PRINCETON UNIVERSITY

Practical Computing Considerations

David Lange

November 26, 2018

Introduction and Caveats

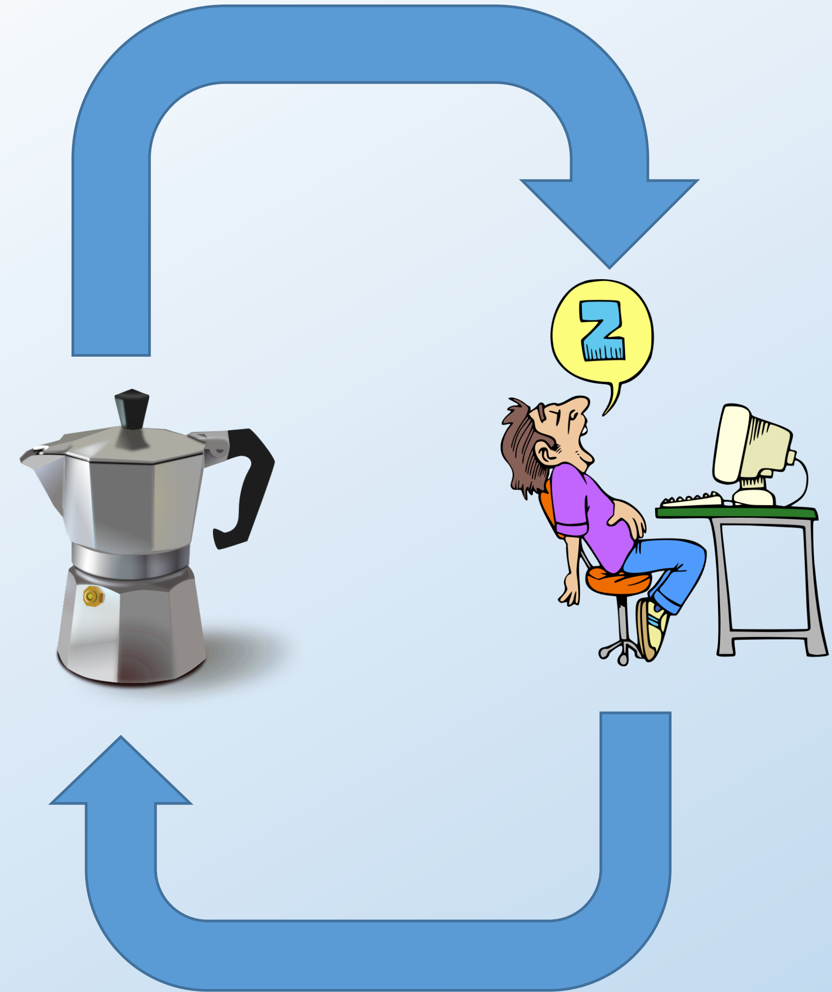
- I was asked to describe how generators fit into the experimental production system and some issues that are commonly encountered
  - A relatively mundane topic – Apologies..
  - I'm from CMS, so perspectives and specifics are sometimes CMS specific.
- I'm not a generator expert [ok, in another era I was one (for BABAR..)]. My perspective for this talk is mostly from the side of experiment software and production computing environments

Monte Carlo simulation – ideal world

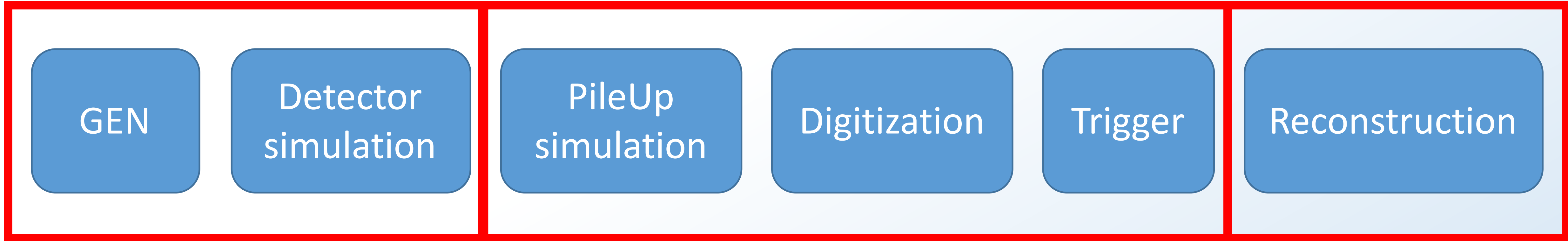
Given the scale of LHC simulation needs, reality is more complicated

“Typical” way we create Monte Carlo simulation on the grid

- Requests come in bursts – naturally driven by data and software availability as well as conference deadlines
- MC production (GEN, detector simulation, pileup simulation, digitization and reconstruction) is the major consumer of CPU on the Grid. ”Queues” are typically deep, so individual requests take weeks
- Gridpacks are a major enterprise for largely non-experts.
  - Sizeable set of experts and a pre-production validation check are needed to filter out most mistakes in this process

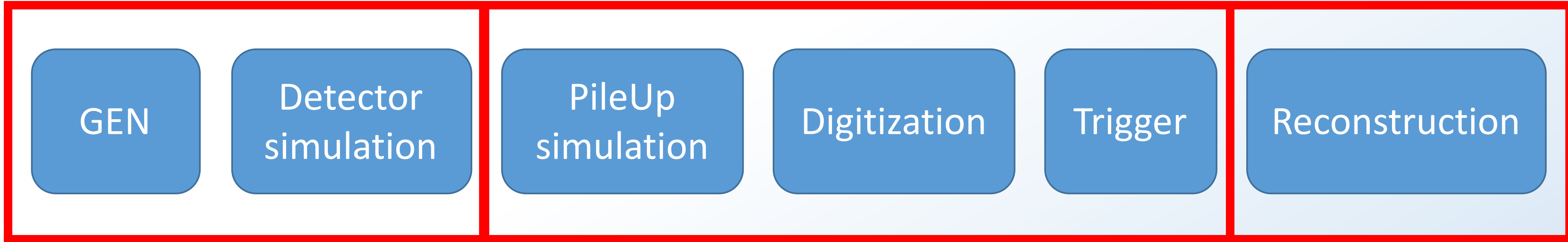


One approach use to create Monte Carlo simulation on the grid



Red boxes represent separate processing jobs that each write output and may be processed by different compute resources at different times

One approach use to create Monte Carlo simulation on the grid



Red boxes represent separate processing jobs that each write output and may be processed by different compute resources at different times

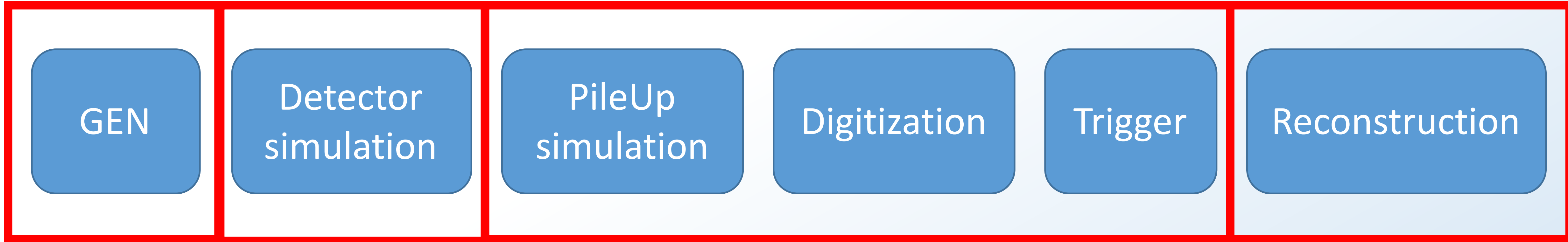
Positive attributes

- Generator small piece of total GEN+SIM processing time (historically)
- Less data handling and fewer processing steps

Negative attributes

- Difficult to accommodate generator developments need more agility than simulation (which prefers stability)
- Long startup times in generators (eg, gridpack handling, once-per-job calculations) have bigger impact on throughput

A second approach use to create Monte Carlo simulation on the grid



Red boxes represent separate processing jobs that each write output and may be processed by different compute resources at different times

Positive attributes

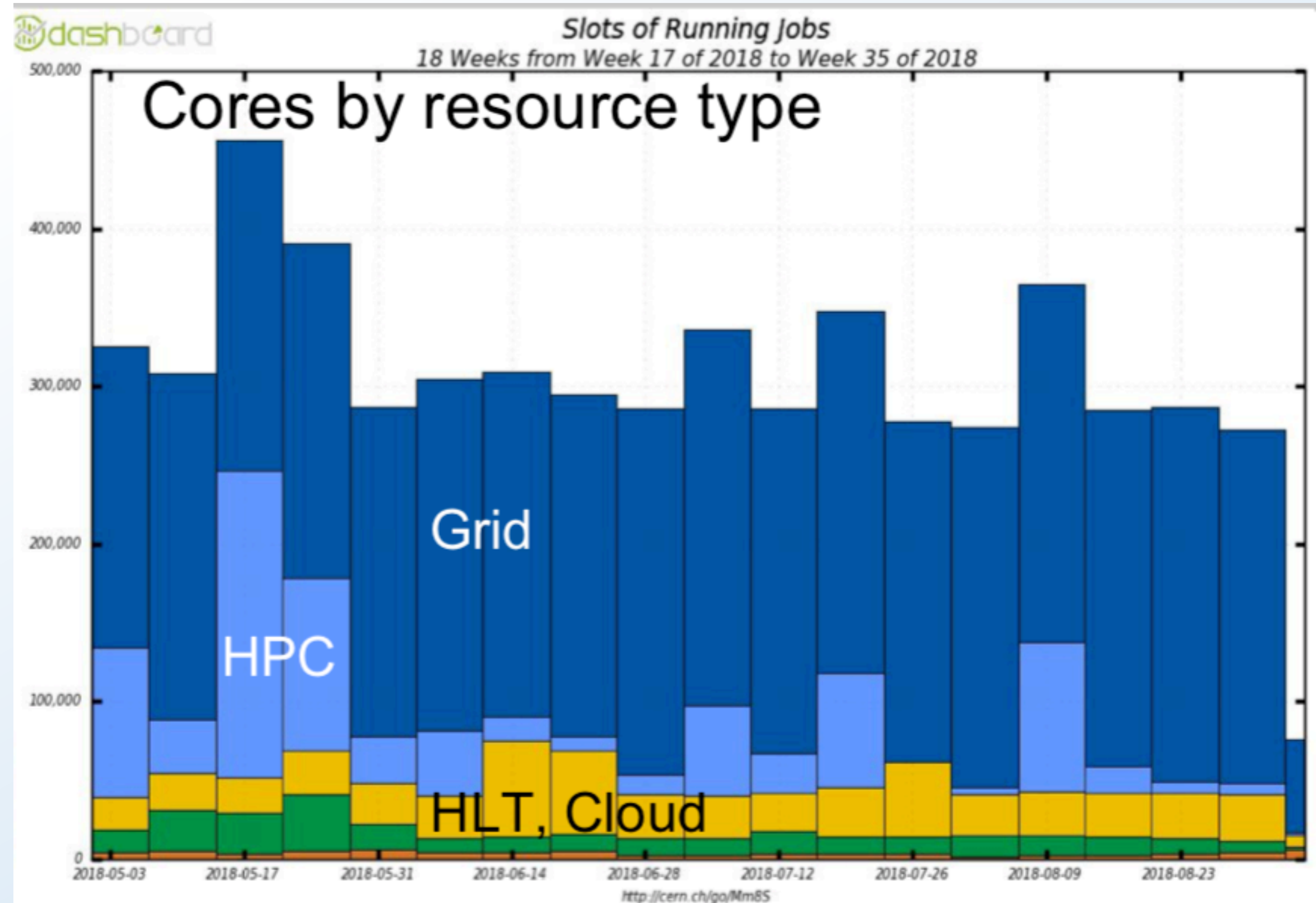
- Easy to include newest generator developments (assuming stable file format for output)
- Job requirements (cores, resources, etc) can be optimized for generator being used

Negative attributes

- Generators must run many events per job to match grid requirements
- More data handling
- Potentially more latency in processing MC events

The scale of grid (and its friends) usage is far beyond the LHC vision at the start of data taking

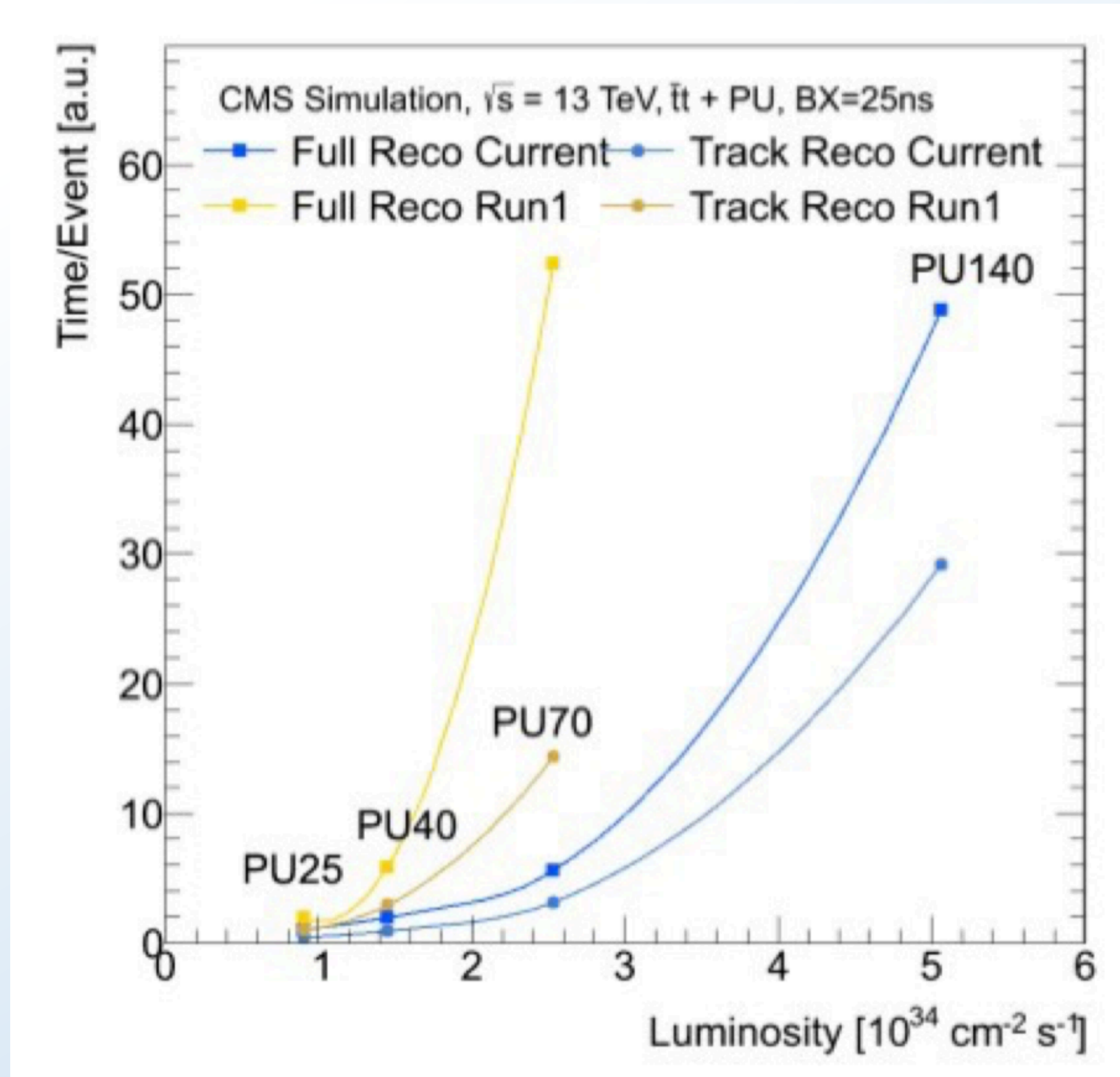
- 10s of billions of Monte Carlo events per year
    - Saved to disk, so generators produce even more
- Workflow management systems orchestrate 500k+ cores distributed across ~100 sites.
    - Most sites have a very similar compute environment, but there are specialized environments
- Jobs slots have 1-8 cores and 1-2 days maximum wall clock time
- Experiments provide evolving, but controlled and thus reproducible, software environment
    - Experiment frameworks work closely with workflow management systems to most efficiency use allocated resources



Aggressive code optimization is an important factor enabling this scale of operations

- Illustrative example: CMS reconstruction has achieved consistent speed-ups while maintaining or extending physics performance
- Achieved via
  - Pure technical improvements (eg, code refactoring)
  - Algorithmic improvements

Typically achieved by small groups of people



What does this mean to a software developer? Ideally…

- Code should be robust enough to run billions of events without errors (Physics-wise or technical)

What does this mean to a software developer? Ideally…

- Code should be robust enough to run billions of events without errors (Physics-wise or technical)

- Code startup time should be minutes not hours

What does this mean to a software developer? Ideally…

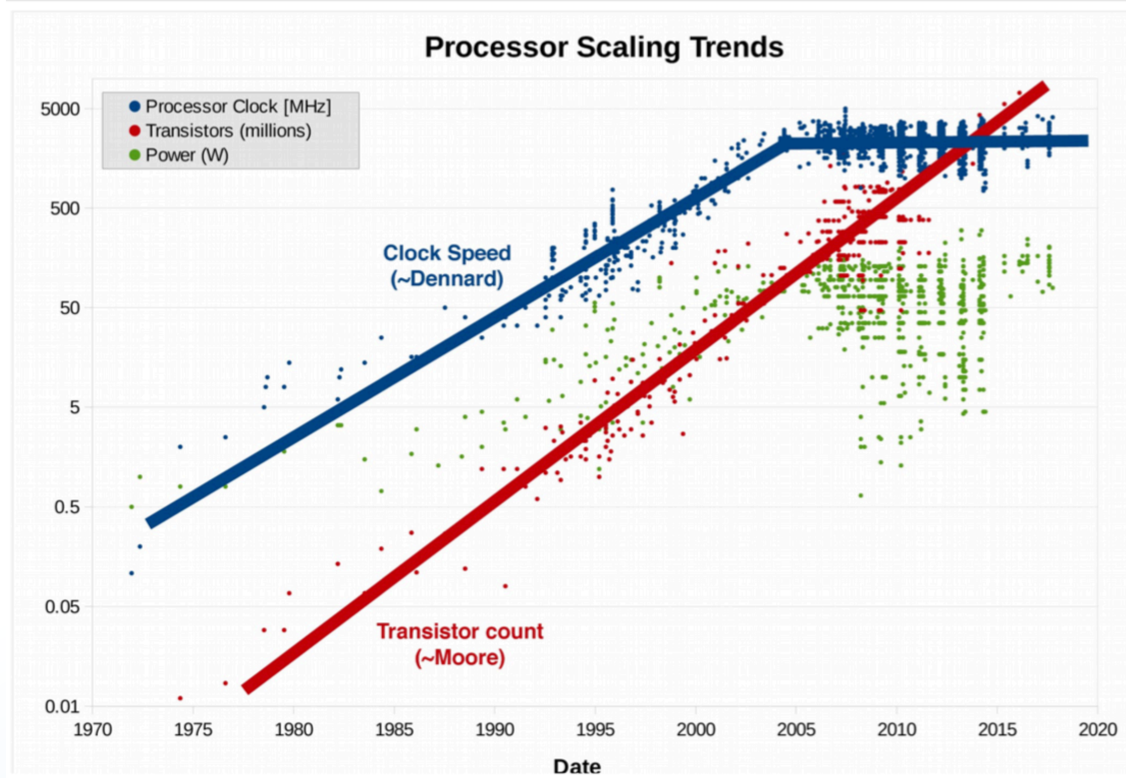
- Code should be robust enough to run billions of events without errors (Physics-wise or technical)
- Code startup time should be minutes not hours
- Code should be robust enough to run for hours without problems

What does this mean to a software developer? Ideally…

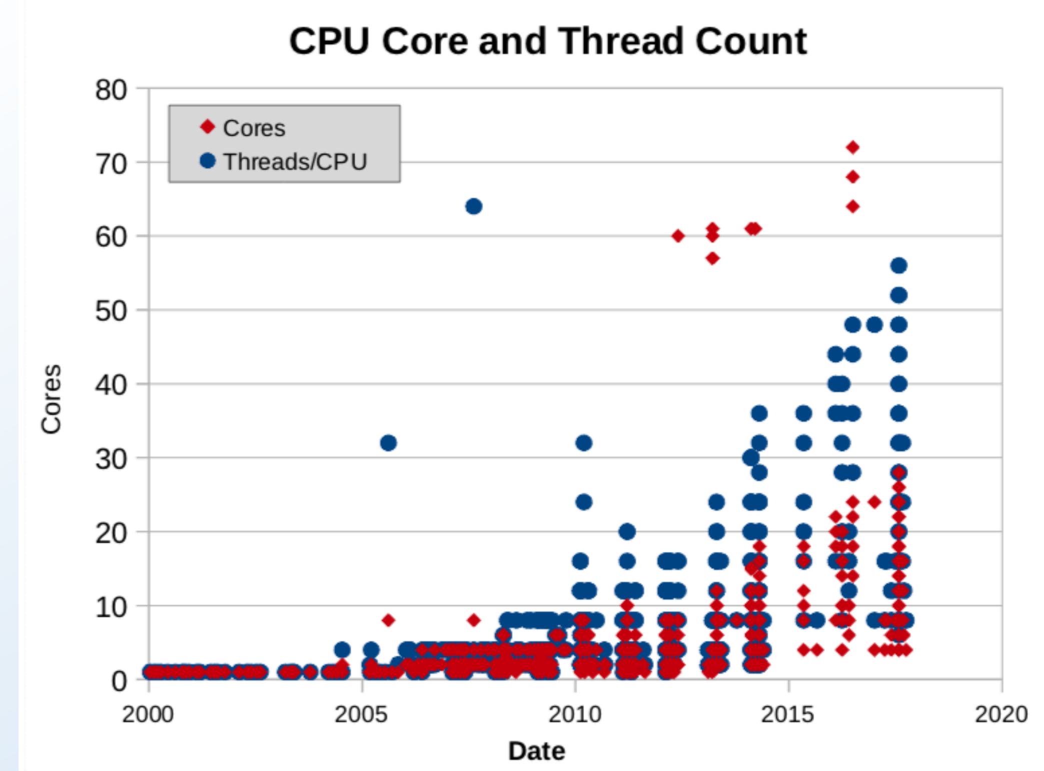
- Code should be robust enough to run billions of events without errors (Physics-wise or technical)
- Code startup time should be minutes not hours
- Code should be robust enough to run for hours without problems
- Code should be runnable as a library not just in its own framework

Technology challenges mean that threaded frameworks are either already in use or soon will be

End of processor speedup



Towards many-core computing



Trend towards more cores and slower memory access. This is directly at odds with ”traditional” HEP applications: Memory-heavy and single threaded

Common wisdom: Generator piece of the workflow is small compared to the rest

- Often incorrect (and increasingly so). Efficient operations of generator workflows is more and more important to experiments
- Examples:
  - Copying/untarring of gridpacks can be a long process (and makes grid node disks unhappy)
    - I understand that a solution to this is in the works or maybe already complete…
  - Complexity of calculations as ”N”s are added
    - Unavoidable? [potentially mitigated via code optimizations?]
  - Filters with a particularly low efficiency for selecting events:
    - Events are are thrown away as “uninteresting” according to generator level results. This avoids the usually much more expensive simulation + reconstruction processing
    - Is there an opportunity to work together to improve ?
(eg, advise on better configurations, implementing biasing possibilities, etc)

Coding thoughts that have enabled experiment applications to run at scale with good throughput per cost

- Avoid where ever possible
  - Repeated calculations
  - Frequent (small) memory allocations
  - Global non-const variables and statics (threaded applications)
  - Unneeded inheritance (virtual function calls are expensive)
  - Divisions (when a multiplication will do)

Coding thoughts that have enabled experiment applications to run at scale with good throughput per cost

- Avoid where ever possible
  - Repeated calculations
  - Frequent (small) memory allocations
  - Global non-const variables and statics (threaded applications)
  - Unneeded inheritance (virtual function calls are expensive)
  - Divisions (when a multiplication will do)
- Be careful with
  - Small function calls that are not inlined by the compiler

Coding thoughts that have enabled experiment applications to run at scale with good throughput per cost

- Avoid where ever possible
  - Repeated calculations
  - Frequent (small) memory allocations
  - Global non-const variables and statics (threaded applications)
  - Unneeded inheritance (virtual function calls are expensive)
  - Divisions (when a multiplication will do)
- Be careful with
  - Small function calls that are not inlined by the compiler
- Strive to
  - Keep algorithms thread safe

Recent example exercise..

- By chance discovered that a NLO workflow used for CMS nightly regression checks took ~2 hours to run (10 events)
  - 99%+ was initialization of NLO generator. Doing this in production (likely we do already…) has big impact on throughput for those workflows.
  - We looked into possible ways to improve

- Bottom line: 3.5x speedup achieved via two different approaches
  - Hacky method (~100 lines of code change): Optimizing handful of functions that were identified to use most of the CPU time
  - Expert method(~70 lines): Ensure that initialization happens ~1 time

We expected to find that mathematical functions dominated

```
% total       Self   Function
  17.82   1'192.52   binomial_dd_ [67]
  17.80   1'191.60   __combinatorics_MOD_calcncoefs [63]
   6.95     465.37   __cache_MOD_setcachemode_cll [79]
   4.29     287.23   __combinatorics_MOD_calcfactorial [78]
   3.09     206.67   init_dd_global_ [65]
   2.64     176.84   __combinatorics_MOD_calcfactorial'2 [86]
   2.27     151.82   __ieee754_log [83]
   1.88     125.90   _init [115]
   1.73     115.86   __ol_ew_renormalisation_dp_MOD_ew_renormalisation [49]
   1.57     104.96   __mul [125]
   1.35      90.25   free [133]
   1.29      86.44   malloc [96]
   1.05      70.35   __combinatorics_MOD_calcncoefsg [153]
   0.94      63.19   __combinatorics_MOD_calcfactorial'3 [157]
   0.94      62.74   __ieee754_hypot [141]
```

- Again, just an illustrative example [CMS code had (still has) plenty of much worse behaviors – the only way to find them is to investigate and improve…]

Example fix (4x faster with the CMS compilers)

```
443  -  recursive function CalcFactorial(n) result(fact)
444
445        integer, intent(in) :: n
446  -     integer :: fact
447
448        if (n < 0) then
449          write (*,*) 'factorial not defined for negative integer'
450          stop
451        end if
452
453  -     if (n .eq. 0) then
454  -       fact = 1
455  -     else
456  -       fact = n * CalcFactorial(n-1)

457        end if
458
```

```
443  +  function CalcFactorial(n) result(fact)
444
445        integer, intent(in) :: n
446  +     integer :: fact,i
447
448        if (n < 0) then
449          write (*,*) 'factorial not defined for negative integer'
450          stop
451        end if
452
453  +     fact = 1
454  +     if (n .gt. 1) then
455  +       do i=2,n
456  +         fact = fact * i
457  +       end do
458        end if
459
```

After (the expert fix)

```
% total        Self   Function
   6.80      132.20   __ieee754_log [58]
   5.13       99.73   __mul [97]
   3.03       58.95   __ol_ew_renormalisation_dp_MOD_ew_renormalisation [55]
   2.95       57.36   malloc [117]
   2.83       55.00   __ieee754_hypot [112]
   2.50       48.50   free [134]
   2.33       45.27   __ieee754_exp [136]
```

- Looks much more like what we expected

About software licenses

- The field has converged on open-source software
  - We are “just” left with discussions about which license is most appropriate and the impact that has on “users”
- Experiments build large stacks of software from many components and getting harmony in licensing is difficult
  - Respecting GPL licensed code means that large parts of the experiment software code becomes GPL by inheritance. Following this policy interpretation also introduces conflicting license requirements (given ~100 dependencies)
    - Defining “Derivitive works” seems to be an open point with big implications.
      We aren’t lawyers (and don’t want to be…)
  - Ways to resolve this dilemma involve a lot of unnecessary gymnastics of different builds and complicated applications of licenses

About software licenses

- Software communities in general are being more aware of this issue. We aren’t alone
  - Experiments are trying to solidify their situations
  - CERN’s viewpoint has evolved in recent years with the desire to work more closely with industry. They have chosen Apache2
- We have to improve our acknowledgements and citations for software
  - MCNet guidelines make this point well and the HSF very much on board with this
- We don’t want to dwell on this point in this workshop, but we would like to raise awareness of it and return to it in a later meeting

Conclusions

- Experiments deal with large software stacks. Much of this code is experiment specific, but the use of external packages are essential
  - Generators, Geant4, ROOT, math libraries, etc
- Consistent (but not expansive) effort looks for opportunities for technical improvements has enabled the current scale of operations.
  - Looking towards HL-LHC, we must not only continue this trend, but also to evolve with changing computing resources
- I didn't talk about accelerators. Are they an area for common R&D?
  - We all need to learn if/how/when accelerators can provide more events per ChF for our workflows (a calculation complicated by HPCs)
  - Lots of R&D across the community, with success stories to build on

Backups

Software distribution

- Releases consist of experimental software + numerous ”external” dependencies.
  - Run 2 LHC software releases are large, and include many components
- Distribution: Releases need to be easily made available to sites. Currently >90% use CVMFS for this
- Consistency: The evolution of major release versions should not change results once in production. They instead should evolve to include additional features
- Reproducibility: Production depends on released software versions (sources, libraries, conditions, etc) not changing
- Self contained: Production systems rely on the software release to declare all of its dependencies