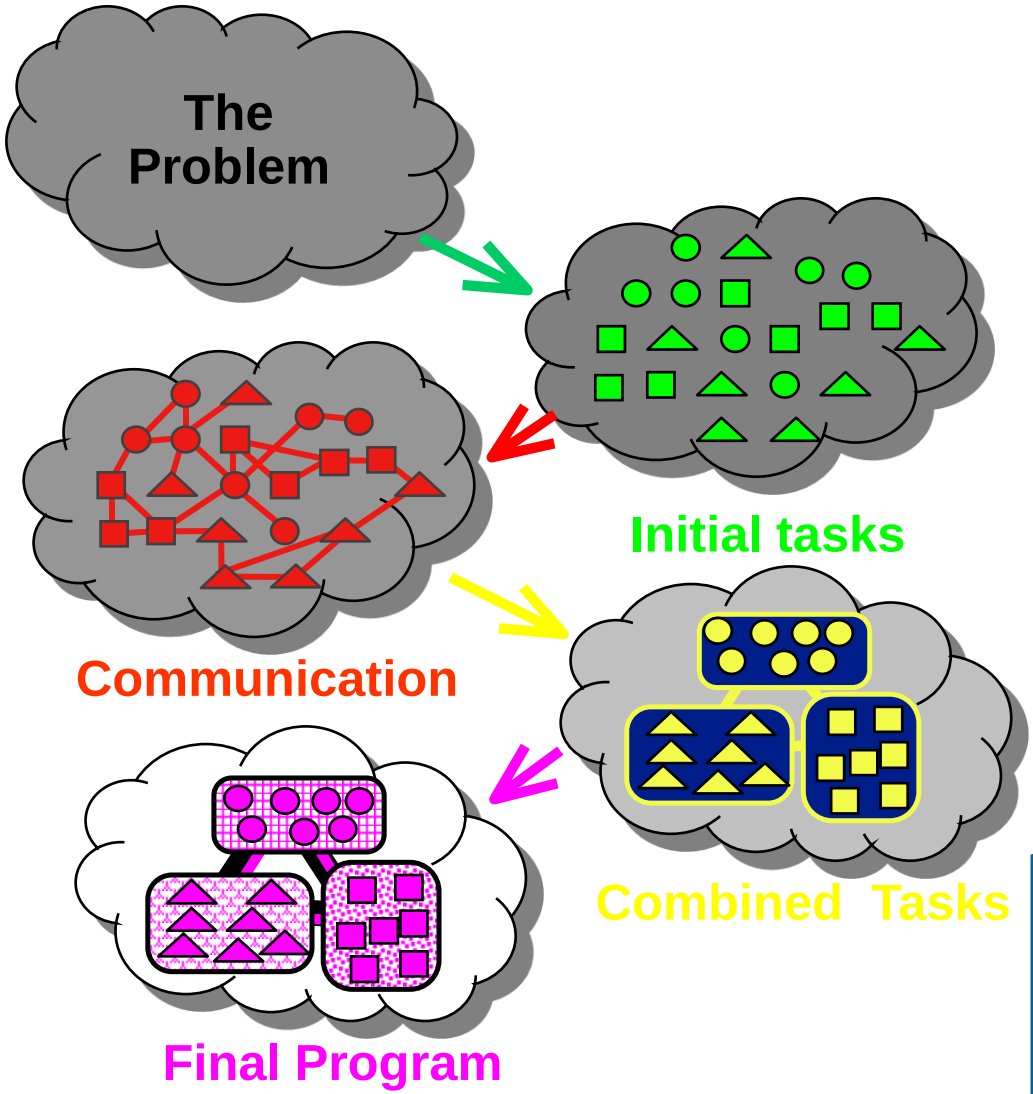


- **Four Steps:**
  - **Partitioning**
    - Dividing computation and data
  - **Communication**
    - Sharing data between computations
  - **Agglomeration**
    - Grouping tasks to improve performance
  - **Mapping**
    - Assigning tasks to processors/threads

CERN School of Computing

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



CERN School of Computing

More on decomposition

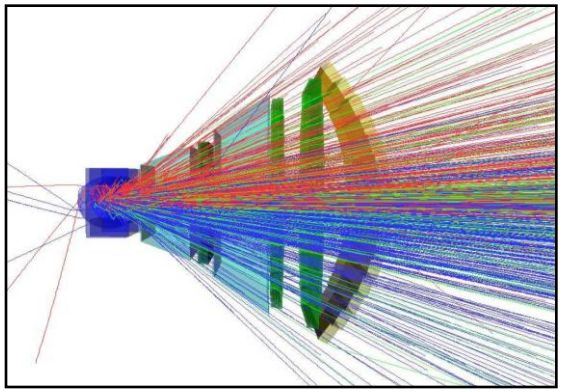
- **Divide the total work into smaller parts,**
  - Which can be executed concurrently
- **Some techniques:**
  - Data decomposition
    - Partition the data domain
  - Task/functional decomposition
    - Split according to “logical” tasks/functions
  - Recursive decomposition
    - Divide-and-conquer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

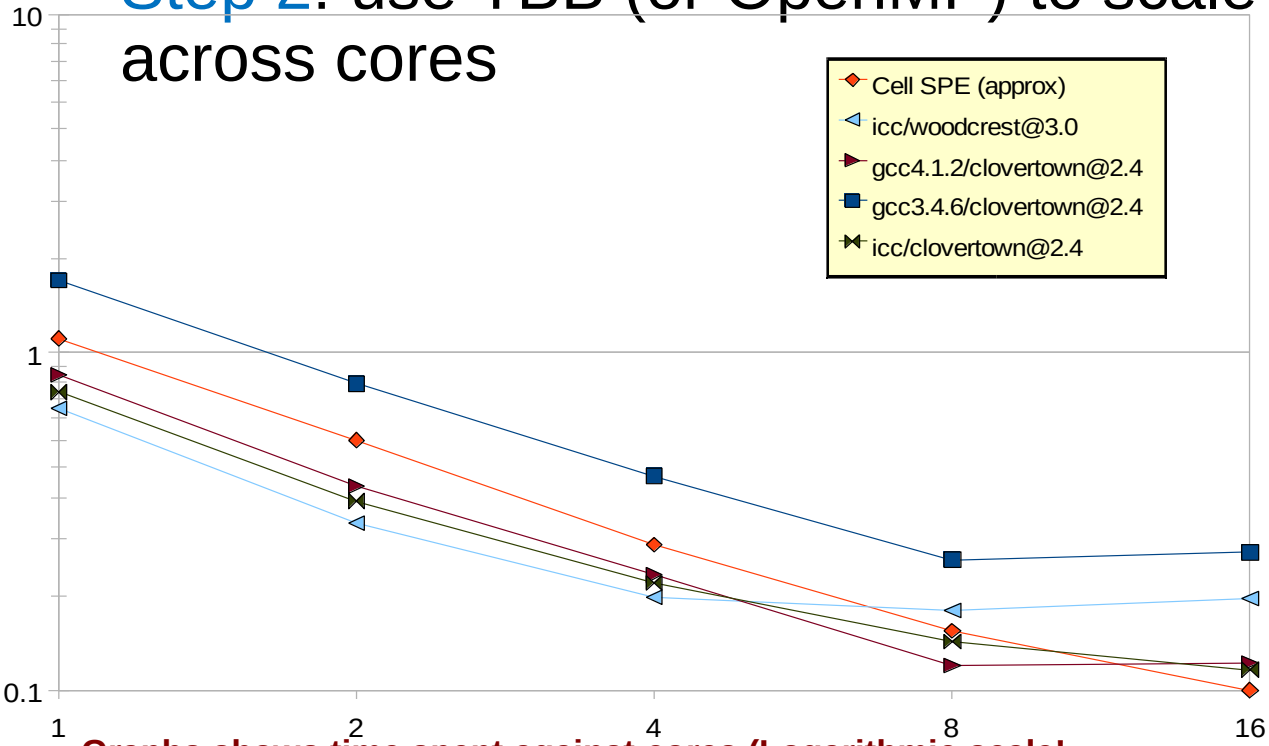
C++ parallelization support

- **Parallelization is not defined inside the language itself**
- **Large selection of low-level tools:**
  - Native: pthreads/Windows threads
  - OpenMP
  - Intel Threading Building Blocks (TBB)
  - OpenCL (www.khronos.org/opencl)
  - CILK++ (www.cilk.com)
  - RapidMind (www.rapidmind.com)
  - TOP-C (from NE University)
  - Ct (in preparation from Intel)
  - MPI, etc.

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



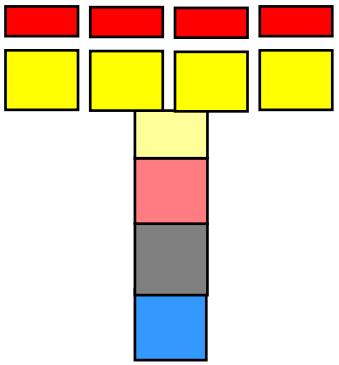


Graphs shows time spent against cores (Logarithmic scale!

From H.Bjerke/CERN openlab

Examples of parallelism: GEANT4

- **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 example
  - Required change of lots of existing classes:
    - Especially *global*, "*extrn*", and *static* declarations
  - First, the geometry was converted
  - Then, the physics tables
    - Ion tables are still shared (protected by locking)

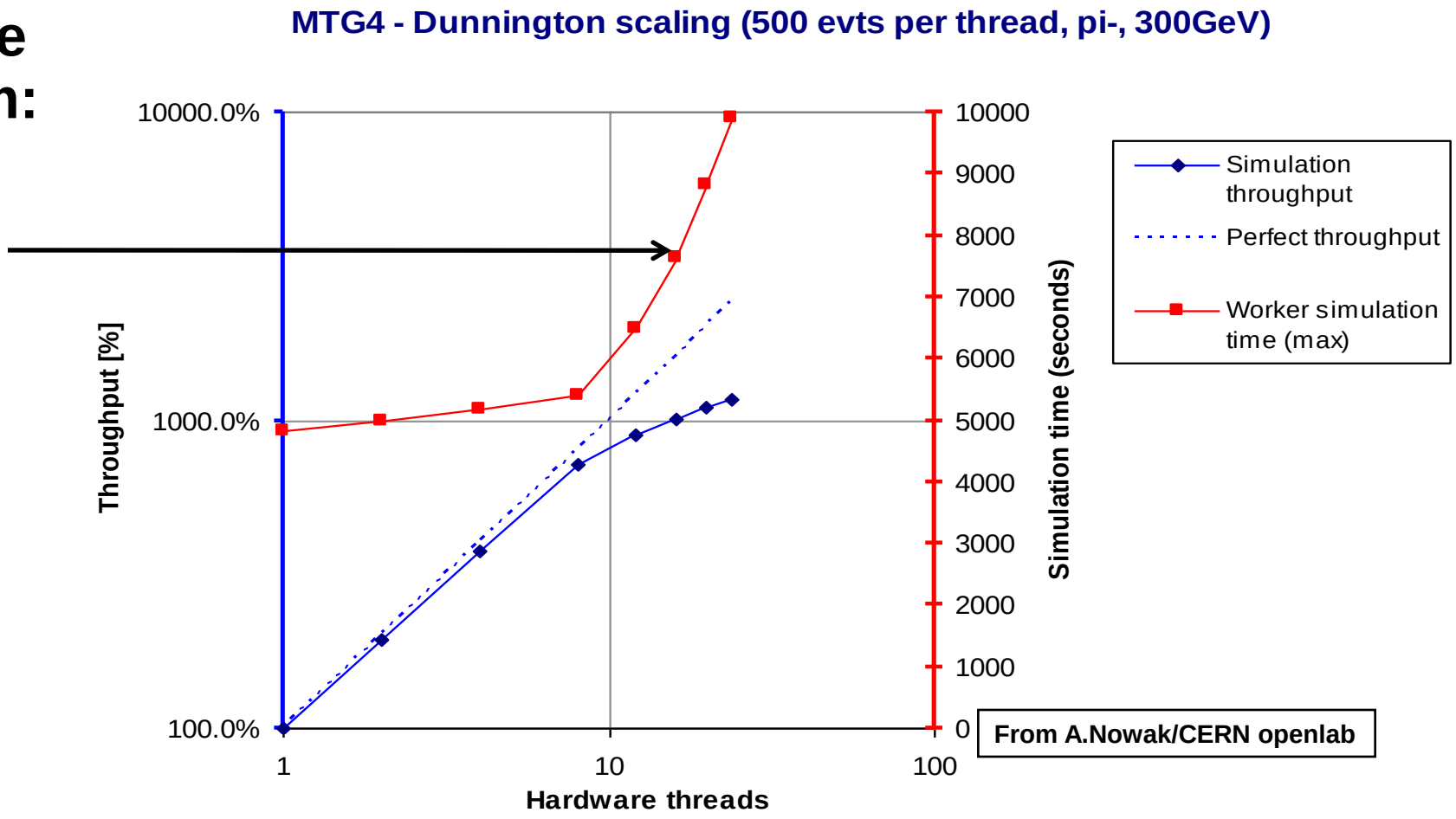


- **Additional memory: Only 22MB/thread (!)**

MTG4/FullCMS measurements

- **Using a 24-core system:**

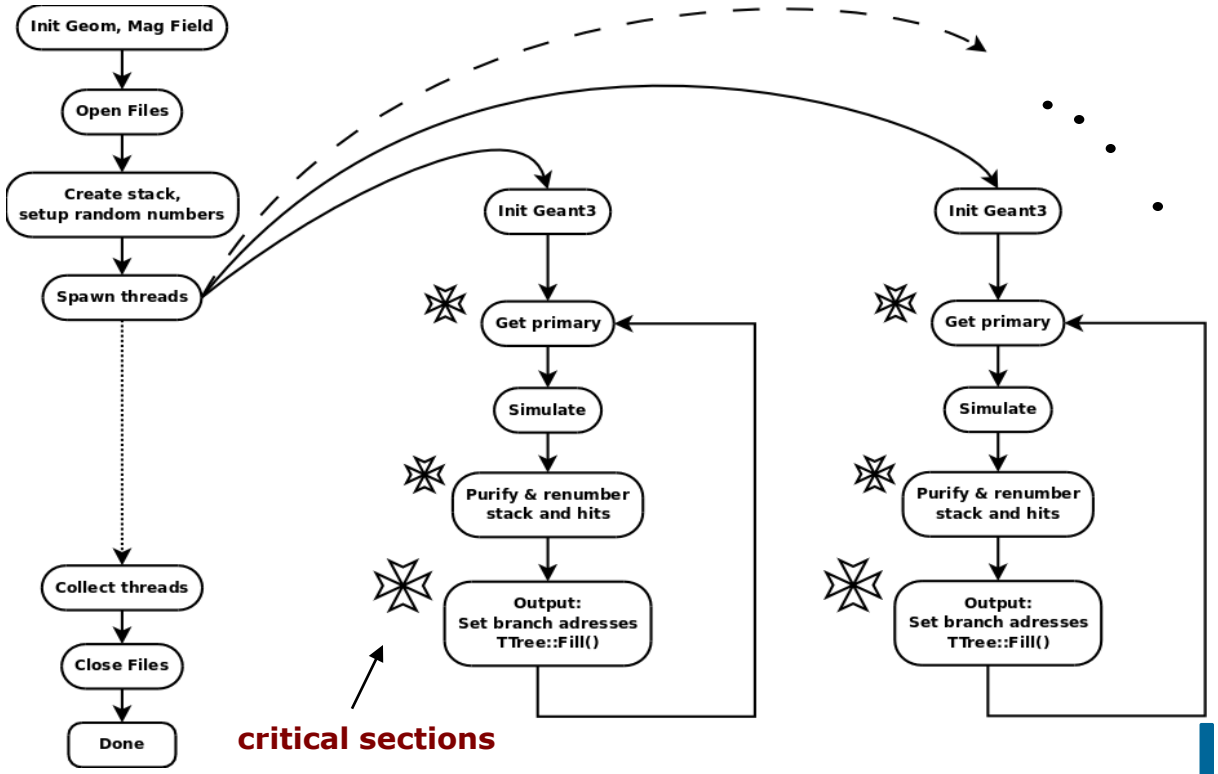
Red curve should have been flat!



More work is needed, but extremely interesting first step!

Multithreaded ALICE simulation

- **Another very interesting prototype**
  - Track level parallelism
- **Simulation of a Pb-Pb event (no output)**
  - 65k primary tracks (!)
  - 5 h CPU time (!)
- **With G4-VMC/Example03**
  - 3.92x speedup
  - w/4 core AMD system
  - 35MB additional per thread

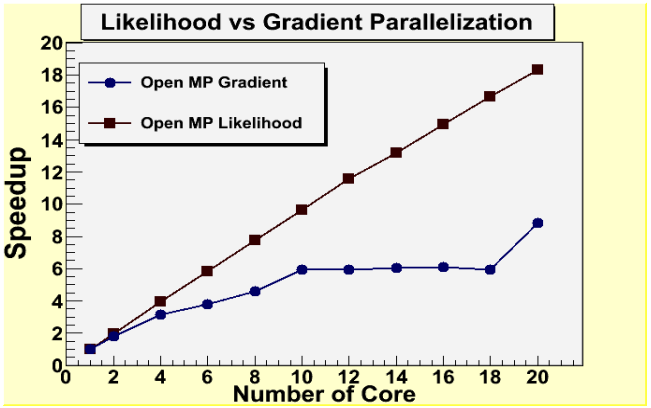


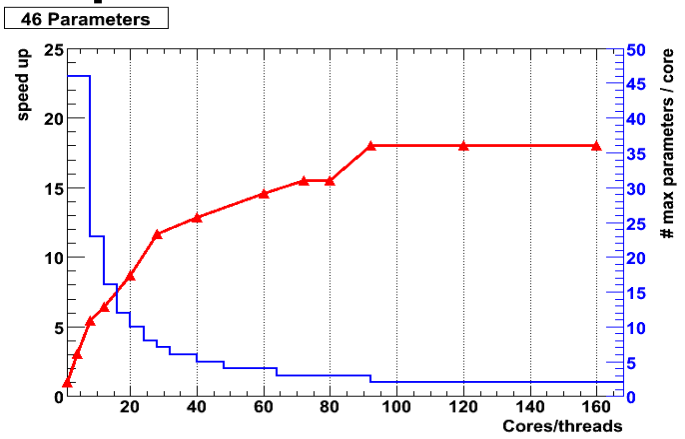
CERN School of Computing

Example: ROOT minimization and fitting

- **Minuit parallelization is independent of user code**
- **Log-likelihood parallelization (splitting the sum) is more efficient**
  - more demanding on thread safety of provided code
- **Example: unbinned fit with 20 parameters**

$$\ln L = \sum_i \ln f(x_i, \theta)$$





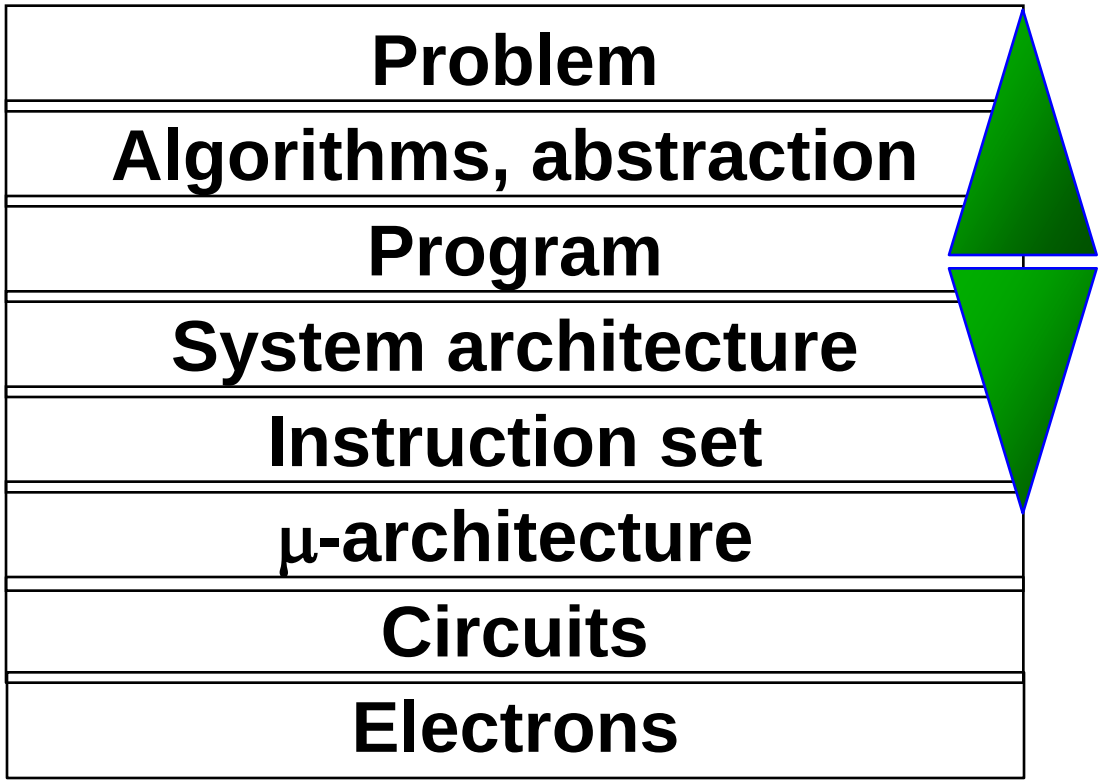
complex BaBar fitting provided by A. Lazzaro and parallelized using MPI

- **Can have combination on both**
  - parallelization via multi-threading in a multi-core CPU
  - multiple process in a distributed computing environment

Code is now available as of ROOT version 5.22

Back to our Complicated Story

- **In these lectures, we tried to move into several layers**
  - Avoiding being “boxed in” !



CERN School of Computing

Concluding remarks

- **The aim of these lectures was to help understand**
  - Changes in modern computer architecture
  - mpact 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?**
- **Learn to master the seven hardware dimensions!**

CERN School of Computing

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

CERN School of Computing

BACKUP

Items not covered today

- **Systematic tuning approach**
- **Performance tuning versus correctness**
  - FP accuracy and reproducibility
- **Amdahl's law (in detail)**
  - Also: Gustafson's law
- **Emerging parallel programming languages**
- **Detailed compiler "control"**
  - Including regression avoidance

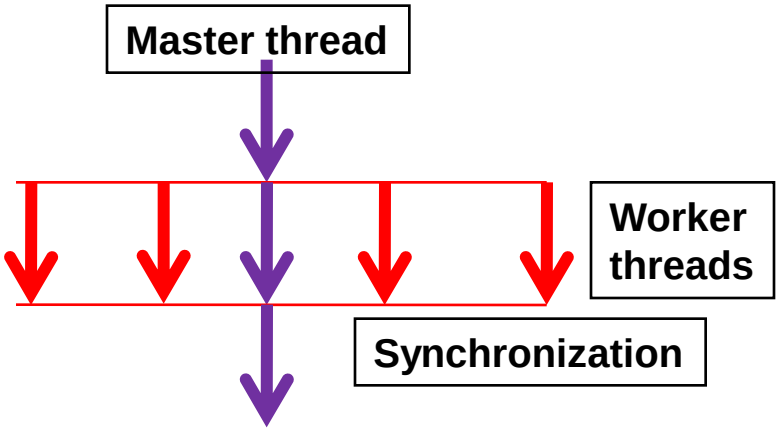
CERN School of Computing

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
```



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

CERN School of Computing

Intel TBB 2.0 overview

- **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

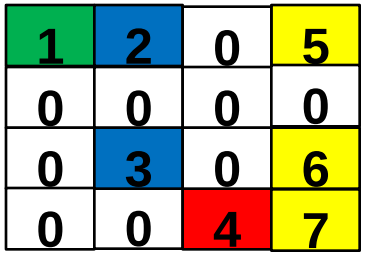
```
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
using namespace tbb;
//
task_scheduler_init init;
  tasks = atoi( argv[1] );
//
 parallel_for(blocked_range<int>(0,
NTracksV, NTracksV / tasks),
ApplyFit(TracksV, vStations, NStations));
```

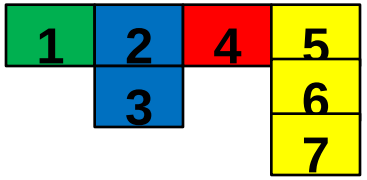
More features in preparation

Ct Language

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**
- **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





TOP-C Overview

http://www.ccs.neu.edu/home/gene/topc.html

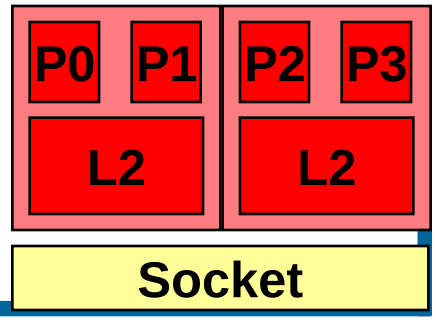
- **Task-oriented Parallel C/C++**
  - Runs on top of most UNIX/Linux flavours
  - Its programming model is based on three key concepts:
    - *tasks* in the context of a master/slave architecture
    - global *shared data* with lazy updates
    - *actions* to be taken after each task
  - Provides a single API to support three primary memory models:
    - *distributed memory*
    - *shared memory*
    - *sequential memory*
      - a single sequential, non-parallel process.

Intel CPU parameters

- **Core 2 processor (Clovertown)**

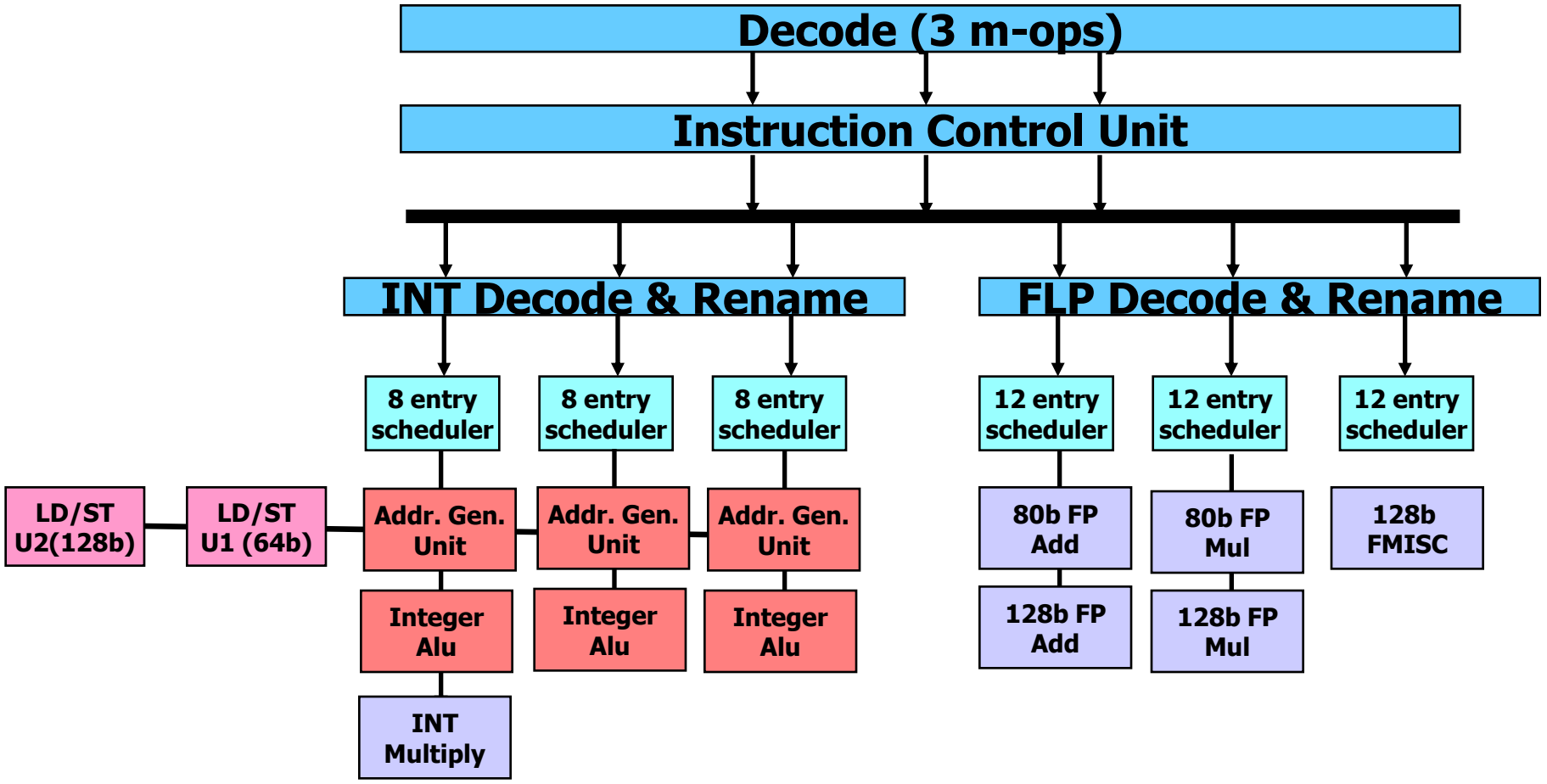
Caches/TLB	Size (total / line)	Access (cycles)	Porting	Associativity (N-ways)
L1I	32 KB / 64 B	-		8-way
L1D	32 KB / 64 B	3	dual	8-way
L2 (semi-shared)	2 * 4 MB / 64 B	14		16-way
ITLB0 entries	128	-		-
DTLB0 entries	16	-		4-way
DTLB1 (4K pages)	256	2		4-way

Instruction issue	4 * 4 μ-ops
CPU speed	3.0 GHz
Bus speed	1333 * 8 B



AMD micro-architecture

- **Execution units in the Barcelona processor:**



AMD CPU parameters

- **Barcelona processor:**

Caches/TLB	Size (total / line)	Access (cycles)	Porting	Associativity (N-ways)
L1I	64 KB /64B			2-way
L1D	64 KB	3	dual	2-way
L2	512 KB	12		16-way
L3 (shared)	2 MB	<38		32-way
L1-ITLB entries	48			fully
L2-ITLB entries	512			-
L1-DTLB entries	48			fully
L2-DTLB entries	512			

Instruction issue	4 * 3 μ-ops
CPU speed	2.0 GHz
Bus speed	2 * 8 * 667 MB/s
HyperTransport	2 * 8 * 2 GB/s