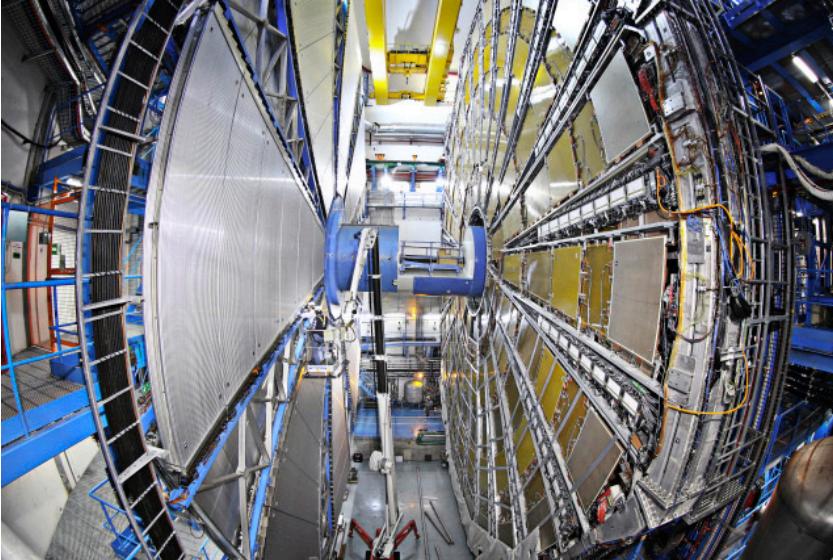


Software Design in the Many-Cores era

A. Gheata, S. Hageböck

CERN, EP-SFT

CERN School of Computing 2024



Lecture I

Parallelism in a Modern HEP Data Processing Framework

Outline of This Lecture

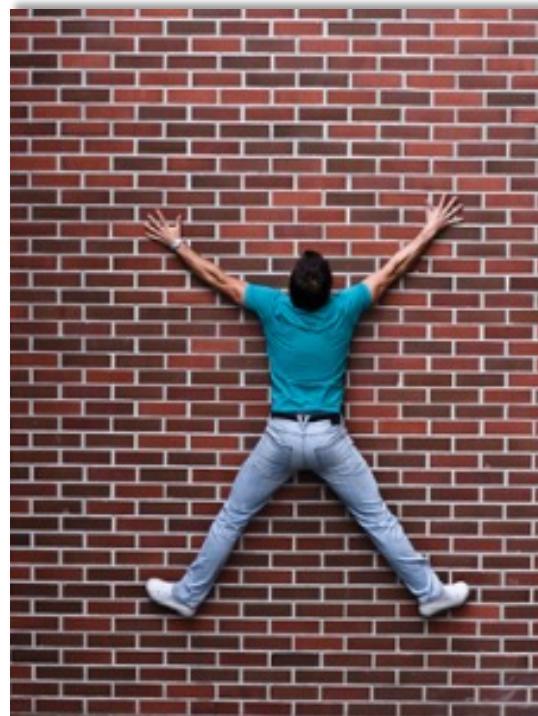
The Goals:

- 1) *Understand why we need parallelisation*
- 2) *Understand the problem domain of physics processing*
- 3) *Break down big problems into work items that can be tackled in parallel*
- 4) *Be aware of the limitations for parallelisation*

- From sequential to parallel
- Experiment Frameworks: basic principles, design
- Laws of parallelism
- Concurrency Models: task-based parallelism

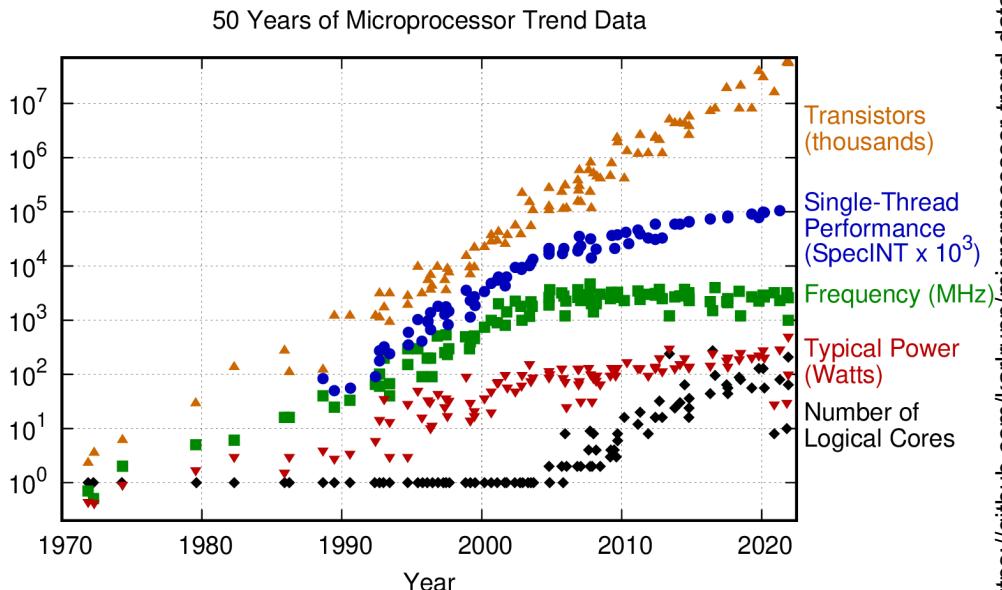
Hitting the Wall(s)

- Once upon a time, the life of software developers was much easier
 - Sequential programming
 - Want your program to run faster? Buy yourself a new machine!
- The fairy tale ended in the early 2000s
 - Processor manufacturers had to rethink CPU architectures
 - No more free lunch for software



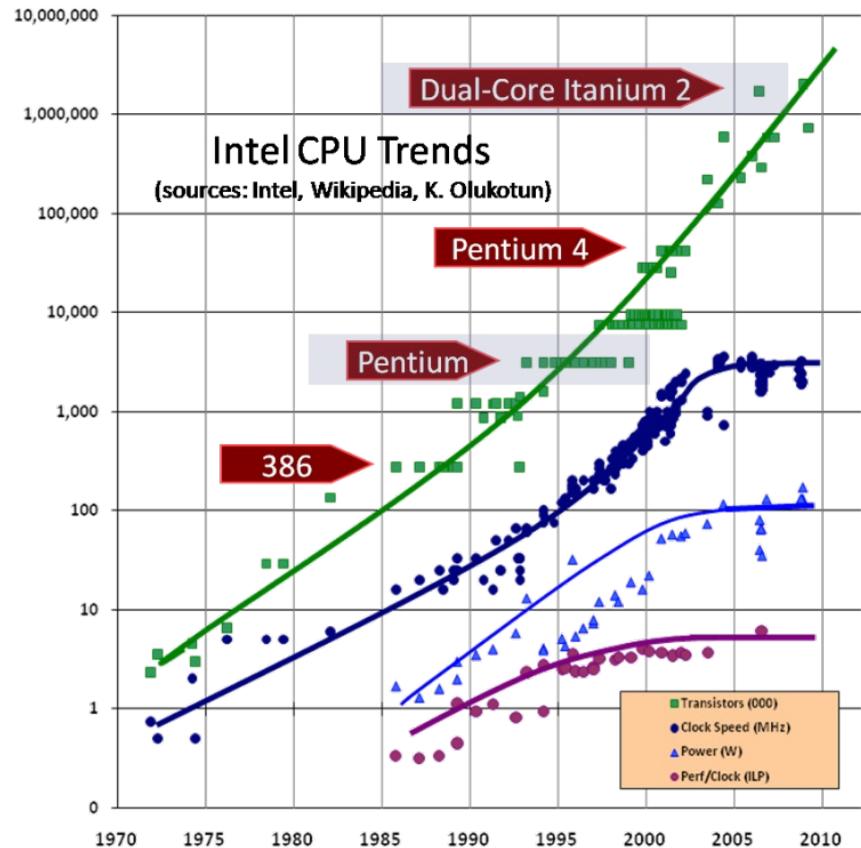
The Power Wall

- Manufacturers could not keep improving processor performance by increasing frequency
 - Not at the same rate at least
- **Power consumption and dissipation** became limiting factors
 - Higher clock rate could lead to overheating



The ILP Wall

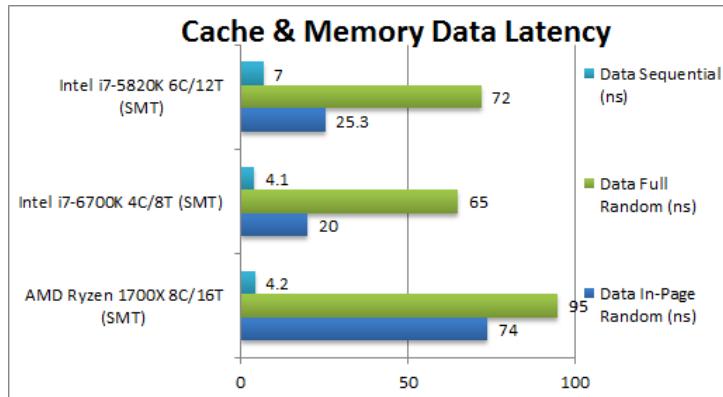
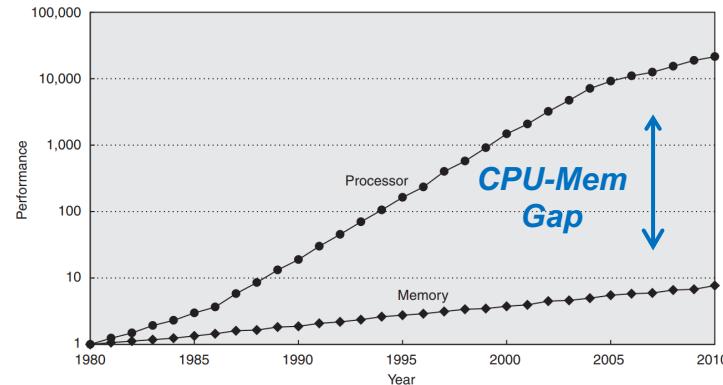
- Processors apply multiple techniques to optimise the execution flow
 - Pipelining
 - Branch prediction
 - Out-of-order execution
 - ...
- **Instruction-Level Parallelism growth also flattened**
 - Hard to squeeze more work out of a clock cycle



<http://www.gotw.ca/publications/concurrency-ddj.htm>

The Memory Wall

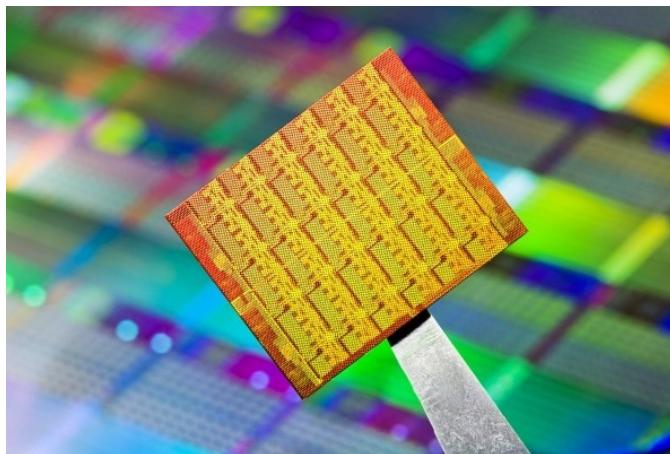
- Processor clock rates have been increasing faster than memory clock rates
- Latency in memory access** is often the major performance issue in modern software applications
- Larger and faster cache** memories help alleviate the problem but do not solve it
- Often the CPU is just waiting for data...



Multi/Many Core to the Rescue

- Let's change strategy
 - Grow by **combining simpler processing units**
 - Moore's law reinterpreted: number of cores per chip will double every two years

*How to make
the most of all
these
resources?*



From Single to Multi/Many core

| | Irwindale | Woodcrest | Gaines-town | Haswell | Broadwell | Skylake | Ice Lake | AMD Epyc 9000 Series |
|-------------------|------------------|------------------|--------------------|----------------|------------------|----------------|--------------------|-----------------------------|
| Year | 2005 | 2006 | 2009 | 2015 | 2016 | 2017 | 2021 | 2024 |
| Cores | 1 | 2 | 4 | 18 | 24 | 28 | 40 | 128 |
| Freq (GHz) | 3.8 | 3.0 | 3.33 | 2.1 | 2.2 | 2.5 | 2.3 (3.4 boost) | 2.2 (4 boost) |
| LL Cache | L2 (2MB) | L2 (4MB) | L3 (8MB) | L3 (45MB) | L3 (60MB) | L3 (38MB) | L3 (60MB) | L3 (256M – 1G) |

Evolution of server processors
<https://ark.intel.com> / <https://amd.com>)

Need for Parallelism

- Change of programming paradigm
 - Need to deal with systems with many parallel threads
 - Improvement in performance comes with **exploitation of concurrency**
- Will all programmers have to be parallel programmers?
 - Different levels of exposure: **explicit vs. implicit** parallelism
 - First step is to **change the way of thinking!**

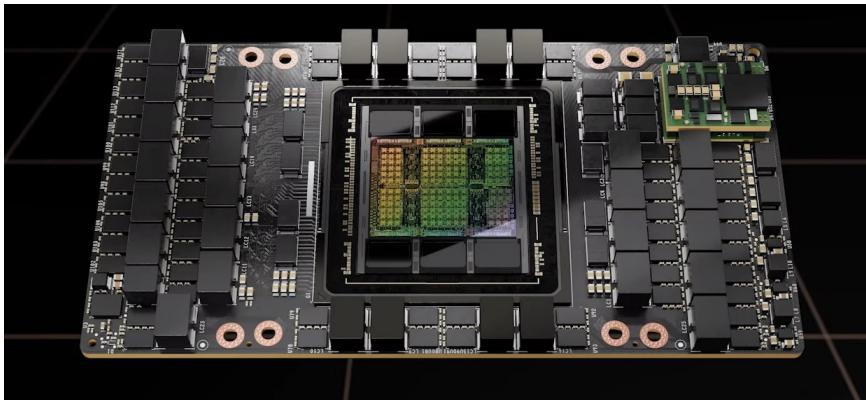
Parallelism is here to stay

A Supercomputer

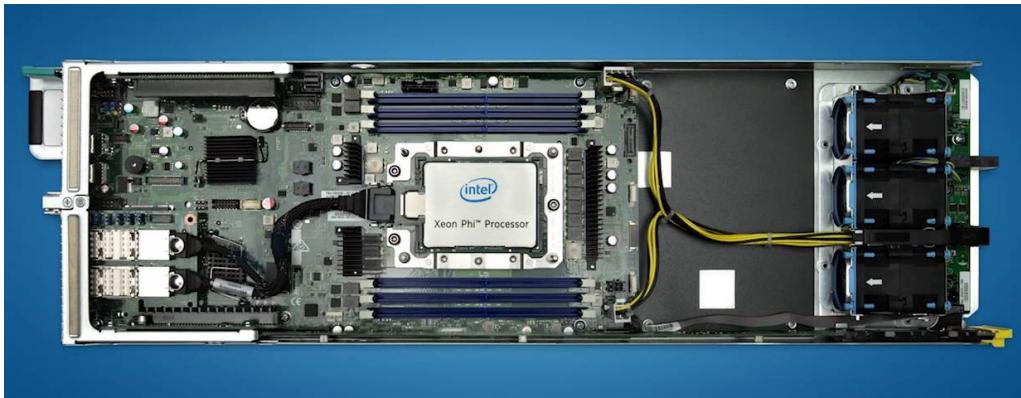
- Perhaps the most striking example of parallelism
 - top500.org: approaching 10M cores (Frontier 2024: 8.7M)
 - Parallelism intra-node and inter-node
 - Multi/many core, hybrid setups: CPU - GPU



Parallel Hardware



Accelerators for massive parallelism



How is Parallelism Achieved?

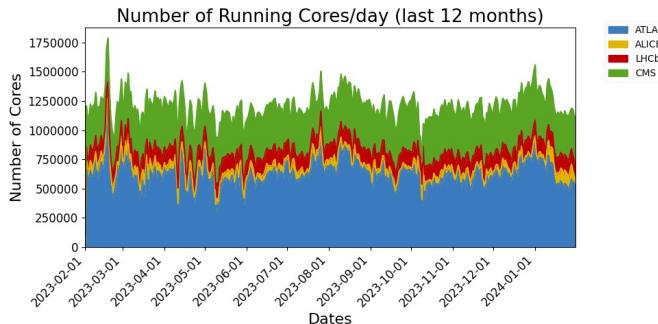
- Supercomputer design tailored for **High-Performance Computing**:
 - Homogeneous nodes (+ accelerators)
 - High-bandwidth low-latency networks (**InfiniBand**)
 - Parallel distributed file system (**Lustre**, GPFS)
- **Explicit** low-level parallelism dominates
 - **MPI** for distributing processes, message passing
 - **OpenMP** inside a node (+ CUDA, OpenCL, SYCL, Kokkos)

Parallelisation in HEP

LHC Computing Grid (WLCG)

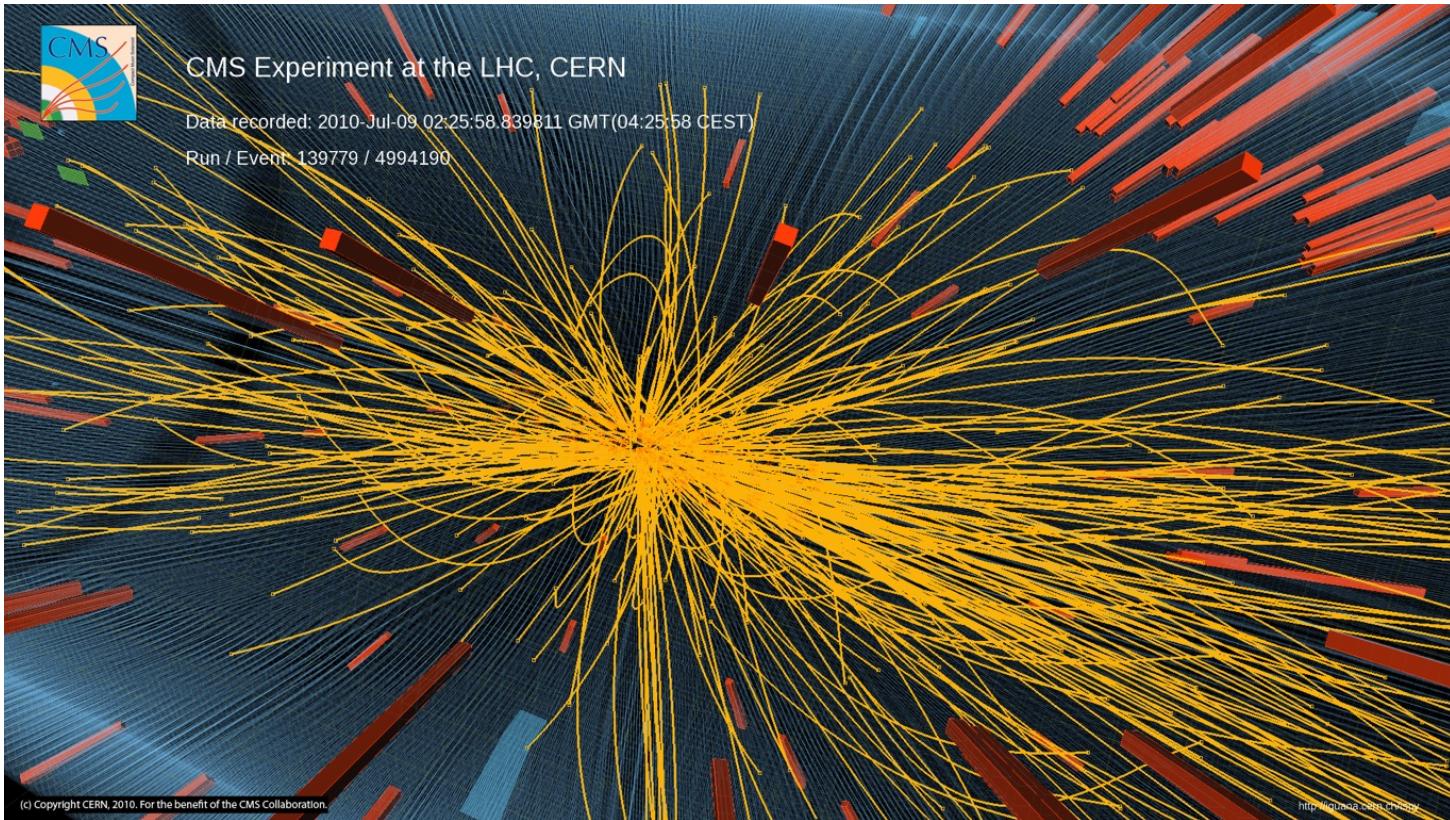
- HEP is parallel since more than a decade
- Computations are distributed among hierarchically organised data centres spread around the globe
- Tens of billions of LHC events are processed per year, running on > 1M cores 24/7 365 days a year

Huge parallel infrastructure!



<https://wlcg.web.cern.ch/using-wlcg/monitoring-visualisation/monthly-stats>

Physics Challenges



Physics Challenges II

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

- Due to the beam intensity (“luminosity”) at the LHC multiple proton-proton collisions take place at once (**pile-up**)
- Pile-up increased further in Run 3 and especially in **HL-LHC**
- As a result, memory consumed by experiments’ reconstruction jobs will go up, making it hard to run many simultaneous jobs on a single computer
 - Independent jobs do not share memory!

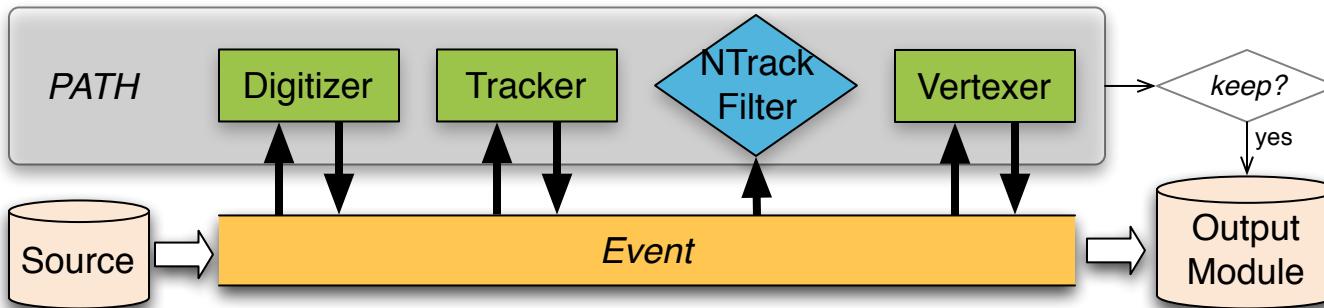
Furthermore:

- **Merging** of results of independent jobs takes significant amount of time

Another parallelisation strategy is needed!

Framework Primer

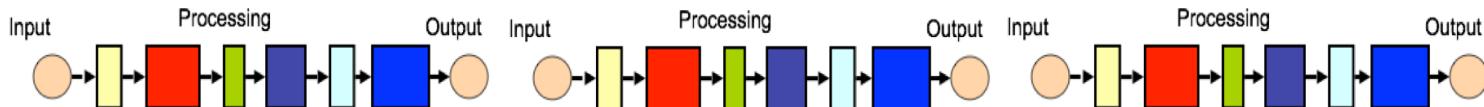
Experiment Software Follows the Idea of a Software Bus



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

Framework Primer II

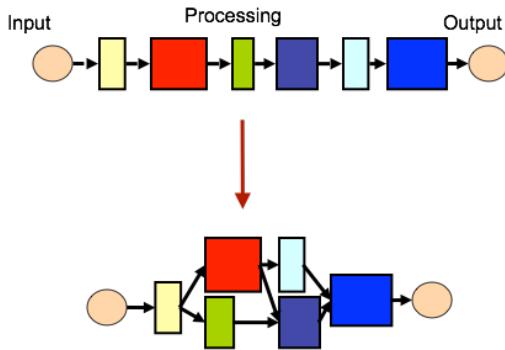
- Multiple events are being processed **sequentially**



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

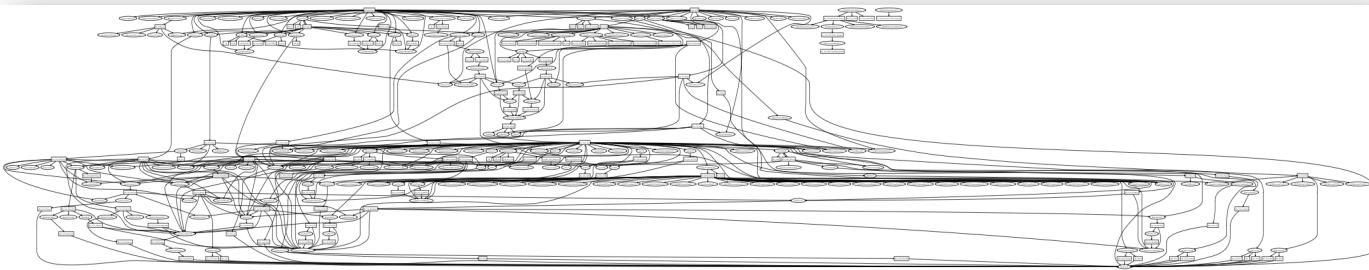
How to Introduce Concurrency

- The algorithms and their data dependencies form a **DAG** (directed acyclic graph)
 - Schedule the algorithms according to the DAG

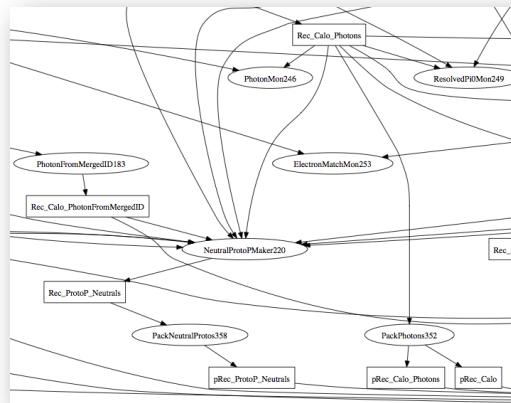


- Sounds more trivial than it is
 - Existing HEP software has many “backdoor” communication channels making the DAG non-obvious.

Real World Example



- Particular example taken from LHCb reconstruction program “Brunel”
- Gives an idea for the potential concurrency
- ATLAS/CMS/Alice/full LHCb just don’t fit on a slide...



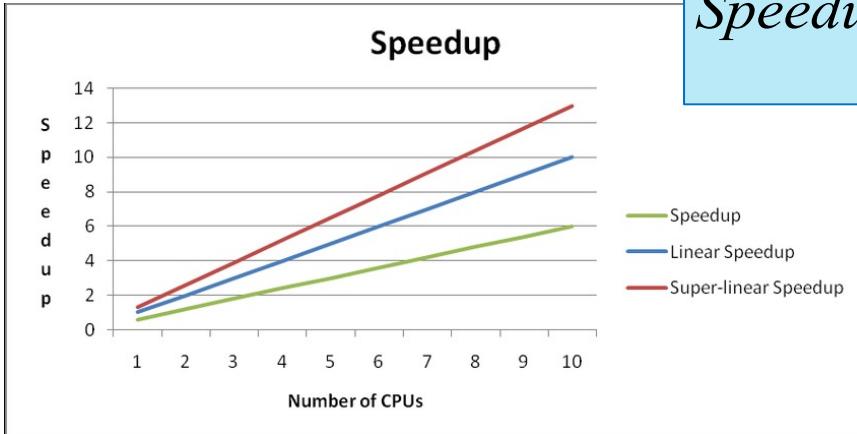
Is Parallelisation Worth It?

- We hit the wall very early – game over and that's it?
- Whenever thinking about parallelisation, one should spend some thoughts on whether the effort is worth it
 - The total cost of ownership of one additional box might be smaller than the design-implementation-maintenance costs
- What is the performance gain we can expect?

Amdahl's and Gustafson's laws can help you there!

Need for Speed(up)

- We parallelise because we want to run our application faster
- **Speedup**: how much faster does my code run after parallelising it?
 - Indicator of **scalability**



$$\text{Speedup} = \frac{\text{Time}_{\text{serial}}}{\text{Time}_{\text{parallel}}}$$

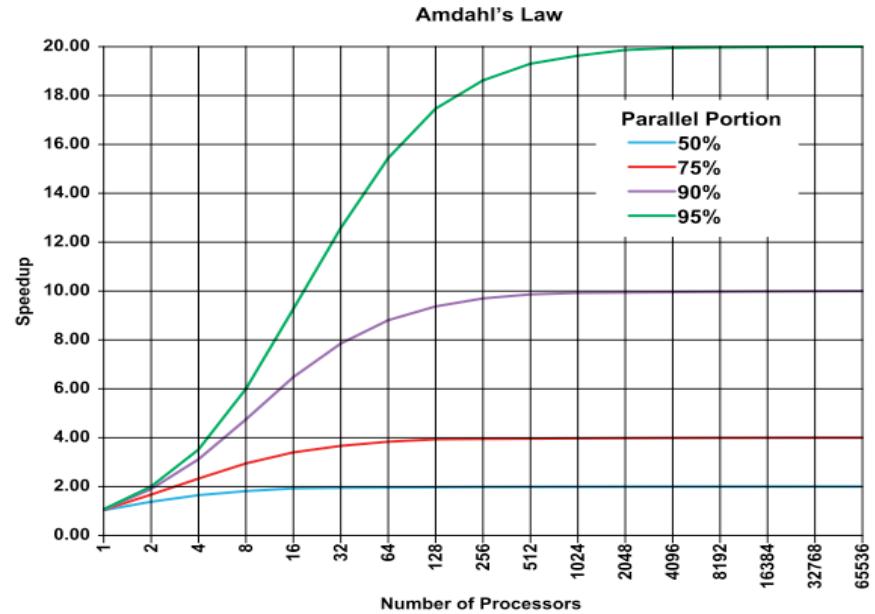
Amdahl's Law

- It predicts the **maximum speedup achievable** given a problem of **fixed size**

$$\text{Speedup} = \frac{1}{(1-p) + \frac{p}{n}}$$

n: number of cores

p: parallel portion



“the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used” – 2011

Reddy, Martin(2011) API Design for C++

Gustafson's Law

- Often **problem size increases**, while serial parts remain constant
- If problem size increases, so does the opportunity for parallelisation
- Solve bigger problems in the same amount of time by using more resources

$$\text{Speedup} = 1 - p + np$$

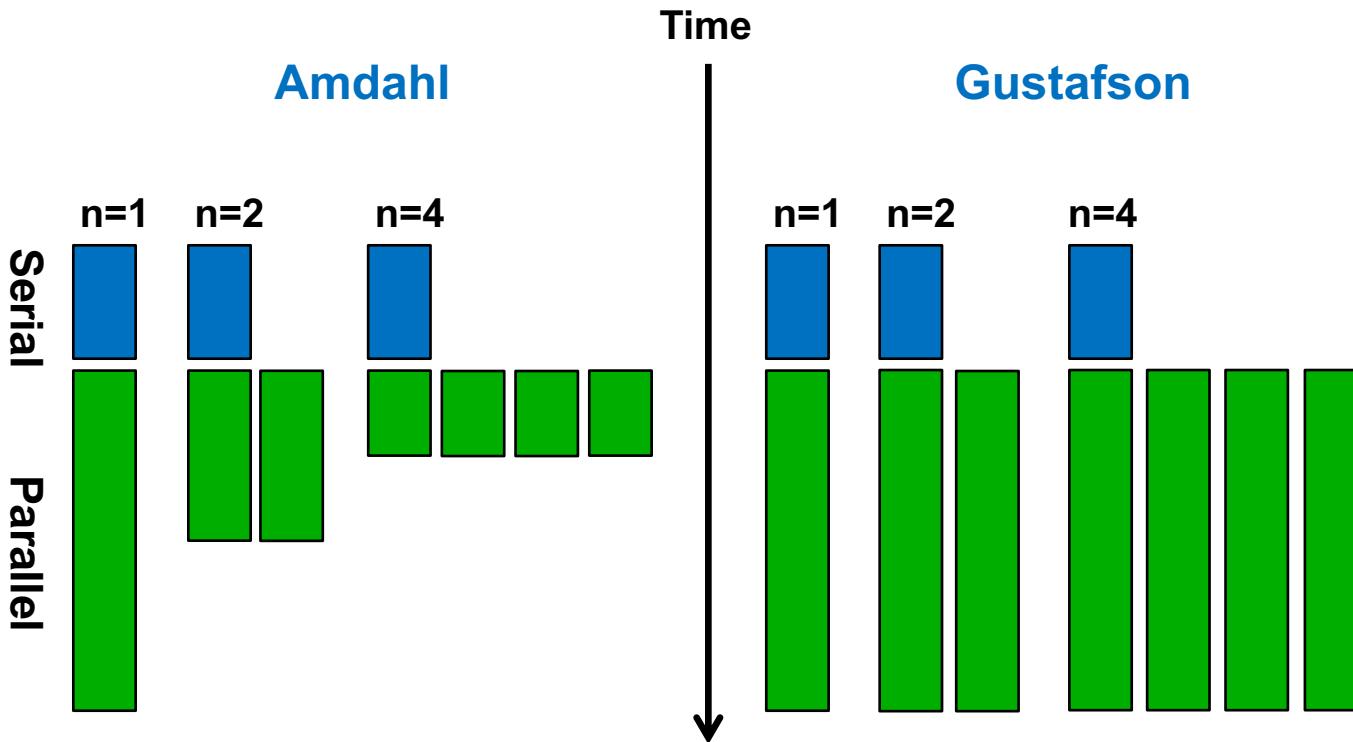
n: number of cores

p: parallel portion

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

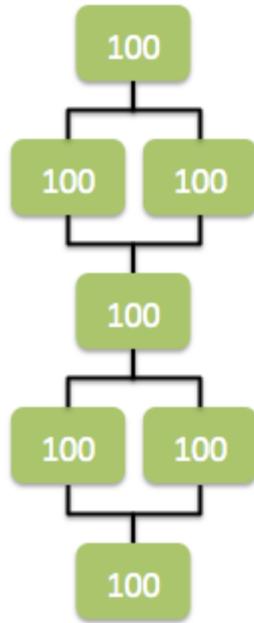


Amdahl vs Gustafson

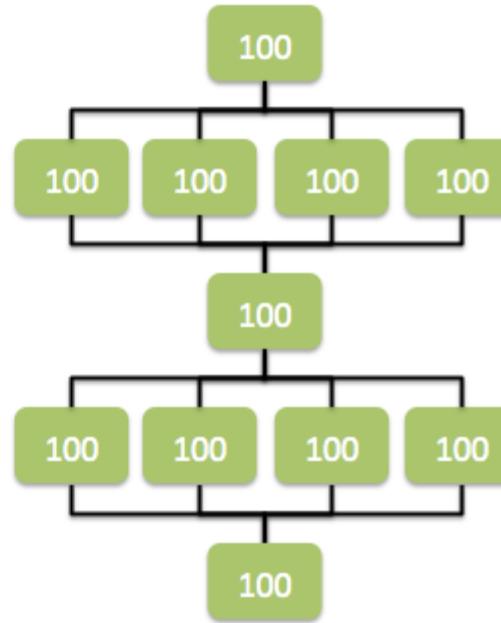


Increase the Problem Size!

↓
Time



Work 700, Time 500
Speedup 1.4x



Work 1100, Time 500
Speedup 2.2x

Strong and Weak Scaling

Case A

- A human is waiting in front of the terminal: **strong scaling**
- A problem of a fixed size is processed by an increasing number of processors
- Best modelled with **Amdahl's law**

Case B

- Want to get the most done in a certain amount of time: **weak scaling**
- Every processor has a specified amount of work to do, and when adding processors, we also add work
- Best modelled with **Gustafson's law**

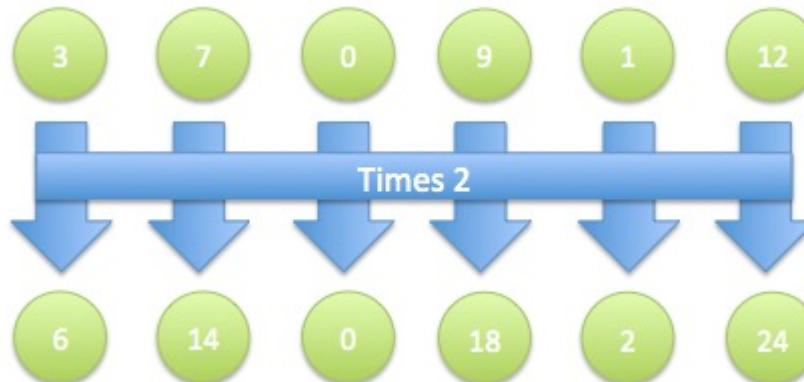


Two sides of the same coin!

Data Parallelism

Definition: parallelism achieved through the application of the same transformation to multiple pieces of data

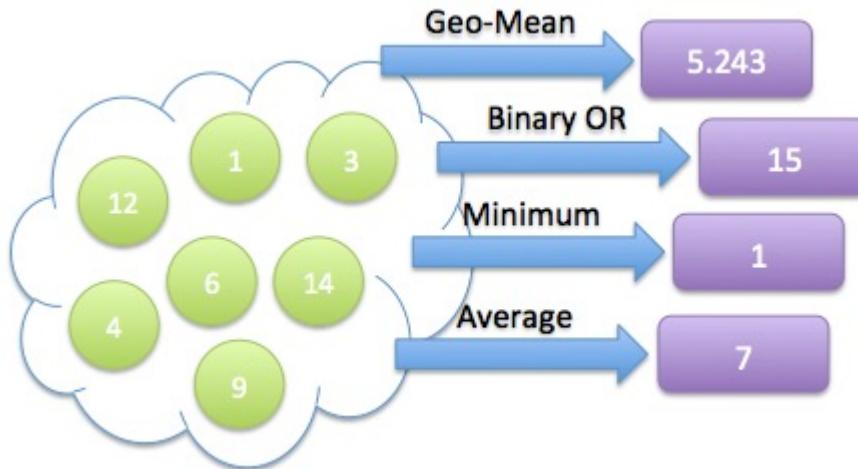
Example of pure data parallelism: multiplication of an array of values (ordinary administration for vector units and GPUs!)



Task Parallelism

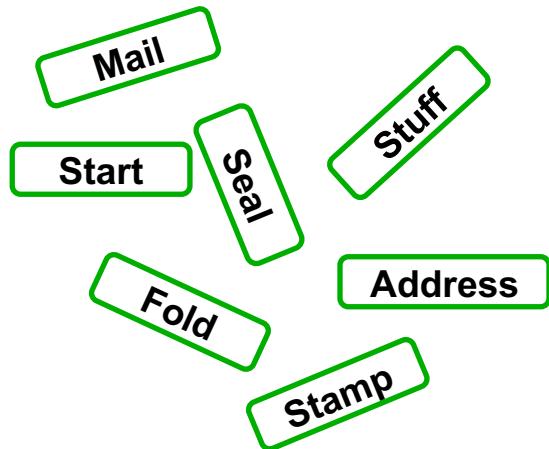
Definition: parallelism achieved through the partition of load in small work baskets consumed by a pool of resources.

Example of pure task parallelism: calculate mean, binary OR, minimum and average of a set of numbers



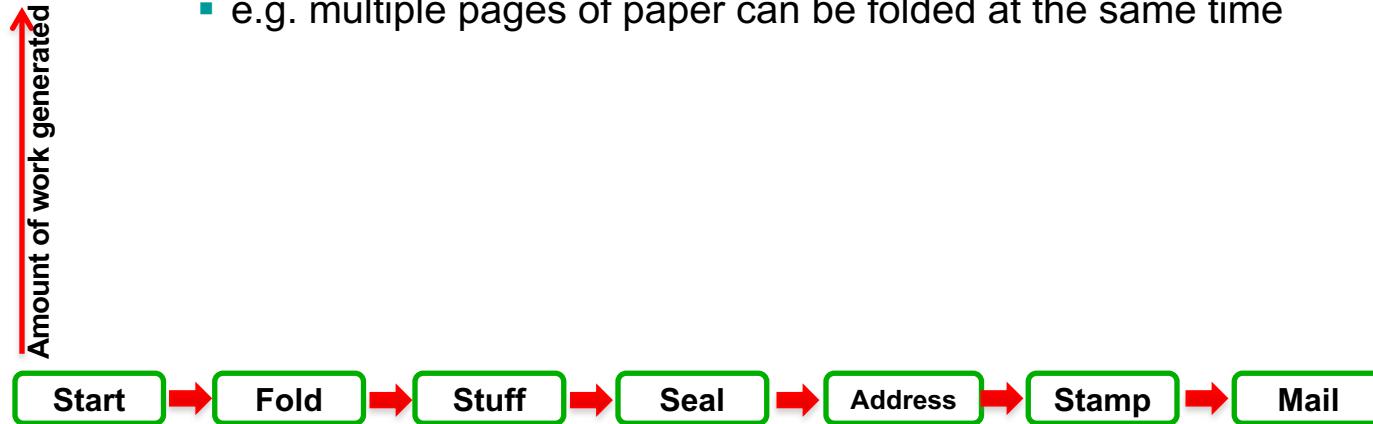
Mixed Solutions

Mandate: Build an efficient letter sending system mixing data and task parallelism



Mixed Solutions

- Fixed order of steps
- Data parallelism is already evident
 - e.g. multiple pages of paper can be folded at the same time



Mixed Solutions

- These operations require different amount of work though



Finding Concurrency

What can be executed concurrently?

Some techniques to figure this out:

- ***Data decomposition***
 - Partitioning of the data domain
- ***Recursive decomposition***
 - Divide and conquer
- ***Task decomposition***
 - Split according to **logical tasks**



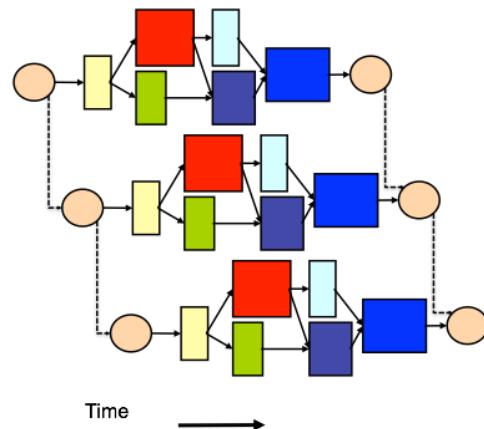
DIVIDE
ET
IMPERA

Mixed Data and Task Parallelism

- Pure task/data parallelism is difficult to achieve in reality
 - Sometimes close enough to real use cases!
- **Mixing data and task parallelism** is the key
 - Many different algorithms applied to a **stream of data**
 - Items processed in **stages** where data parallelism is expressed
 - Many items can pass through the **pipeline** simultaneously
 - Think of items as “collision events” and algorithms as “HEP data processing”!

Rethinking the Parallel Framework

- **Need to change the problem size**
 - Process **multiple events concurrently**
 - Helps on tails of sequential processing
- **Contradicts a lot of the basic assumptions in existing code**
 - Code prepared to process only one event at a time in memory
 - But existing code can't be thrown away easily
 - Need to localise distributed states
- **Major effort ongoing in all LHC experiments**
 - Exciting times for curious programmers!



A Glimpse on Complications

- 1. The DAG is not known to its entirety**
 - Hidden dependencies
- 2. Shared states are rarely safe**
 - “Caches” that do not behave like... well... caches
- 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?

We need a **smart scheduling** environment

- 1. The DAG must be “fixed” by changing the existing code**
- 2. Shared states are replaced by task-local data, avoid locks!**
 - More in the next lectures
- 3. If an algorithm requires a non-thread safe resource, it has to ‘reserve’ it beforehand**
 - No two algorithms using the resource are scheduled at the same time

Scheduling Directions

Three ways of coding up a scheduler for the DAG:

- ***On demand***

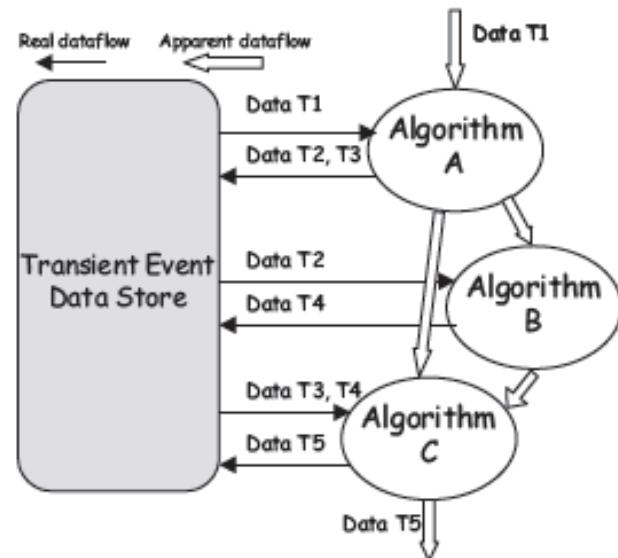
Start with the last algorithms in the DAG and invoke whatever algorithm is needed on-the-fly.
(backward scheduling)

- ***Data driven***

Start with the first algorithms in the DAG and start new algorithms whenever the necessary inputs are there.
(forward scheduling)

- ***Global view***

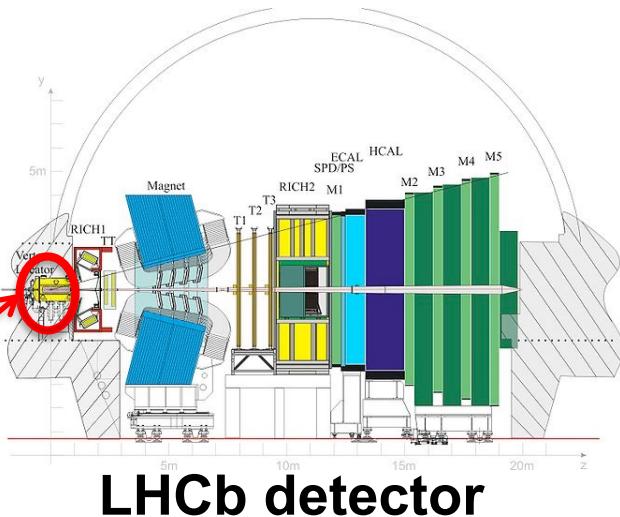
Analyse the entire DAG and schedule algorithms according to the dependency order (*task graph scheduling*)



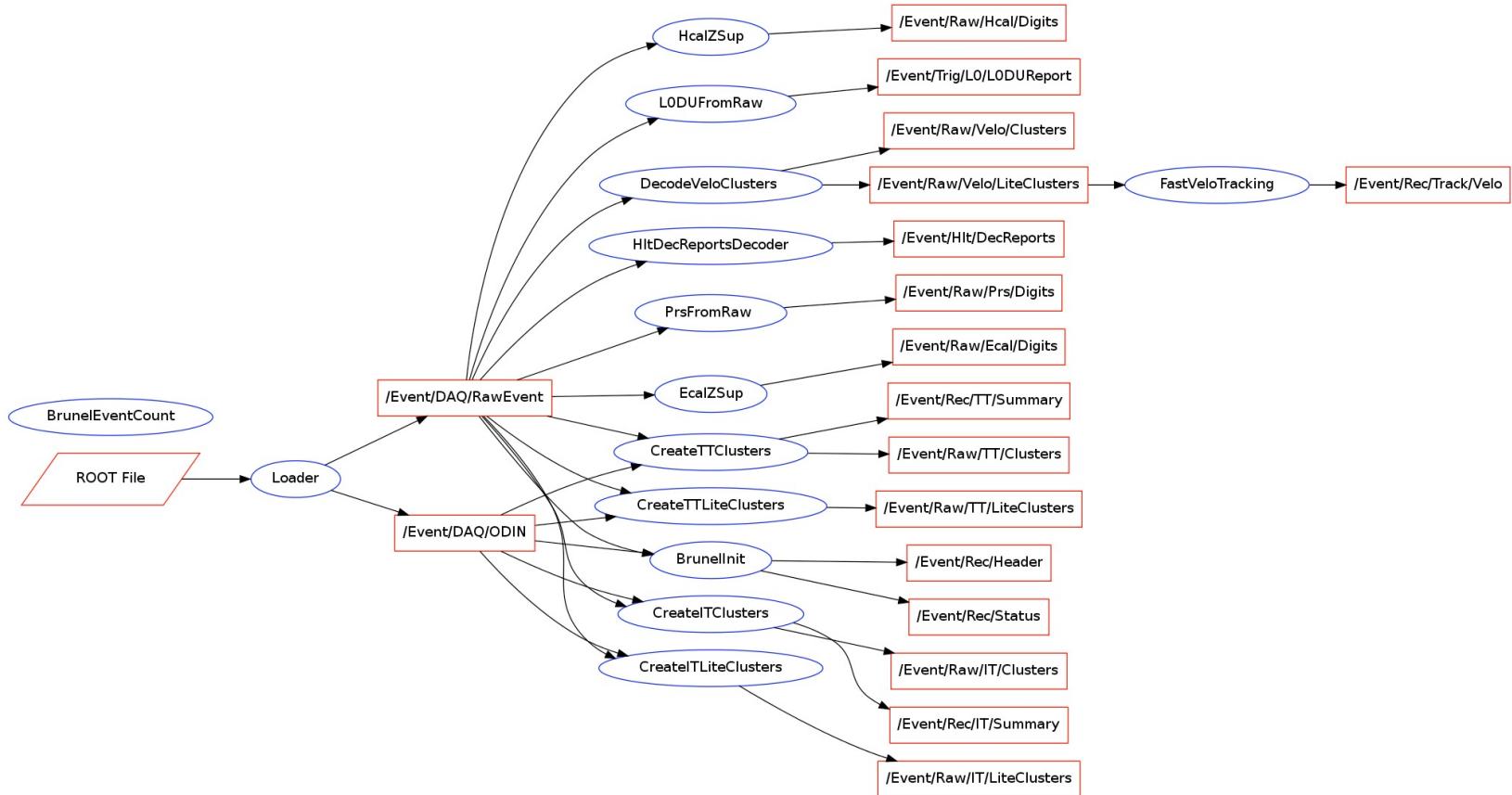
A Simplified Example

- Such a parallel framework is not only theory
- They are realised in experiment software
 - CMS offline software (CMSSW)
 - ATLAS/LHCb framework (Gaudi)
 - Alice (O2)
- Let's have a look at an example workflow
 - A slice of the LHCb reconstruction
 - Only the low-level objects of the vertex locator (VELO)

This part of
the detector



The Velo Low-Level Reco DAG



Take-Away Messages

- **Dealing with parallelism is inevitable**
 - Software must exploit parallel hardware
 - But there are different levels of exposure to parallelism
- **High energy physics has a history of parallelisation**
 - However, at a rather naïve level
 - The next steps require a harder approach
- **Parallelisation can be exploited in multiple ways**
 - Data parallelism and task parallelism
- **Amdahl's and Gustafson's laws** give a handle for scaling behaviour
- There is a clear strategy for parallelising HEP software
 - Use of a **task-based** approach