



# Multi-core computing in HEP

Benedikt Hegner  
CERN



*Disclaimer:*

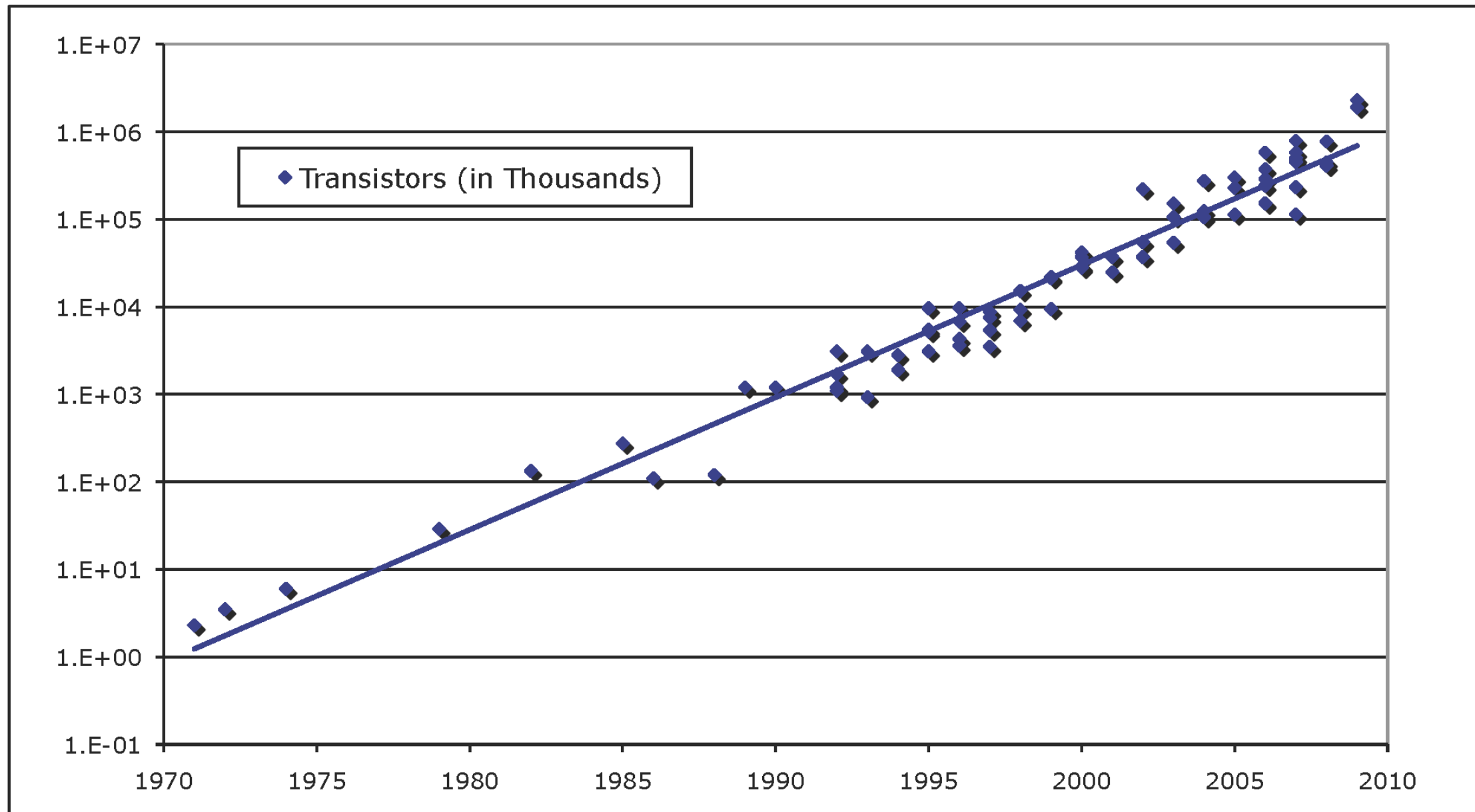
*I am showing only a very tiny part of the picture  
(in particular no GPU goodies)*

- Many cores and parallelism ('Power Wall')
- Ongoing R&D Efforts
- Memory Speed and Bad Programming ('Memory Wall')
- Summary



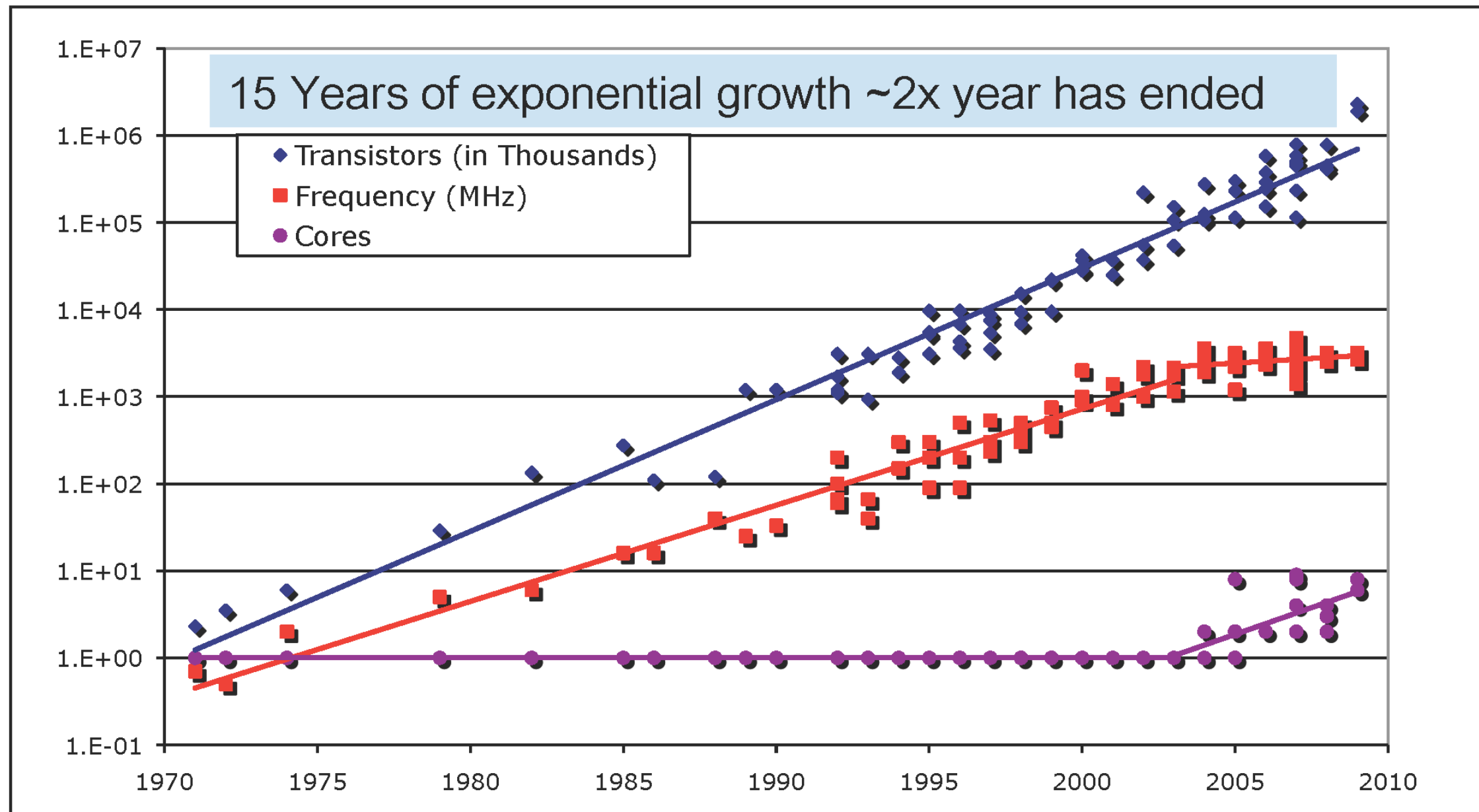
## The 'Power Wall'

# Moore's law alive and well?



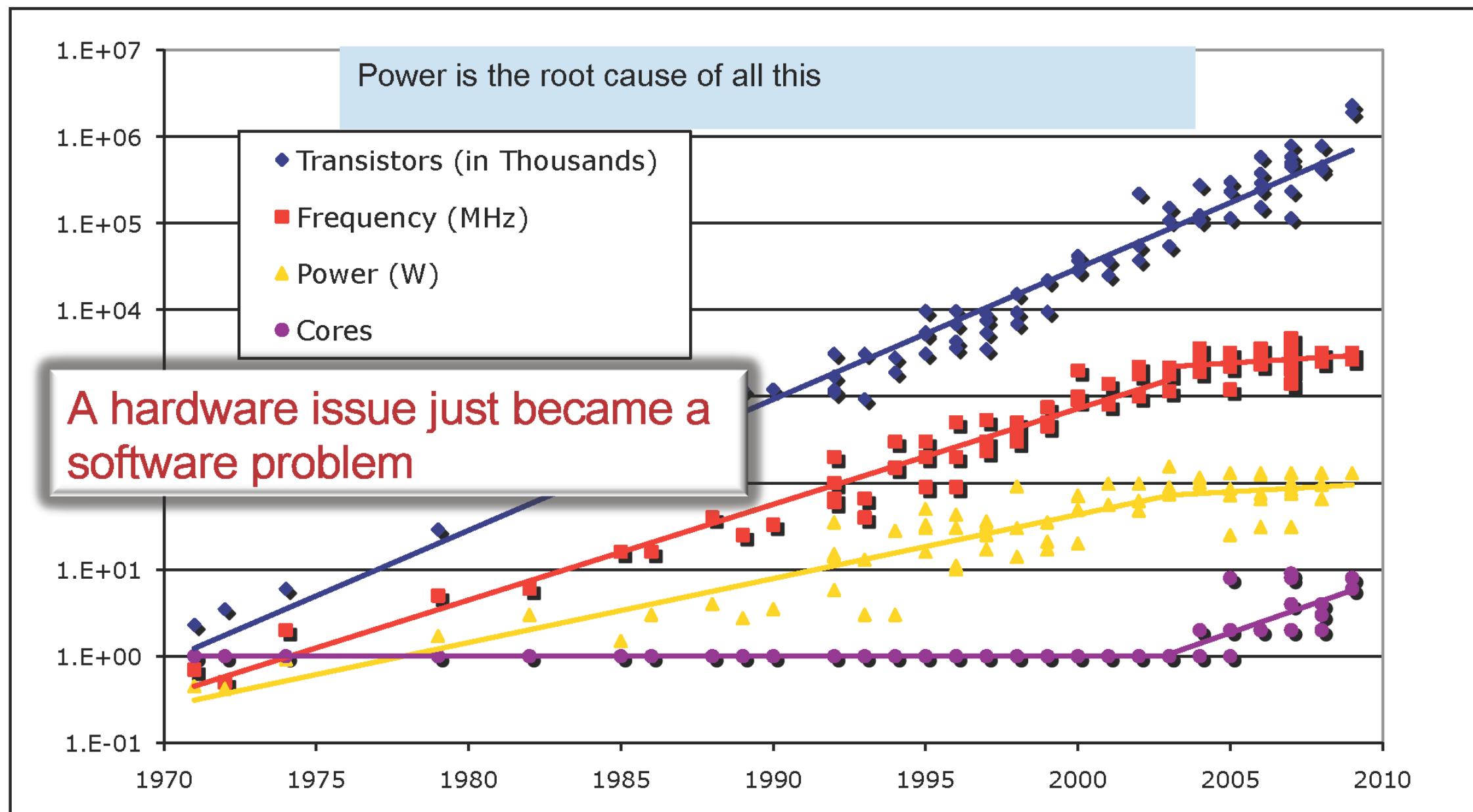
# Moore's law alive and well?

...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:

$$\textit{Power} \propto \textit{Frequency}^3$$

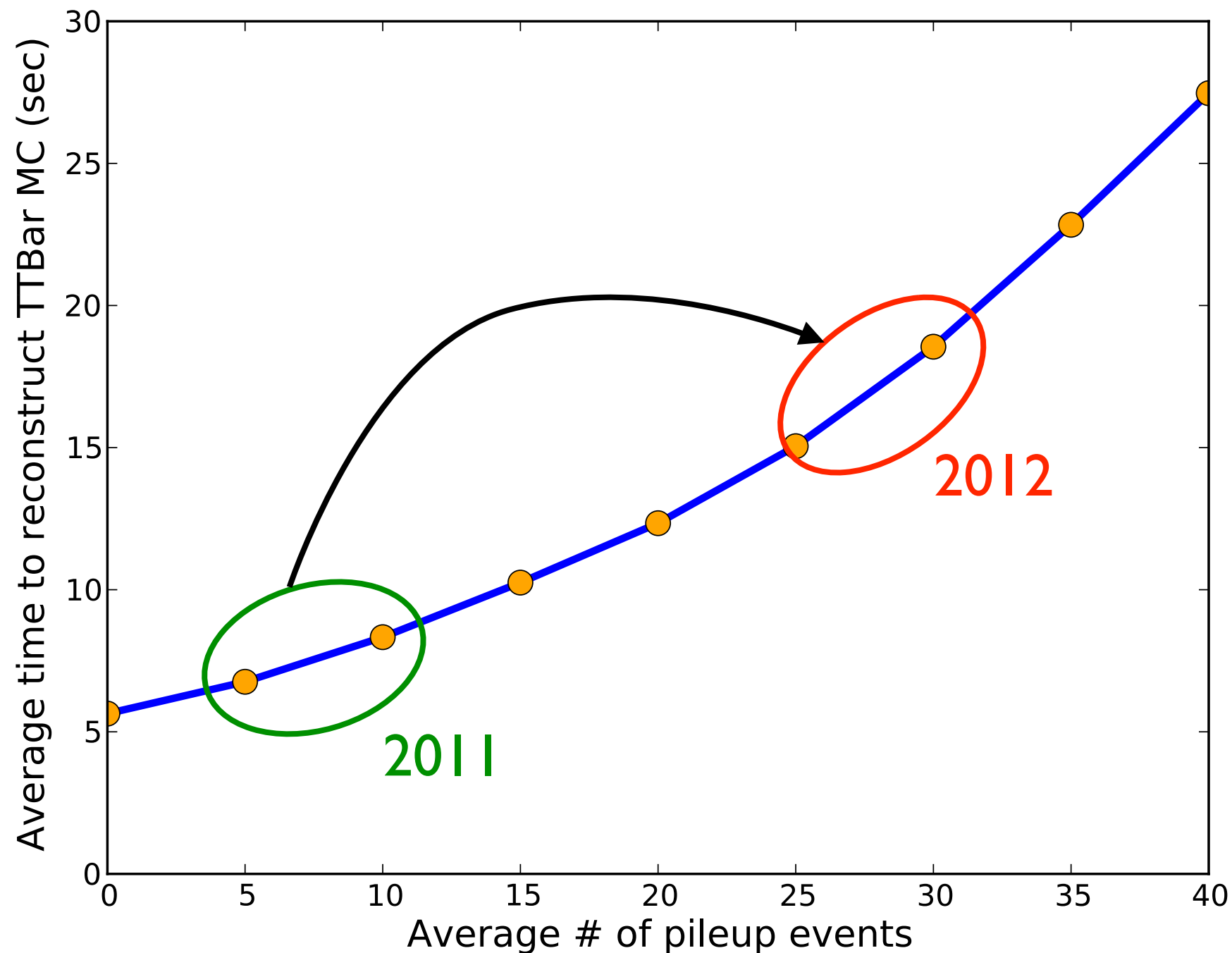
- Need to deal with systems of tons of concurrent threads and calculations
- In GPUs that's reality already now
- We can learn a lot from game programmers! (\*)

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

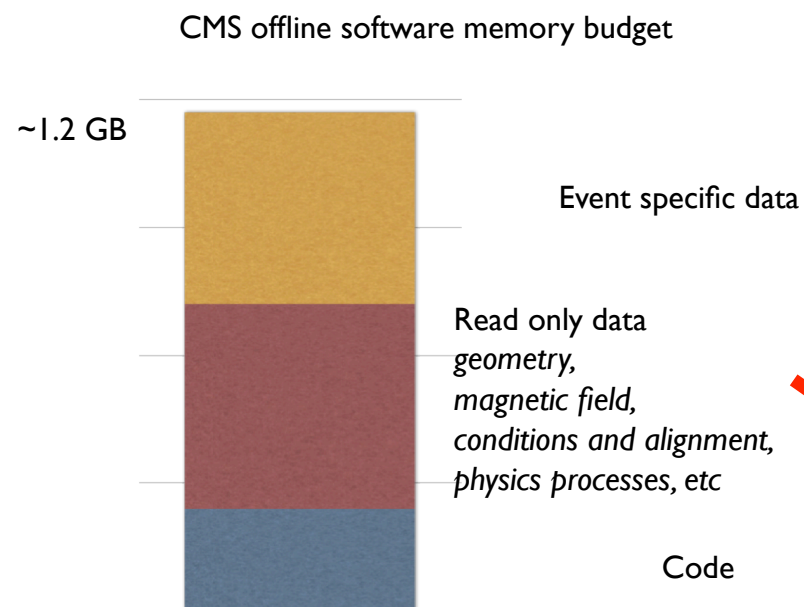


# From 2011 to 2012

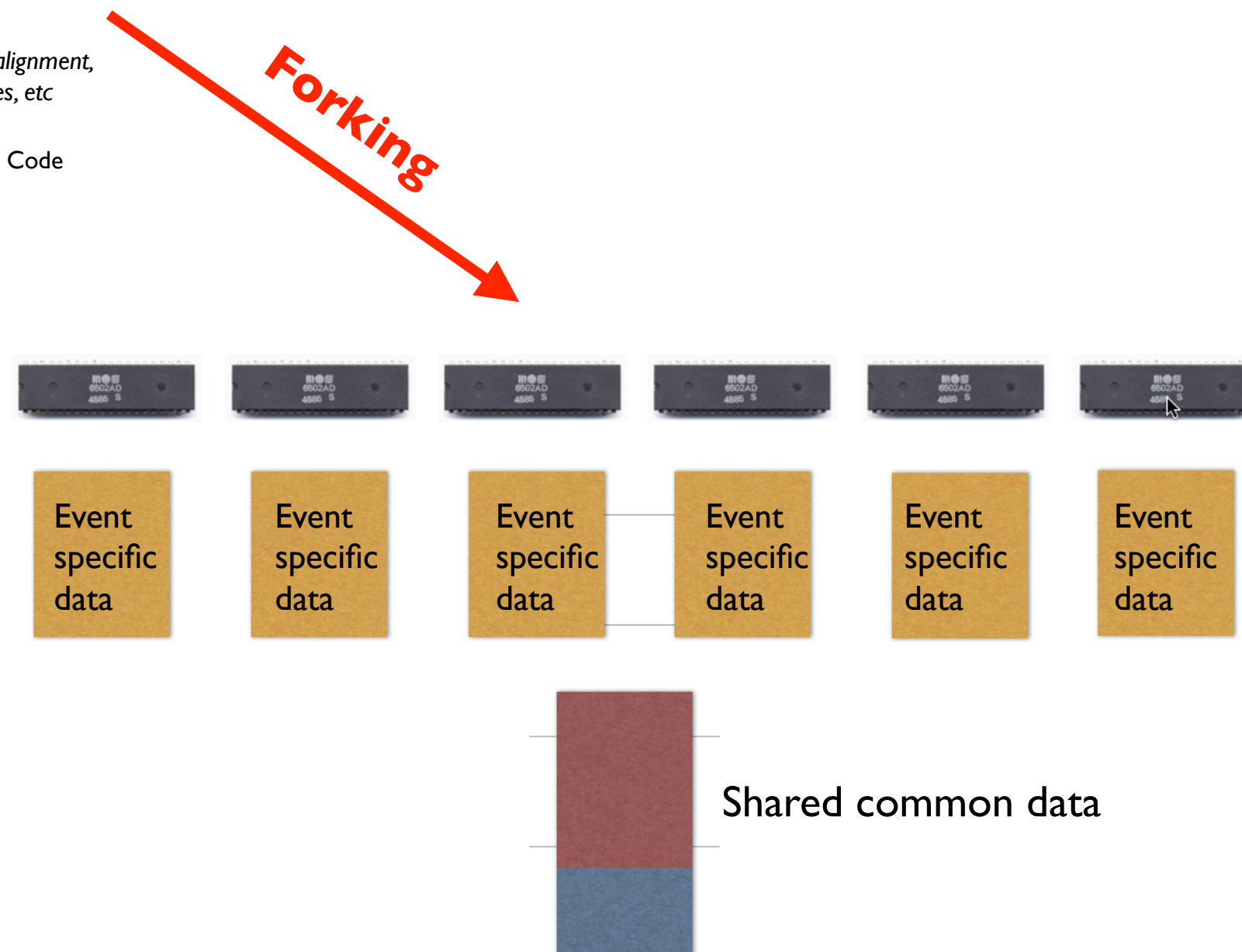
It is not an academic problem !



# Surviving i

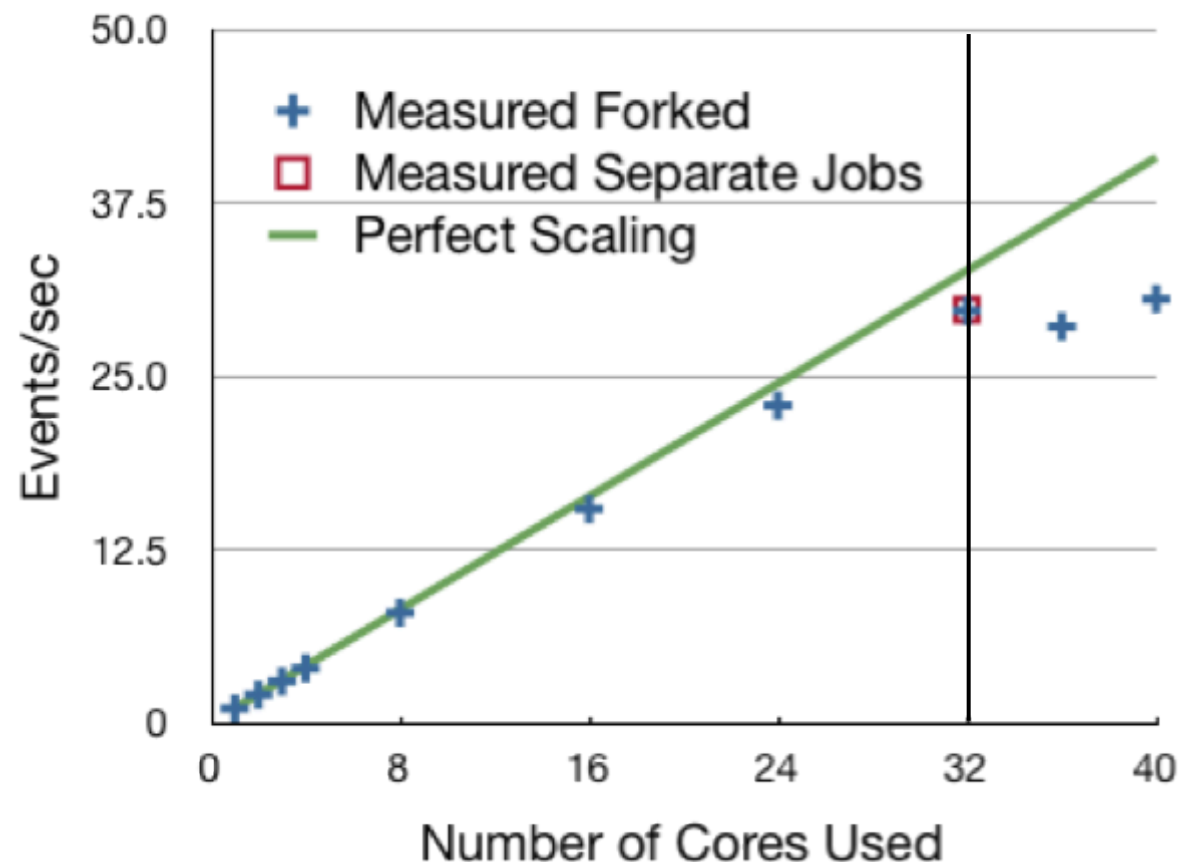


Possible because of our  
“embarrassingly parallel” problem

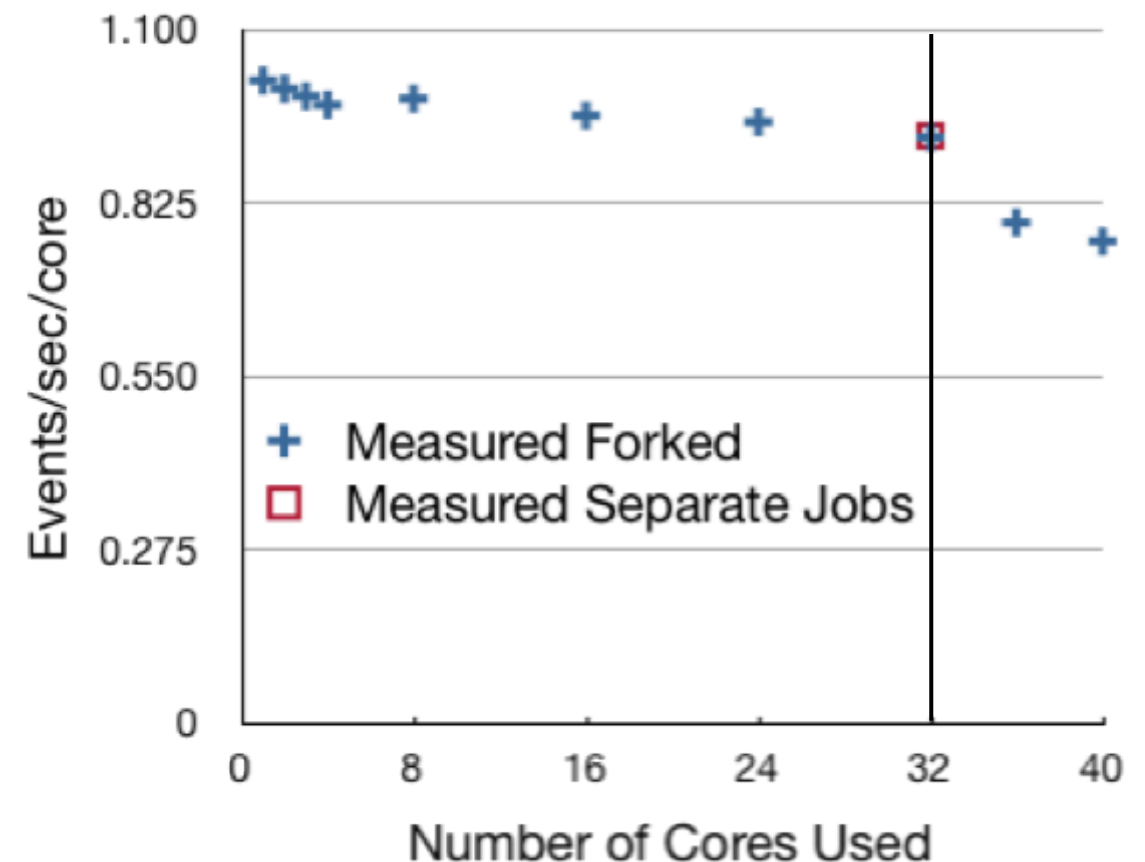


# Surviving in the short term (contd)

Events/sec vs Number of Cores

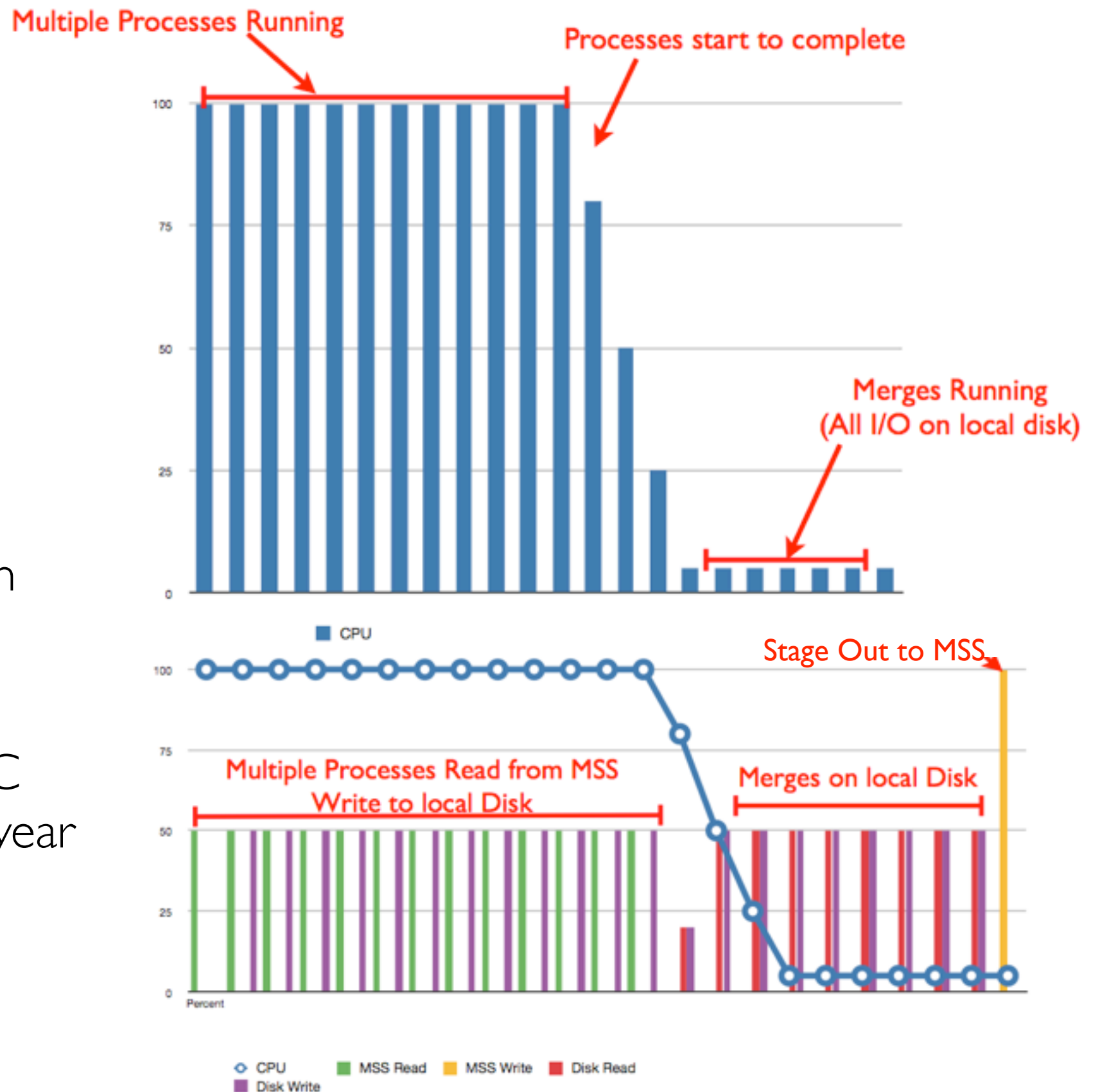


Events/sec/core vs Number of Cores



Problem for current Grid infrastructure:  
**Not all Grid centres are prepared for  
full allocation of a machine**

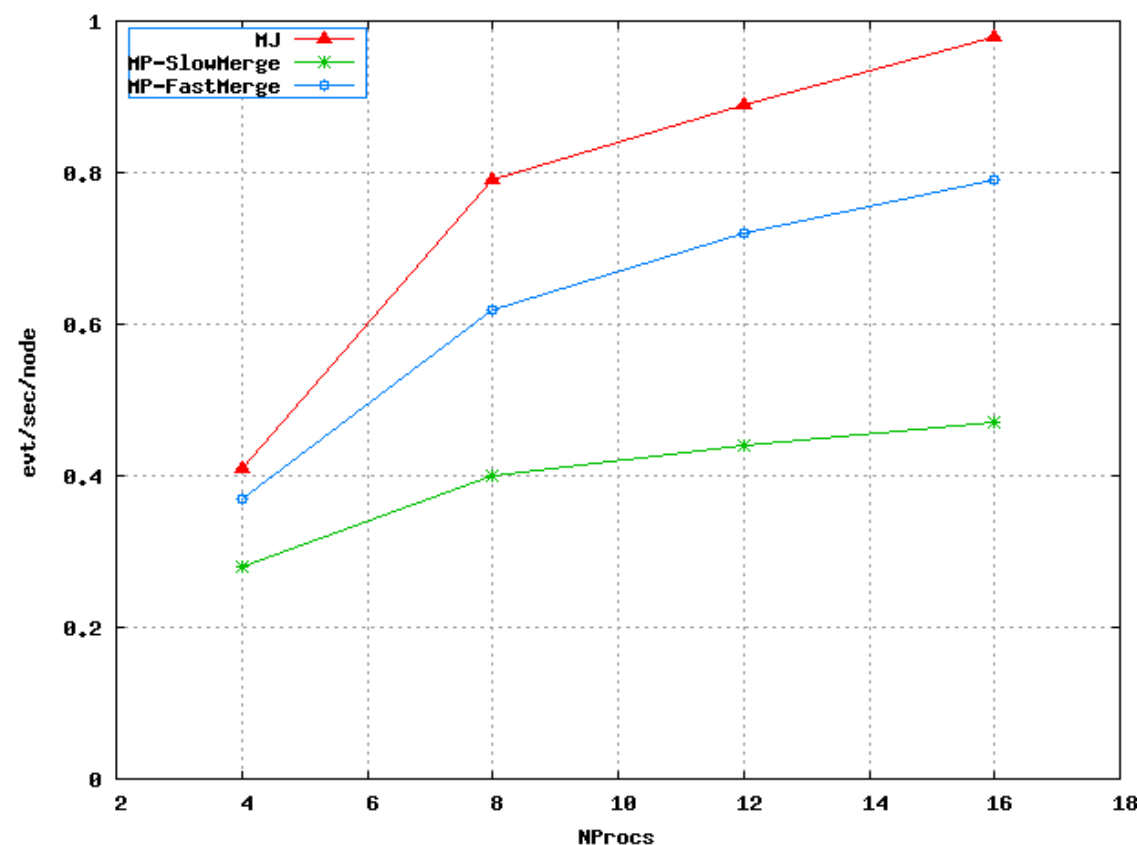
- No requirements on code quality, thread safety, etc
- Output file merging is the bottleneck in this approach
- However, it brings the LHC experiments through this year





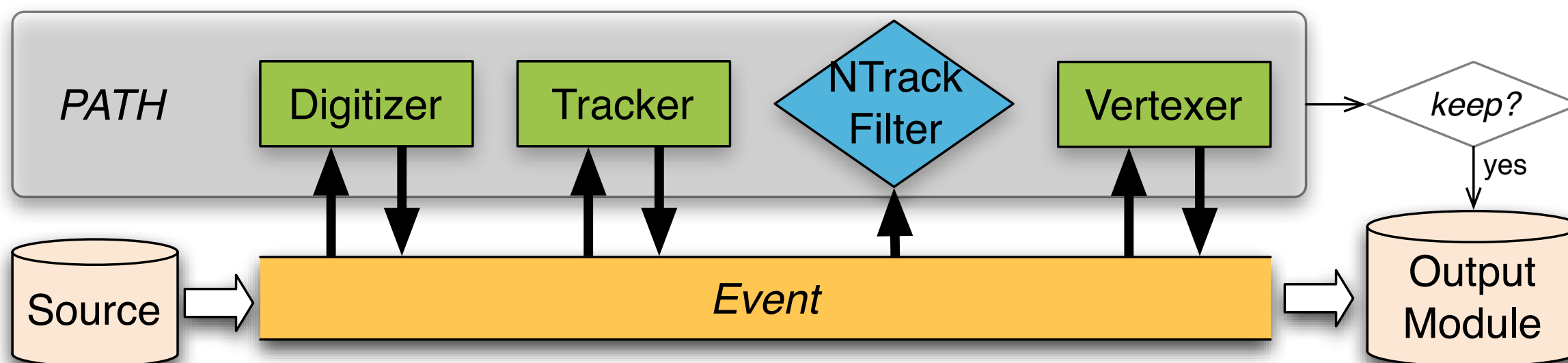
## Efficiency: job size

- In order to compete in CPU efficiency with N single process Athena jobs (assuming that we have enough memory for those), we need to **increase Athena MP job size**
  - Run **one Athena MP job over N input files** instead of running N Athena MP jobs over single input file each



# Framework Primer

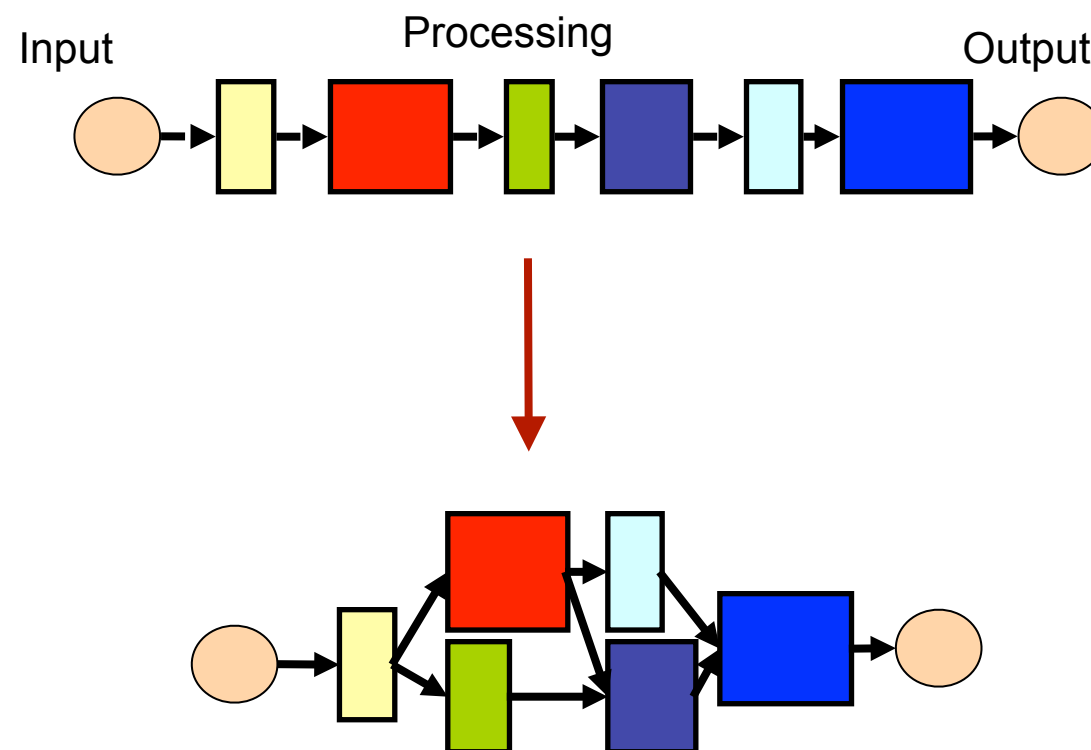
Experiment software follows the  
concept of a 'software bus'



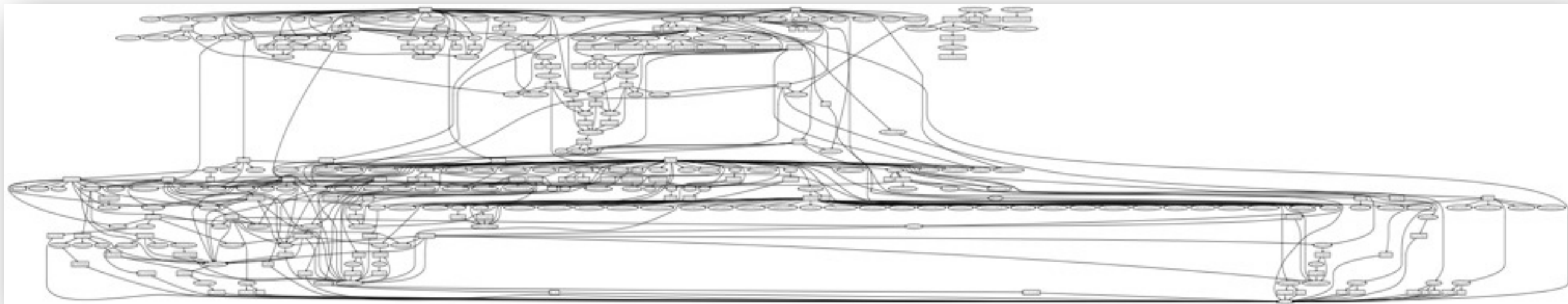


# Surviving in the mid term

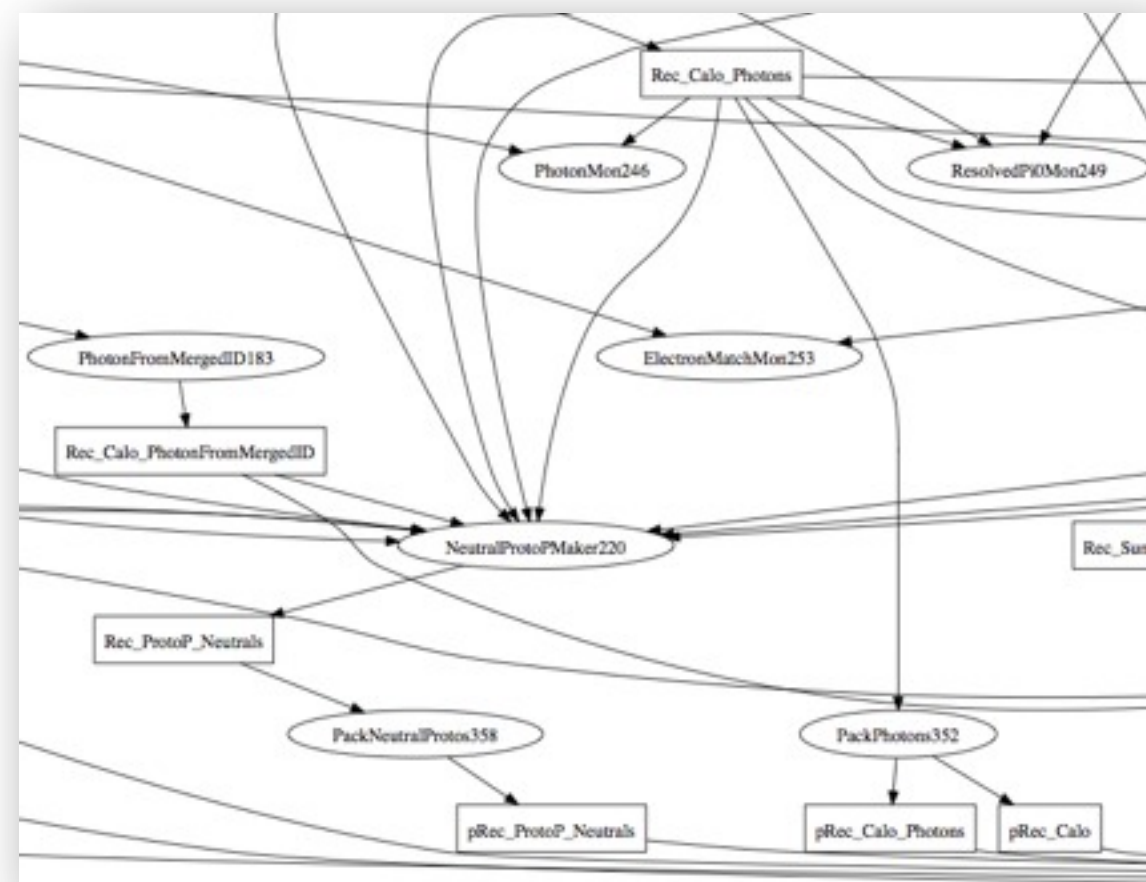
- Framework with the ability to schedule modules/ algorithms concurrently
- Full data dependency analysis would be required (no global data or hidden dependencies)
- Need to resolve the DAGs (Direct Acyclic Graphs) statically and dynamically



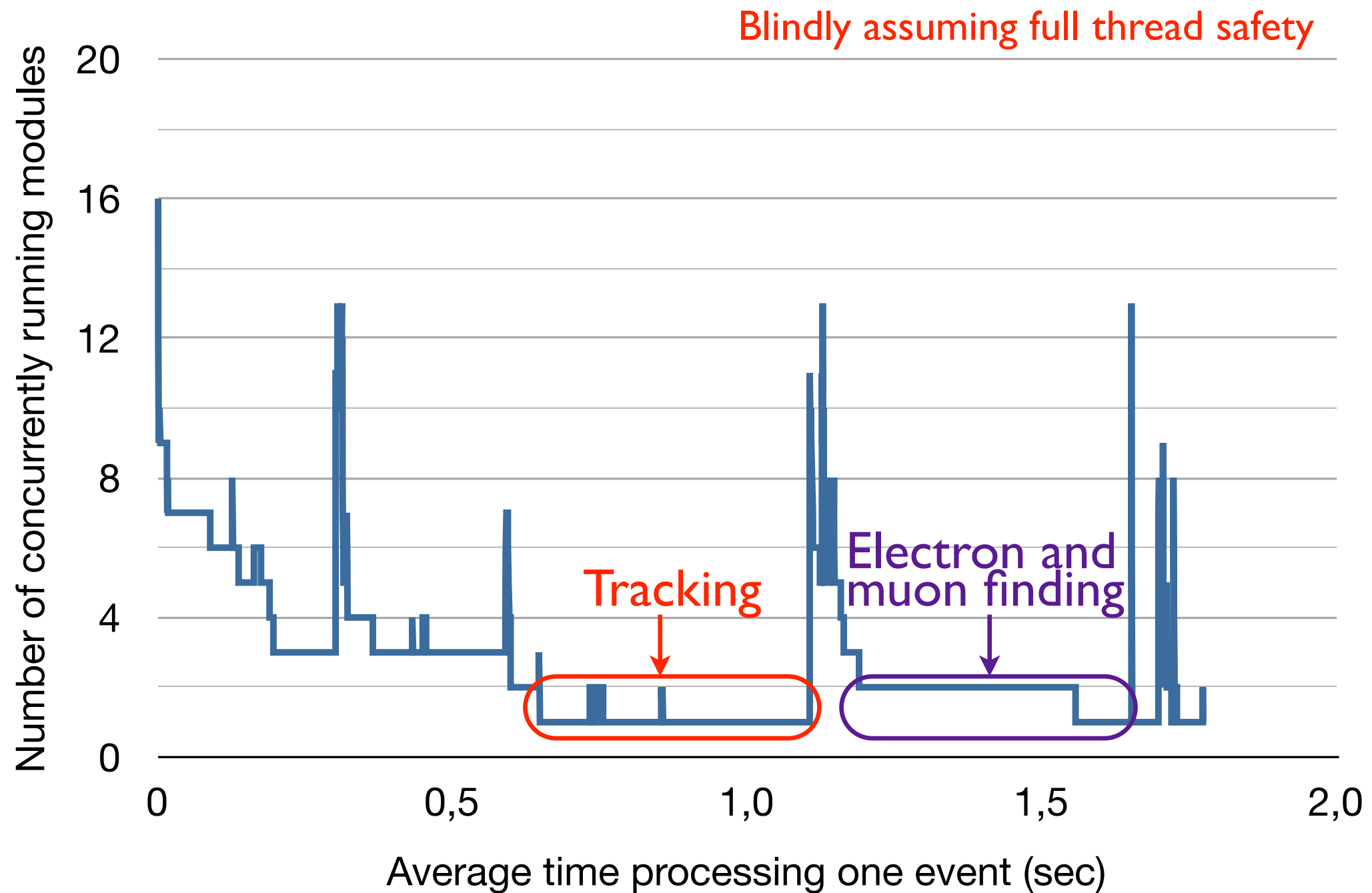
# Real-world example



- 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...

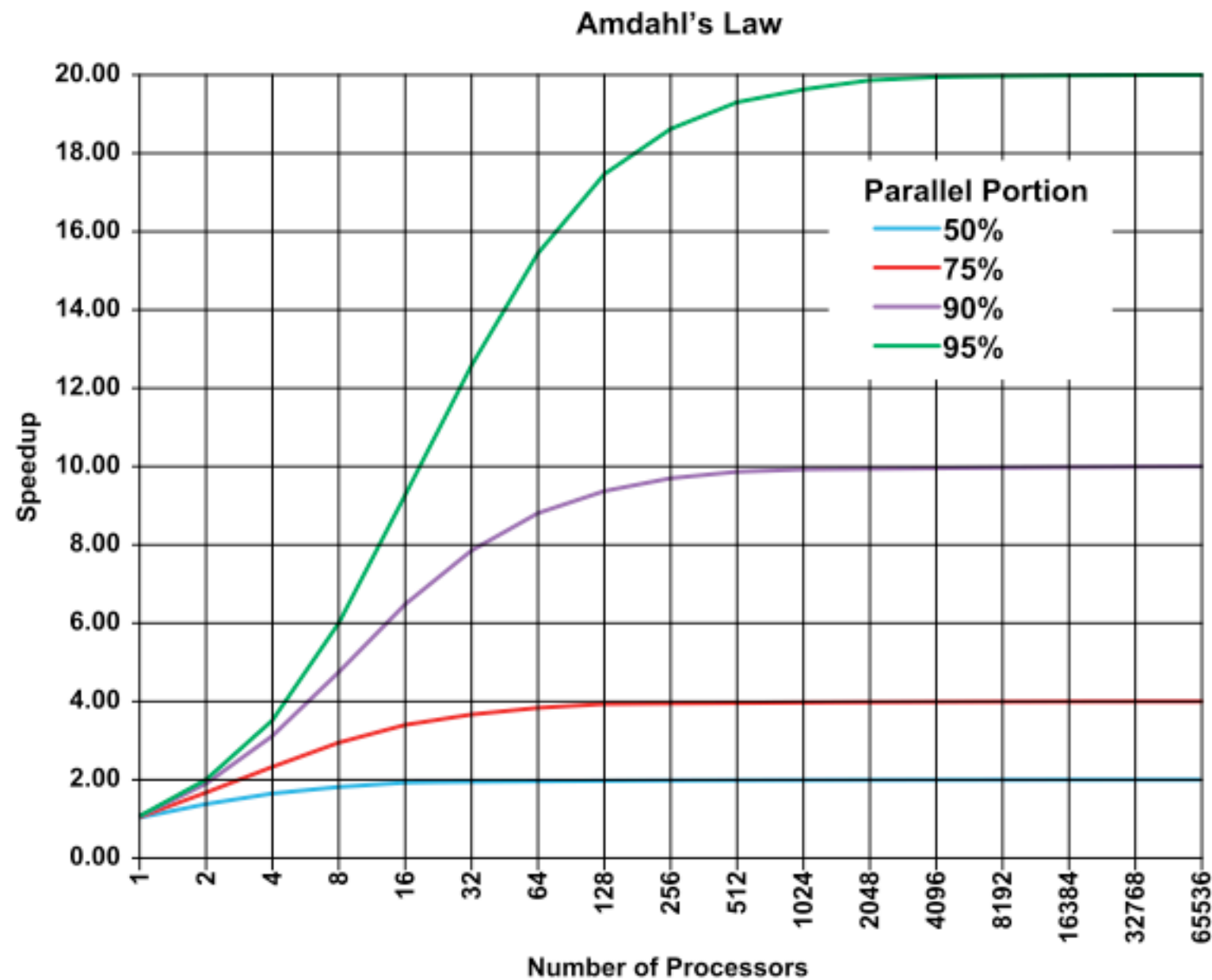


# Unfortunately it doesn't work too well



Remember: tracking will become even worse with more with pile-up

# Amdahl's Law

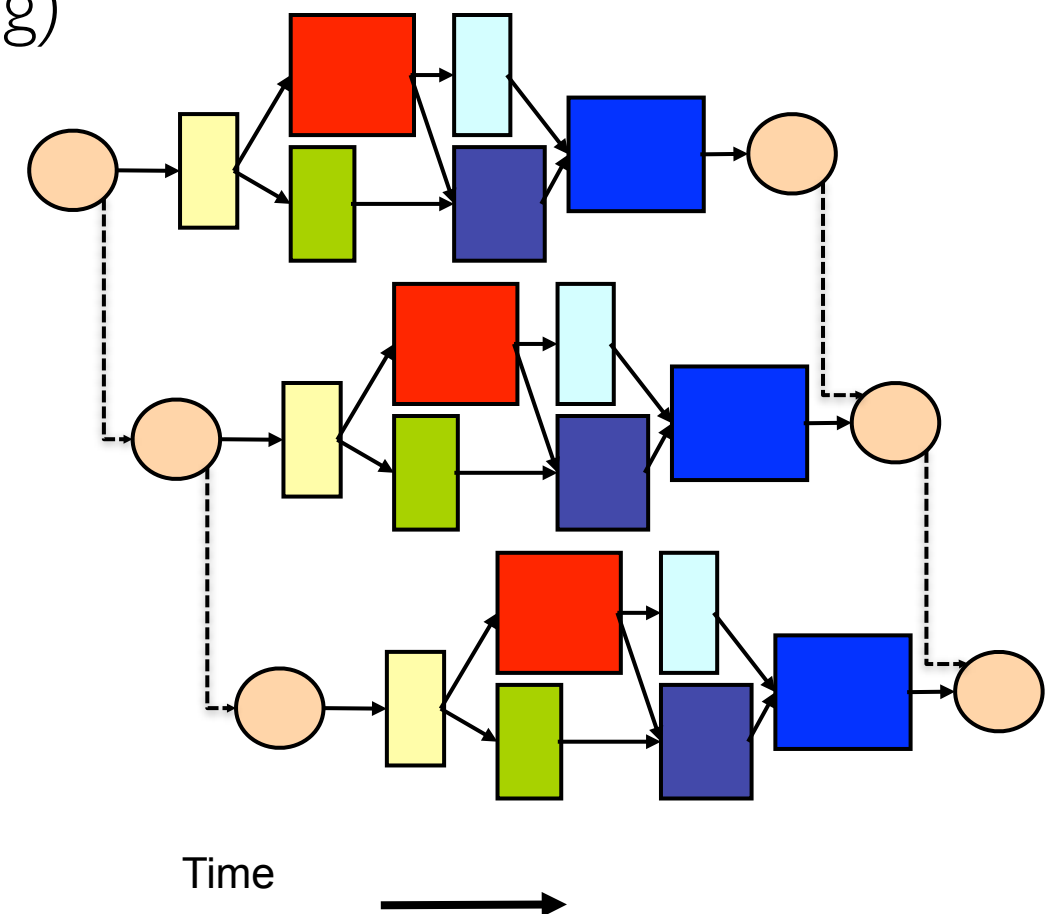


Maximum processing speed is limited by the serial part



# Many concurrent events

- Need to deal with the tails of sequential processing
- Introducing Pipeline processing
  - Never tried before in this context!
  - Exclusive access to resources or **non-reentrant** algorithms can be pipelined (e.g. file writing)
- Need to design or use a powerful and flexible scheduler
- Need to define the concept of an “event context”




$$t = t_1 / N_{\text{thread}}$$

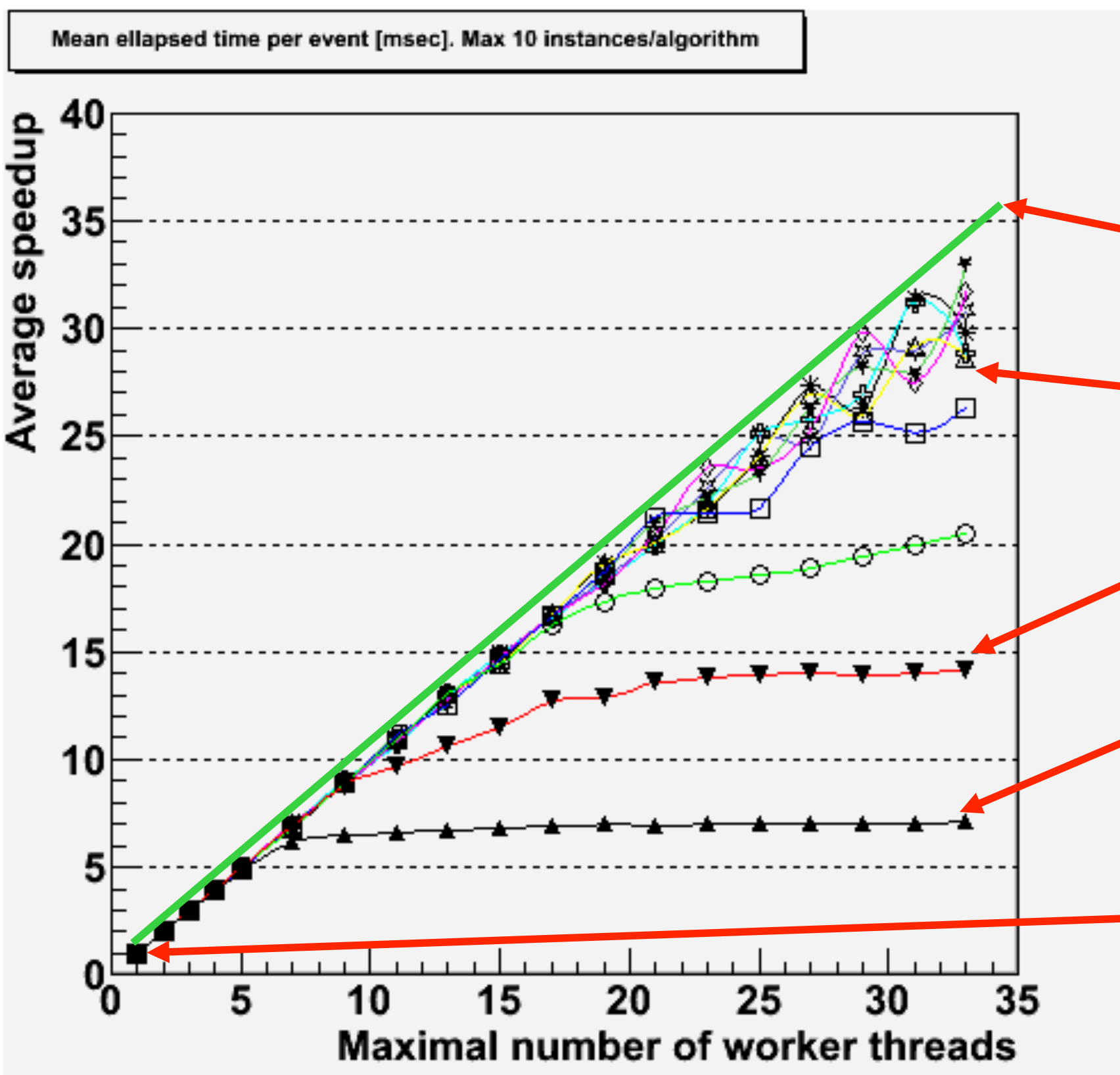
Speedup up to  $\sim 30$

1 event \* 2

Algorithmic parallel limit  
Speedup:  $\sim 7$

= classic processing ( $t_1$ )





- Max 10 events in parallel
- TOP 4 algorithms reentrant with max 10 instances
- Cut 25 msec [4.3 %]

**Theoretical limit**

**Max evts > 3**  
Speedup up to ~30

**Max 2 events**  
1 event \* 2

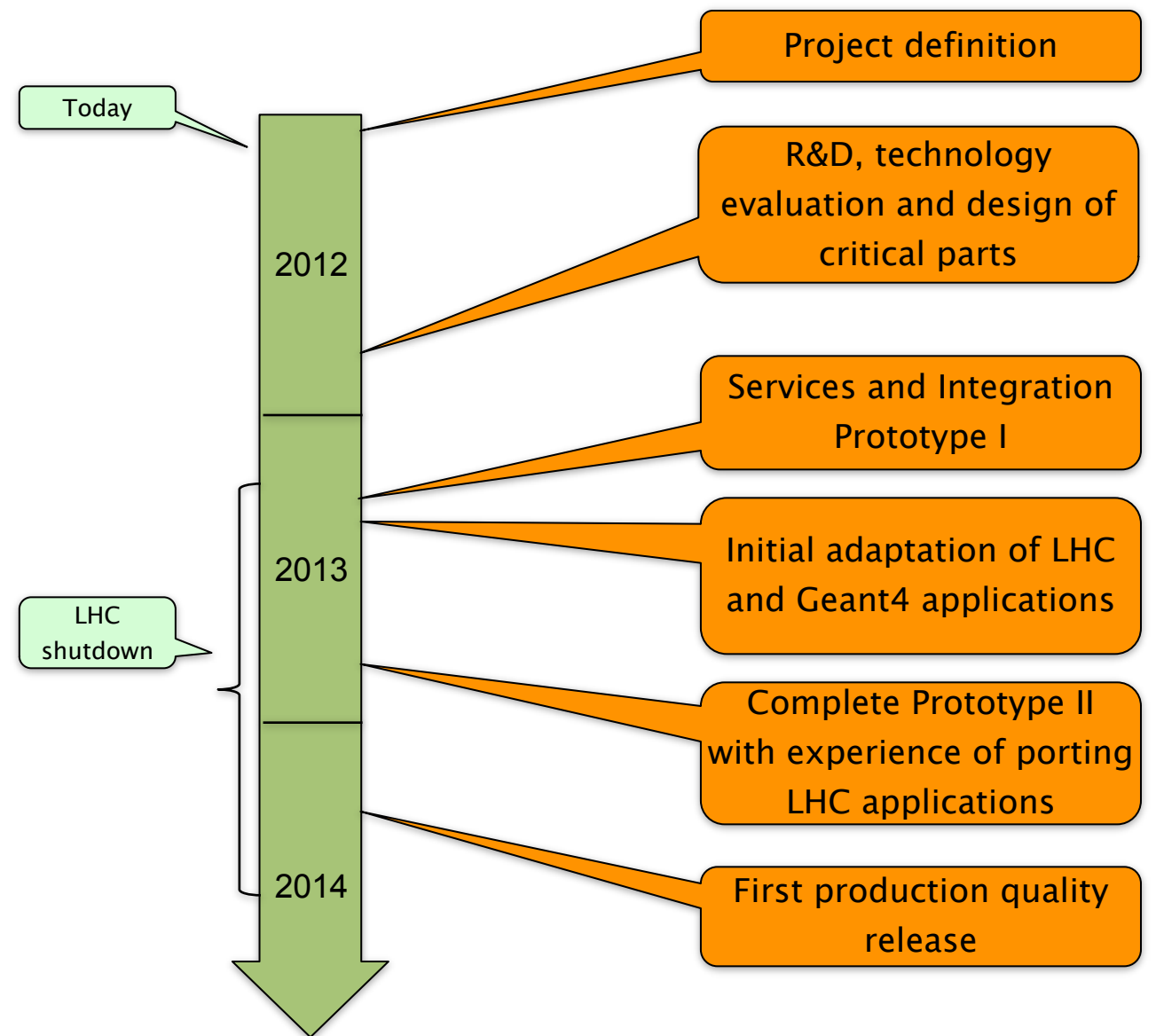
**Max 1 event**  
Algorithmic parallel limit  
Speedup: ~7

**One thread**  
= classic processing ( $t_1$ )

# Moving to implementations

- LHC experiments and big labs started **concurrency forum** (\*) to discuss and tackle problems
- Development of functional components the experiments can choose from
- Collaborations have a huge investment in 'algorithmic' code based on their frameworks
- Currently in 'demonstrator' phase
- Adiabatic migration of algorithms
- Ambitious, but needs to be there in time for LHC re-start in 2015

## Straw Man Project Timeline

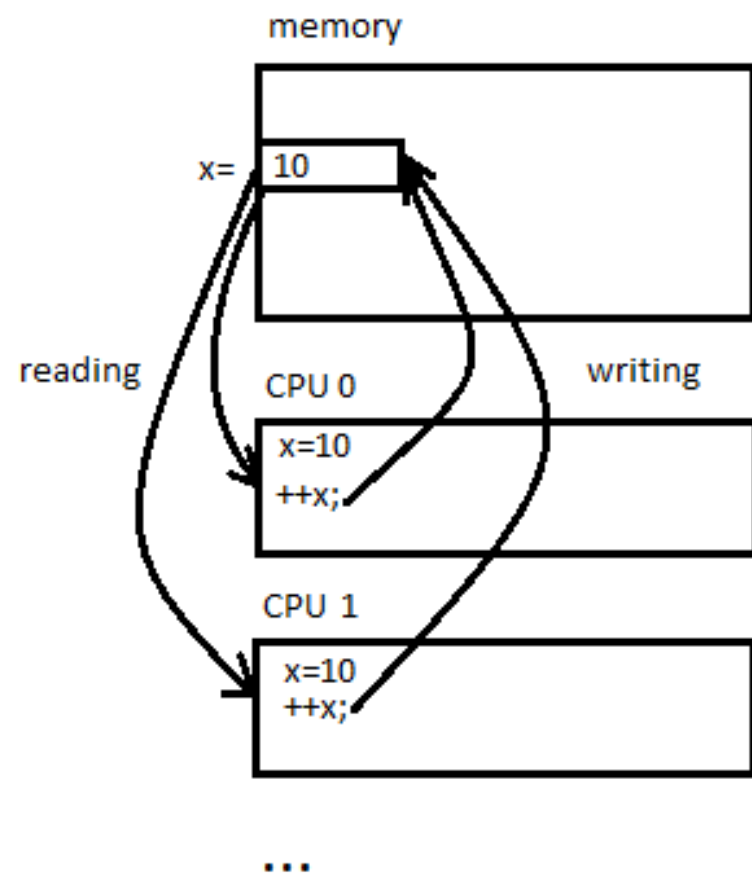


(\*) <http://concurrency.web.cern.ch/>

# Writing thread safe data

- All event state is contained in an object accessed concurrently by hundreds of objects
- How to make such data thread safe?
- Transactional memory could be a solution!
  - Treat memory access like DB transactions
  - gcc 4.7 contains experimental support
  - Future Intel CPUs will support it on HW level (starting with Haswell in 2013)

# What's the problem?



Serial result:  $x=12$

Parallel result:  $x=11$

# What are the solutions?

## Lock based

x= 10

CPU 0

```
try lock;  
lock;  
  
++x;  
  
unlock;
```

CPU 1

```
try lock;  
wait;  
  
try lock;  
lock  
  
++x;  
  
unlock;
```

time

## transactional memory

x= 10

CPU 0

```
transaction{  
  ++x;  
  commit{  
    x was 10  
    x now 10  
    update x;  
  }  
}  
x=11;
```

commit successful

CPU 1

```
...  
  
trasaction{  
  ++x;  
  commit{  
    x was 10  
    x now 11  
    repeat;  
  }  
  ++x;  
  commit{  
    x was 11  
    x now 11  
    update x;  
  }  
}  
x=12
```

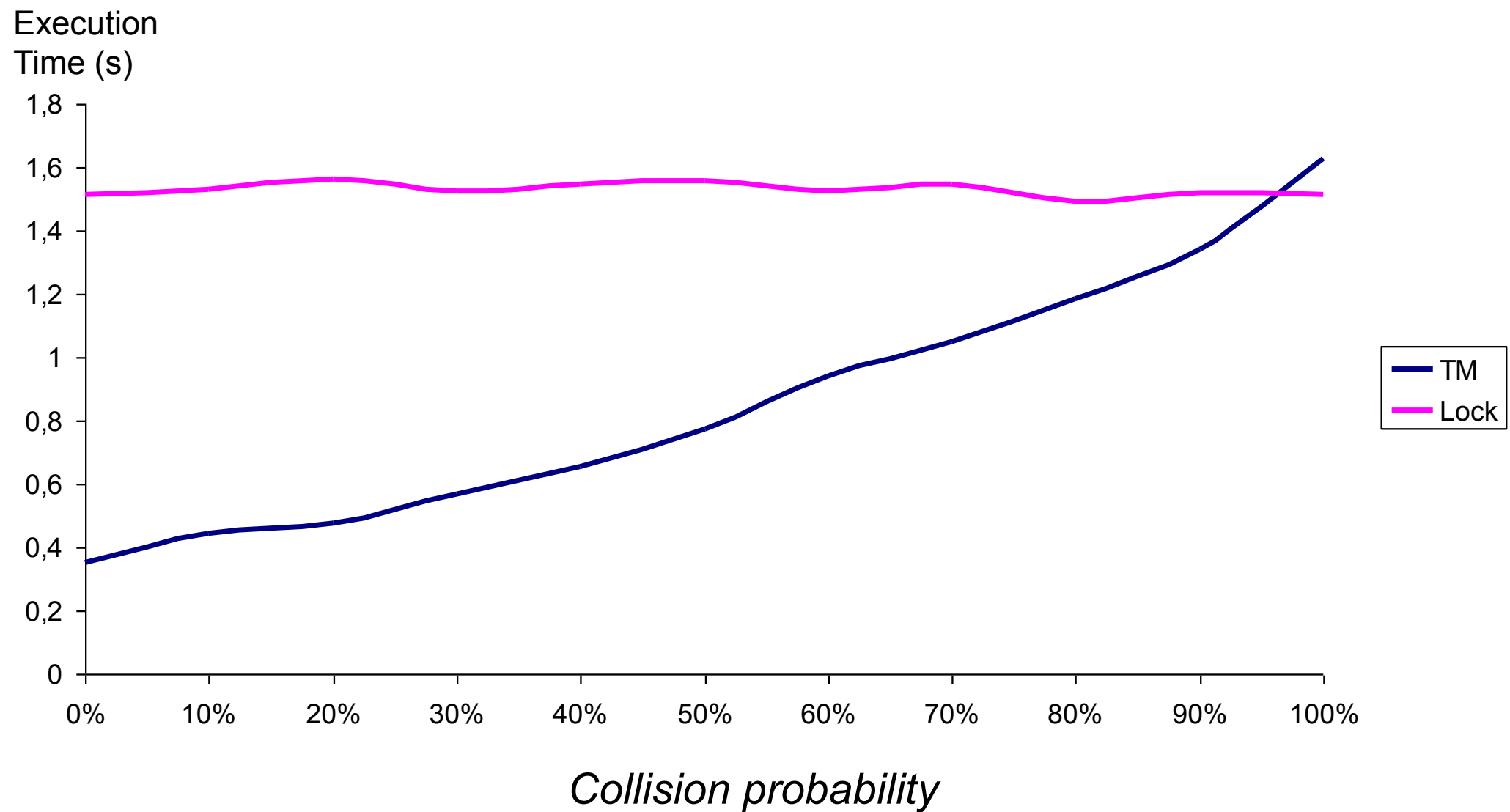
commit unsuccessful

commit successful



# Performance behaviour

based on transactional memory in gcc 4.7



plot by Audrius Pakalniškis  
(CERN summer student)



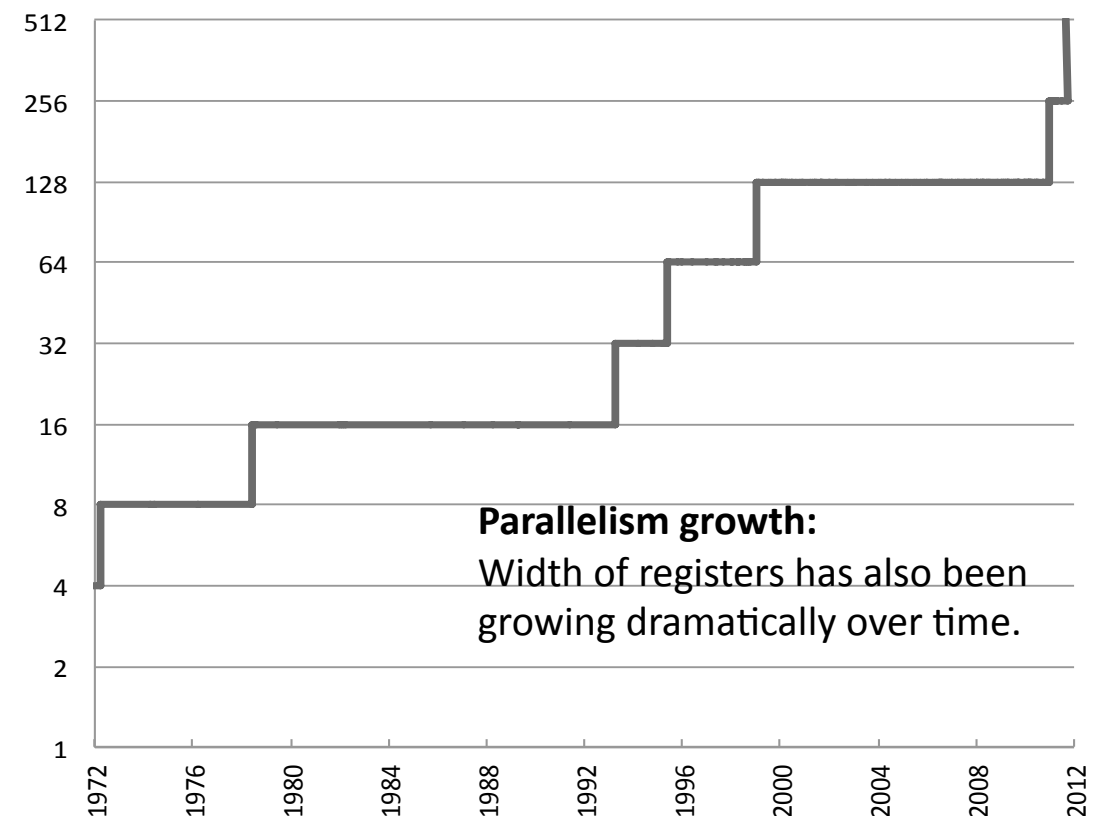
The top of the slide features a blue header with a background of particle physics diagrams. On the left, a graph shows a curve with a peak labeled  $\mu = 500 \text{ GeV/c}$ . Below it, text reads  $H, A \rightarrow \tau^+\tau^- \rightarrow \text{two } \tau \text{ jets} + X, 60 \text{ fb}^{-1}$ . On the right, there is a circular diagram resembling a particle detector cross-section.

# **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



## 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:

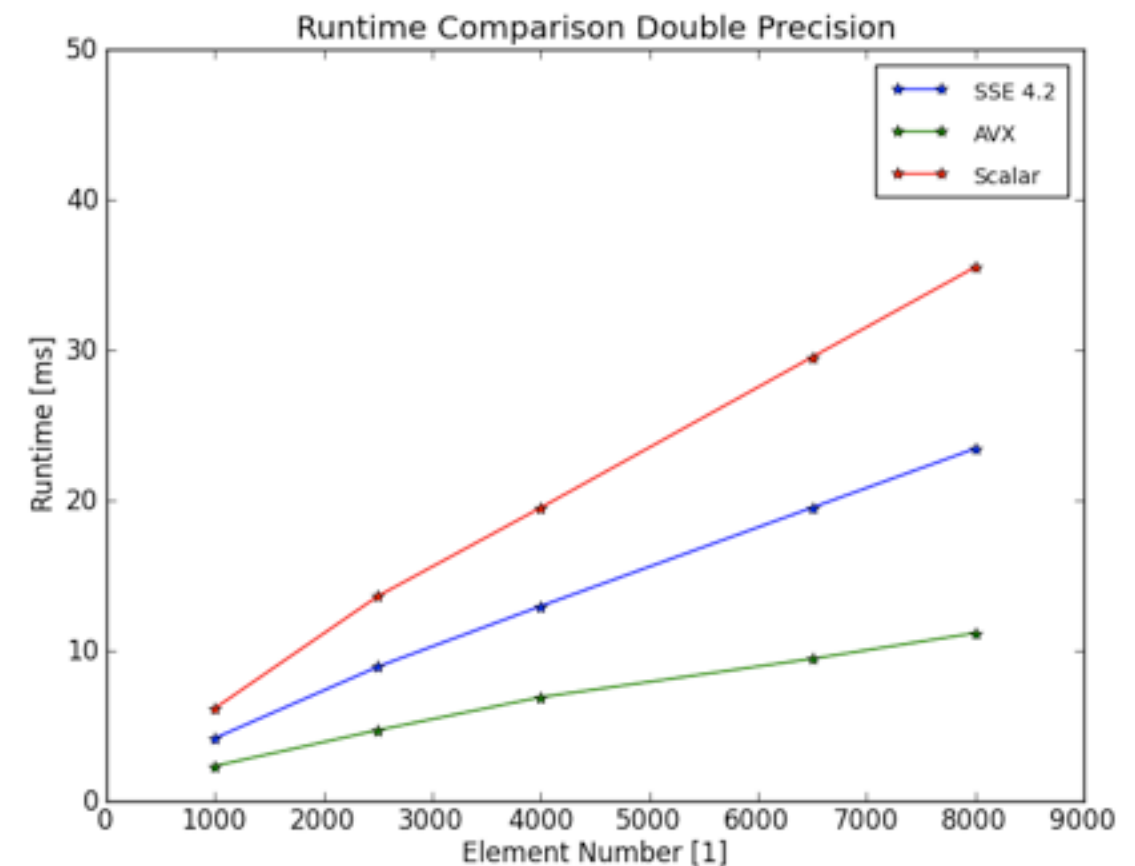
```
double* x = new double[ArraySize];
double* y = new double[ArraySize];

...

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;
    }
}
```

+

gcc4.6 and -ftree-vectorize



# SIMD example

Just an 'academic' example:

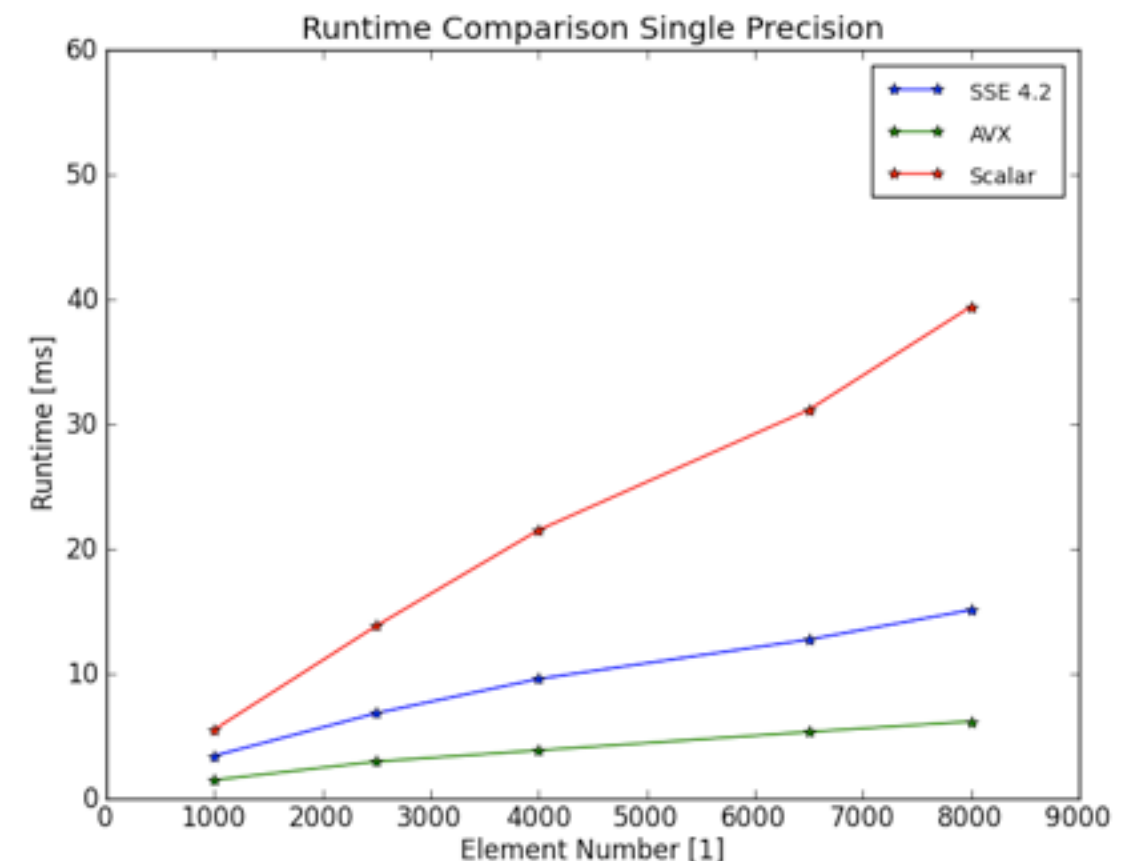
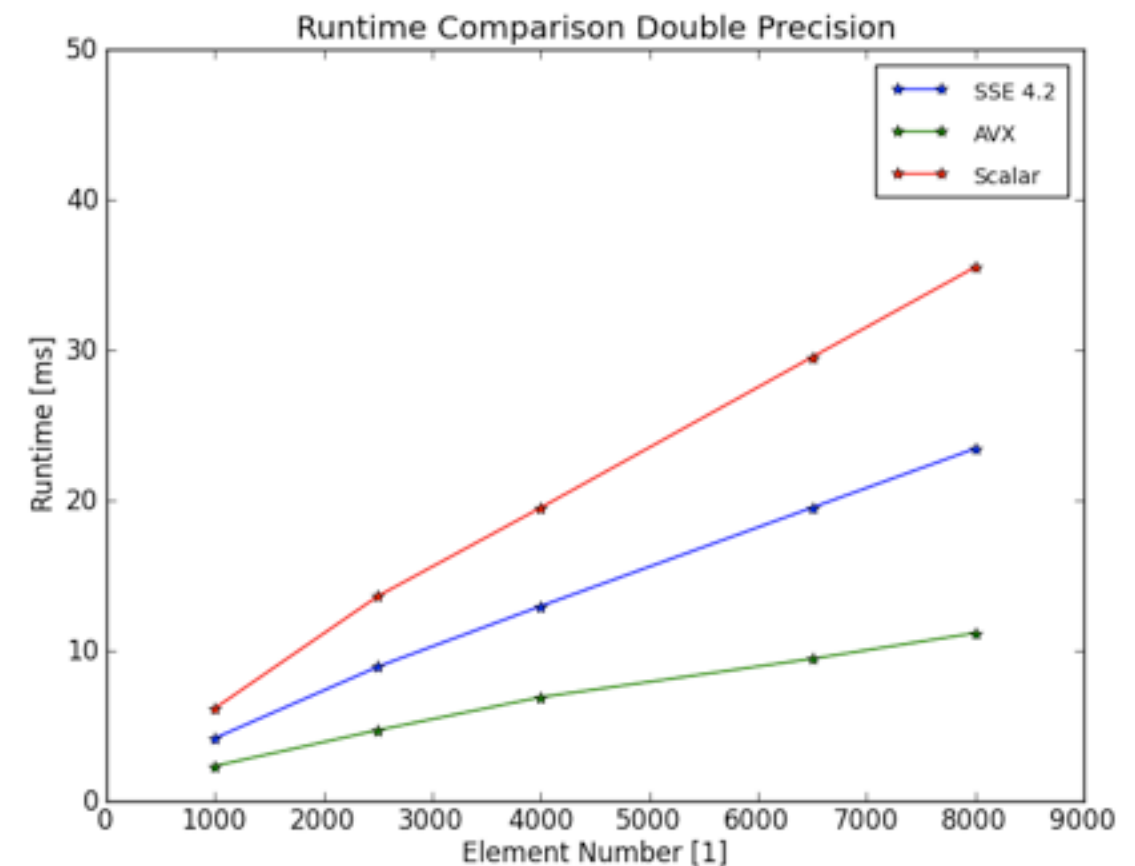
```
double* x = new double[ArraySize];
double* y = new double[ArraySize];

...

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



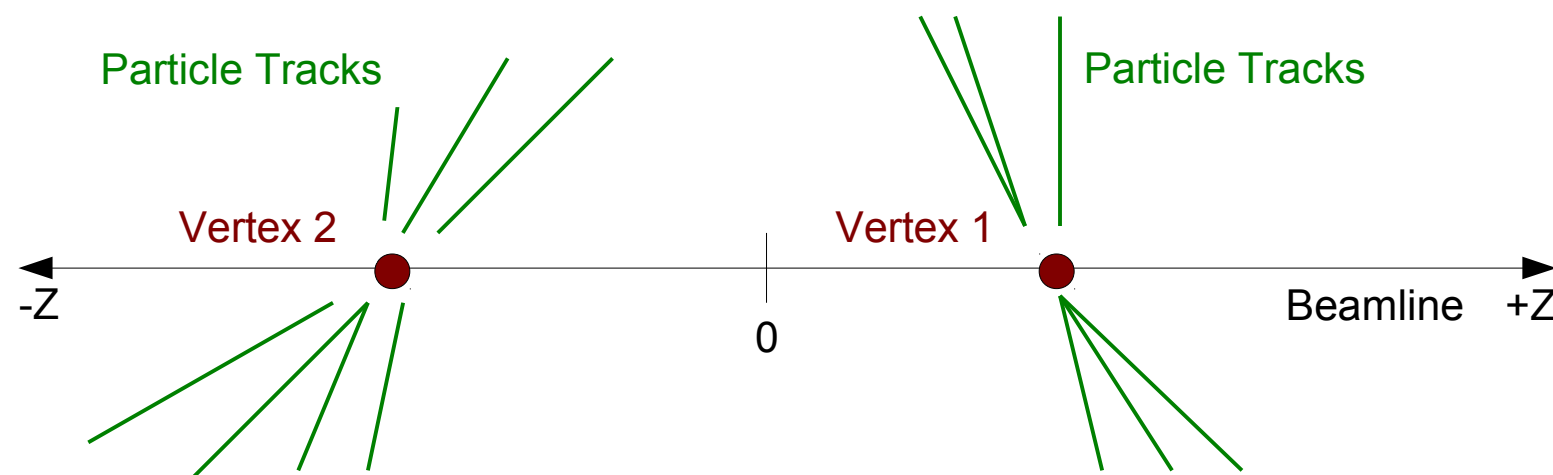


# Turning that into reality

## 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

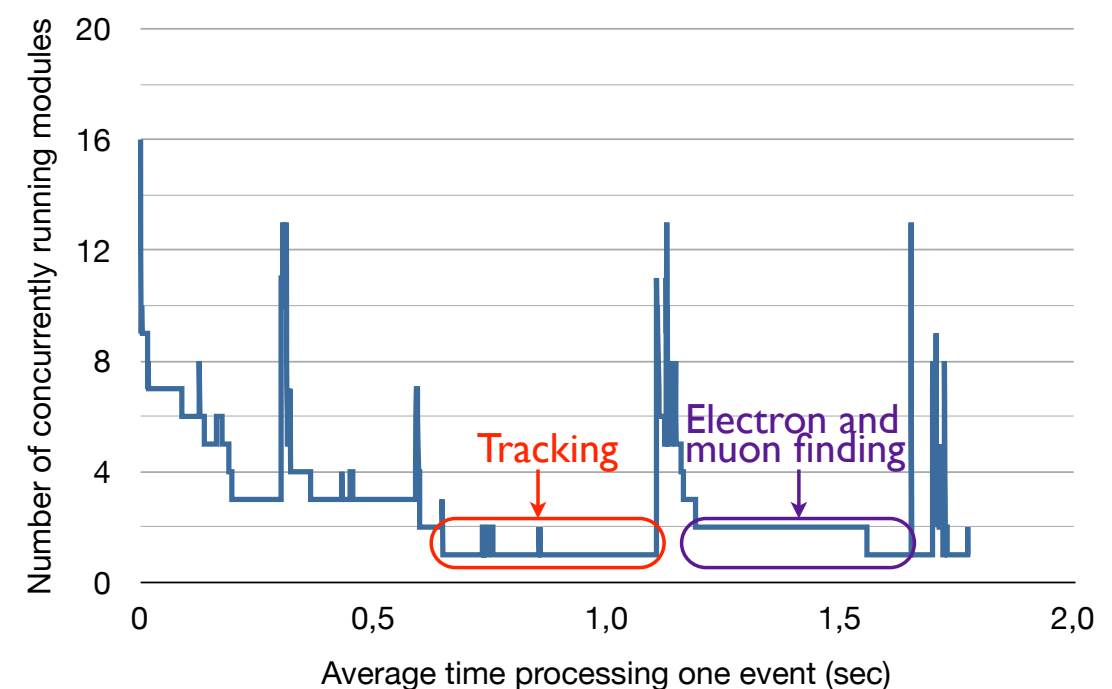


Version	Runtime for 50 Events [s]	Ratio [1]
Regular	26.64	1.0
Vectorized	19.96	0.74
Vectorized + vdt math	11.46	0.43



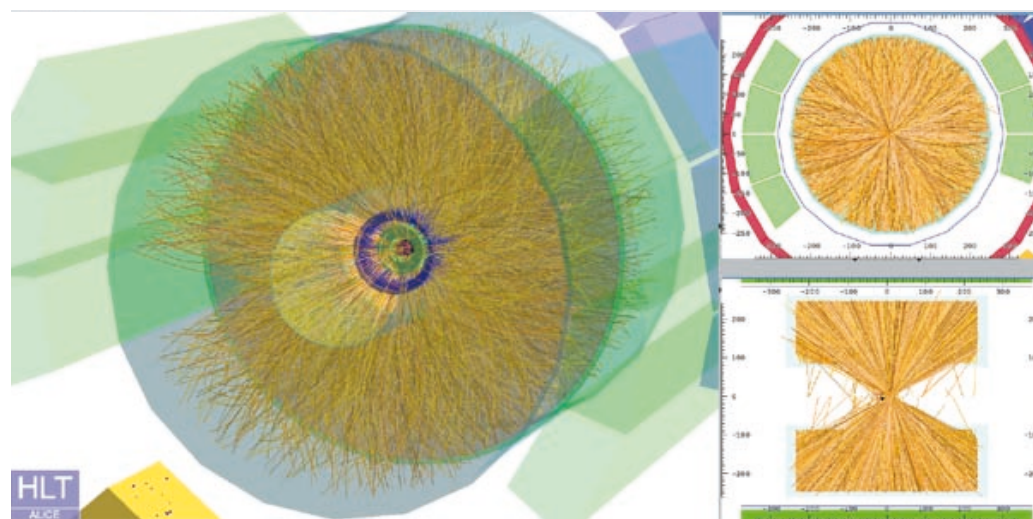
# Long-term solution

- 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 Status

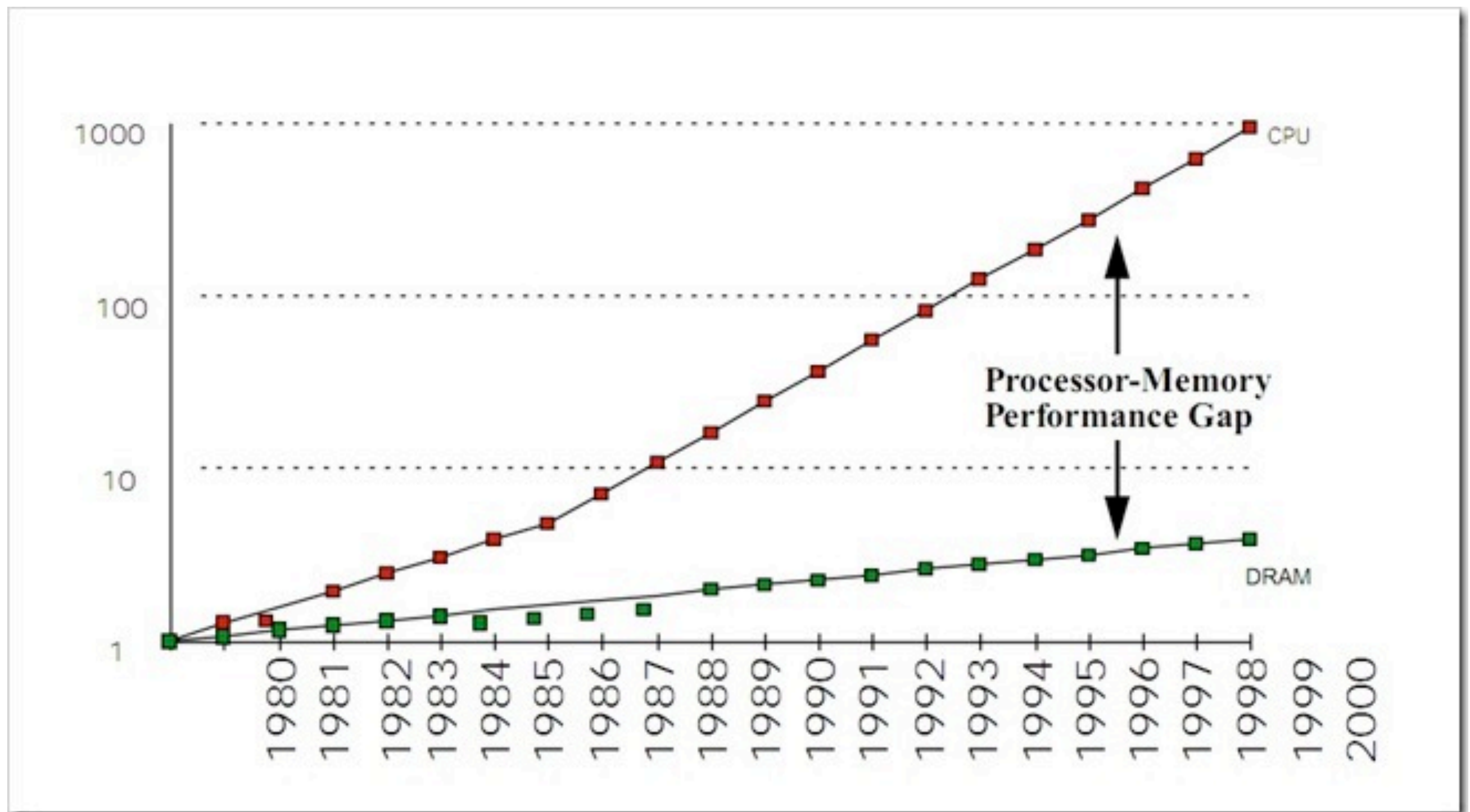
- ATLAS already made some efforts
  - Found a potential for an improvement by an order of magnitude
  - Implemented seed finding for Level-2 trigger
  - Raw data pre-processing for Level-2 trigger
- ALICE trigger using simplified GPU-based tracking





# The Memory Wall

# Memory Speed Development

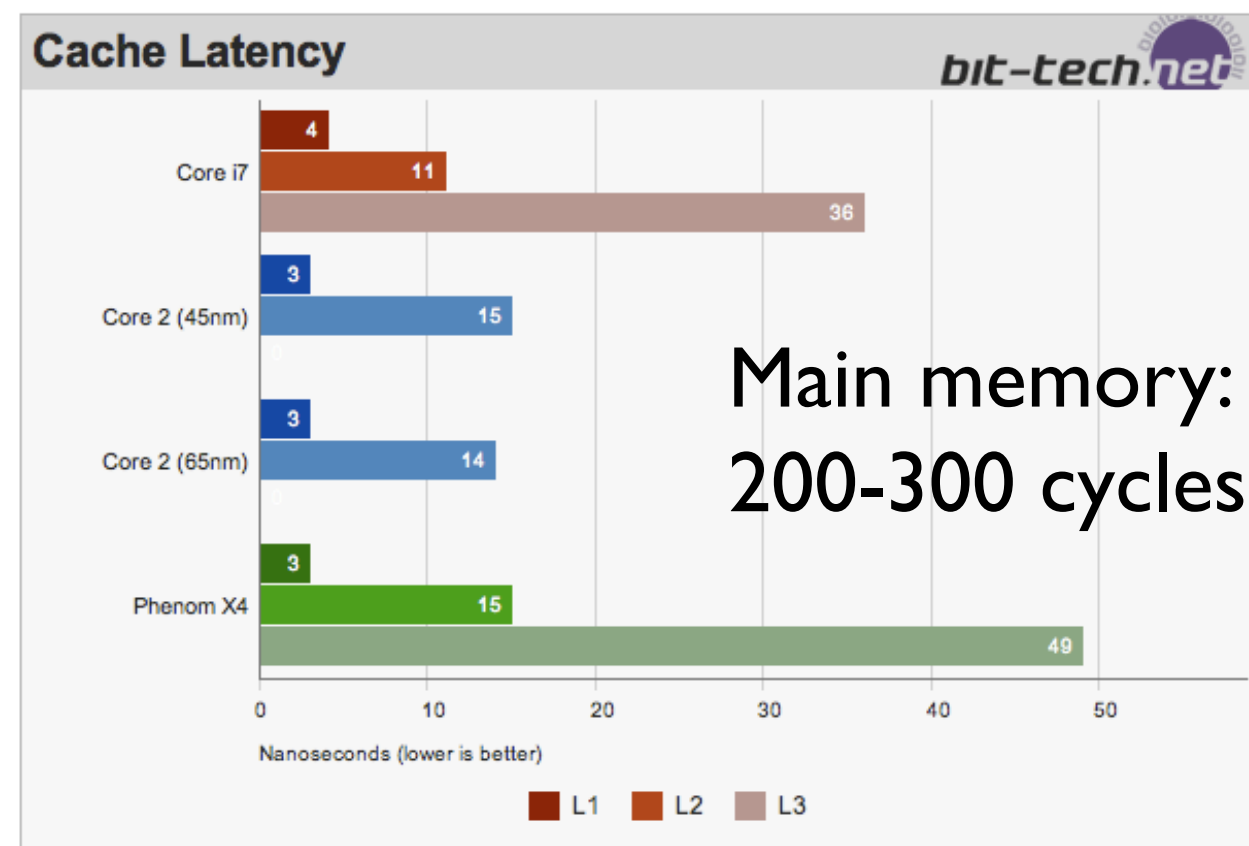
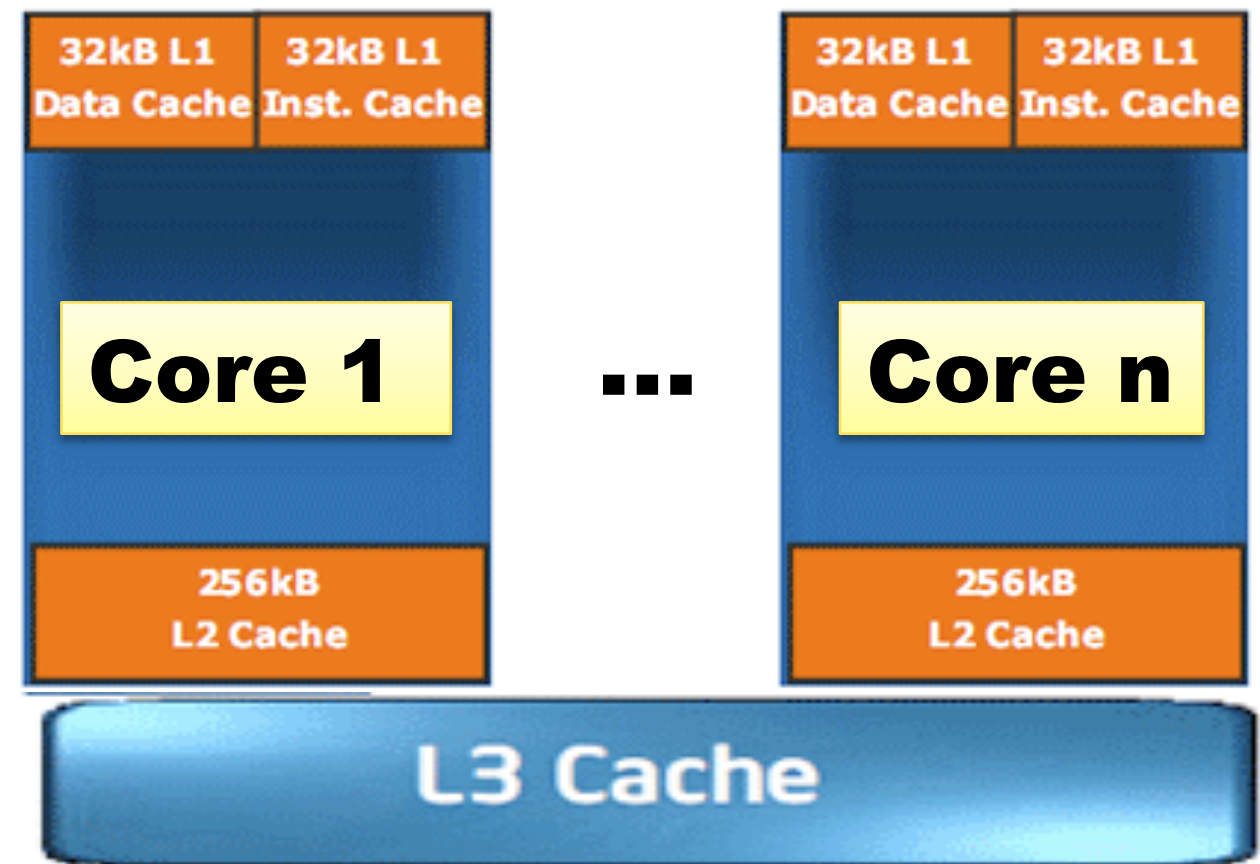


More than a factor 100 !



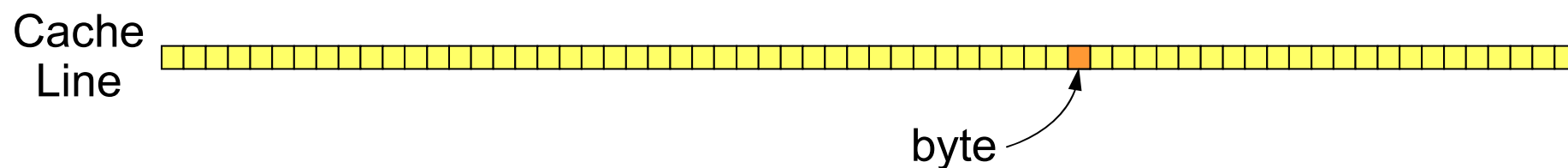
# The 'Memory Wall'

- 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 does not solve it





- 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)

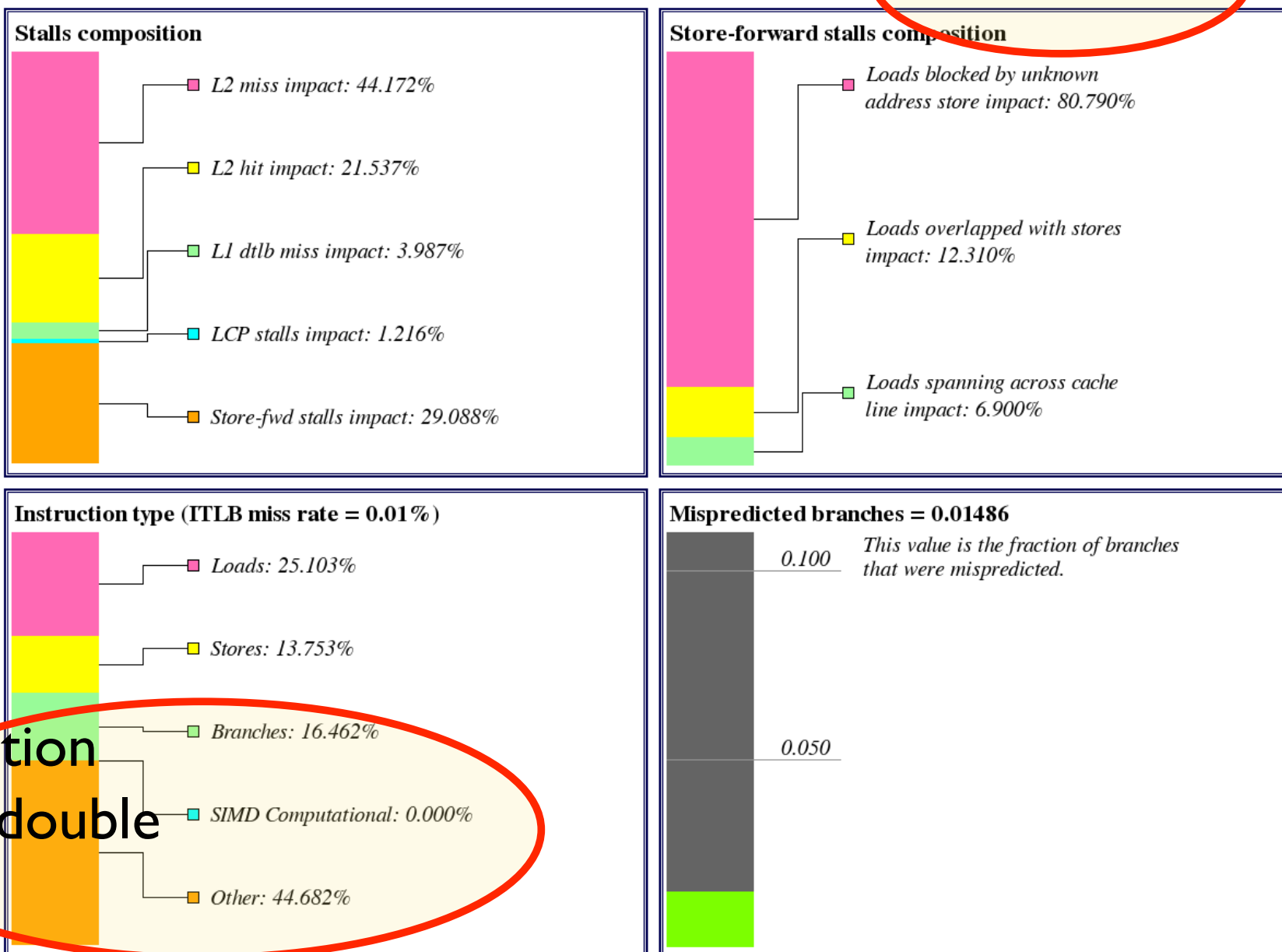
**Very tiny compared to main memory!**

# Dominated by data movement NOW!

## We use only 15% of available “d”flops

60% “active”

EcalRawToRecHitProducer\_hltEcalRecHitAll - CYCLES: 25708037 - STALLED: 40.2% - CPI: 0.98



50%  
“computation  
on single/double  
word”

# Where are we now?

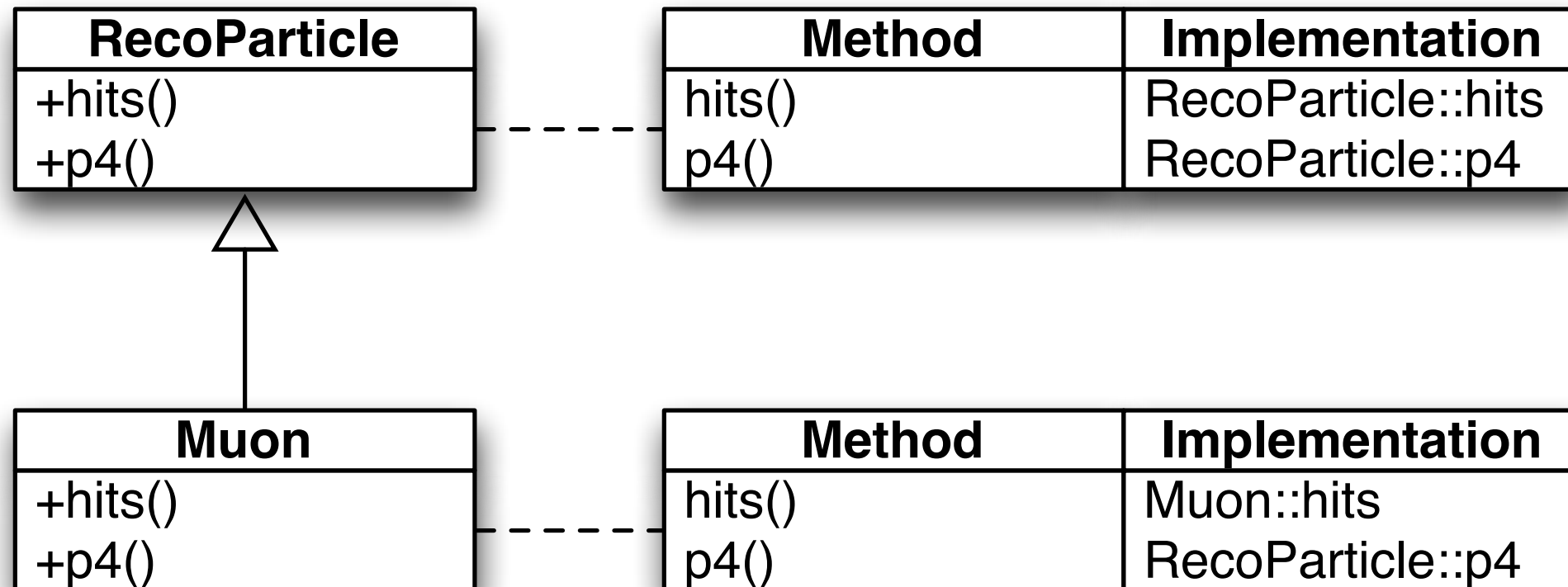
	SIMD	ILP	HW THREADS	CORES	SOCKETS
MAX	4	4	1.35	12	4
TYPICAL	2.5	1.4	1.25	8	2
HEP	1	0.55	1	6	2

Thanks to Andrzej Nowak for the table

<http://indico.cern.ch/getFile.py/access?contribId=2&resId=0&materialId=slides&confId=186554>

# Little Reminder - vtable

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



# The death for any cache

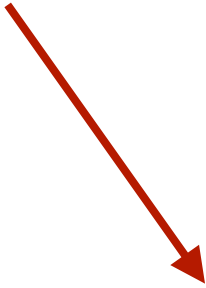
Let's consider the following code and its first execution:

```
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

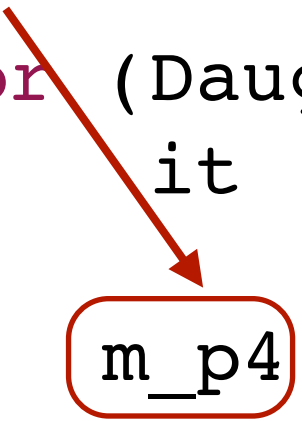
```
for (DaughterIt it = m_daughters.begin();  
     it != m_daughters.end(); ++it)  
{  
    m_p4.Add( it->p4());  
}
```

**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?

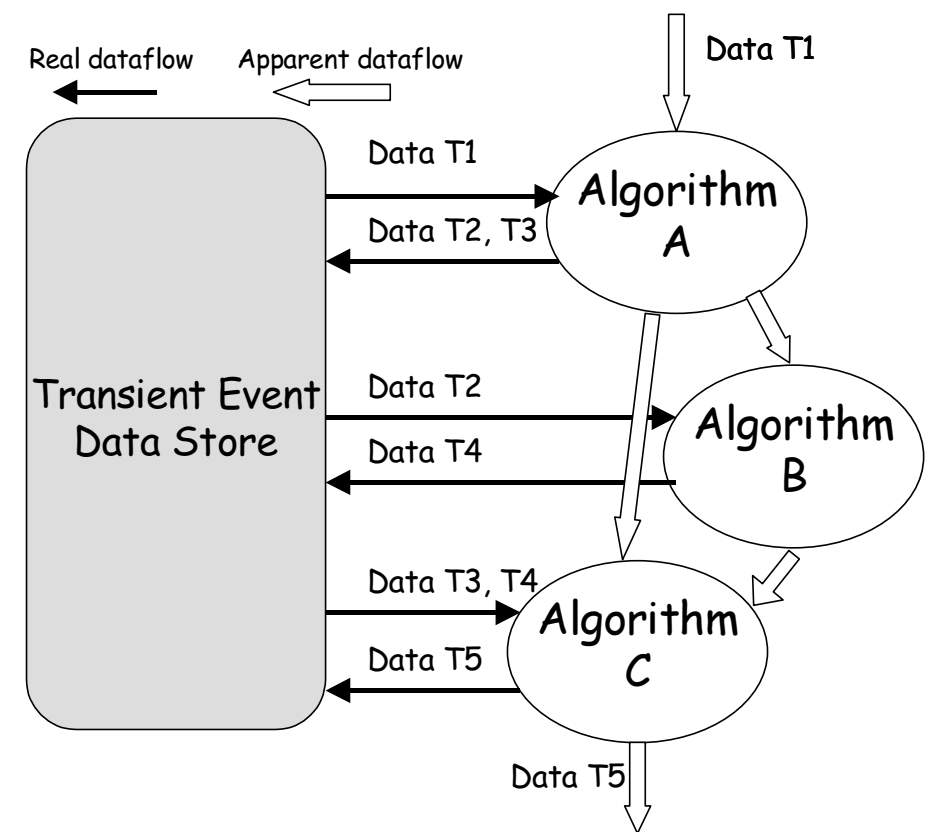
- **OOP as dreamed of in the books**

- 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

- **‘Deprecated’ FORTRAN-legacy might turn out to be not so bad after all...**





# What's ahead of us?

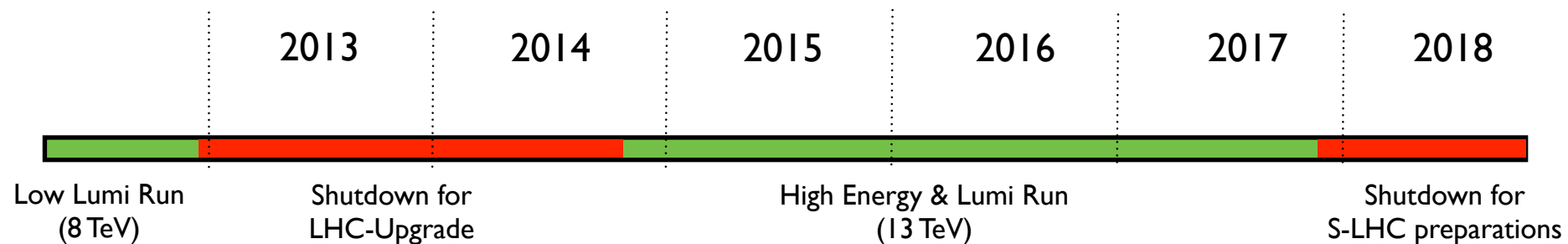
- **We have to choose with more thought when to follow which programming paradigm**
  - Many identical data chunks & high throughput => data oriented
  - Small number of objects & heterogenous data => object oriented
- **For reconstruction we have to redesign our data formats to become even dumber**
  - 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
- Various R&D efforts going on these days
- Exciting times if you are interested in programming!

# Challenges Summary

- Process multiple forked jobs in parallel  
( ~ now )
- Run modules and events concurrently  
( ~ 2014 )
- Split our algorithms into smaller chunks which can be run in parallel, potentially on co-processors (e.g. GPU)  
( ~ 2014 )
- We have to start using Single-Instruction-Multiple-Data (SIMD)  
( as soon as possible )
- We have to learn how to properly separate the “object world” from the “data world”  
( ILD / Belle II / SuperB ? )



**That's it :-)**

