Many-core computing in HEP

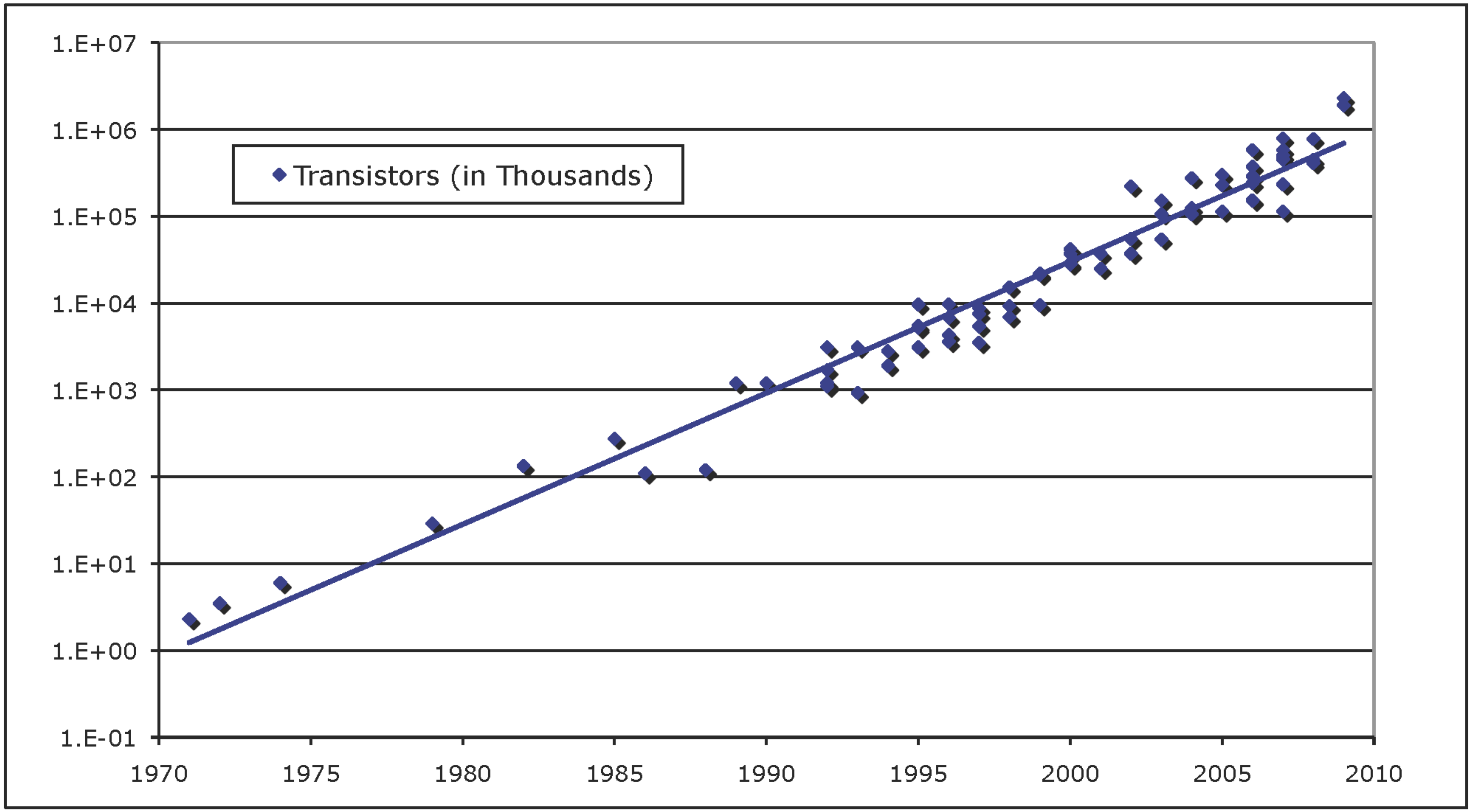
Benedikt Hegner
CERN

Outline

- Many cores and parallelism
  ('Power Wall')
- New Design Paradigm for HEP
- Memory Speed and Bad Programming
  ('Memory Wall')
- Summary

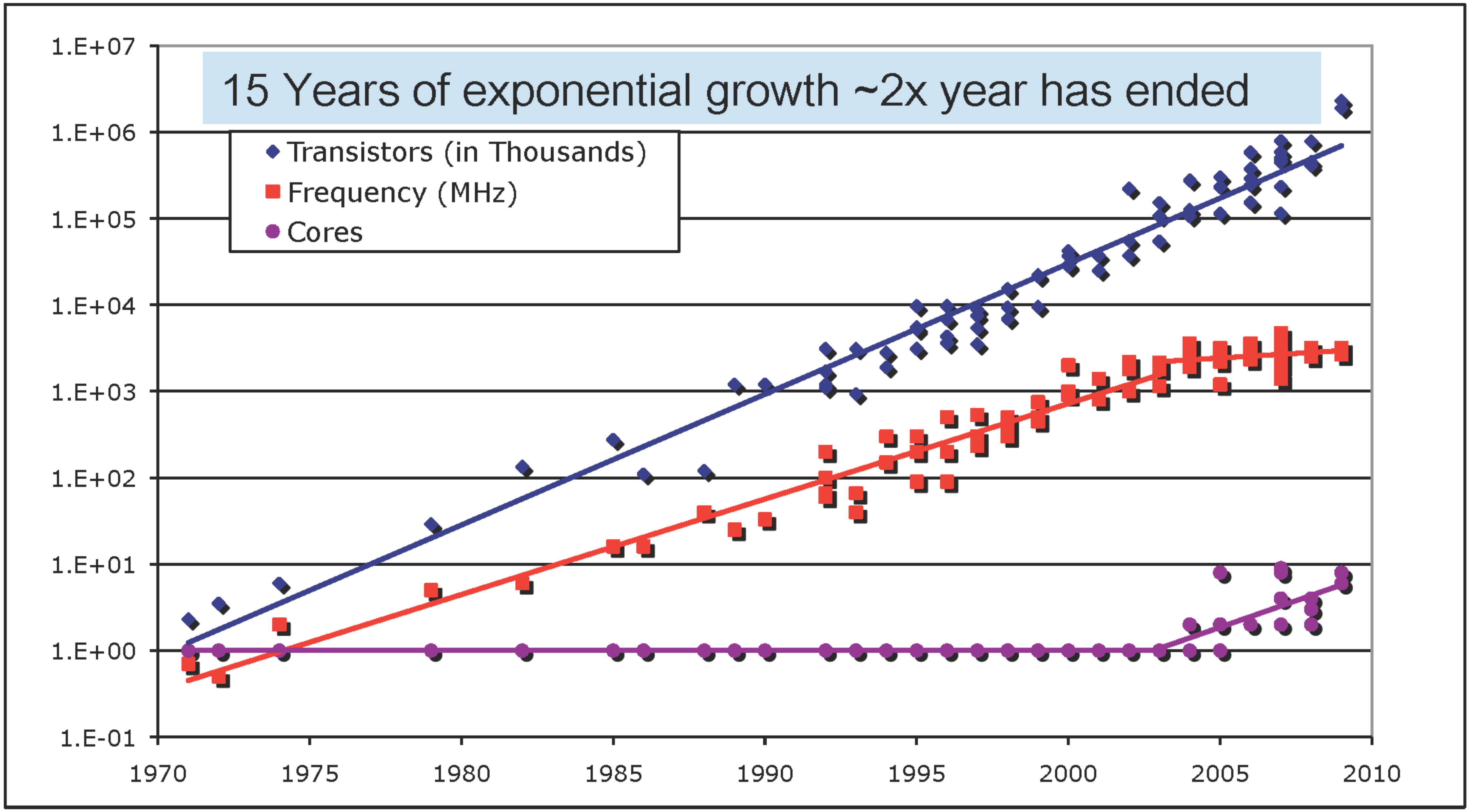
The ‘Power Wall’

Moore's law alive and well?



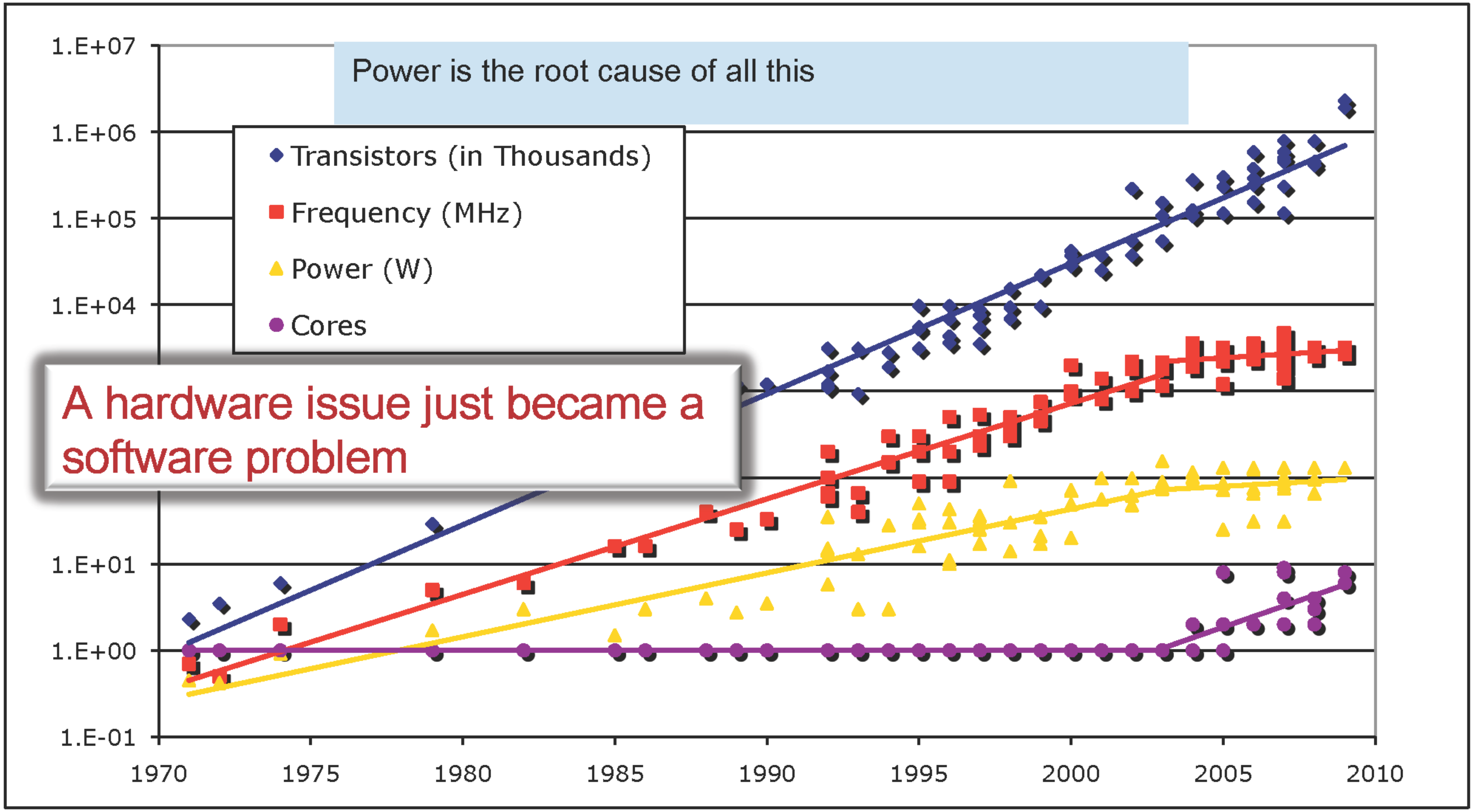
Moore's law alive and well?

...but clock frequency scaling replaced by cores/chip



Moore's law alive and well?

The reason is that we can't afford more power consumption



Moore's Law reinterpreted

- Number of cores per chip will double every two years
- Instruction parallelization (vectorization) increases
- Clock speed will not increase (or even decrease) because of Power consumption:

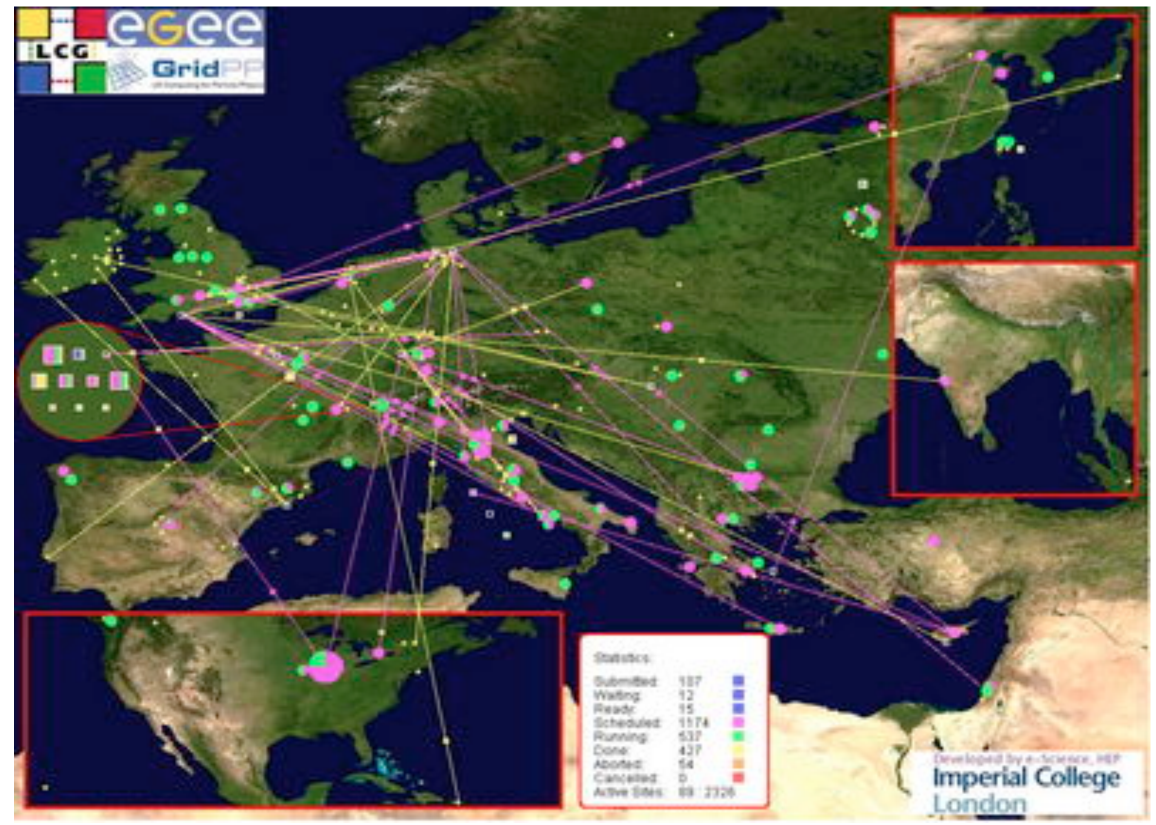
$$Power \propto Frequency^3$$

- Need to deal with systems of tons of concurrent threads and calculations
- In GPUs that's reality already now
  - We can learn a lot from game programmers! (*)

(*) thanks to all of you who fund their "research" by playing during office times ;-)

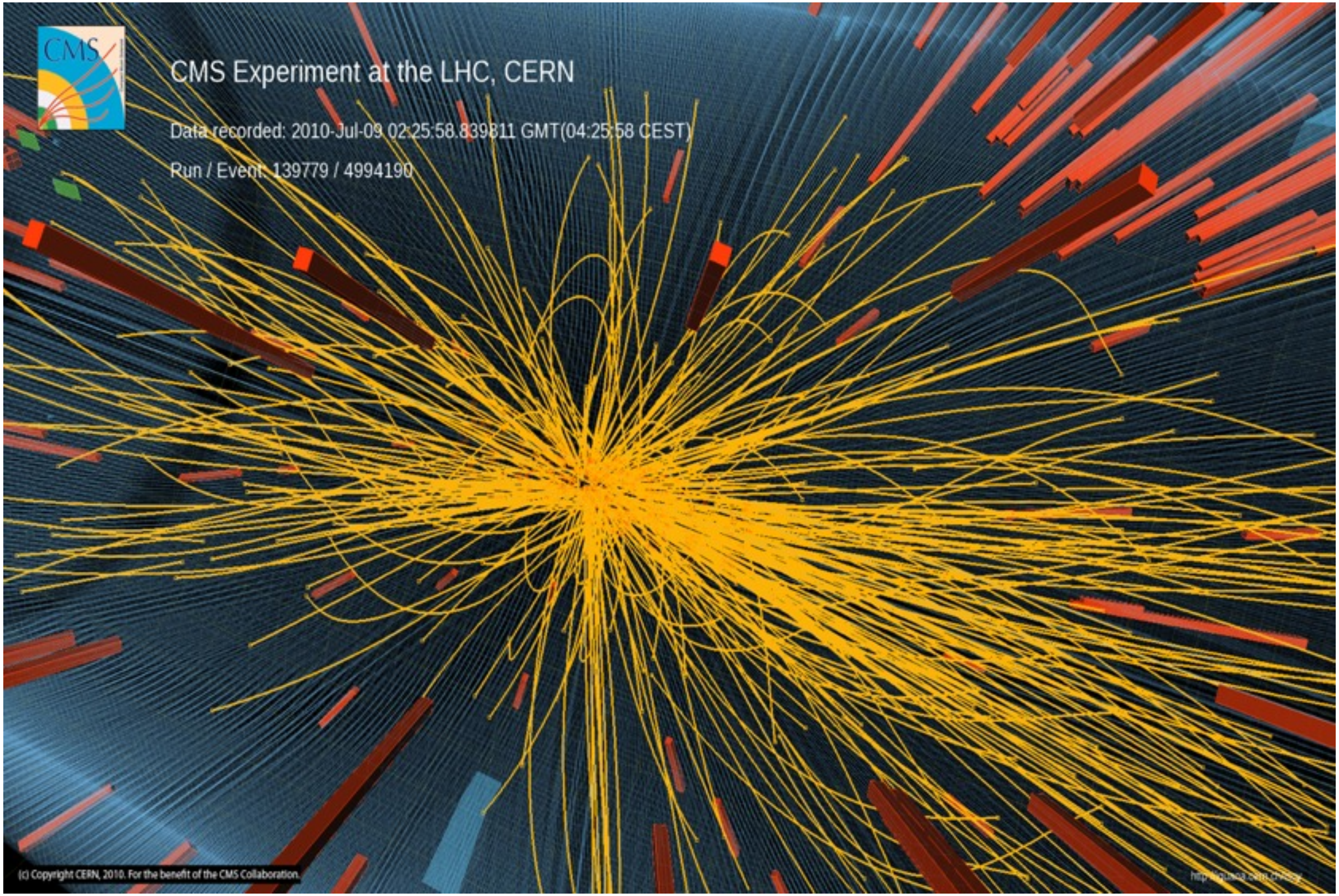
"Easiest" Way of Parallelization

- HEP is parallel since ~a decade!
  - "embarrassingly" parallel
- LHC Computing Grid
  - Processing tens of billions of LHC events/year
  - Running 24/7 365 days a year
  - Largest parallel application ever!



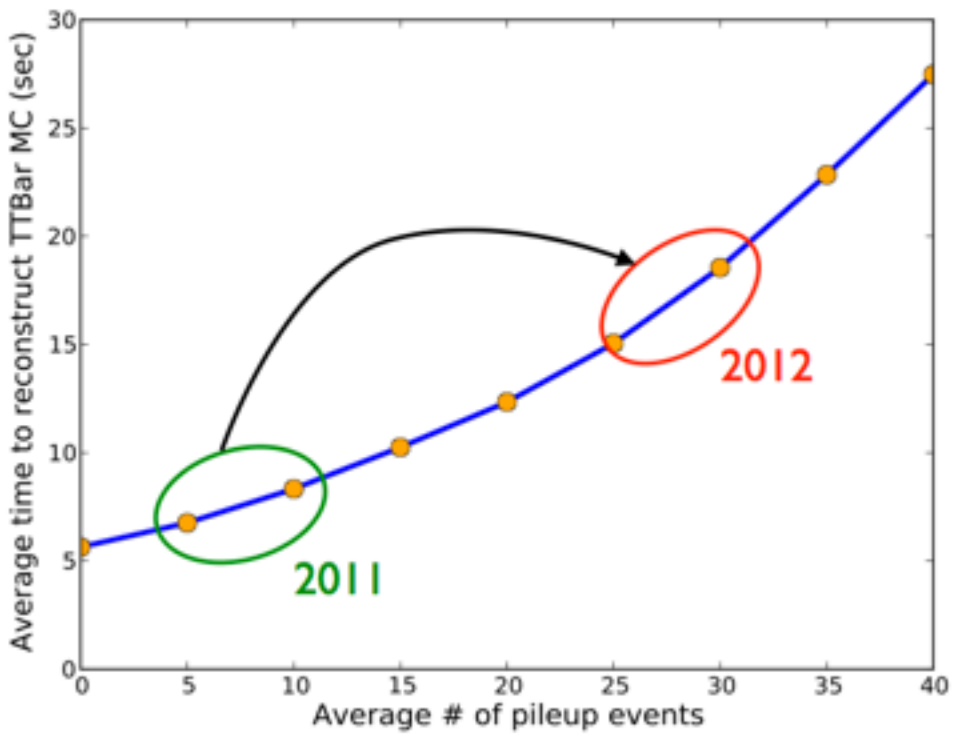
So why not treating every many-core computer as a computing centre of its own with many independent jobs on it?

Physics Challenges

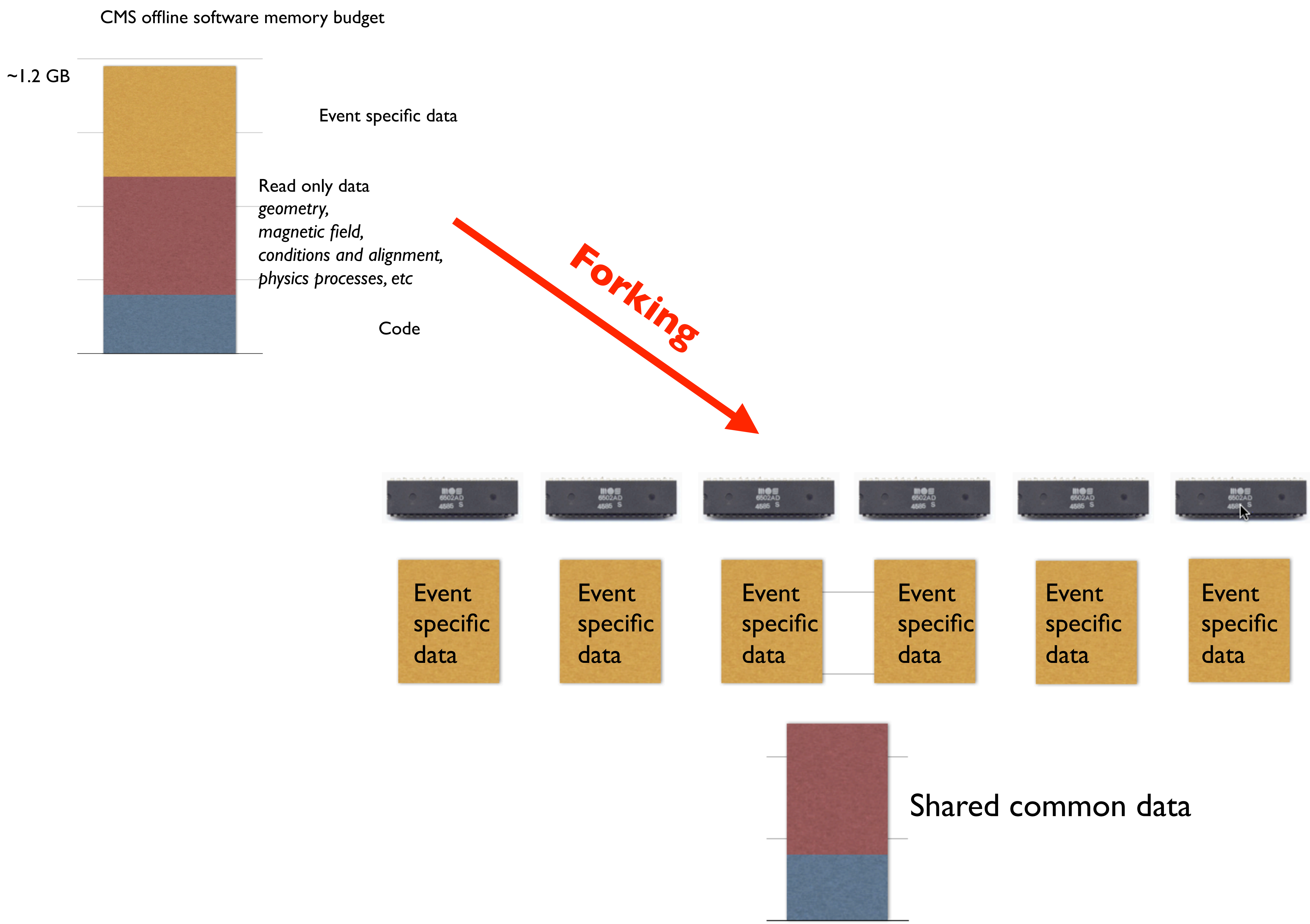


Physics Challenges

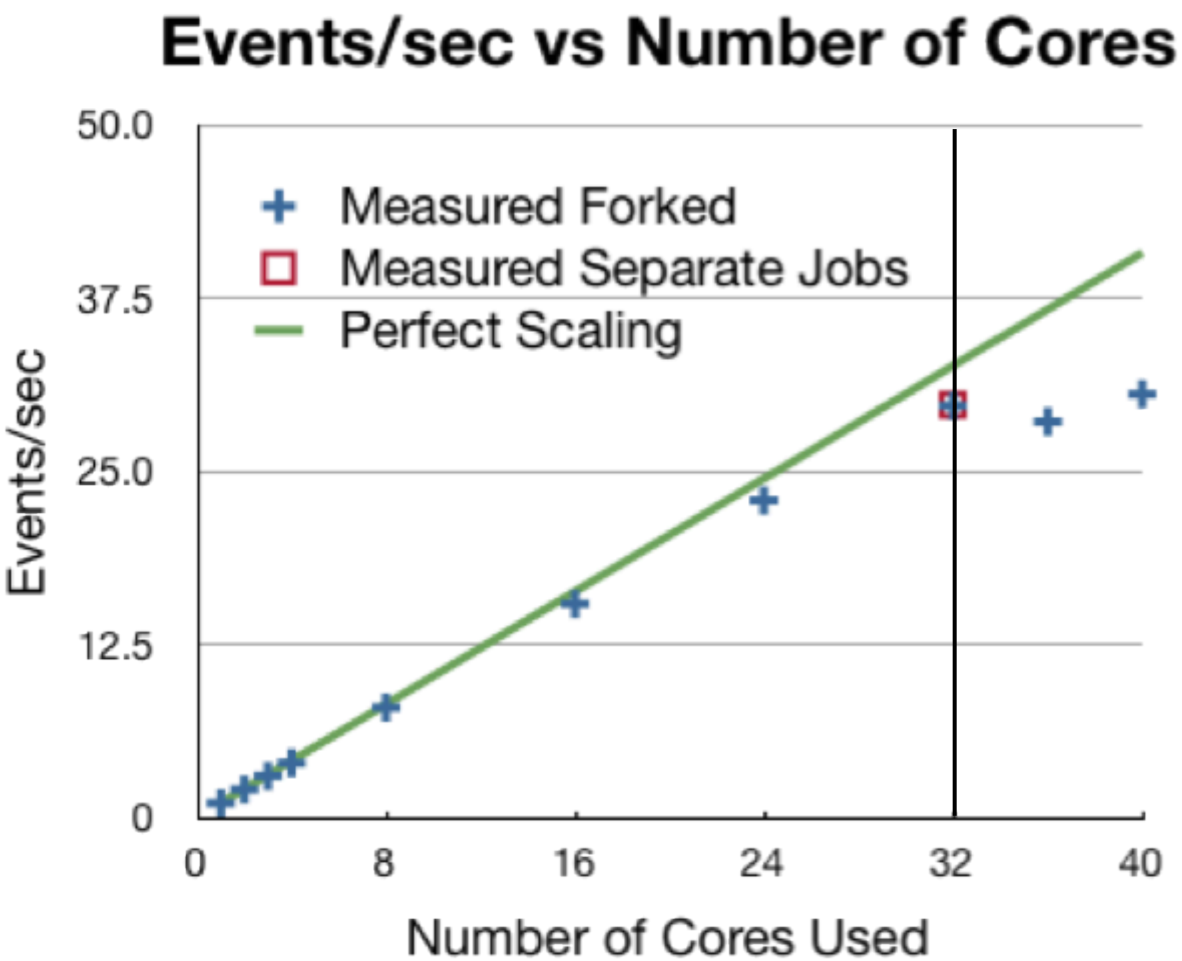
- Due to the beam intensity ("luminosity") at the LHC multiple proton-proton collisions take place at once ("pile-up")
- Experiment's reconstruction takes up to 4 GB of memory per job
- This is expected to increase further
- Running multiple jobs on a computer is not really an option
- Furthermore:
  - **Independent jobs give no handle on cache optimized parallelization**
  - **Merging of results of independent jobs takes significant amount of time**

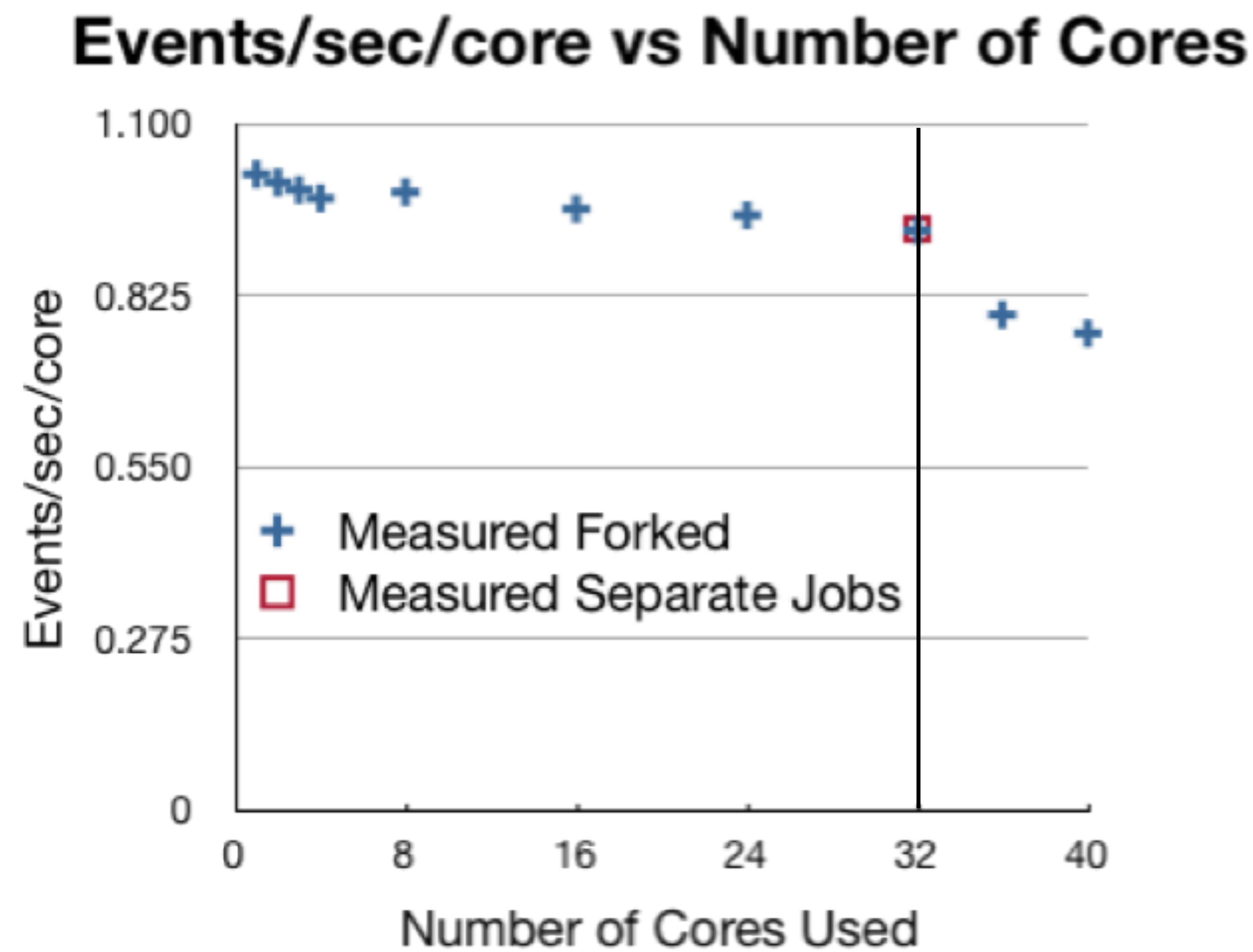


Stop-Gap solutions



Stop-Gap Solutions II





Problem for Grid infrastructure:

"Whole-node scheduling"

+

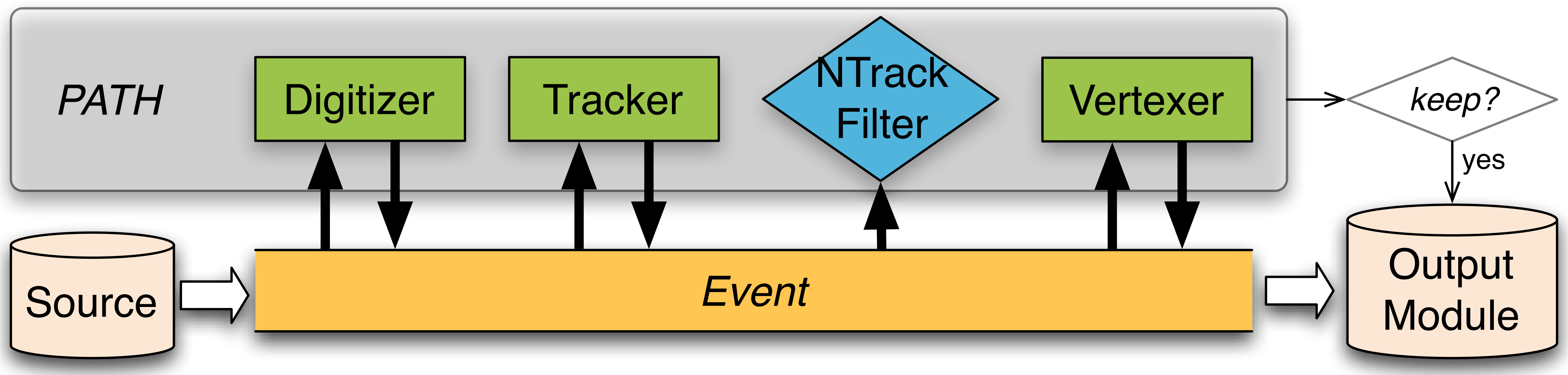
merging

We need to enable the application itself for parallelism

But how to make such huge code bases thread-safe?

Framework Primer

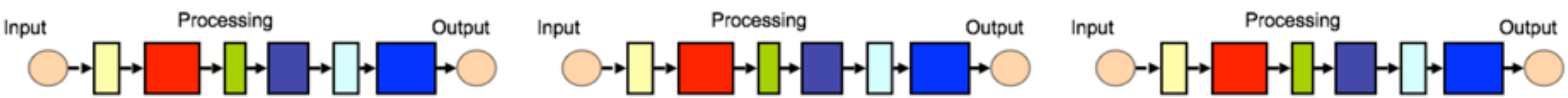
Experiment software follows the concept of a ‘software bus’



Each LHC experiment has software with about 5 million lines of code based on this model

Framework Primer II

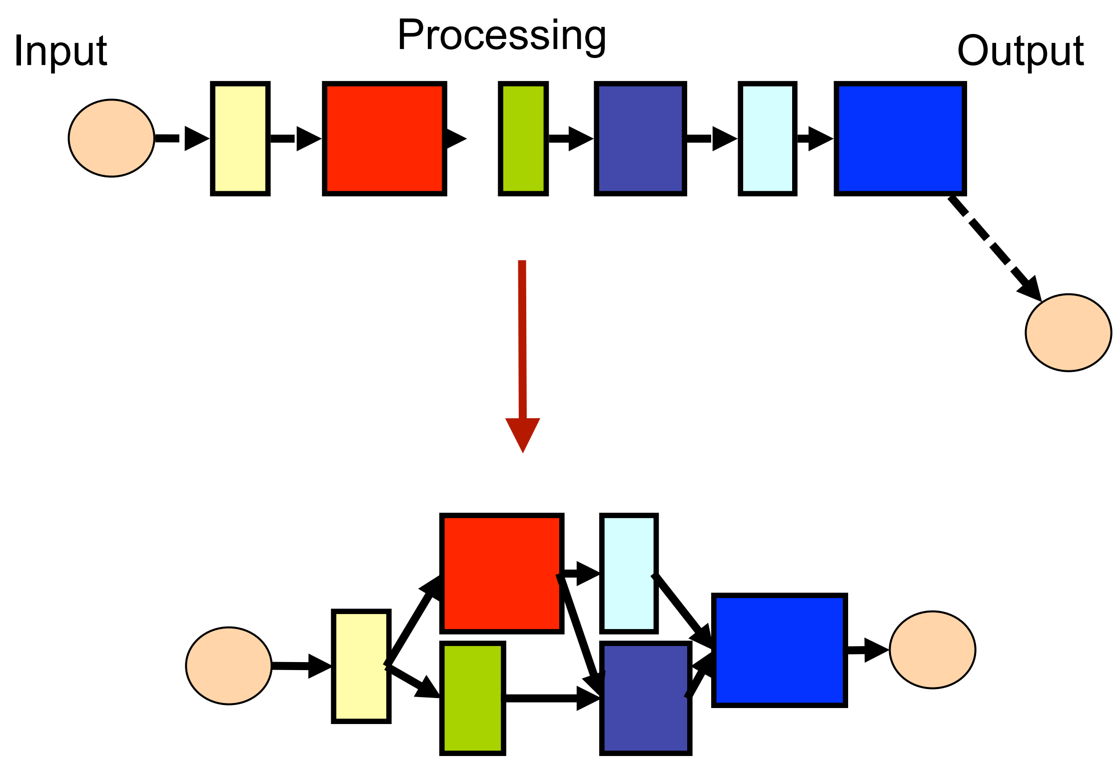
- Multiple events are being processed sequentially



- The result is the being put into a single output file
- This keeps only one core busy at a time

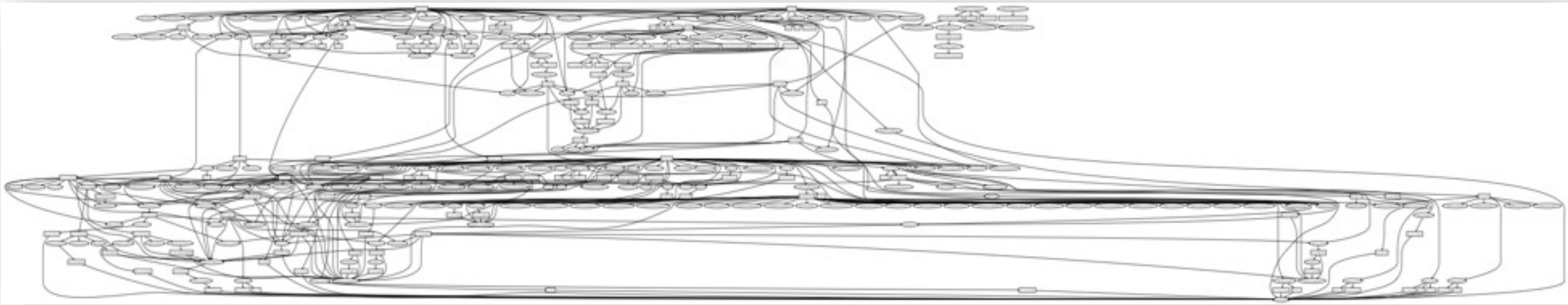
How to introduce Concurrency

- Our software is great candidate for ***task based parallelism***
- The algorithms and their data dependencies form a DAG (directed acyclic graph)
  - Schedule the algorithms according to the DAG

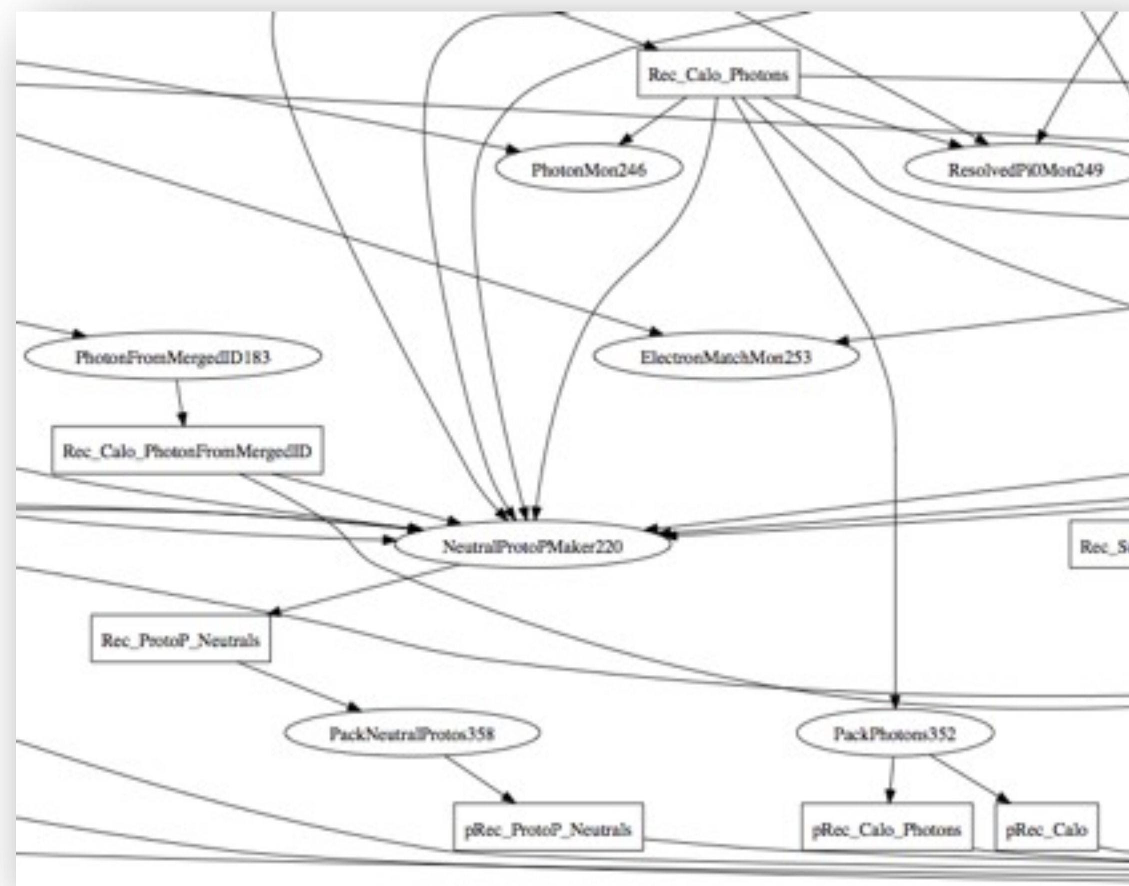


- Sounds more trivial than it is!
  - Existing HEP software has many "back-door" communication channels making the DAG non-obvious.
  - The software-bus and infrastructure need to be made thread-safe

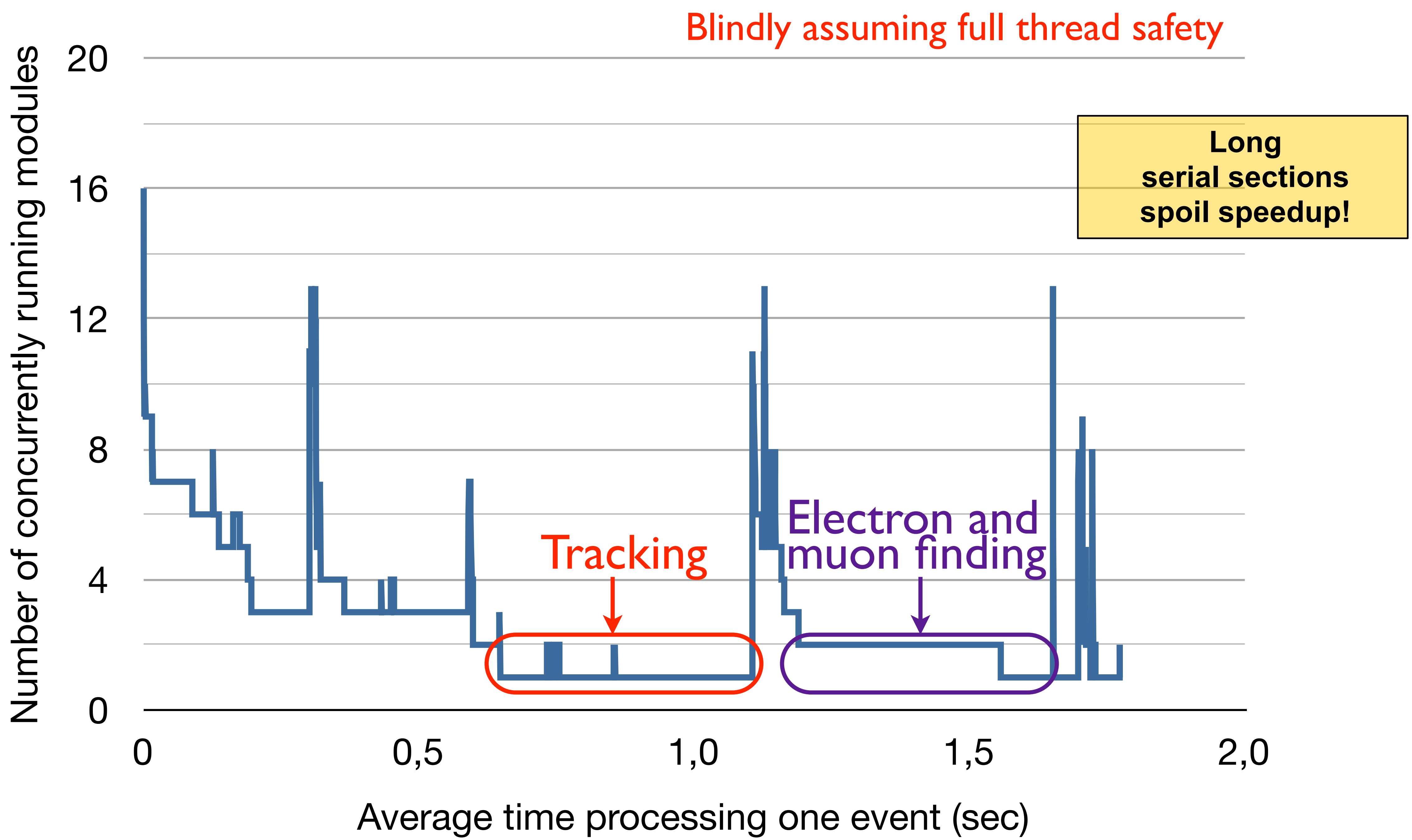
Real-world example



- Particular example taken from LHCb reconstruction program Brunel
- Gives an idea for the potential concurrency
- ATLAS or CMS just don't fit on a slide...

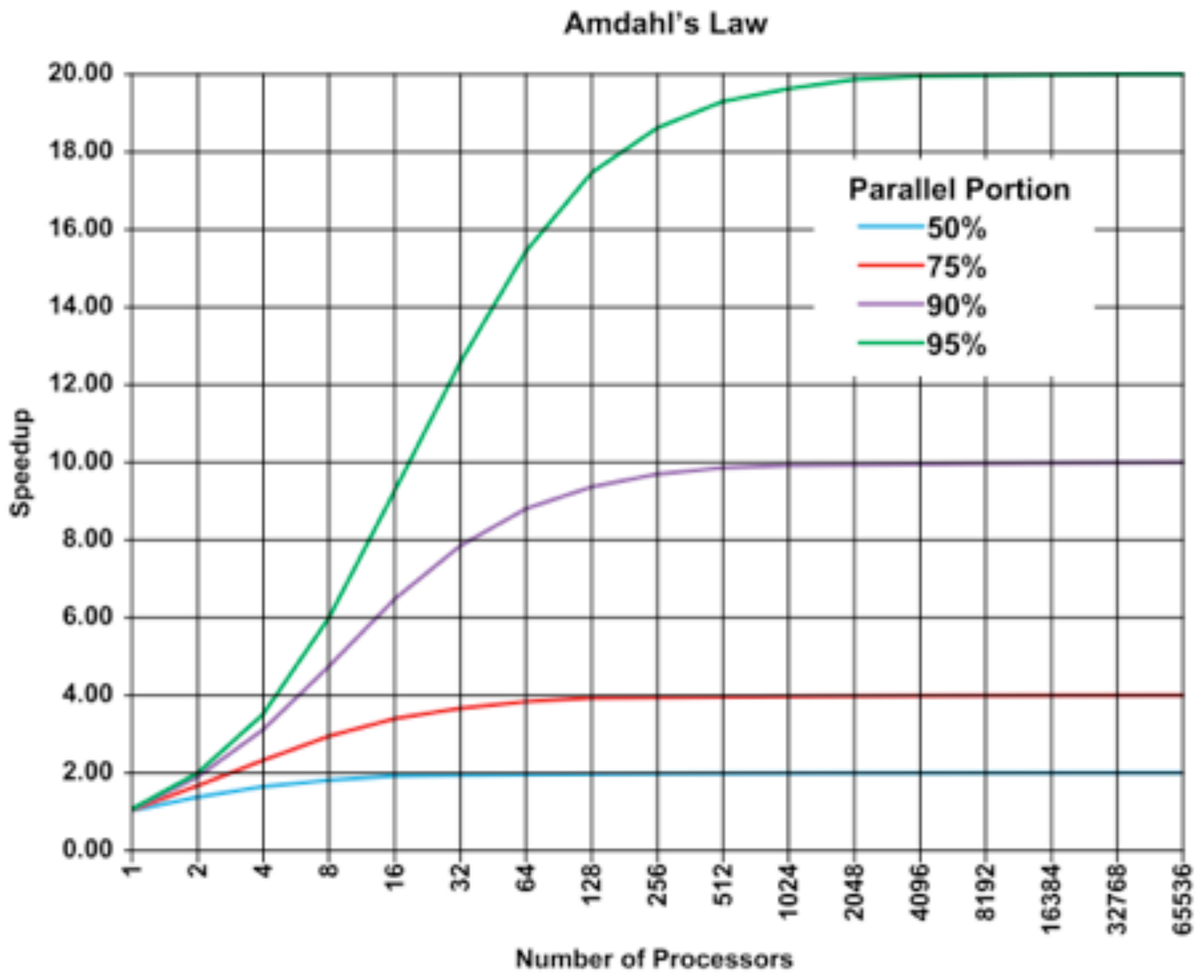


Unfortunately it doesn't work too well



Typical **theoretical** speedup is only a factor **3 to 5**

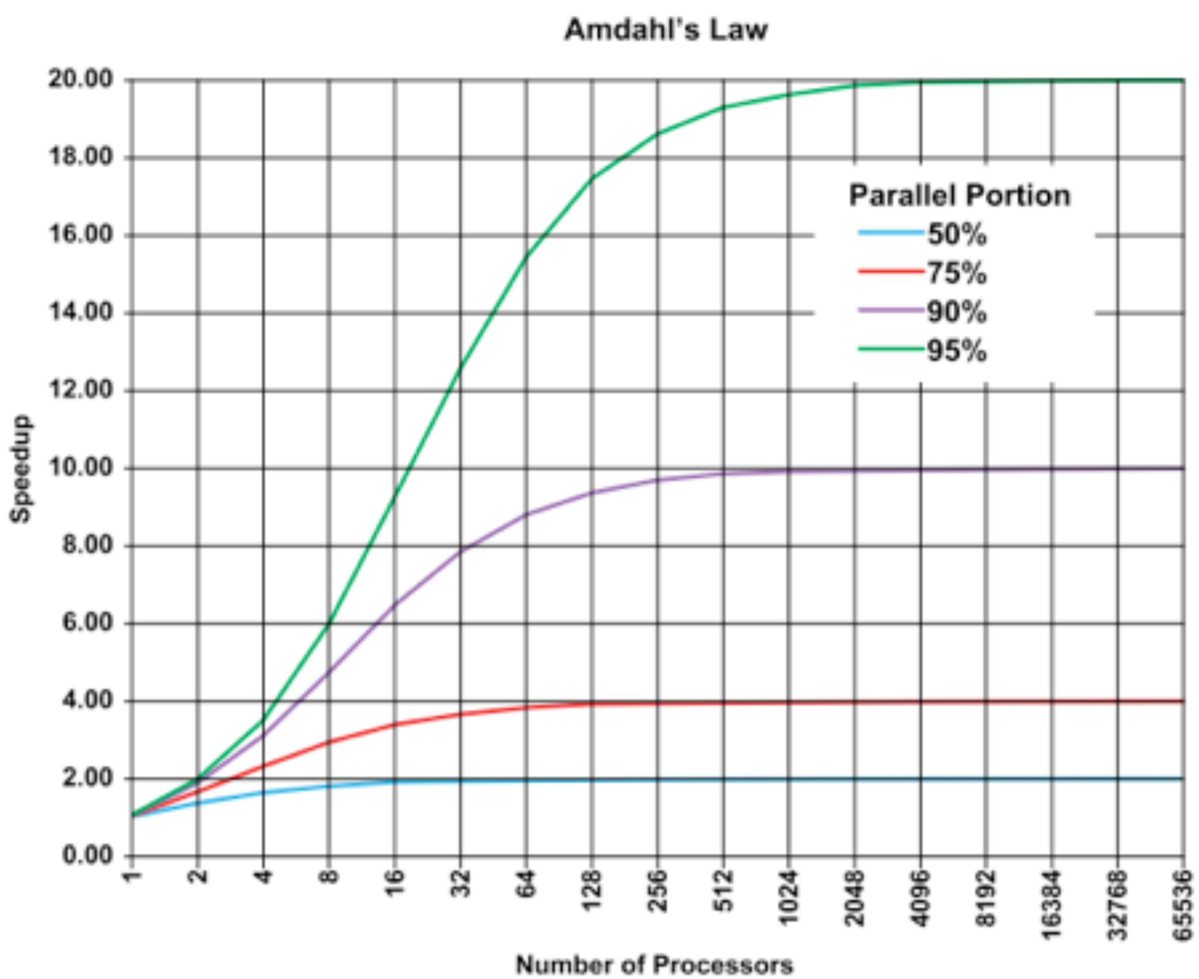
Amdahl's Law



Amdahl's Law:

"… the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." - 1967

Amdahl's Law



Amdahl's Law:

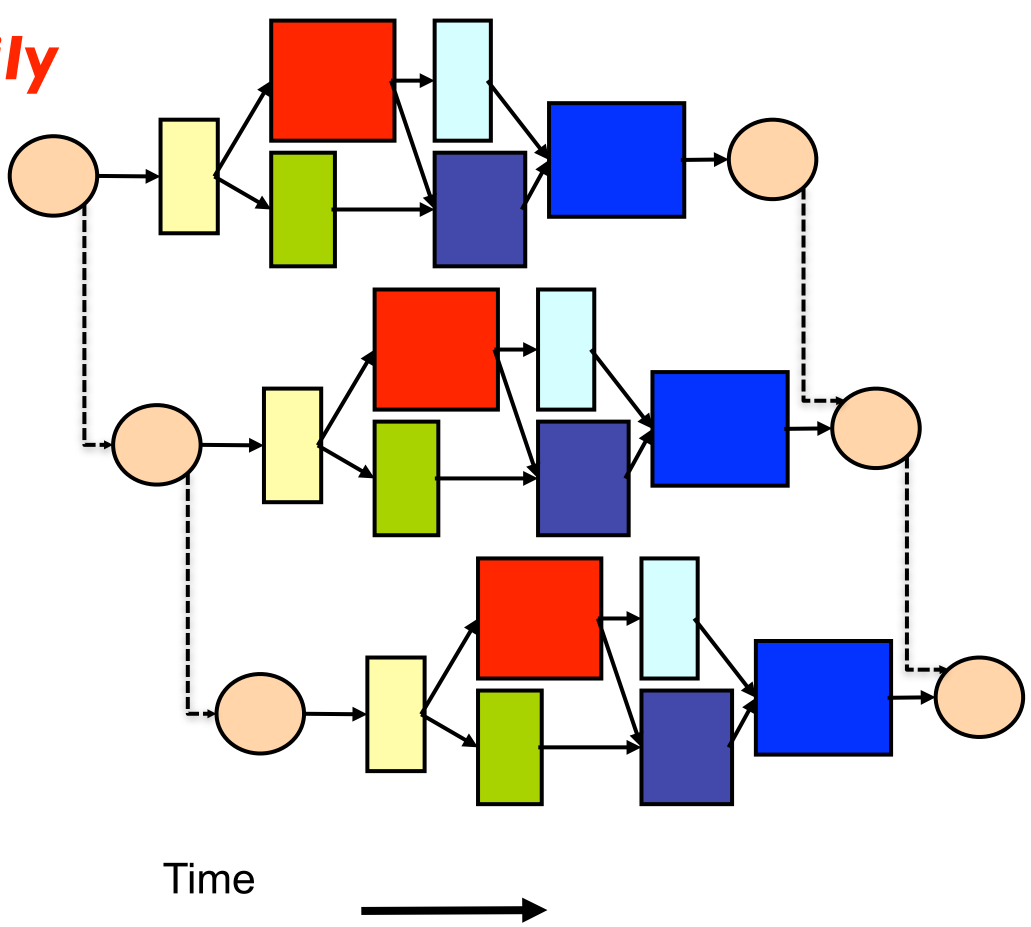
"… the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." - 1967

Gustafson's Law

"… speedup should be measured by scaling the problem on the number of processors, not by fixing the problem size." - 1988

Re-thinking the Parallel Framework

- **Need to change the problem size**
    - Process ***multiple events concurrently***
    - Helps on tails of sequential processing
- **Contradicts a lot of basic assumptions in existing code**
    - State machines expect to only hold one event at a time in memory
    - ***But existing code can't be thrown away easily***
    - Need to localize distributed states into "event context" that is passed around
- **Major efforts in all LHC experiments!**



A Glimpse on Complications

1. **The DAG is not known to its entirety**
    - Many stateful entities acting as back door w.r.t. official event store
2. **Shared states are rarely safe**
    - "Caches" that do not behave like… well… caches
    - Physicists programmers are creative in every respect!
3. **Algorithms are not thread-safe**
    - E.g. track reconstruction cannot be run on two events concurrently
    - Making all algorithms thread-safe is an impossible task
4. **External libraries are not thread safe**
    - But independent parts of the framework access them
    - Not all of the libraries will be thread safe ever!

Solutions?

The solution to three of the four problems is creating a smart scheduling environment

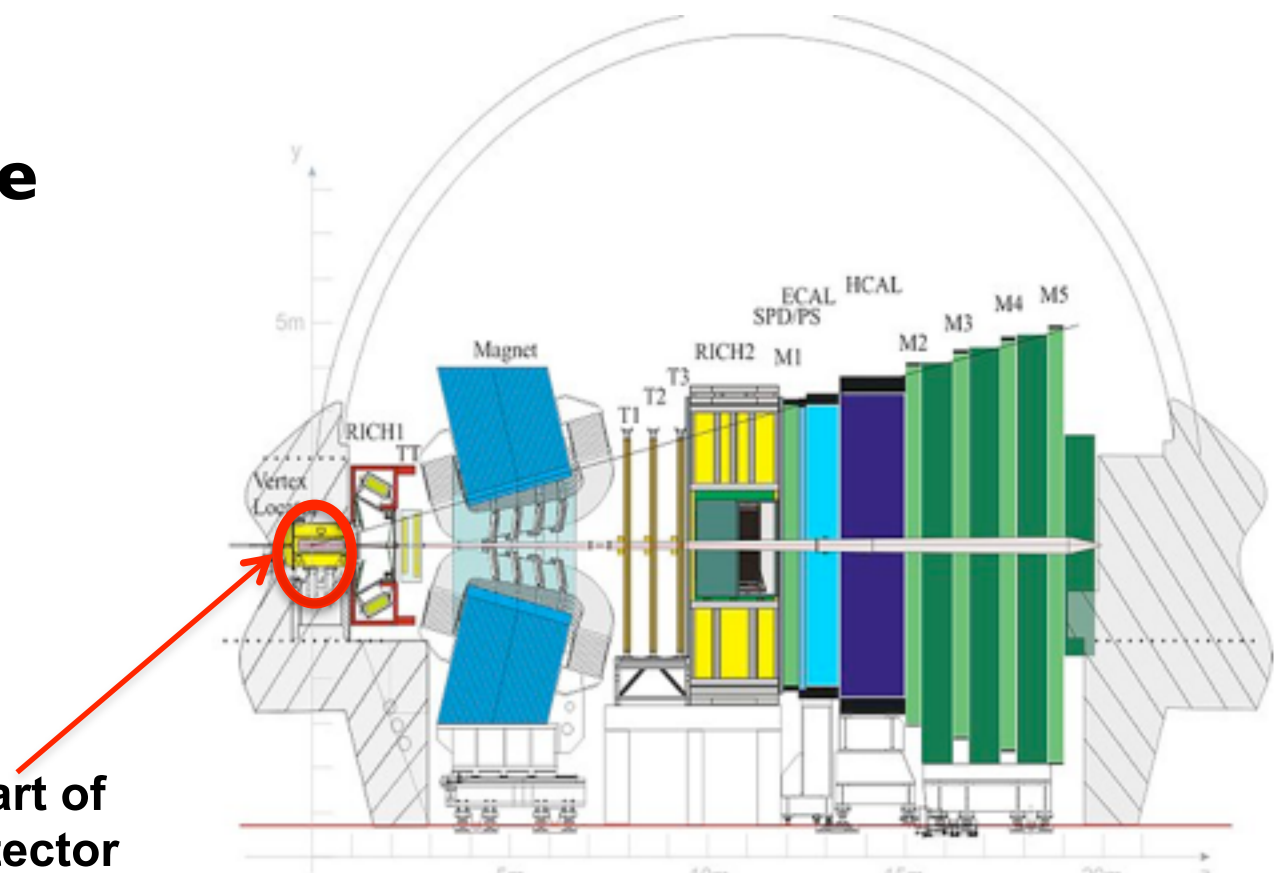
1. **The DAG has to be "fixed" by changing the existing code**
2. **Shared states are replaced by task-local data and thread-safe constructs**
3. **If an algorithm requires a non-thread safe resource, it has to "reserve" it beforehand**
   - Be careful: "reserving" is different from "locking" (compare with hotel rooms)
4. **Algorithms are being treated as resources that are being reserved**
5. **If a particular resource causes a bottleneck, make it thread-safe or provide exchangeable clones**

The Golden Rule of Software Design

Don’t develop theories,
write a prototype!

State of the Art

- **Such parallel frameworks are not only theory**
- **First implementations exist already for**
  - CMS offline software (CMSSW)
  - ATLAS/LHCb framework (Gaudi)
- **Let's have a look at an example workflow and its scaling**
  - A slice of the LHCb reconstruction
  - Only the low level objects of the vertex locator (VELO)

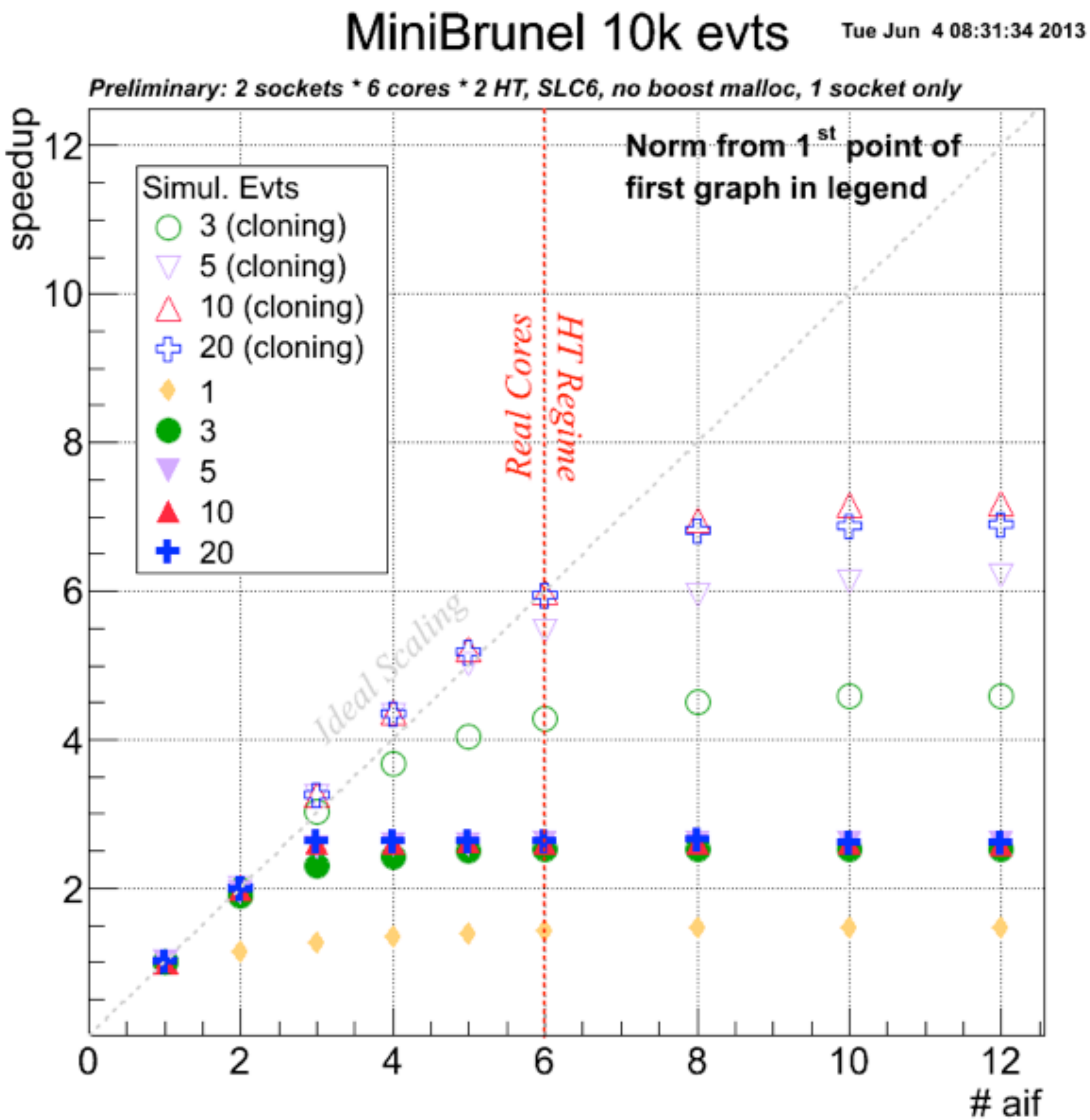


This part of the detector

LHCb detector

VELO Low-Level-Reco DAG

Scaling result on 6-core processor



One event processed at the time: ~30% speedup

No cloning of algorithms: Speedup saturates

Cloning:

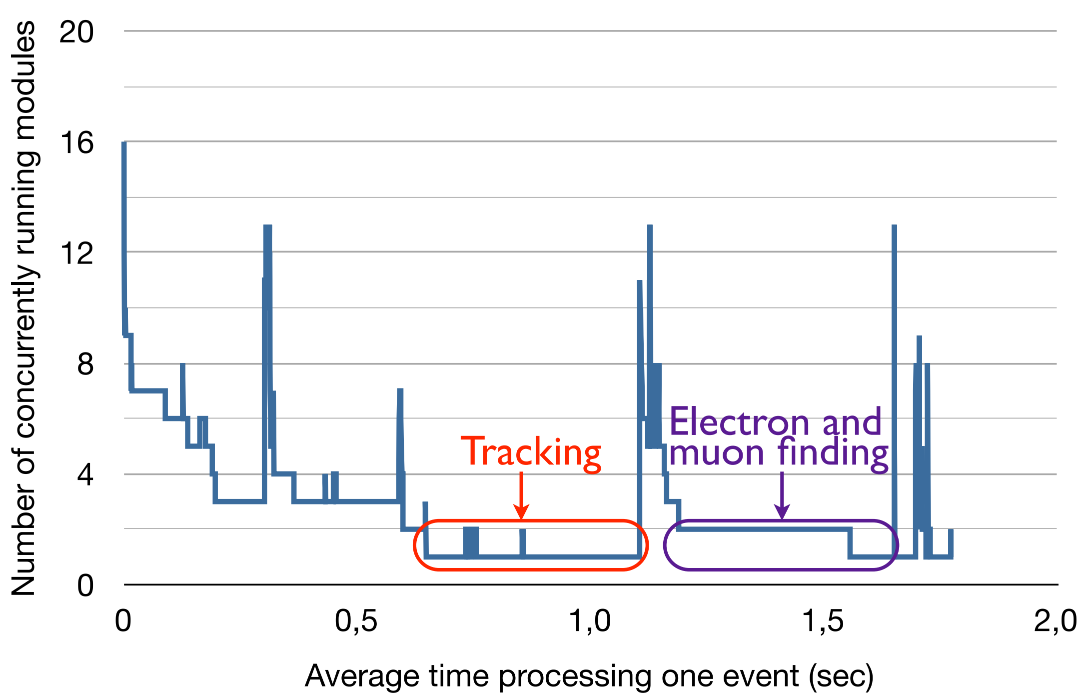
ideal (linear) scaling reached

Cloning of the 3 most time consuming algs only

Most of the code doesn't need to be thread-safe

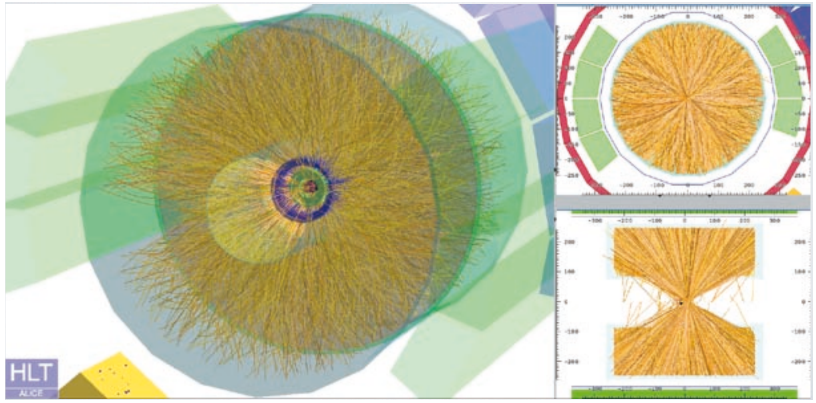
Long-term solution

- What do we do once the parallel scheduling of modules doesn't work any more?
- We need to split up our modules and algorithms into smaller pieces ('kernels') that run parallel in the CPU or on GPUs
  - Tracking will be the most important piece
  - I/O will rank second
- Many competing technologies around:
  - MIC, GPGPU, OpenCL, CUDA, ...
- So what's the potential?
  - Let's have a look at what people already did...



Parallel Tracking

- ATLAS already made some efforts
  - Discovered a potential for improvement by an order of magnitude
  - Implemented seed finding for Level-2 trigger
  - Raw data pre-processing for Level-2 trigger
- ALICE trigger using simplified GPU-based tracking



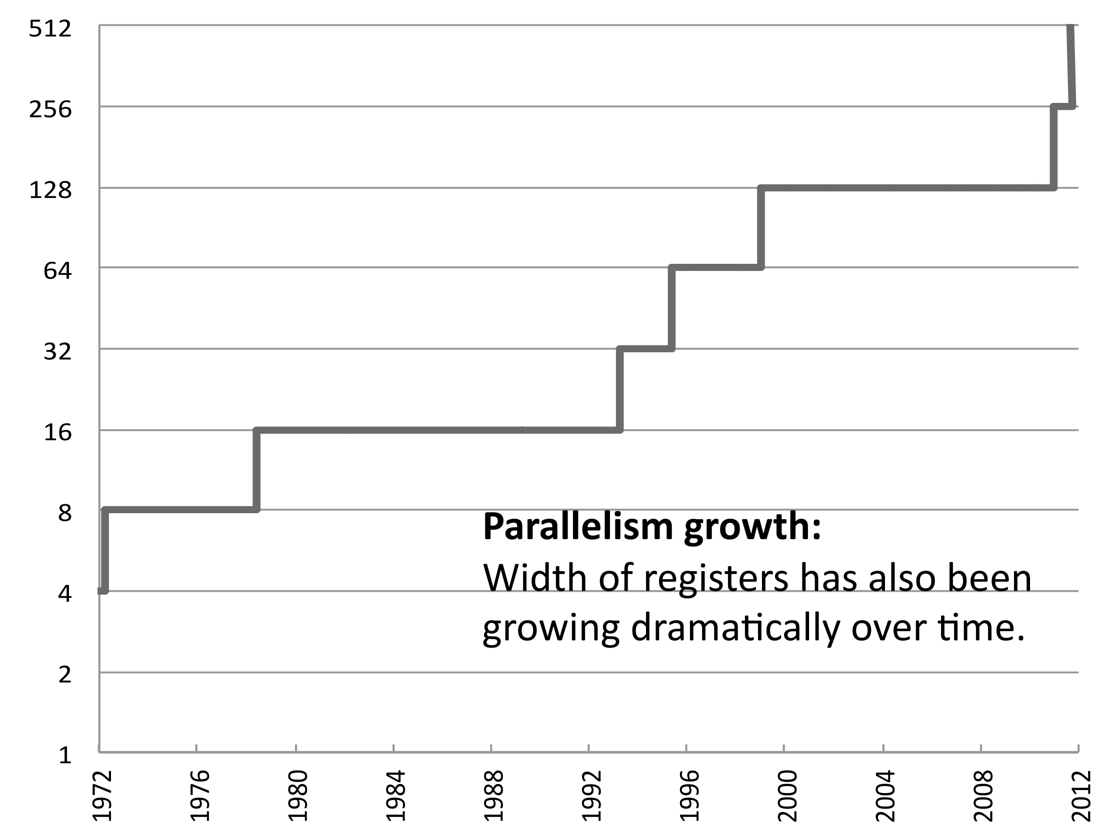
- Very hot topic these days
  - CBM (@FAIR) and CMS have PhD students from KIT on parallel tracking

SIMD
(Single Instruction - Multiple Data)

SIMD instructions

- Processors supporting Single Instruction, Multiple Data (SIMD) can execute ***one instruction*** on ***multiple data***
- Successive standards of SIMD instruction sets exist (MMX, SSE, SSE2, ... , AVX) with ever increasing register size

- **SSE2**
  - Basically all CPUs since 2003
  - ***Two*** double precision floating point values
- **AVX**
  - Since 2011 (Intel Sandy Bridge)
  - ***Four*** double precision floating point values



http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/

SIMD example

Just an 'academic' example:

```
double* x = new double[ArraySize];
double* y = new double[ArraySize];

...

for (size_t j = 0; j< iterations ; j++)
{
   for ( size_t i = 0; i < ArraySize; ++ i)
   {
      // evaluate polynom
      y[i] = a_3 * ( x[i] * x[i] * x[i] )
         + a_2 * ( x[i] * x[i])
     +  a_1 * x[i] + a_0;
   }
}
```

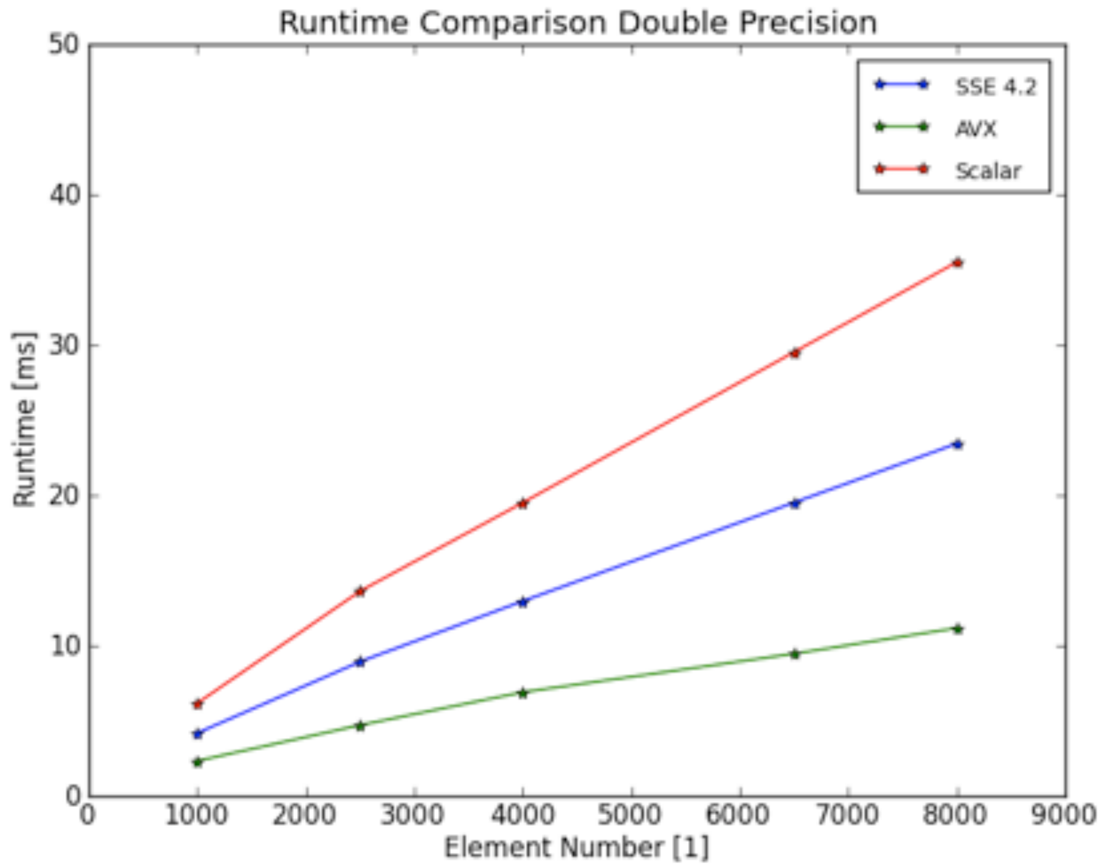
SIMD example

Just an 'academic' example:

```
double* x = new double[ArraySize];
double* y = new double[ArraySize];

...

for (size_t j = 0; j< iterations ; j++)
{
   for ( size_t i = 0; i < ArraySize; ++ i)
   {
      // evaluate polynom
      y[i] = a_3 * ( x[i] * x[i] * x[i] )
         + a_2 * ( x[i] * x[i])
     +  a_1 * x[i] + a_0;
   }
}
```

+

recent gcc and -ftree-vectorize



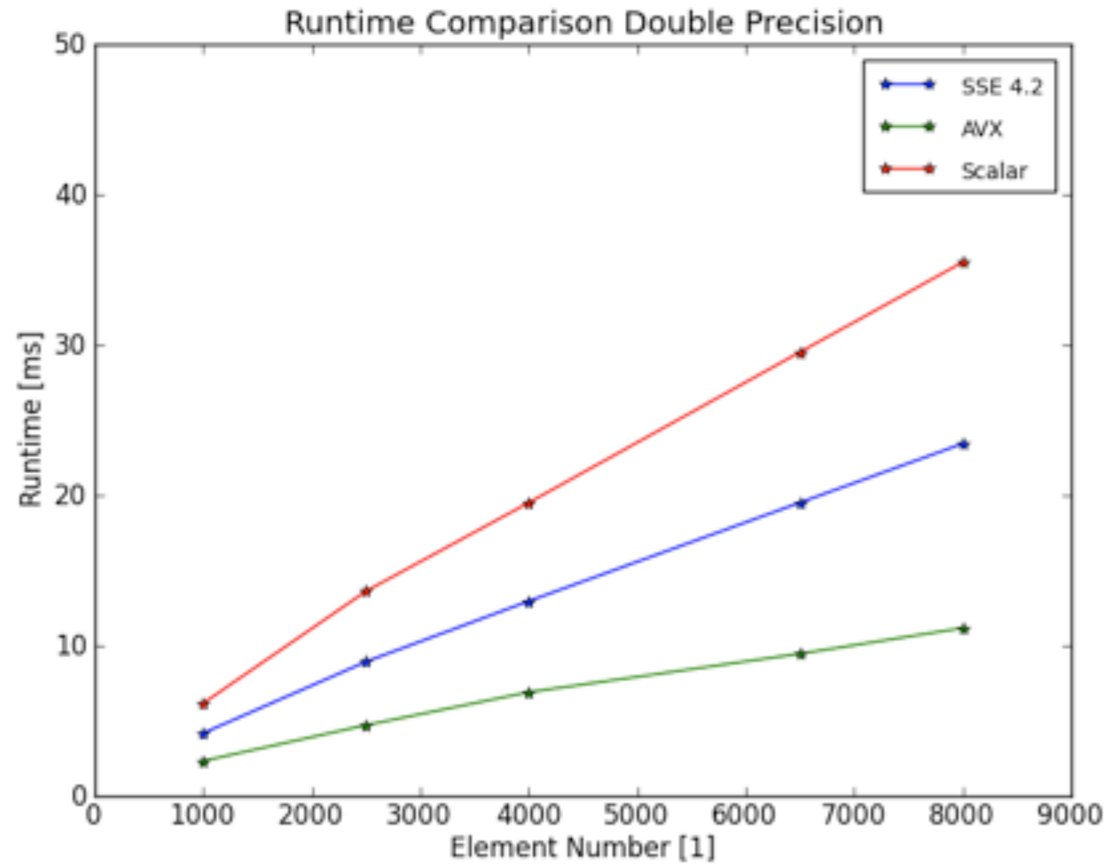
SIMD example

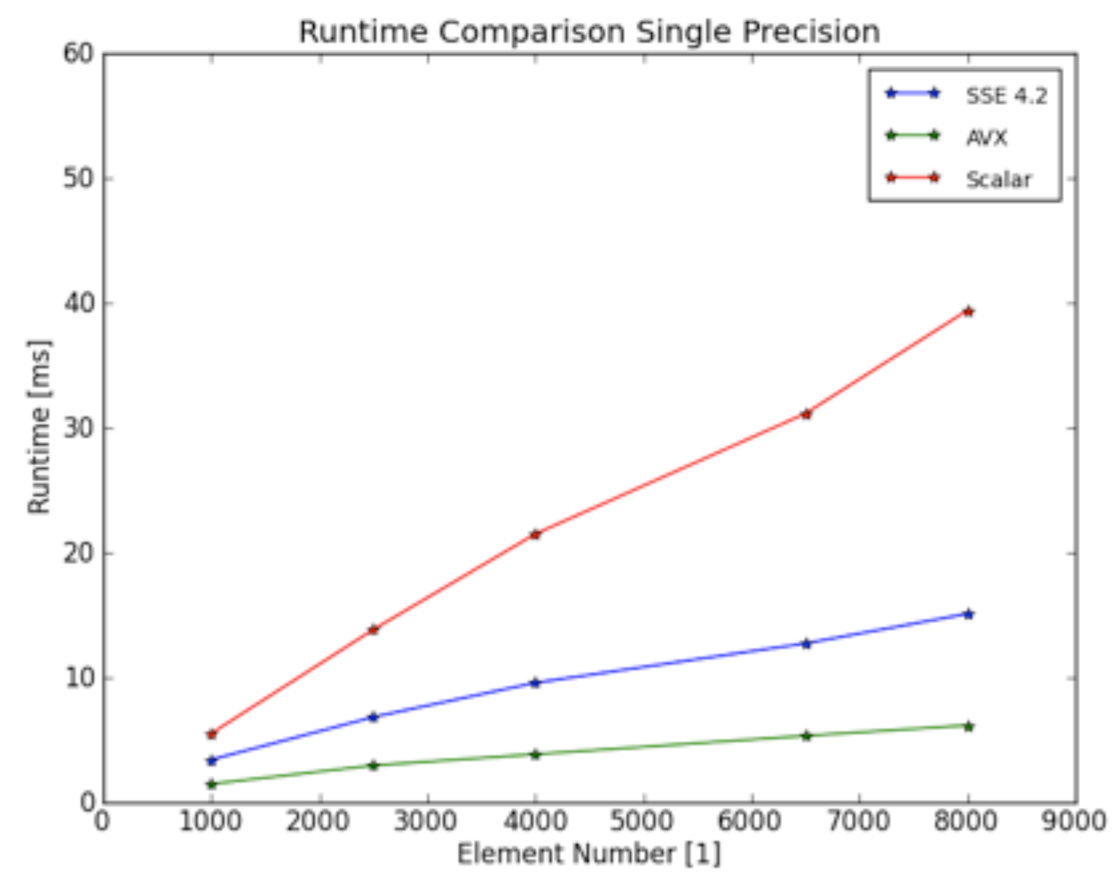
Just an 'academic' example:

```
double* x = new double[ArraySize];
double* y = new double[ArraySize];

...

for (size_t j = 0; j< iterations ; j++)
{
   for ( size_t i = 0; i < ArraySize; ++ i)
   {
      // evaluate polynom
      y[i] = a_3 * ( x[i] * x[i] * x[i] )
         + a_2 * ( x[i] * x[i])
    +  a_1 * x[i] + a_0;
   }
}
```

+

gcc4.6 and -ftree-vectorize

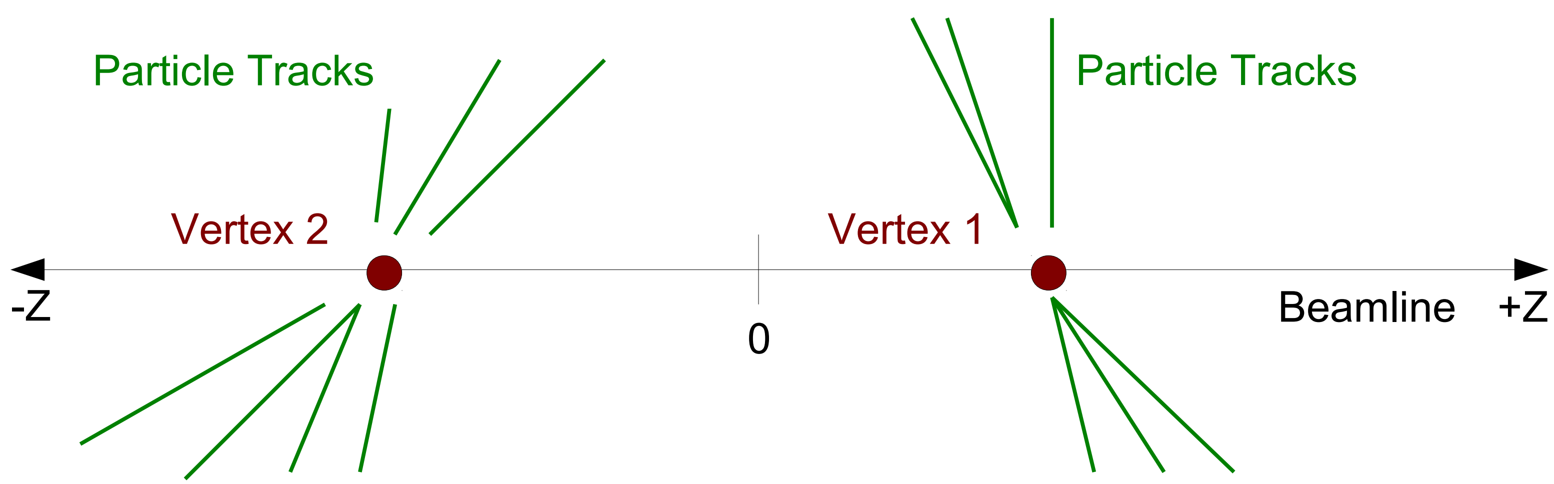




Turning that into reality

Real World Example: Vertex Clustering

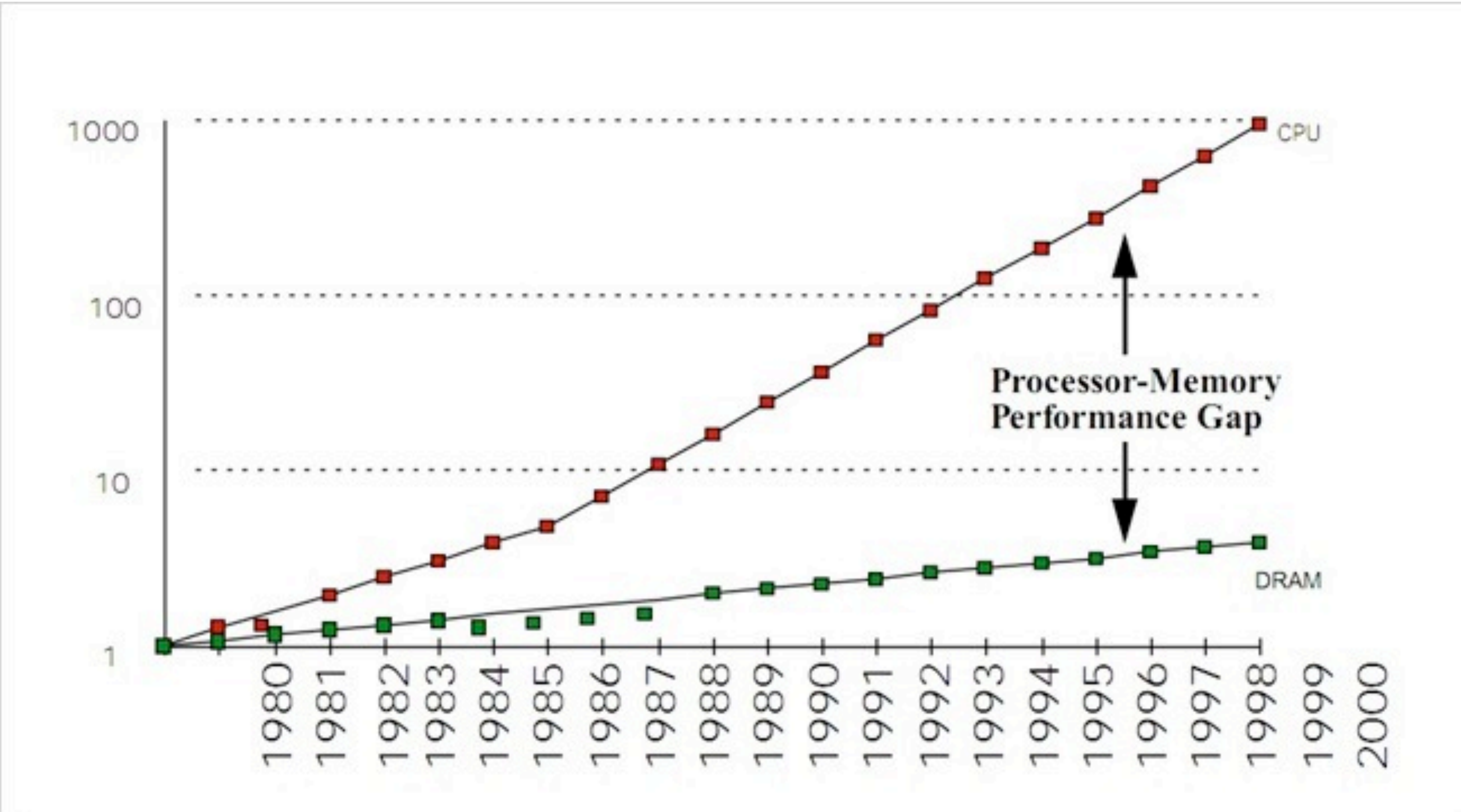
- Part of the CMSSW Reconstruction software
- Tracks are the input and the amount and location of primary vertices along the Z-Axis is computed using the Deterministic Annealing algorithm
- Nested loops over tracks and vertices have to be performed many times → Ideal for vectorization
- This clustering step represents 3% of the overall reconstruction runtime



Version	Runtime for 50 Events [s]	Ratio [1]
Regular	26.64	1.0
Vectorized	19.96	0.74
Vectorized + vdt math	11.46	0.43

The Memory Wall

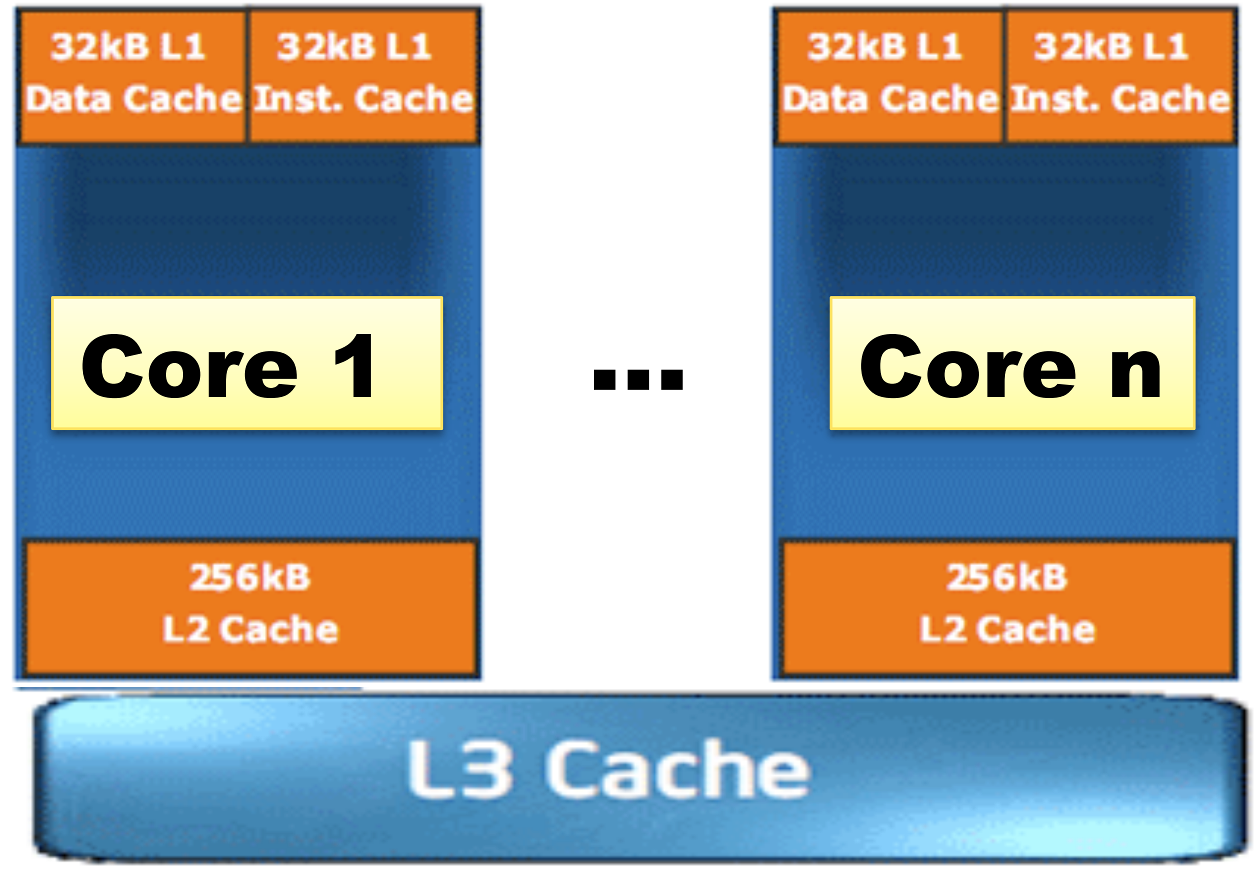
Memory Speed Development

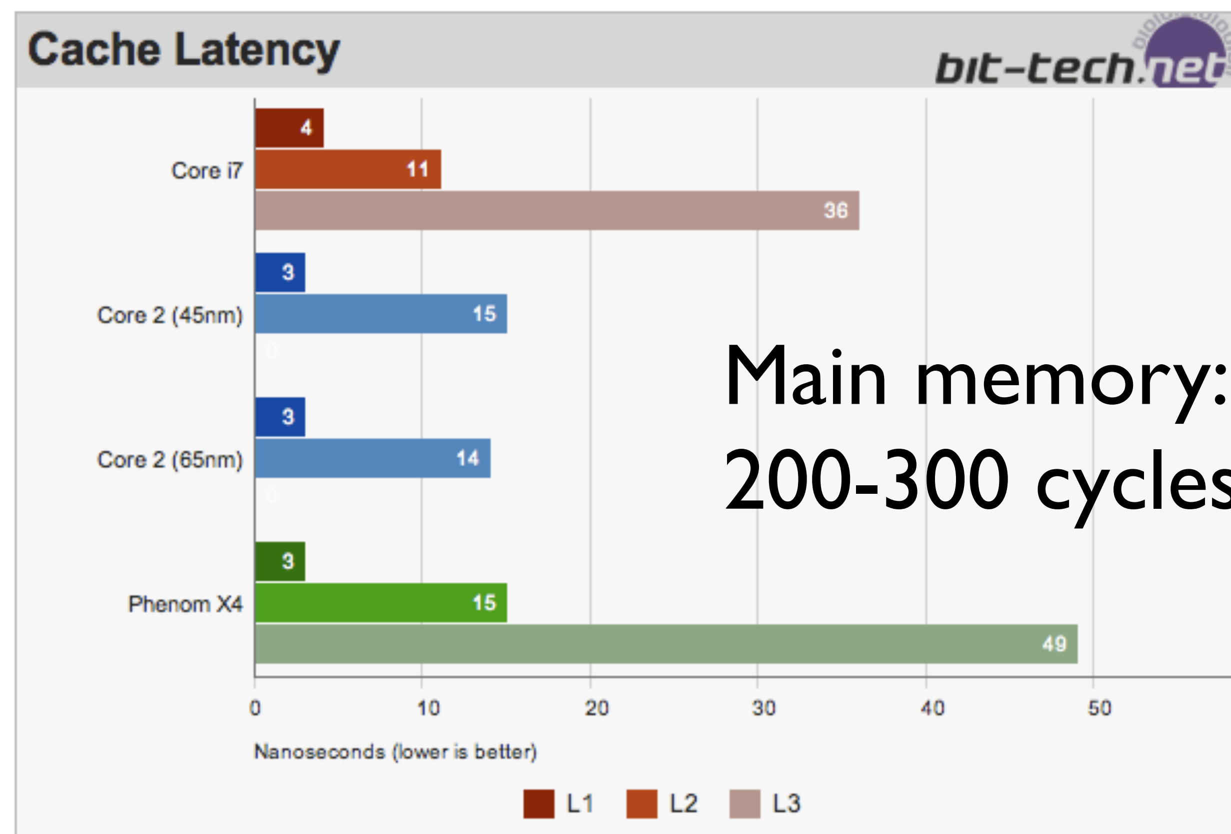


More than a factor 100 !

The 'Memory Wall'

- Processor clock rates have been increasing faster than memory clock rates
- Latency in memory access is often the major performance issue in modern software applications
- Larger and faster "on chip" cache memories help alleviate the problem but do not solve it
- Often CPU just waiting for the data

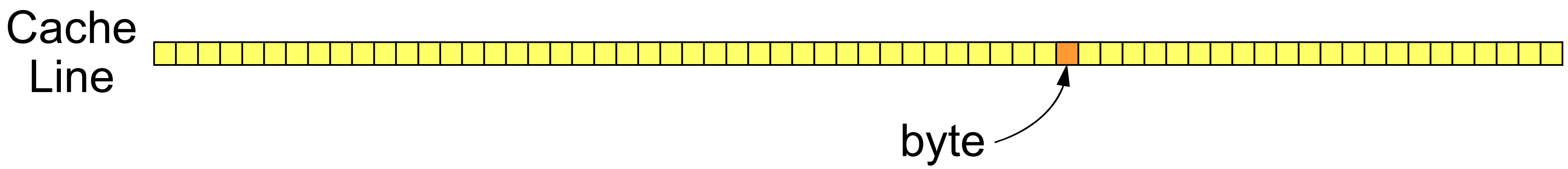




Main memory: 200-300 cycles

Caching

- Caching is - at distance - no black magic
- Usually just holds content of recently accessed memory locations

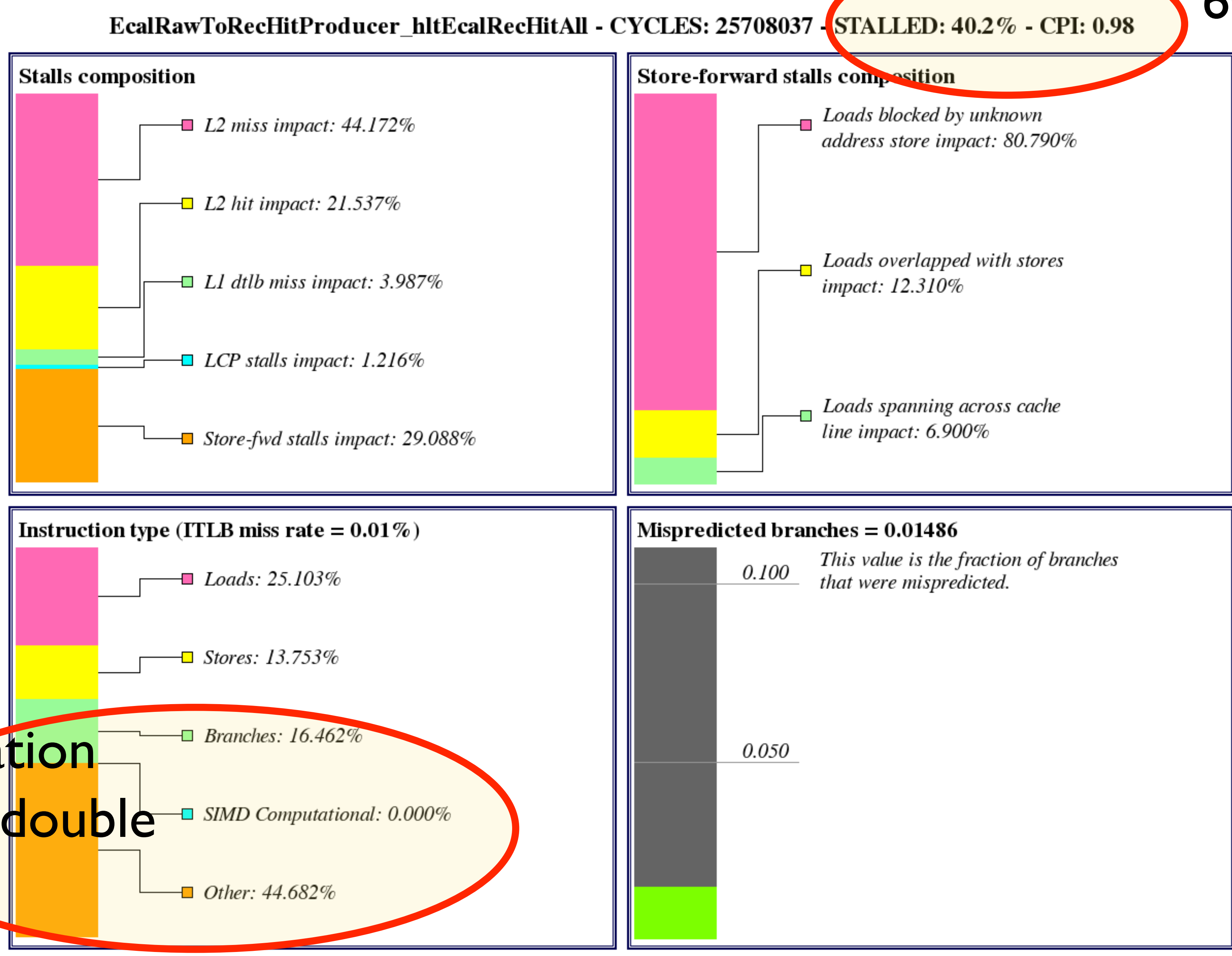


- Caching hierarchies are rather common:
  - 32KB L1 I-cache, 32KB L1 D-cache per core
    - ➡ Shared by 2 HW threads
  - 256 KB L2 cache per core
    - ➡ Holds both instructions and data
    - ➡ Shared by 2 HW threads
  - 8MB L3 cache
    - ➡ Holds both instructions and data
    - ➡ Shared by 4 cores (8 HW threads)

Very tiny compared to main memory!

Dominated by data movement NOW!
We use only 15% of availiable "d"flops

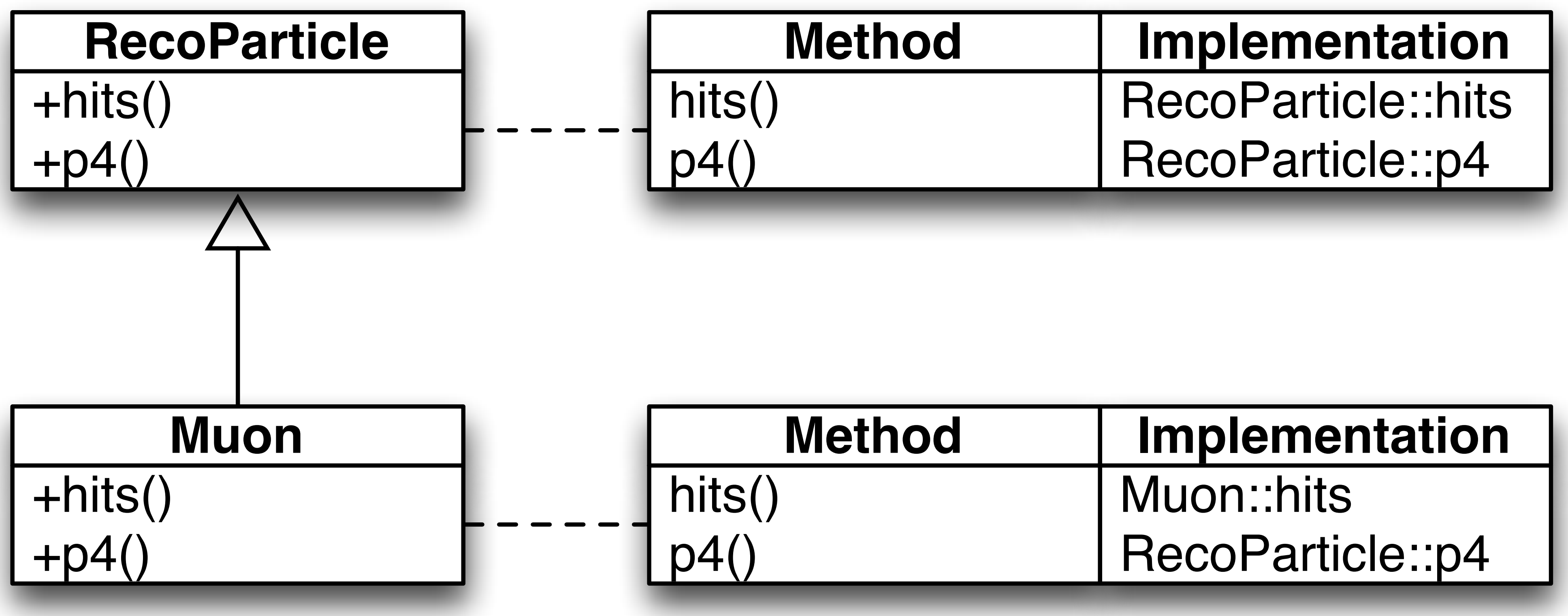
60% "active'



50% "computation on single/double word"

Little Reminder - vtable

The virtual table tells which code to execute when dealing with polymorphism



The death for any cache

Let's consider the following code and it's first execution:

```
for (DaughterIt it = m_daughters.begin();
     it != m_daughters.end(); ++it)
{
    m_p4.Add( it->p4());
}
```

The death for any cache

Create Iterator

```
for (DaughterIt it = m_daughters.begin();
      it != m_daughters.end(); ++it)
{
    m_p4.Add( it->p4());
}
```

The death for any cache

```
for (DaughterIt it = m_daughters.begin();
     it != m_daughters.end(); ++it)
{
    m_p4.Add( it->p4());
}
```

vtable of Iterator

The death for any cache

```
for (DaughterIt it = m_daughters.begin();
     it != m_daughters.end(); ++it)
{
    m_p4.Add( it->p4());
}
```

vtable of object
+ object

The death for any cache

```
for (DaughterIt it = m_daughters.begin();
      it != m_daughters.end(); ++it)
{
     m_p4.Add( it->p4());
}
```

method code

The death for any cache

Fetch into Cache

```
for (DaughterIt it = m_daughters.begin();
        it != m_daughters.end(); ++it)
{
    m_p4.Add( it->p4());
}
```

The death for any cache

```
for (DaughterIt it = m_daughters.begin();
      it != m_daughters.end(); ++it)
{
    m_p4.Add( it->p4());
}
```

vtable + method code

The death for any cache

```
for (DaughterIt it = m_daughters.begin();
     it != m_daughters.end(); ++it)
{
    m_p4.Add( it->p4());
}
```

+

every ugliness inside the method code

That were quite a few cache misses,
for a rather simple operation:

...

```
m_px += x
m_py += y
```

...

Identifying a way out

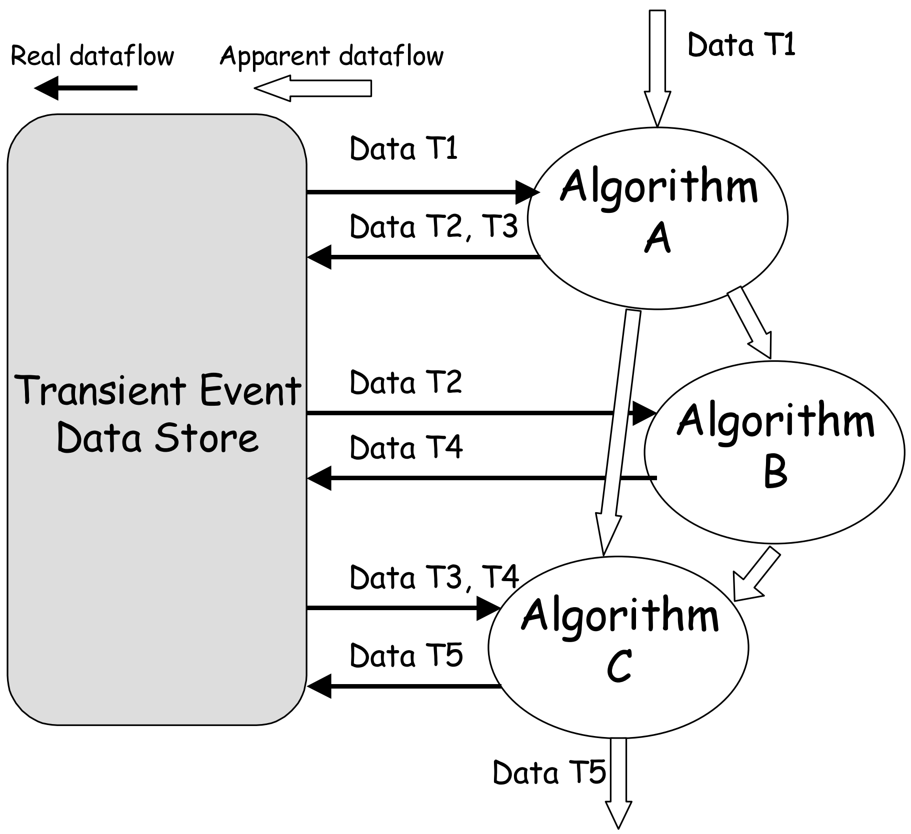
- **Cache misses are evil**
- **Put data that are used together closer together**
  - This usually crosses object boundaries
  - But only rarely collection boundaries
  - "Arrays of Structs" vs. "Structs of Arrays"
    - A particle collection becomes a collection single px, py, pz, ... vectors
- **vtables cause a good fraction of cache misses**
  - In principle every conditional statement spoils branch prediction and caching
- **Design your software around the most efficient data structures**
  - "Data Centric Programming"
- **Doesn't data locality contradict OOP principles and requirements?**

%&*!&   !!!

But is that really such a big problem?

- **OOP as dreamed of in the books**
  - It combines data and algorithms into a single entity
  - It ensures that the developer does not need to code up the control flow explicitly.
- **We already violate this with the software bus model**
  - The stored objects are mainly only data
  - We define the control flow explicitly
  - Data transformations happen in modules

Software design **at the big scale**
and efficient code **at the small scale**
can't be looked at in isolation!



What's ahead of us?

- **We have to choose with more thought when to follow which programming paradigm**
  - Many identical data chunks & high throughput => data oriented
  - Small number of objects & heterogenous data => object oriented

- **For reconstruction we have to redesign our data formats to become even dumber**
  - Expert operation !
  - Helps with auto-vectorization as well!

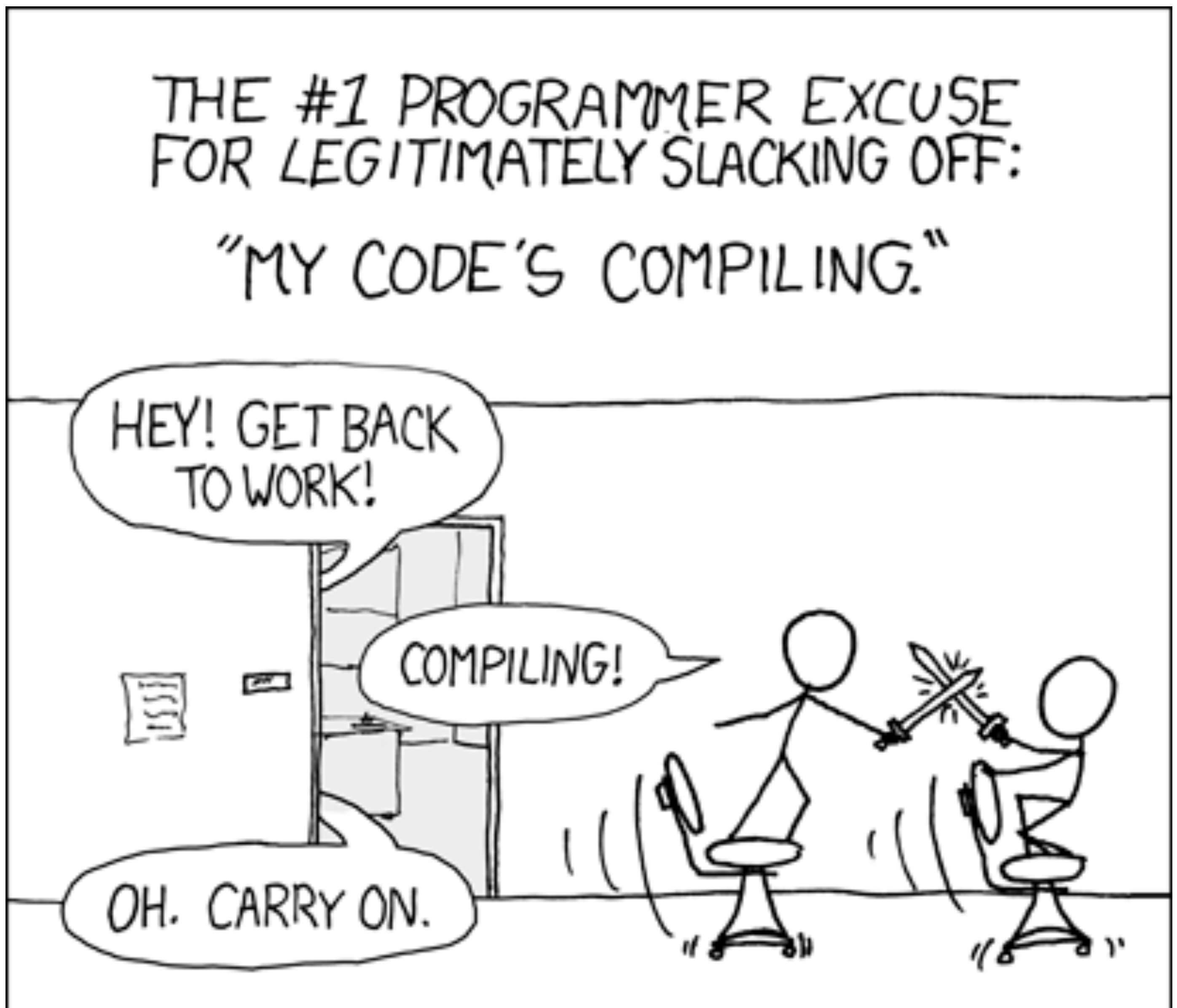
- **Analysis and other cases much more heterogenous**
  - We need a "data-to-smart object" translation layer. But where?!
  - A lot of trial-and-error R&D needed

Situation Summary

- There are limits to "automatic" improvement of scalar performance:
  - **Power Wall:** clock frequency can't be increased any more
  - **Memory Wall:** access to data is limiting factor
- Explicit parallel mechanisms and explicit parallel programming are essential for performance scaling
- LHC experiments converged on basic design approach for parallel applications
- Software design and efficient code have to go hand in hand

- **Challenging times ahead**
- **Exciting times for curious programmers!**

That’s it :-)



xkcd