Science and
Technology
Facilities Council

Computing

Stewart Martin-Haugh, James Walder

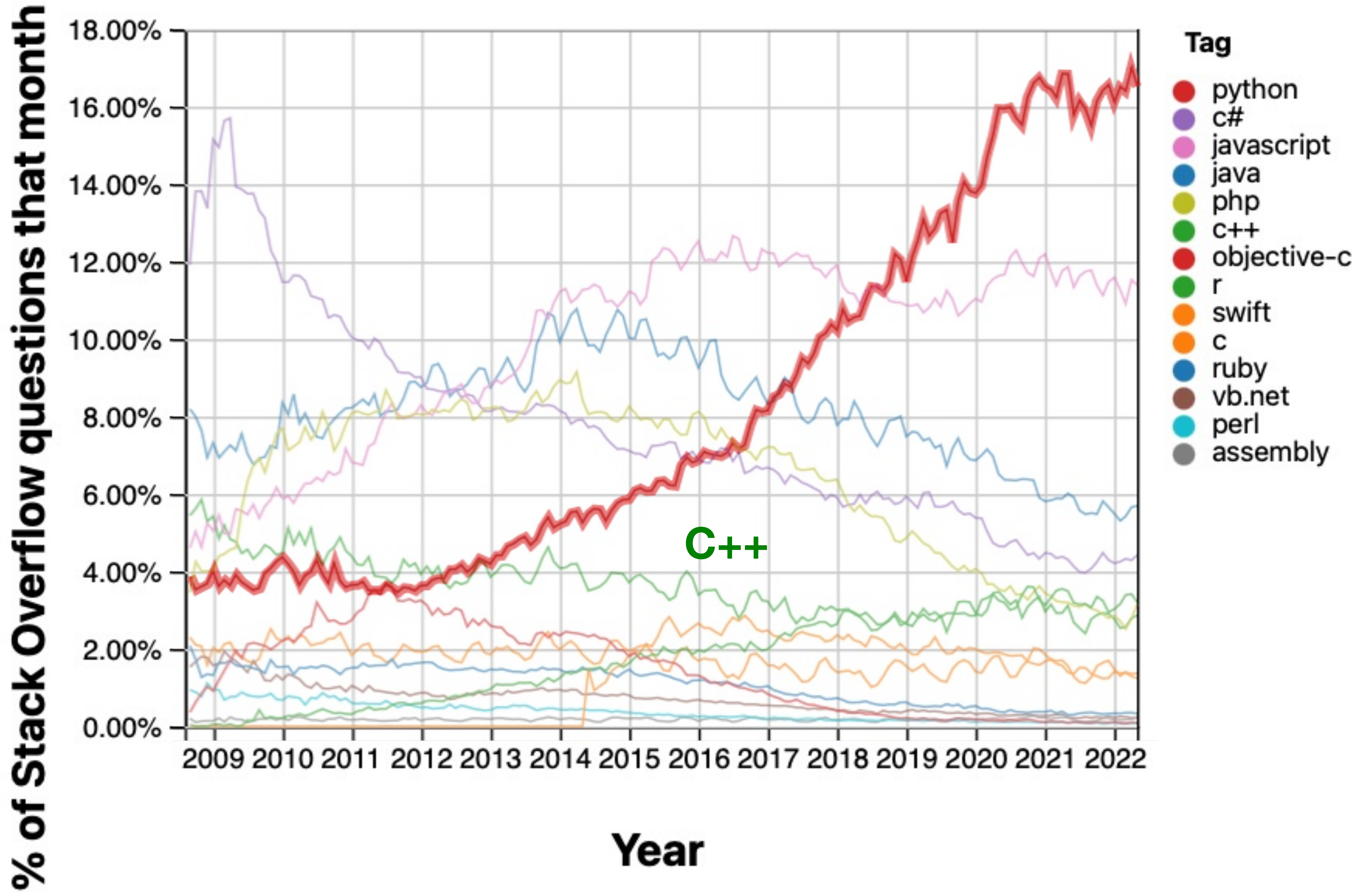
- Computers:
    - Historically, the word dates back as far as the 17th Century,
        - Referred to humans who carried out carried out calculations or computations.
    - Modern (machine) computers provide advantage of humans of:
        - Computers offer two principal advantages over humans:
            - Correctness, and repeatability / reproducibility
            - Speed:
                - For ‘simple’ and repetitive operations
                  (although modern techniques of machine learning are making complex tasks accessible for computers; e.g. chess, visual recognition tasks).
            - Although - these are not guaranteed …

Programming Languages

- C++ and Python most prevalent within Particle Physics community

- C++ is rather stable over recent years

- TIOBE Index
    - Python replaced C as #1 ranking in 2022
    - #3 Java
    - #4 C++



Source: insights.stackoverflow

Python

- Python:
  - Excellent as a scripting language:
    - Easy to read and write (also easy to introduce typos, etc; Type Hints can help spot these).
    - Slower than C++ (interpreted, versus compiled):
      - Can call / leverage other languages (e.g C++ / fortran) when speed is required (numpy, numba, ROOT)

- In HEP:
  - Used as the 'glue' within framework software to combine C++ code (e.g Athena, CMSSW)

  - Recent evolution of using python at the 'analysis stage' of a physics analysis (e.g pyhf)
    - PyHEP Workshops
    - https://scikit-hep.org :
      - Community-driven effort for creating an ecosystem for data analysis in Python

C++

- Large and complicated language; many ways to accomplish same things:
  - C (`arrays, pointers, functions`)
  - Traditional C++ (`new, delete, classes`)
  - Templates (`template class<T>`)
  - Modern C++ (`std::unique_ptr,  for (auto x: y) {}`)

- Will find all of these approaches in large project codebases: (e.g. HEP experimental software frameworks);
  - Can co-exist (for better, or worse …)
- Evolving language; new standards every 3 years;
  - C++11 standard introduced modern C++, (C++14 considered incremental, with C++17 and C++20 bringing additional functionality).

- C and C++ are probably the best-supported ways to write fast code

Other languages

- Java, Javascript, C# and various languages with more domain-specific utility
- A few recent ‘newcomers’ attracting some attention:
  - Go: ~ modern C (with some less complication)
  - Rust: aims to be like a correct-by-default C++
    - Better to avoid memory leak, buffer overwrites
  - Julia: aims to be like a fast Python

Software Engineering

- Everything around the actual writing of code
  (and distinct from the theoretical aspects of computer science)

- Use of sound engineering best practices, and encompasses at least:
  - Requirements gathering and specification definition
  - Debugging
  - Testing
  - Packaging
  - Documenting
  - Operating environments
  - Considerations around ongoing maintenance and evolution

Term "Software Engineering" promoted by Margaret Hamilton, lead programmer for the Apollo Mission guidance computer

Testing for Correctness

- Demonstrating an algorithm is correct with respect to its specification:
  - Usually interested in functional correctness;
  - that for a given input for a computation the correct output is returned.

- General approaches could be:
  - Find version of the calculation (not an edge case) where the answer is known;
  - Make changes that should not affect the answer and verify the result.
  - Also try to test on edge-cases.
  - Important to test that code handles problematic cases correctly:
    - e.g. passing in an incorrect type (e.g. python)
    - Out-of-range values
    - Invalid values (nan)
    - Might be sufficient just to crash the code;
      - Or handle any and all possible exceptions

Debugging: Thoughts

- Brian Kernighan (C, Unix etc):
  - *The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.*
  - *Everyone knows that debugging is twice as hard as writing a program in the first place.*
    - *So if you're as clever as you can be when you write it, how will you ever debug it?*

- Rubber ducks and teddy bears:
  - Explain your problem to anyone – doesn't need to be an expert
    - Doesn't even need to be a living entity; *rubber ducks and teddy bears*
  - Draft and email for an expert, explaining what you have done / what you think is happening.
  - – Documenting / explaining your thought process and problems found leads:
    - to better understanding of the issue, and
    - possibly its solution.

Debugging: Process

- Reproduce the problem;
- document exact setup instructions (environment, command, extra software / settings)
    - If necessary distill the problem to it's simplest components:
        - e.g could write a separate small program that generates the same errors
- Find the fix …
- Establish that the fix has no undue side effect
  (e.g. that tests unrelated to the fix still gives expected results)

- Consider creating tests to make sure issue can't reoccur (see later)

Static analysis

- Static analysis (compared to Dynamic analysis) is the study / debugging of source code before executing the program.
- Many automated tools to help.
- Code review process with another human also important.

- Python:
  - Flake8
    - (Static analysis plus pep8 style guide)
  - Vulture (find unused/unreachable code)
  - Type hints: (introduced in python 3.5;
    - Improved static type checking

```
def hello_world (name: str) -> str:
    return f'Hello {name}'
```

- C++:
  - Compiler Errors and Warnings
    - Enable as many Warning flags as possible, and consider them as code problems until fixed.
  - cppcheck (fast)
  - Coverity (slow …)

- Include these tools into your testing cycle (more later)

Dynamic analysis

- Study of your code while the program is being executed
  - Ideally run tests for variety of scenarios / edge cases to test all coverage of code.

- Usually accomplished by adding extra information to the compiled binary (or running code) that augments the output
  - Including when things go wrong (or even to make it crash)

- Extra print statements / logging info
- Debug symbols: e.g `g++ -g` to add line numbers, etc to your crashes
- GDB and similar: use the debugger to trace error points / add breakpoints, read stack traces
- Valgrind: run in a virtual environment to identify memory errors
- Sanitizer tools (part of Clang project):
  - tell you if you're writing outside allowed memory, using uninitialised data

Memory errors (C++)

- Most common: reading/writing beyond an array

-
  ```
  vector<float> vec;
  vec.push_back(1);
  std::cout << vec[2] << std::endl;
  ```
- Results are undefined, as random memory is read:
  - Might print 0, could print 1.1755e^{-38} , could crash
  - Results irreproducible
  - AddressSanitizer or running under valgrind

- vec.at(i) will throw an exception when attempting to access past the end of the array
  - It is slightly slower (but unlikely to be significant in most cases)
  - Much easier to find the bug

Floating point errors

- 1.0/0.0, 0.0 / 0.0, sqrt(-1), inf, NaN, -NaN,
    - All floating point exceptions
- Mathematical operations involving NaN (C++) will produce NaN; can pollute results and should be fixed:
    - In numerical computations, may need to identify and catch at runtime (.e.g isinf, isnan)

- Floats are most precise close to 0 - avoid using very small and very large numbers

- Strongly recommend reading Floating point demystified for a more thorough understanding

- NB. In python, 1.0/0.0 type operations raise exceptions (ZeroDivisionError).
    - Also, in higher level applications (e.g. Pandas) NaN may be used to represent ‘missing data’ or similar

Testing

- Testing correctness of code can manifest at several levels:
  - Types of Tests:
    - **System / Integration tests**: test that the different components work together - usually at scale.
      - For example; run reconstruction code over a large dataset. Does it crash? Are the results the same as the previous (night's) test - and if not, is it understood.
      - Large scale validation runs.
      - Generally easiest to write, as mirrors the nominal operation of the code;
        - Might be difficult to ensure full coverage.
    - **Regression tests**: check that a fixed bug does not reappear;
      - Harder to write; need to keep track of test cases; only to prevent a reoccurrence of previous bugs.
    - **Unit tests**: check functionality at a ~ function level - should be quick to run
      - Hardest to write: think about what each function does, and to test all cases of input.
      - Might need extra code to set up, fake input data, or even a mocked-up backend
      - Test driven development; first write the test, then develop the code to pass the tests.
  - **Static analysis**:
    - Add to your tests (e.g. in continuous integration); preferable to catch any problems at the static analysis stage, if you can
- Writing any tests is good; the more you do, the better at writing them you'll become.

Unit test examples: python

- Using asserts:

```
def squ(x):
    return x**2 + 1

def test_squ():
    assert squ(42)==1764, “should be 1764”

test_squ()
```

```
Traceback (most recent call last):
  File "square.py", line 6, in <module>
    test_squ()
  File "square.py", line 5, in test_squ
    assert(squ(42)==1764), should be
AssertionError: should be 1764
```

- Useful for checking that a change that shouldn’t change the output, doesn’t change the output
- `assert` in python is in general very useful; apply where needed.

Unit test: frameworks

- In C++, CppUnit, GoogleTest is useful for functional tests.
- Python has: unittest, nose, pytest, …
- Unittesting libraries provide a framework to run tests, setup and tear down initial states and collate the outputs into various formats.

```
import unittest
from square import squ


class TestSquare(unittest.TestCase):
    def test_square(self):
        self.assertEqual( squ(42), 1764)


if __name__ == '__main__':
    unittest.main()
```

```
python test_square.py
F
======================================================================
FAIL: test_square (__main__.TestSquare)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_square.py", line 7, in test_square
    self.assertEqual( squ(42), 1764)
AssertionError: 1765 != 1764

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

- Other methods of automation tests - beyond function level testing also exist; e.g. https://robotframework.org for automation testing of application level processes (e.g. logging into a website, API call correctness, etc).
- Very easy to 'go overboard' when first exploring tests; think about the cases you really want to be checking;
  - Also consider testing failure mode correctness, as well as checking for correct behaviour.

Random and comprehensive testing

- Random input testing not truly well used within particle physics:
    - Although large MC and data samples with inherent randomness is used

- Fuzz testing (more important from security concerns) attempts to construct malformed or ‘almost-valid’ inputs in order to expose limitations in the code (e.g. poor parser logic).

- Coverage:
    - Coverage is a measure of how much of the code is actually being tested:
        - High coverage does not necessarily mean that it is exhaustively testing possible test cases of a function.

    - Good to aim for high coverage, but experience / code complexity may suggest where you need to place effort on your test cases

Continuous Integration

- Use tools, such as GitLab CI, Jenkins, Travis CI to automate your build / testing phase
- Catch problems before they're part of the main codebase
    - Demonstrate functionality (or rather, lack of changing other outputs) to project leader
- Run as many tests as you can for a merge / pull request

```
#example gitlab CI for LaTeX

stages:
- build

build:
    image: thomasweise/docker-texlive-full stage: build
    script:
      - apt update -y
      - apt install -y biber
      - make
         artifacts:

            paths:
               - "*.pdf"

            expire_in: 1 week
```

Containers

- Useful for many reasons:
  - debugging, versioning, sharing code, reproducibility
  - Run a lightweight virtual operating system on top of your real OS
  - Similar to a virtual machine (some OSs, e.g. Mac run docker within a small VM)
  - Easily run Linux programs on Mac, Windows
  - A description of an environment that someone else can run
    - e.g. in docker; a Ddockerfile declares the base image and all changes needed to build the container image
    - Can run on the Grid
- Provides a repeatable / reproducible environment:
  - Docker is most well known:
    - Apptainer / singularity also well used within (and outside) HEP community

Correctness: Summary

- Expect debugging and testing of code to take longer than the code implementation
- Make use of:
  - Human code reviews
  - Tools for static and dynamic analysis:
  - Unit tests, integration tests, etc.

- Remember the next person who will maintain your code (it might still be you …)
  - Documentation; inline, external, …

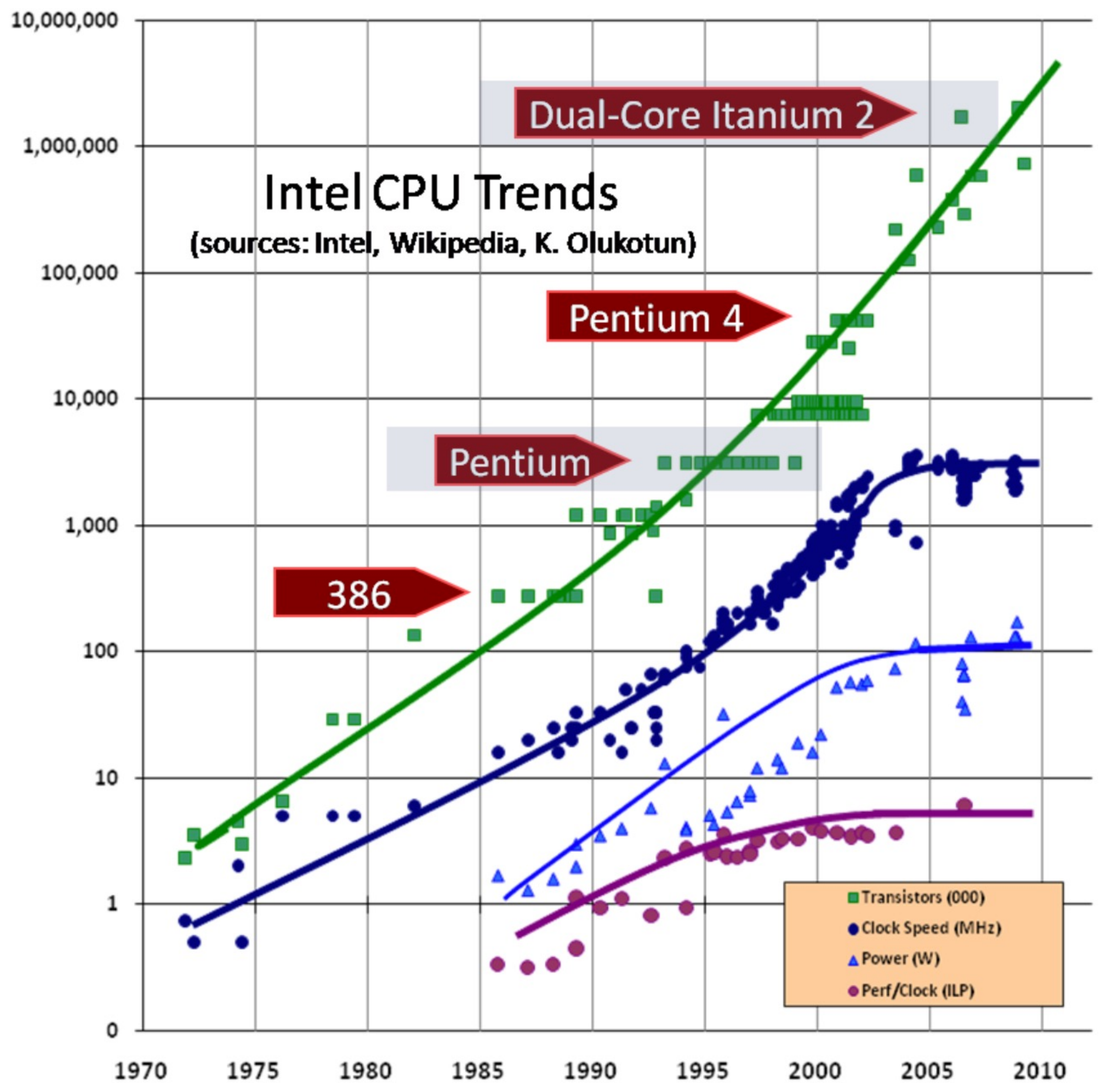
- Questions?

Fast computing

- High throughput computing
  - Can parallelise and buffer data for later processing
  - LHC;
    - Generally ‘embarrassing parallel’ class of problems
    - Events (typically) independent of each other (modulo detector conditions, etc.)
  - Maximise throughput = events/second
- Low latency computing
  - impossible or not useful to buffer
  - High frequency trading, autonomous vehicles
- High performance computing
  - Problems that don’t parallelise easily - supercomputer
  - Climate modelling (inter-grid communications)
  - Fast connections between processors, lots of RAM

CPUs

- CPU clock speed no longer following Moore's law: (since ~ 2006)
  - 'Free code speed-up'
  - Transistor density does continue to follow Moore's law
- Memory has fallen behind CPU - big bottleneck frequently memory access
- More processing power available through parallelism
  - More intelligence and complexity now required to keep increasing performance gains



Source: Herb Sutter

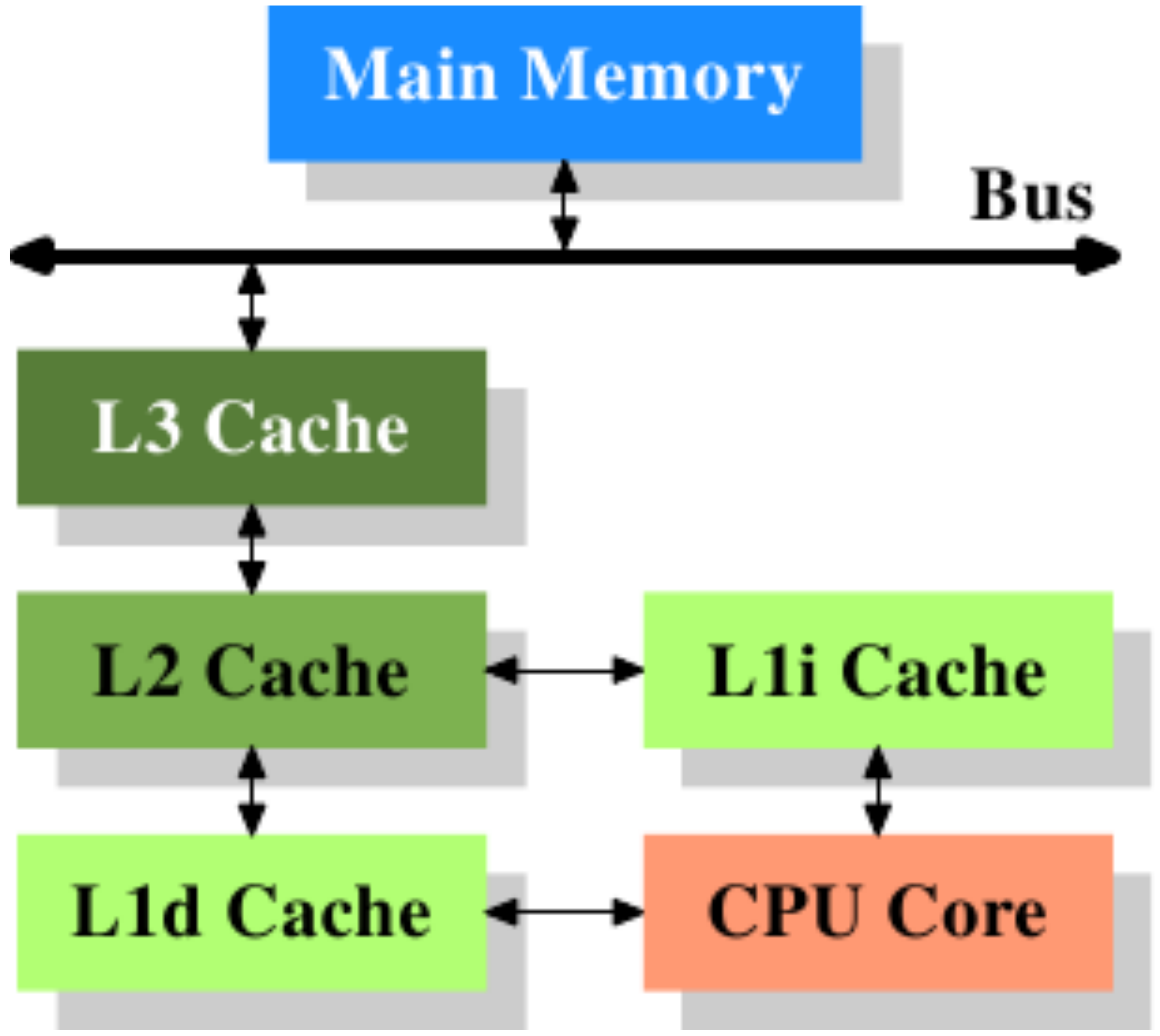
Memory

- Fast memory is expensive (in cost)
- Moving data between memory caches (and to/from io devices is expensive (in time)

- Fastest memory in L1 caches, closest to the CPU,
  - Then L2, L3,
- Next fastest is RAM
- Slowest then is on the physical storage:
  - (HDD, SSDs, Tape)

- Cache miss => retrieving data from a different cache

- NUMA: modern servers with multiple sockets and processors:
  - Memory allocated to individual sockets, sharing and moving data between memory adding to overheads

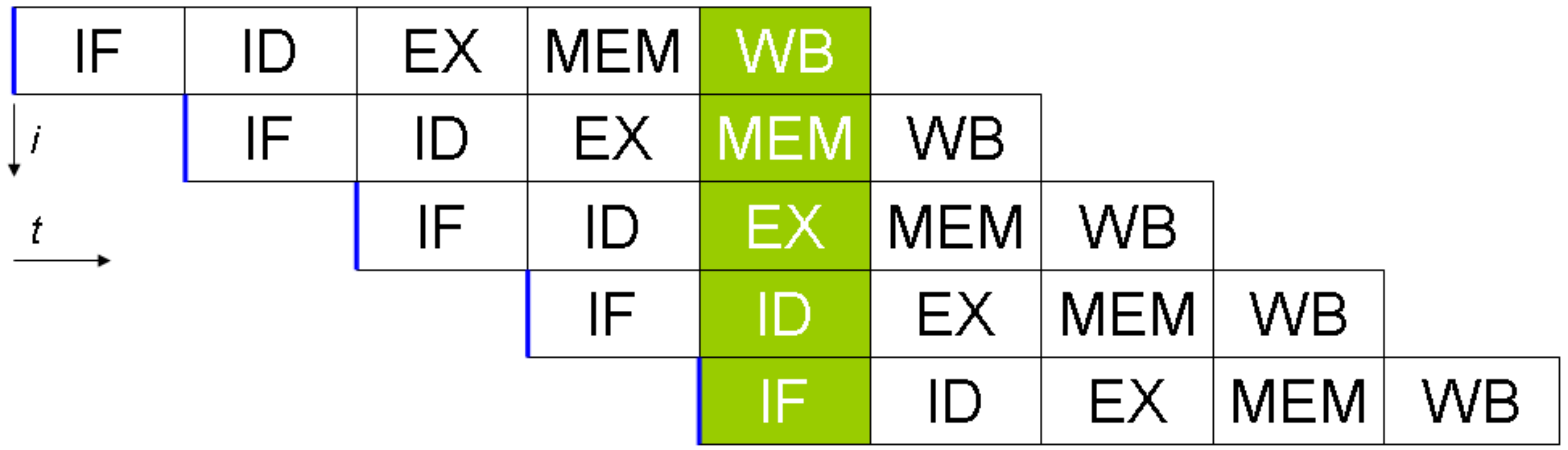


Source: What Every Programmer Should Know About Memory

- *NUMA: Non-uniform memory access*

Pipelining

- Pipelining allows processors to execute multiple instructions per clock cycle:
  - Aim to keep each processor unit busy per cpu by dividing instructions into a series of parallel steps.



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Five stage : Instruction pipeline

IF = Instruction Fetch,
ID = Instruction Decode
EX = Execute
MEM = Memory access
WB = Register write back

- Only works if code is linear
  - Branching is an issues;
    - e.g. conditional statements, if / else

- Unable to load instructions past the branch point

Branch prediction

- Aim to solve problem of not allowing instruction loading past branch point

- Module within CPU decides which branch to take (see here)
  - Allows CPU to pipeline code with branches;
  - However, significant penalty if non-predicted branch is selected.
    - CPU has to load new code into pipeline

```
If (x) {
…
} else {
…
}
```

- General Solutions:
  - remove branches if possible
  - Unbalance the branching (e.g. 10 / 90% vs 50/50%).

- Sort the data:
  - See stackoverflow.com
  - i.e. branch prediction might be optimal on sorted data
    - However, penalty for sorting operation might make this ultimately slower …

Parallelism / concurrency

- An entirely parallelisable calculation is referred to as embarrassingly parallel
    - Event generation: every collision has no dependency on the previous
    - Simulate each collision on a separate CPU: scale to number of CPUs available

- Most calculations have a parallel and serial component, which limits the speedup

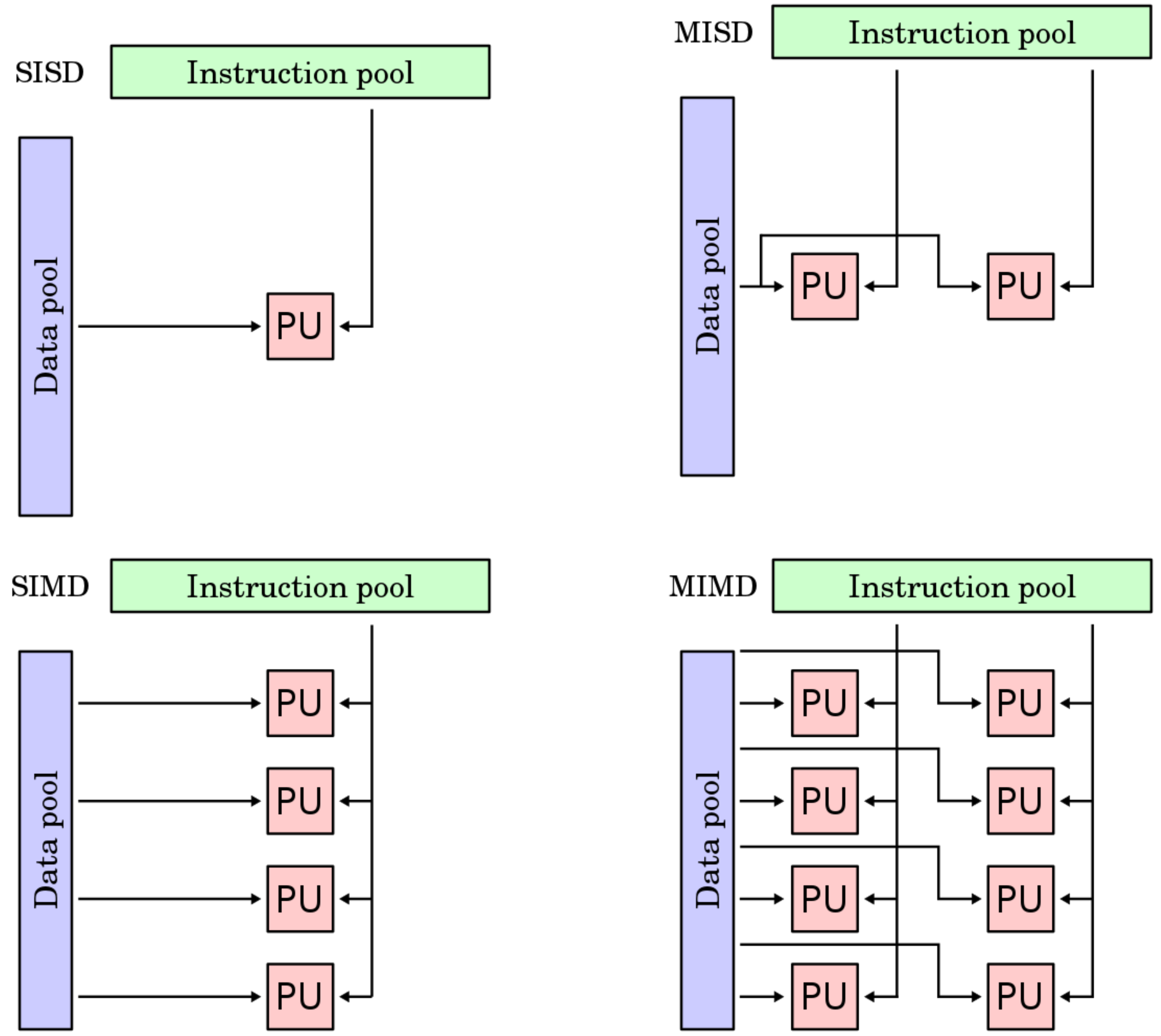
Parallel and serial components

- Naive example
  - Assume:
    - Making a car requires 1000 identical parts: each part takes 1 minute to make
- The 1000 identical parts must be assembled in a final step: this takes 60 minutes
- Serial path
  - 1000 parts assembled by a single worker + final assembly = 1000 + 60 minutes = 1060 minutes
- Parallel path
  - 1000 parts assembled by different workers simultaneously + car assembly = 1 + 60 minutes = 61 minutes
- Maximum speedup 1060/61 = 17.4
- No further gains without improving the final assembly (serial part)
- Amdahl’s law formalises this reasoning to an equation.

Parallel architectures

- A broad definition (Taxonomy) of parallel architectures; proposed in 1966 (and extended in 1972) by Michael J. Flynn.

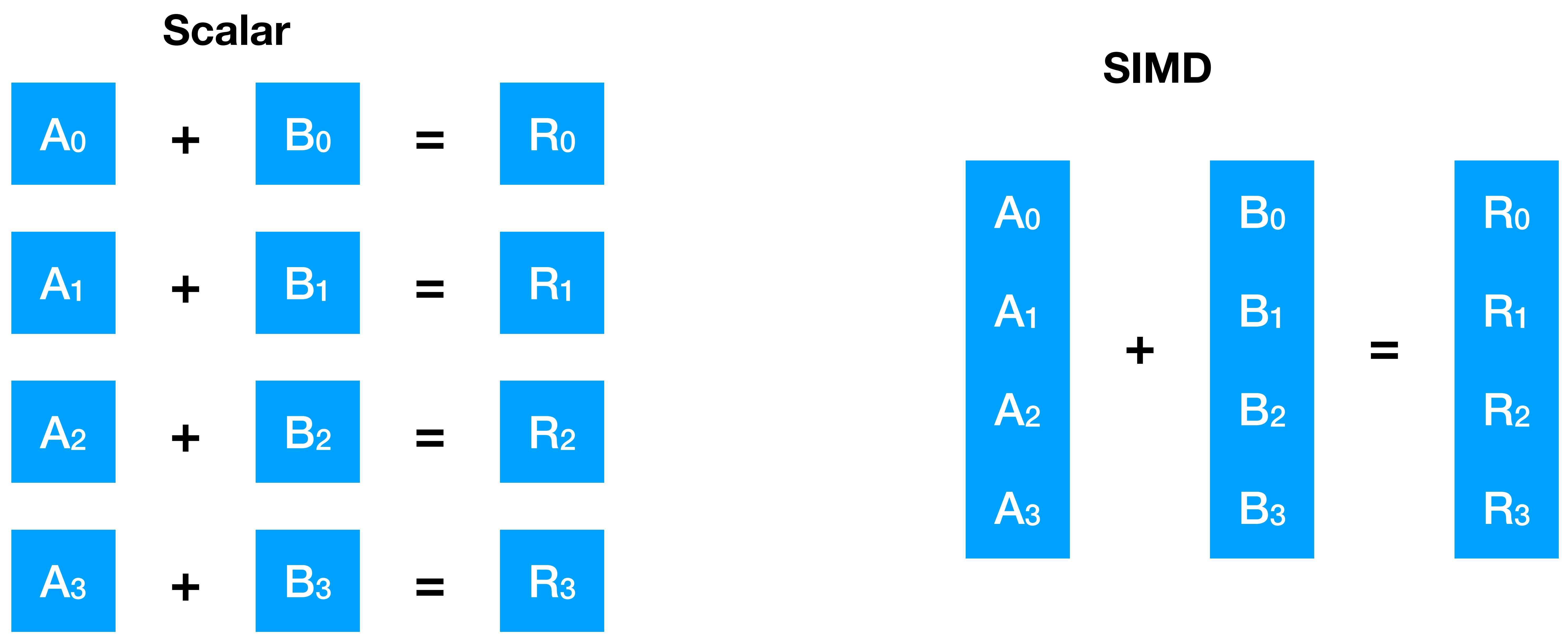
- **SISD**: Single Instruction, Single Data)
    - single-threaded operation
- **SIMD**: Single Instruction, Multiple Data)
    - vector operations
- **MISD**: Multiple Instruction, Single Data:
    - Not common
- **MIMD**: Multiple Instruction, Multiple Data)
    - multi-threaded operation



Source: Wikipedia

Vectorisation/SIMD

- Modern CPUs can execute the same instructions on multiple data simultaneously
- Consider vector addition of two vectors into a result vector



- Scalar operation:
  - Each element treated separately
- SIMD:
  - Computation performed across multiple elements simultaneously.

- Different architectures depending on CPU generation: MMX, SSE, AVX
  - Code generated for one instruction set will not work with another

Auto-vectorisation

- Compiler can generate appropriate vector instructions for loops etc (see Compiler Explorer)
- Will not always apply it if not beneficial
- Might need re-think of code implementation to be used:
  - e.g. not looping over a vector of Particles (each containing a px,py,pz), but rather looping through vectors of px,py,pz, where a corresponding row of elements represents a given particle.

```
 1  #include <array>
 2  #include <iostream>
 3
 4  int main() {
 5          std::array<int, 100> int_array;
 6          for (unsigned int i = 0; i < 100; i++) {
 7                  int_array[i] = i*4;
 8          }
 9          std::cout << int_array[10] << std::endl;
10          return 0;
11  }
```

```
clang++ -msse4.2 -std=c++11 vec.cpp -O2 -Rpass=loop-vectorize
vec.cpp:6:2: remark: vectorized loop (vectorization width: 4,
    interleaved count: 1) [-Rpass=loop-vectorize]
        for (unsigned int i = 0; i < 100; i++) {
        ^

#similarly: g++ -std=c++11 vec.cpp -O3 -fopt-info-vec
```

Implementing vectorisation

- Autovectorisation is fragile: re-order your code and it can disappear

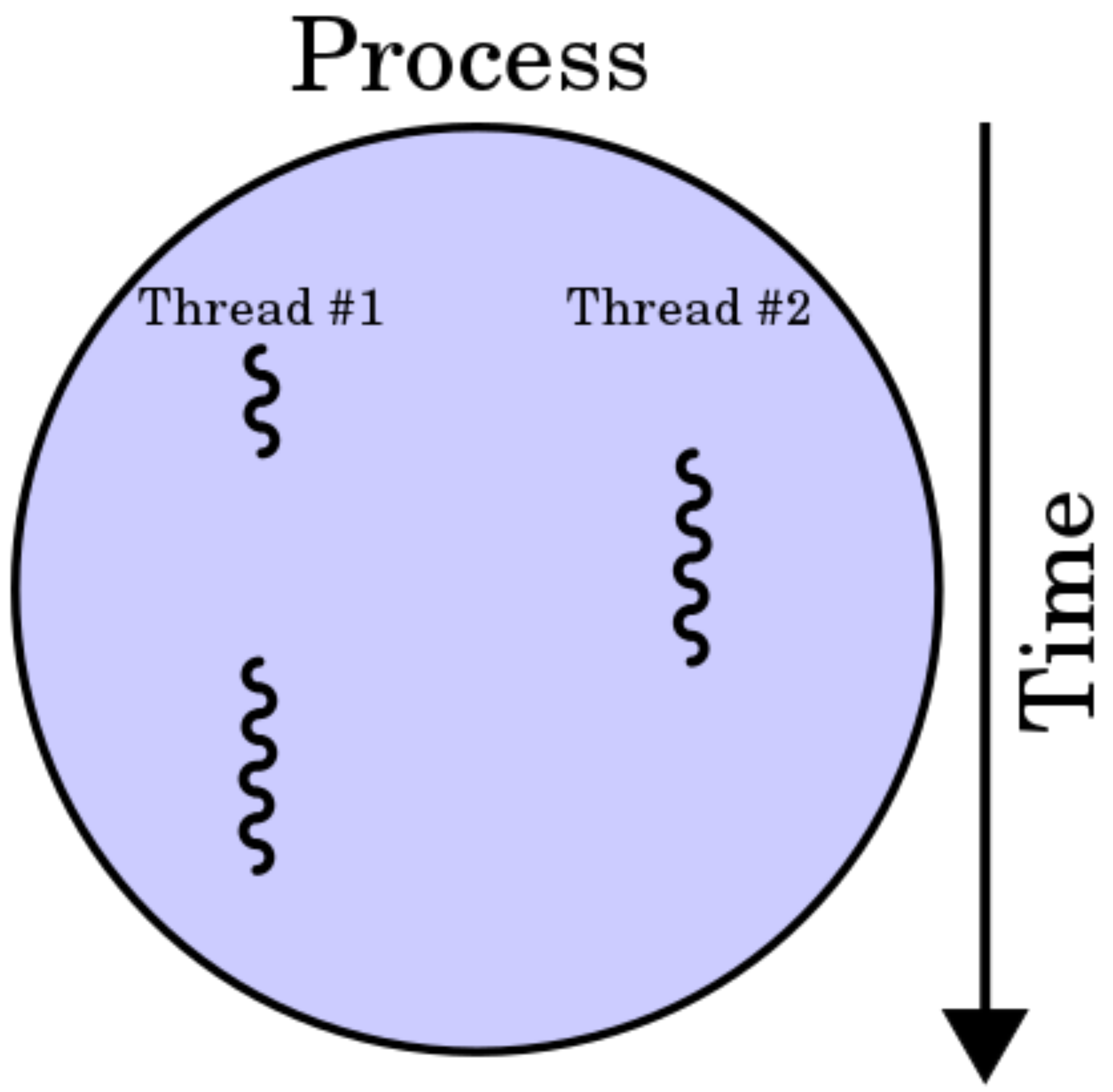
- Can write using vector intrinsics: functions that act on arrays of data and operate accordingly
    - Resulting code will crash on different CPU type (e.g. older Intel/AMD)
- More complicated to write

- Easiest solution: use a library written by an expert
- E.g. for cos(), exp(), atan2()
    - CERN VDT
    - Intel and AMD mathematical function libraries
- For matrix/linear algebra
    - Eigen

Multi-Threading

- Multiple instructions, multiple data : MIMD
- A thread is a sub-program controlled by your main program
- Operating system decides when and on which CPU they run
  - (e.g n threads running across m CPUS)
- Thread order is non-deterministic: can lead to difficult bugs
  - Race conditions, deadlocks, irreproducible output
- Usually one thread per CPU
- Swapping between threads on one CPU: “hyper-threading”

- Simplest case:
  - Single thread runs until blocked by some high-latency operation (e.g. a cache miss, and needed to await for data)
- More complex:
  - simultaneous multithreading (SMT) tries to exploit limited instruction-level parallelism per thread and attempts to call instructions from multiple threads to minimise wasted slots per cycle



Source: Wikipedia

Threading with OpenMP

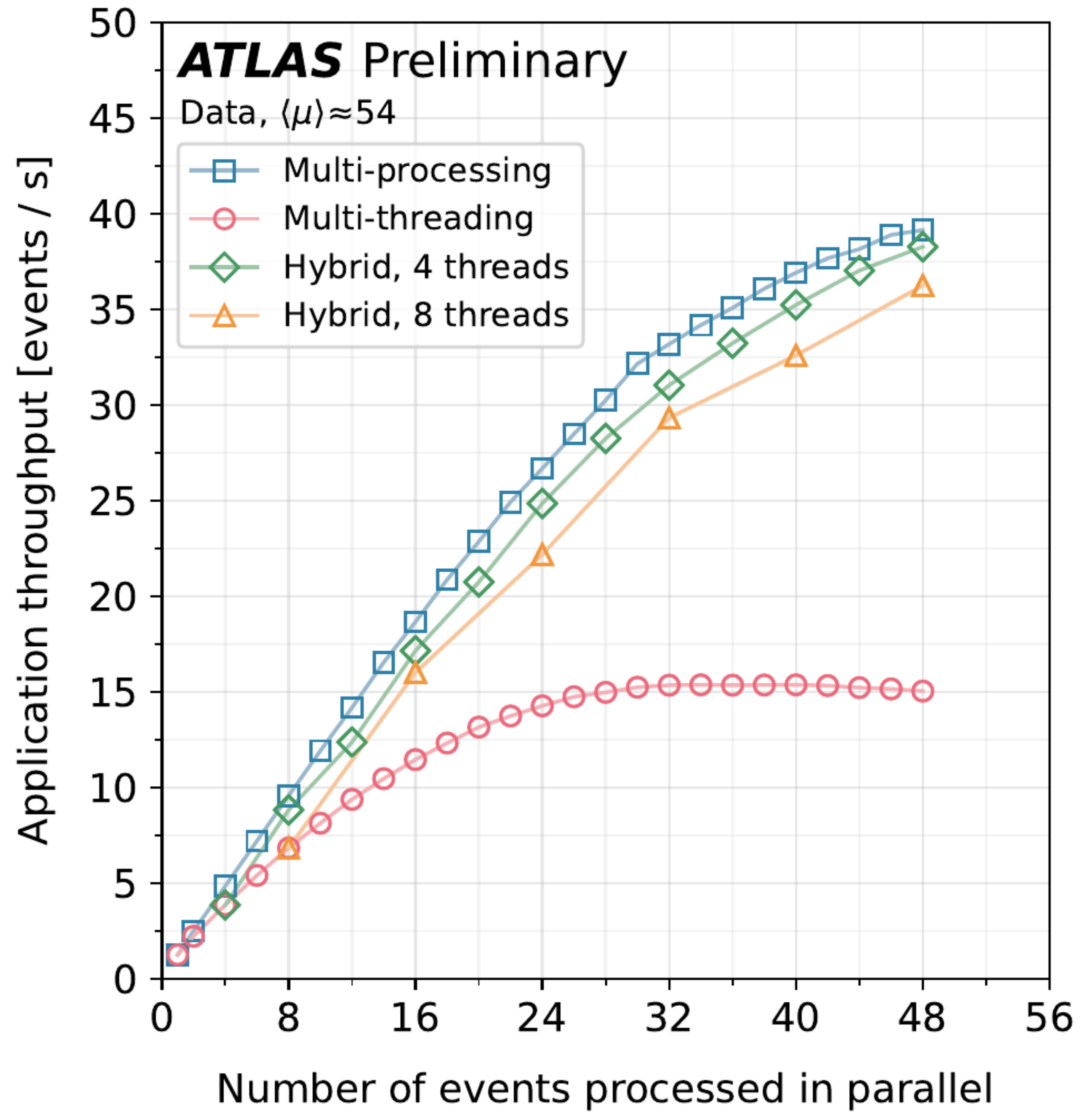
- Popular interface for multiprocessing tasks

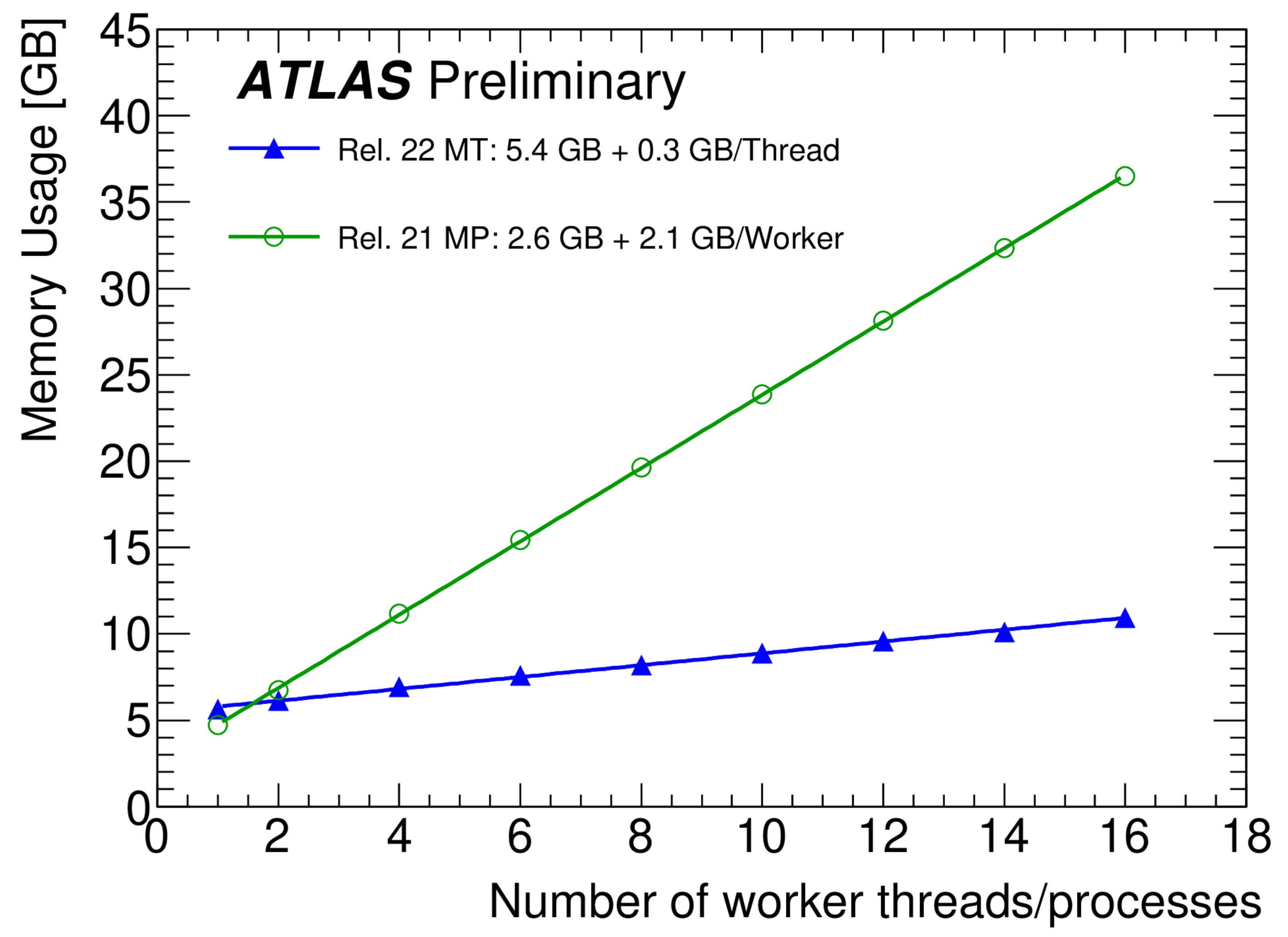
```
#include <iostream>
#include <omp.h>
int main() {
  #pragma omp parallel num_threads(4)
  {
    int thread = omp_get_thread_num();
    int total = omp_get_num_threads();
    std::cout << "Greetings from thread " << thread << "
         out of " << total << std::endl;
  }
  std::cout << "parallel for ends." << std::endl;
  return 0;
}
g++ -fopenmp test_omp.cpp && ./a.out
Greetings from thread 1 out of 4
Greetings from thread 0 out of 4
Greetings from thread 3 out of 4
Greetings from thread 2 out of 4
```

- Code must ensure that there is no dependency on the order that threads run, or on data local to the thread.
  - May need to introduce synchronisation steps to wait for all current threads to complete before moving on.
- Intel Threading Building Blocks: used by ATLAS, CMS, LHCb for multi-threaded data processing
- OpenMP is best for simpler situations with limited relationships between threads
- Other frameworks (e.g. HPX) also exist.

Memory sharing

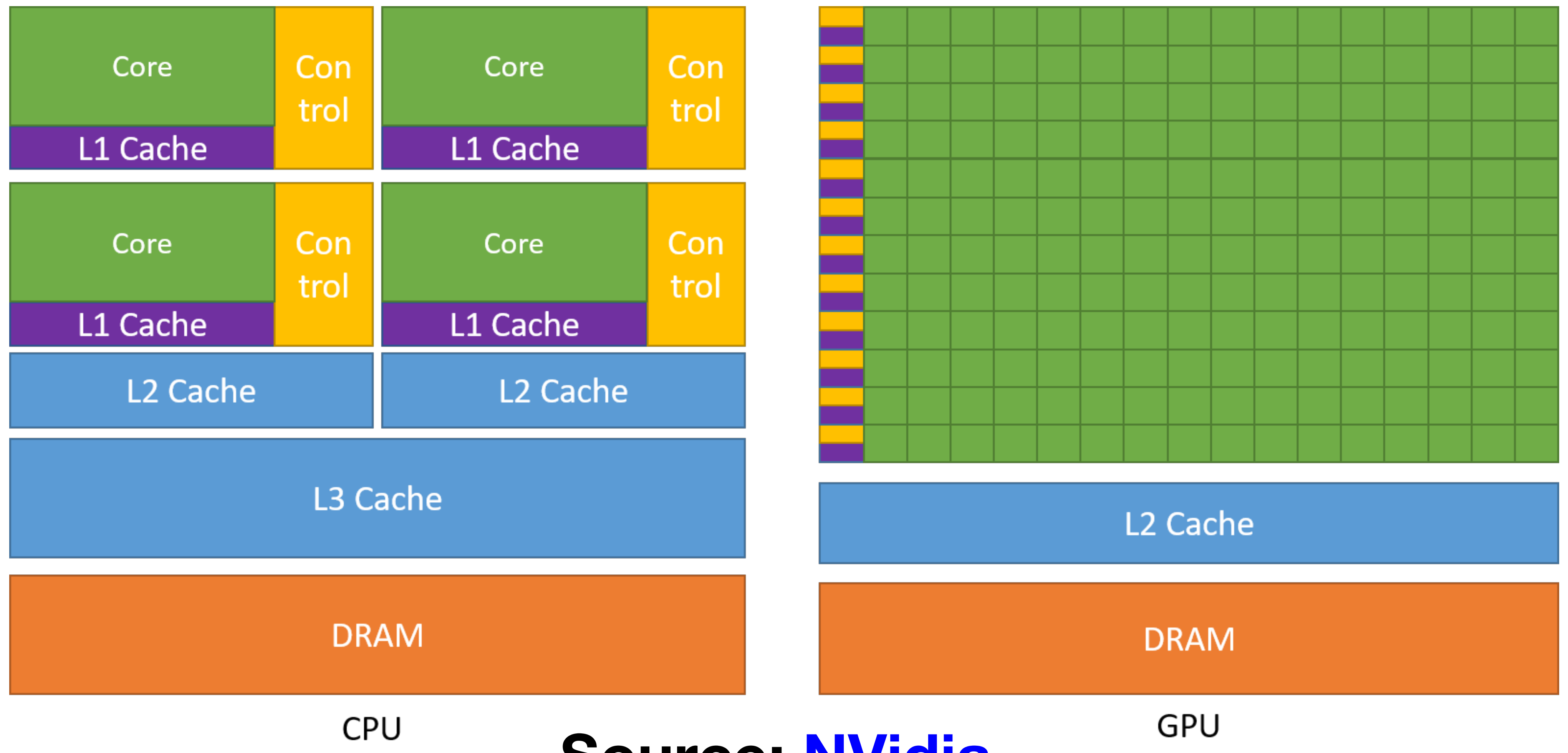
- For particle physics we usually want to run our CPUs at 100% capacity
- Multi-CPU:
  - Because each event is independent, 2x CPUs = 2x events/second
  - Single-threaded application usually gives best throughput
- Multi-process
  - Much of our application memory is read-only (detector layout, magnetic field)
  - Linux mechanism (copy-on-write) means we can fork several subprocesses to process events and share read-only memory
- Multi-threading
  - Threads can share all their read/write memory – big reduction in memory use
- Once software is thread-safe (threads not interfering with each other), can tune no. threads and no. processes to environment





GPU programming

- Initially designed for graphics calculations: matrix and vector operations
- Tailored towards embarrassingly parallel problems
- Increasingly used for scientific applications, machine learning:
    - Tools such as tensorflow allow to easily swap between cpu and gpu backends
- For programming software, several competing options:
    - CUDA for NVidia
    - HIP for AMD
    - OpenCL, SYCL: multi-platform
- Movement of data into the gpu and the results back out require careful design.
- HEP software moving towards these, but difficult/labour-intensive to port:
    - i.e. Challenges is to write code once, which can be effectively and optimally used on the multiple architectures of heterogeneous environments
- Increasingly popular for supercomputers etc: pulling HEP that way



Source: NVidia

Parallelism: Summary

- Vectorisation and multi-threading are harder to work with than single-threaded programming
  - Necessary if you want to get the highest possible performance
- Even if you don’t need the best performance, you can still apply some of this through libraries
  - Compilers will try to optimise your code (but may need some help)

- Programming for heterogeneous environments:
  - Active area of study / discussion

Runtime measurements

- Simples method to measure runtime: the `time` command
  - `time factor 1234567890987654321123456789333333333`

```
1234567890987654321123456789333333333: 3 3 3 23 43 27062723775121 1708375824282413291

real  0m0.867s
user  0m0.866s
sys   0m0.000s
```

- user = time spent in your code
- sys = time spent in (Linux) kernel code
- real = sum of user + sys, referred to as **Walltime**

- You care about real, but you can only affect user

- If you're worried about system calls, you can use strace to see which ones are used (see e.g Julia Evans strace zine)

- Walltime is the most important number for profiling, but also the most difficult to measure accurately
- Varies with CPU
- Some variation from operating system
  - Other running processes can influence the results
- Penalty if running in a virtual machine

Sampling profilers

- Often, it's not the overall time that's most useful,
    - but knowing which calls and sections of your code take the most time.
    - Optimise your effort there
- Simplest sampling profiling: the debugger;
    - Run your code many times in the debugger and interrupt (i.e. Sample) the code at various points
-
```
^C
Program received signal SIGINT, Interrupt.
0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
(gdb) bt
#0  0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
#1  0x00007f8d81f0baaa in frameworkCode() ()
   from frameworkCode.so
#2  0x00007f8d81f0bc0b in main() ()
   from program.so
```

- If your program spends 90% of its time in program X, then your sampling should find it in the call stack 90% of the time.
    - Run your code say 10 times; does it stop in a similar place each time ?

Sampling profilers

- costlyFunction() (top of the stack trace): where program was when halted
  - “Self cost”
- frameworkCall(), main(): call the function doing the work
  - “Total cost”
- Self cost ≤ total cost

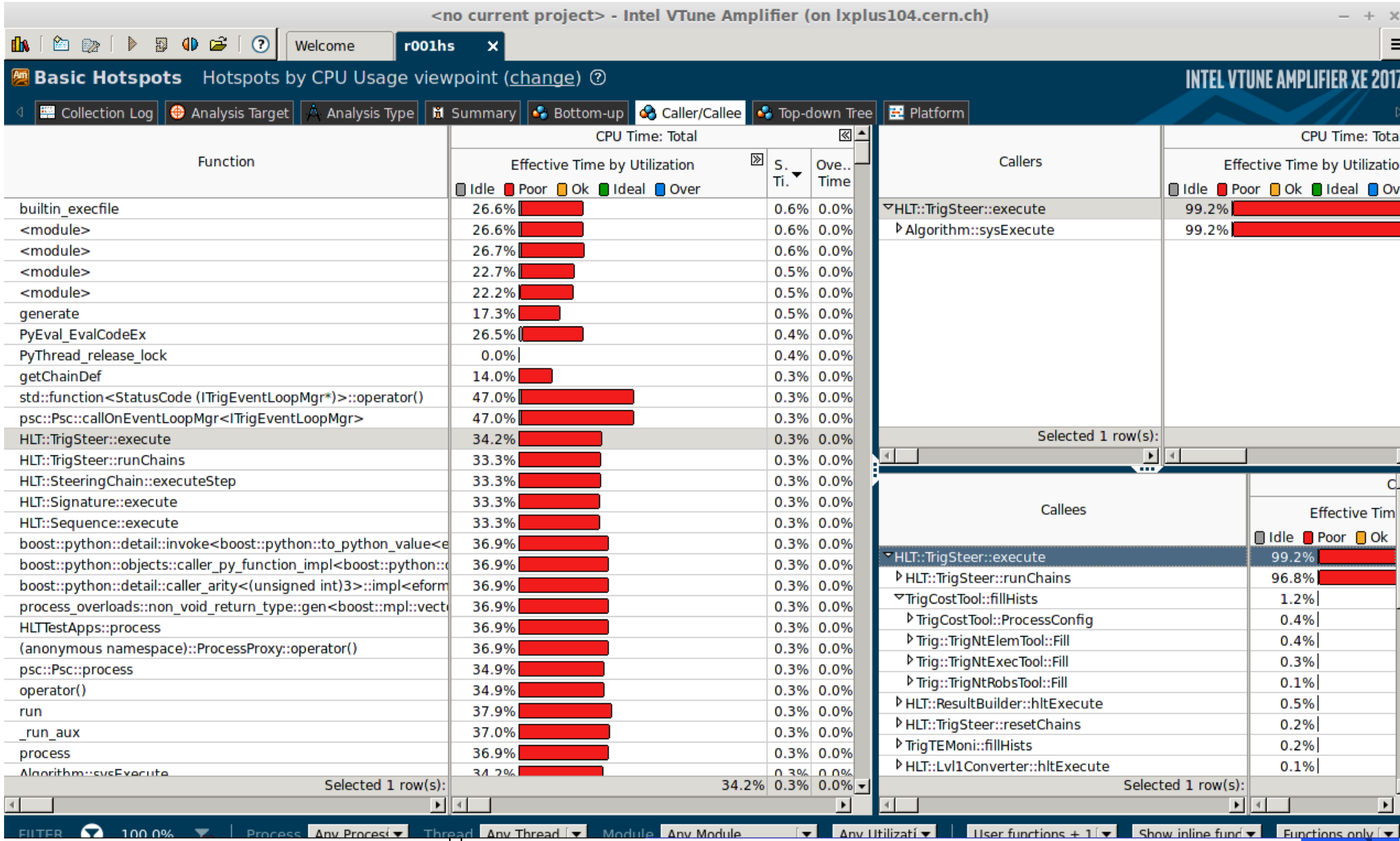
- Focus optimisation efforts on functions with highest self-cost

```
^C
Program received signal SIGINT, Interrupt.
0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
(gdb) bt
#0  0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
#1  0x00007f8d81f0baaa in frameworkCode() ()
   from frameworkCode.so
#2  0x00007f8d81f0bc0b in main() ()
   from program.so
```

- Fortunately several tools exist to sample and visualise (e.g KCacheGrind) the results in a call graph
- gperftools, Intel VTune, igprof

VTune

- Intel VTune is an goof tool
- Free to download, if you register with Intel
- Multi-language support; CPU, GPU, and FPGA



<no current project> - Intel VTune Amplifier (on lxplus104.cern.ch)

Welcome | r001hs

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

INTEL VTUNE AMPLIFIER XE 2017

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Function	CPU Time: Total – Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	S. Ti.	Ove.. Time
builtin_execfile	26.6%	0.6%	0.0%
<module>	26.6%	0.6%	0.0%
<module>	26.7%	0.6%	0.0%
<module>	22.7%	0.5%	0.0%
<module>	22.2%	0.5%	0.0%
generate	17.3%	0.5%	0.0%
PyEval_EvalCodeEx	26.5%	0.4%	0.0%
PyThread_release_lock	0.0%	0.4%	0.0%
getChainDef	14.0%	0.3%	0.0%
std::function<StatusCode (ITrigEventLoopMgr*)>::operator()	47.0%	0.3%	0.0%
psc::Psc::callOnEventLoopMgr<ITrigEventLoopMgr>	47.0%	0.3%	0.0%
HLT::TrigSteer::execute	34.2%	0.3%	0.0%
HLT::TrigSteer::runChains	33.3%	0.3%	0.0%
HLT::SteeringChain::executeStep	33.3%	0.3%	0.0%
HLT::Signature::execute	33.3%	0.3%	0.0%
HLT::Sequence::execute	33.3%	0.3%	0.0%
boost::python::detail::invoke<boost::python::to_python_value<e	36.9%	0.3%	0.0%
boost::python::objects::caller_py_function_impl<boost::python::c	36.9%	0.3%	0.0%
boost::python::detail::caller_arity<(unsigned int)3>::impl<eform	36.9%	0.3%	0.0%
process_overloads::non_void_return_type::gen<boost::mpl::vect	36.9%	0.3%	0.0%
HLTTestApps::process	36.9%	0.3%	0.0%
(anonymous namespace)::ProcessProxy::operator()	36.9%	0.3%	0.0%
psc::Psc::process	34.9%	0.3%	0.0%
operator()	34.9%	0.3%	0.0%
run	37.9%	0.3%	0.0%
_run_aux	37.0%	0.3%	0.0%
process	36.9%	0.3%	0.0%
Algorithm::sysExecute	34.2%	0.3%	0.0%
Selected 1 row(s):	34.2%	0.3%	0.0%

Callers	CPU Time: Total – Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)
HLT::TrigSteer::execute	99.2%
Algorithm::sysExecute	99.2%
Selected 1 row(s):	

Callees	Effective Time (Idle, Poor, Ok)
HLT::TrigSteer::execute	99.2%
HLT::TrigSteer::runChains	96.8%
TrigCostTool::fillHists	1.2%
TrigCostTool::ProcessConfig	0.4%
Trig::TrigNtElemTool::Fill	0.4%
Trig::TrigNtExecTool::Fill	0.3%
Trig::TrigNtRobsTool::Fill	0.1%
HLT::ResultBuilder::hltExecute	0.5%
HLT::TrigSteer::resetChains	0.2%
TrigTEMoni::fillHists	0.2%
HLT::Lvl1Converter::hltExecute	0.1%
Selected 1 row(s):	

FILTER 100.0% Process Any Process Thread Any Thread Module Any Module Any Utilizati User functions + 1 Show inline func Functions only

Code Instrumentation

- High-level languages (e.g. C++, Python) have tools to “instrument” code:
  - As an example: Adding in timing information for C++

```
using namespace std;
using namespace std::chrono;
auto start_time = high_resolution_clock::now();
doSomething();
auto end_time = high_resolution_clock::now();
cout << "Time:␣" << duration_cast<microseconds>(end_time -
    start_time).count() << endl;
```

- Carries a performance overhead; not to be used within tight loops
- Google Benchmark builds this into a useful framework to benchmark functions

Code Instrumentation

- Several tools also available for python: cProfile builtin, e.g.

10000007		function	calls in	2.048 seconds
ncalls	**tottime**	percall	cumtime	percall filename:lineno(function)
1	**0.115**	0.115	2.048	2.048 <string>:1(<module>)
1	**0.000**	0.000	1.933	1.933 test_profile.py:12(run)
1	**1.219**	1.219	1.831	1.831 test_profile.py:3(create_array)
1	**0.000**	0.000	2.048	2.048 {built-in method **builtins.exec**}
1	**0.000**	0.000	0.000	0.000 {built-in method **builtins.print**}
1	**0.102**	0.102	0.102	0.102 {built-in method **builtins.sum**}
10000000	**0.612**	0.000	0.612	0.000 {method '**append**' of '**list**' **objects**}
1	**0.000**	0.000	0.000	0.000 {method 'disable' of '_lsprof.Profiler' objects}

```
import cProfile

def create_array():
    data = []
    for i in range(10000000):
        data.append(i)
    return data

def sum_data(d):
    return sum(d)

def run():
    arr = create_array()
    print(sum(arr))

if __name__ == "__main__":
    cProfile.run('run()')
```

- Use of Profile class and pstats to update the output and fine tune the profiling

ncalls	**tot time**	**per-call**	**cumtime**	**percall**	**filename:lineno(function)**
1	1.270	1.270	1.909	1.909	test_profile.py:3(create_array)
10000000	0.639	0.000	0.639	0.000	{method '**append**' of '**list**' **objects**}
1	0.137	0.137	0.137	0.137	{built-in method **builtins.sum**}
1	0.000	0.000	0.000	0.000	{built-in method **builtins.print**}

```
if __name__ == "__main__":
    import cProfile, pstats
    profiler = cProfile.Profile()
    profiler.enable()
    run()
    profiler.disable()
    stats = pstats.Stats(profiler).sort_stats('tottime')
    stats.print_stats()
```

Instrumentation

- perf is now the gold standard - sampling and instrumenting
- Part of Linux kernel (best results with new kernels)
- Monitor performance monitoring counters (PMCs)
- VTune also has access to these
  - (Some functionality requires root)

```
perf stat -d program
    10 152 172 182      cycles:u                     #
       3,451 GHz                                (49,86%)
    14 584 154 073      instructions:u               #
       1,44  insn per cycle                     (62,43%)
     2 318 605 154      branches:u                   #
        788,130 M/sec                             (74,93%)
        44 768 463      branch-misses:u              #
           1,93% of all branches                   (75,00%)
     4 116 170 377      L1-dcache-loads:u            #
        1399,150 M/sec                             (74,18%)
       167 821 302      L1-dcache-load-misses:u      #
          4,08% of all L1-dcache hits             (25,06%)
        45 252 042      LLC-loads:u                  #
           15,382 M/sec                             (24,89%)
         8 794 669      LLC-load-misses:u            #
```

Emulation

- Callgrind tool (part of Valgrind)
  - Valgrind is very powerful suite of tools for debugging and profiling (particularly with memory issues).
- Emulates a basic modern CPU, with level 1, level 2 caches, branch prediction (somewhat configurable)
- Runs slowly
- Information about cache misses and branch misprediction
- Produces output suitable for KCacheGrind

Compiler optimisation

- Standard compilers (GCC, clang) able to perform significant code optimisations:
  - -O0 = no optimisations applied
  - -O1, -O2 = basic, safe optimisations applied. (e.g. `g++ -O2`)
  - -03 = expensive optimisations (take a long time, may actually make code slower) applied

- -O2 is a good reference level; baseline it, and test against -O3
- Remember to try these optimisation levels, before optimisation by hand.
- Fine-tuned optimisation options available - check GCC/clang documentation for details

- Example: (godbolt)
  - Basic optimisation level applied at compilation:

```
int square(int n)
{
  int k = 0;
  while (true)
  {
    if(k == n*n)
    {
      return k;
    }
    k++;
  }
}
```

$\longrightarrow$

```
int square2(int n)
{
    return n*n;
}
```

Profiling and optimisation: Summary

- Profile helps identify bottlenecks and high ‘cost’ functions in code:
  - Measure and benchmark (Also have an understanding of what is ‘good enough’).
  - Many profilers available
- Biggest improvements usually will come from changing algorithms, rather than minor changes to code
- Once poor performance locations of code identified, you can decide where to focus your effort:
  - (Note - even in some cases, highly optimised code will still take the most time).
- Compiled languages (C++, fortran) faster in general than interpreted (python, ruby).
- Standard libraries exist (FFTW, BLAS, Eigen). Most probably these will be faster than your own implementations:
  - Don’t reinvent the wheel.

Floating-point and mathematical operations

```
y = d*x*x*x + c*x*x + b*x + a; //Bad
y = x*(x*(x*d+c)+b) + a;       //Good
```

- [illegible] han multiplication (Compiler will usually help in these cases)
    - Multiplication is faster than division

- Rearrange calculations to minimise number of operations
- Compiler won't necessarily do this for you (floating point rules)

- Be very careful about loosing clarity of the function of the code.
    - Clear and concise code might be more maintainable / error-free, than slightly more performant implementation.
- Consider if the trade-offs / performance increase is worth it.
    - e.g. if it's really a bottleneck in the code flow

- Square roots are slow
- Trigonometric functions, exp, log are also slow
- Consider using optimised libraries (e.g. VDT), and trig. Identities
- For linear algebra, use a library (e.g Eigen)

Loops and standard algorithms

- Don’t recalculate within a loop,
  - Move outside to the outmost possible level

```
for (i = 0; i < 50; i++) {
  for (j = 0; j < 50; j++) {
    x = sin(5*i) + cos(6*j);
    //Can move sin() into earlier loop
  }
}
```

- Consider memory management within loops.
  - e.g don’t create a vector<int> within a loop, if you can create outside and reserve enough memory.

- C++ (STL) and python (builtin, numpy) contain well tested / optimised standard algorithms (e.g. std::sort);
  - Familiarise yourself with the available ones, and use instead of your own.
- If the algorithm is the bottleneck, consider different options (merge sort, bubble sort, etc …)

Data Structures

- Worth thinking about which data format fits your problem
- In C++, std::vector is probably a good fit in most cases
  (but make sure you reserve enough size in advance!)
  - std::map and std::unordered_map are also useful

- Python (and C++):
  - Consider whether builtin types are sufficient before creating own types (and sub-classing existing types)

- Designing for data optimisation or object representation ?

Other comments

- Optimisation is important; Correctness must come first
- Ensure you understand (and document) any optimisations that are not concise.
  - (Might also want to keep the simpler implementation available, to help test for correctness).
- While using ‘standard libraries’ and algorithms are encouraged for ‘production’ work;
  - Writing and testing your own implementation remains the best way to learn;
    - Even if never deployed.
- Again - Correctness comes first; then strive for optimisations (where meaningful).

Memory

```
int g(int x) {
  int* i = new int(55);//On heap
  return x + *i;
  //Memory for i not given back to OS - leak
}
```

mory: **stack**

- Small amount of memory associated with program
- Fast to access - can be e.g. in CPU L1 cache
- E.g. variables in a function

- **Heap**:
  - Slower to access than stack
  - Can be dynamically allocated
  - If you don't free up memory, this is where it leaks (use … std::unique_ptr)
  - All the RAM available on the machine (if it runs out, it will use hard drive - v slow!)

- Using too much memory is bad:
  - Eventually you run out (memory leak) (or using swap)
  - Allocating memory has CPU overhead; more so if data doesn't fit (e.g. for L1 cache).
  - A single allocation is cheaper than smaller allocation.
- Better to access memory in order - data-locality
  - Appropriate data structures help with this

Memory allocators

```
LD_PRELOAD=/usr/lib/libtcmalloc.so.4 ./my_program
```

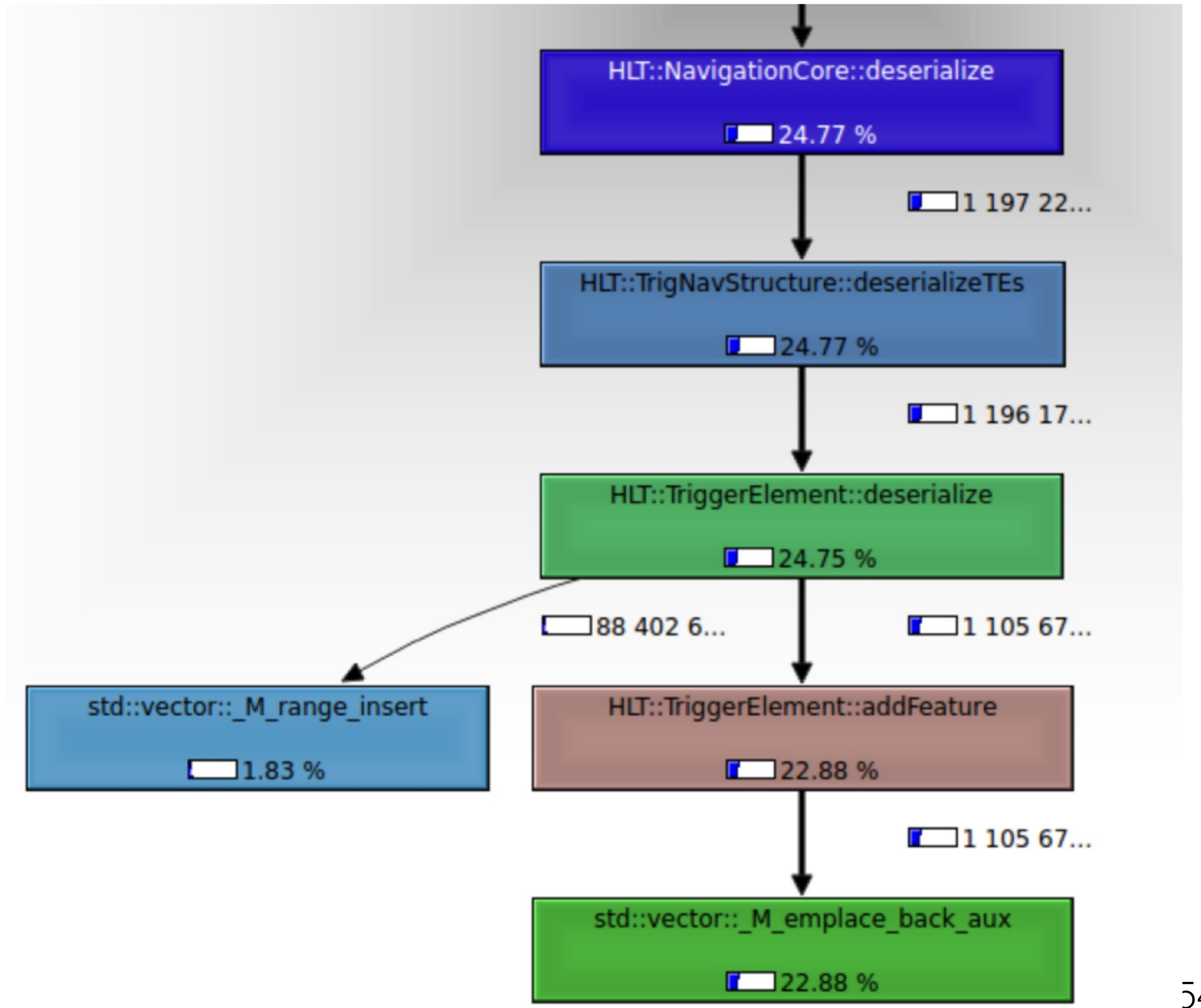
- Allocator decides how much to request at a time and how much should be contiguous
- glibc by default
- Others available, particularly jemalloc (Facebook) and tcmalloc (Google)
- No need to recompile, just preload
- May work better for your memory access pattern than glibc - free speedup!
- tcmalloc can also provides a printout when large allocations are made:

```
tcmalloc: large alloc 2720276480 bytes == 0x73eda000 @
tcmalloc: large alloc 2720276480 bytes == 0x2a96f0000 @
tcmalloc: large alloc 2720276480 bytes == 0x34b932000 @
```

- *Note that --enable-large-alloc-report must be added to ./configure in recent releases of tcmalloc*

Heap profilers

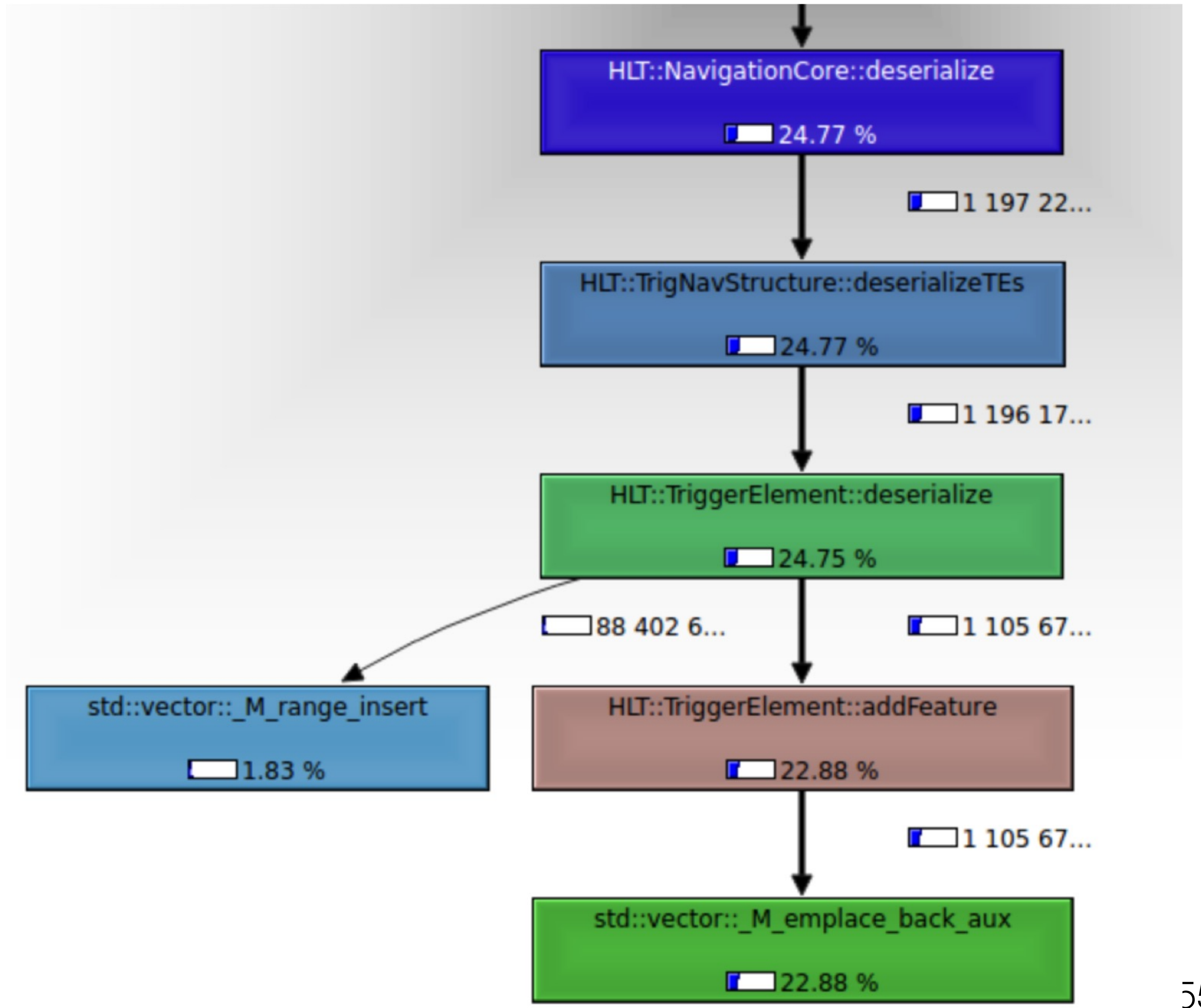
- jemalloc and tcmalloc both come with low-overhead profilers:
  - analyse which functions allocate most memory
- Output can be interpreted much as with a call-graph
- Best overall is heaptrack - see e.g. this ATLAS memory fix that it signposted


HLT::NavigationCore::deserialize
24.77 %
1 197 22...
HLT::TrigNavStructure::deserializeTEs
24.77 %
1 196 17...
HLT::TriggerElement::deserialize
24.75 %
88 402 6...
1 105 67...
std::vector::_M_range_insert
1.83 %
HLT::TriggerElement::addFeature
22.88 %
1 105 67...
std::vector::_M_emplace_back_aux
22.88 %

- Memory profiling is more difficult than CPU profiling - tools less advanced/convenient
- But improving all the time
- Can make a big difference if you're using a lot of memory

Heaptrack

- jemalloc and tcmalloc both come with low-overhead profilers:
    - analyse which functions allocate most memory
- Output can be interpreted much as with a call-graph
- Best overall is heaptrack - see e.g. this [ATLAS memory fix]() that it signposted


HLT::NavigationCore::deserialize
24.77 %
1 197 22...
HLT::TrigNavStructure::deserializeTEs
24.77 %
1 196 17...
HLT::TriggerElement::deserialize
24.75 %
88 402 6...
1 105 67...
std::vector::_M_range_insert
1.83 %
HLT::TriggerElement::addFeature
22.88 %
1 105 67...
std::vector::_M_emplace_back_aux
22.88 %

- Memory profiling is more difficult than CPU profiling - tools less advanced/convenient
- But improving all the time
- Can make a big difference if you're using a lot of memory

Profiling Summary

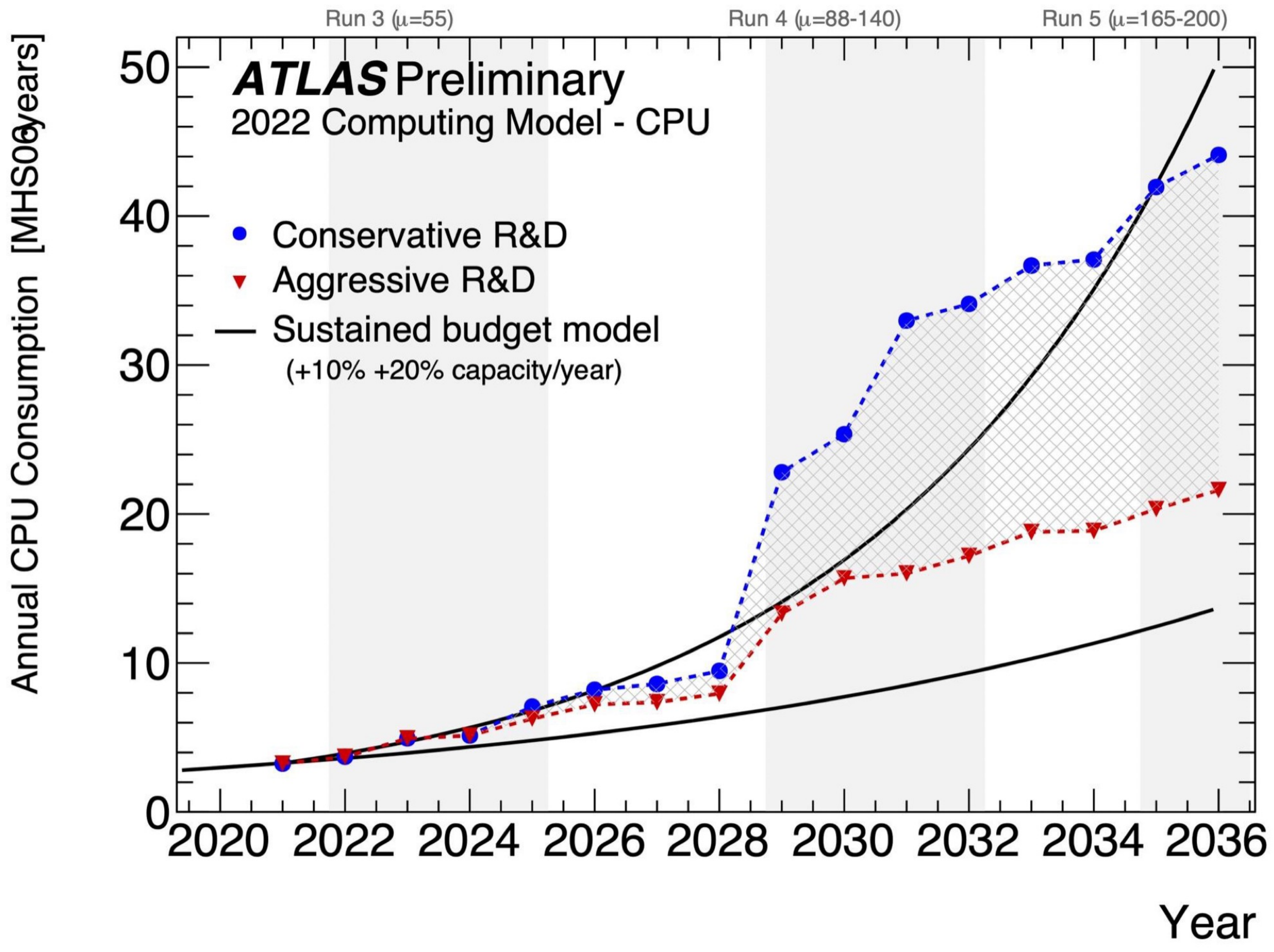
- Continuing developments, particularly in concurrency and memory safe programming
- Modern C++ (e.g. with std::unique_ptr) providing features to help minimise risks of memory leaks, etc.
  - Use them …
- A small amount of profiling/optimisation knowledge can dramatically improve your application performance
  - Profiling is more important than optimisation
  - Good debugging and profiling skills can help you in a lot of areas throughout your PhD (and beyond)

- Advanced techniques useful once you've done the easy bits

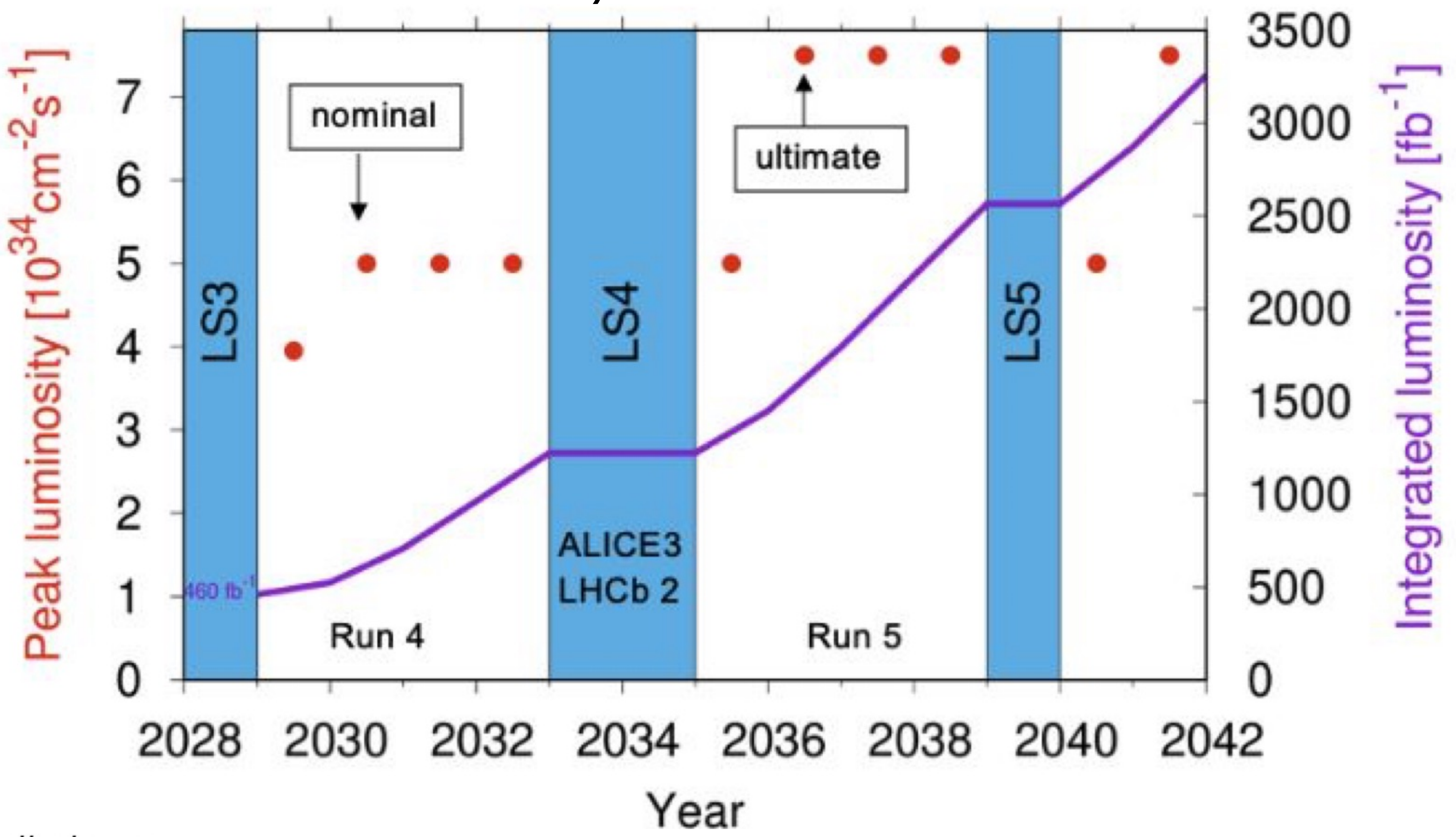
- Particularly for C++: see books (e.g. by Herb Sutter, Scott Meyers) and videos (e.g. from CppCon, pyCon)

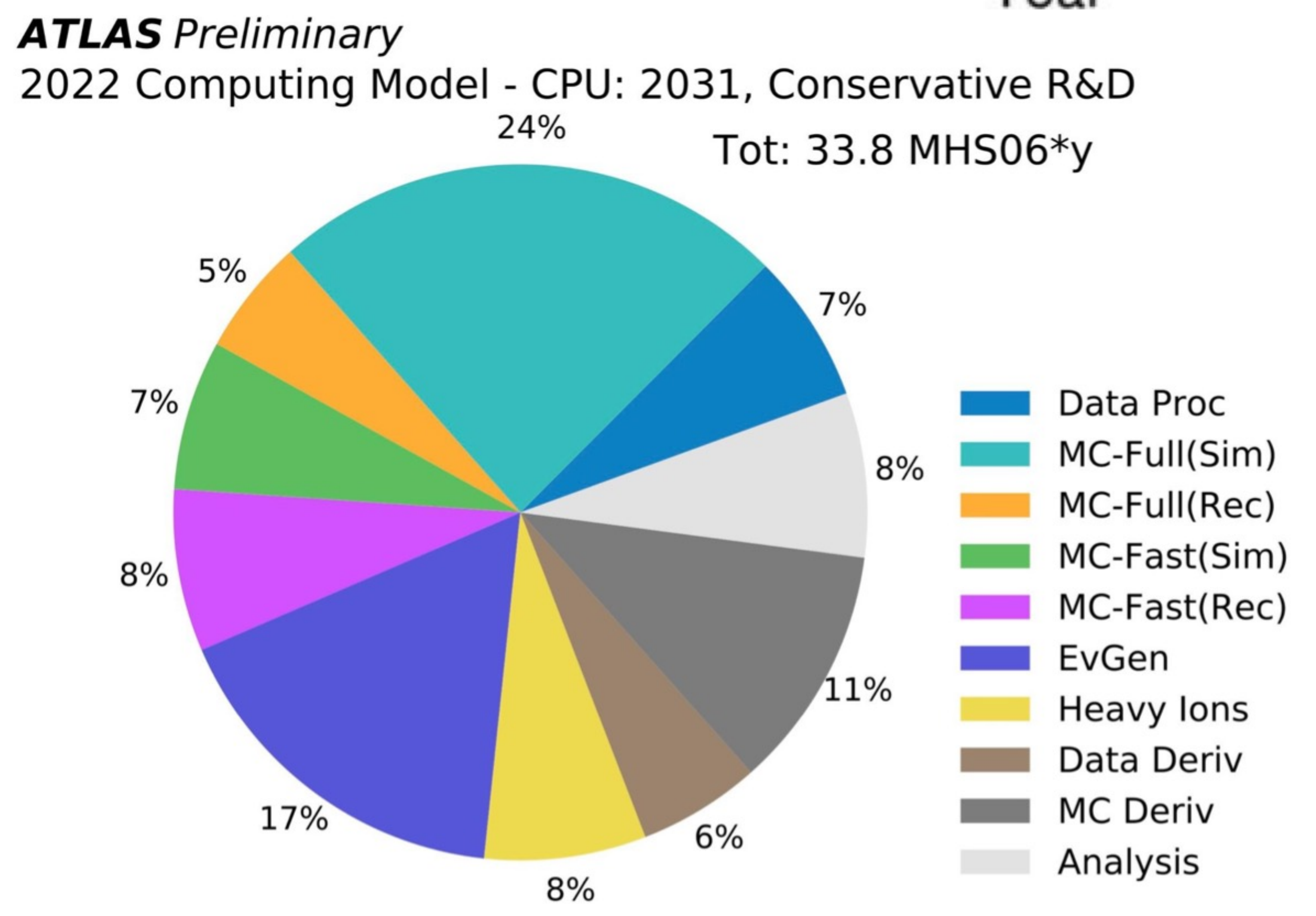
Computing challenges for (HL)-LHC and others

- Modelling of requirements for HL-LHC shows simple scaling of technologies not sufficient to meet needs.
    - Software R&D required now to meet these demands and optimise for physics exploitation.



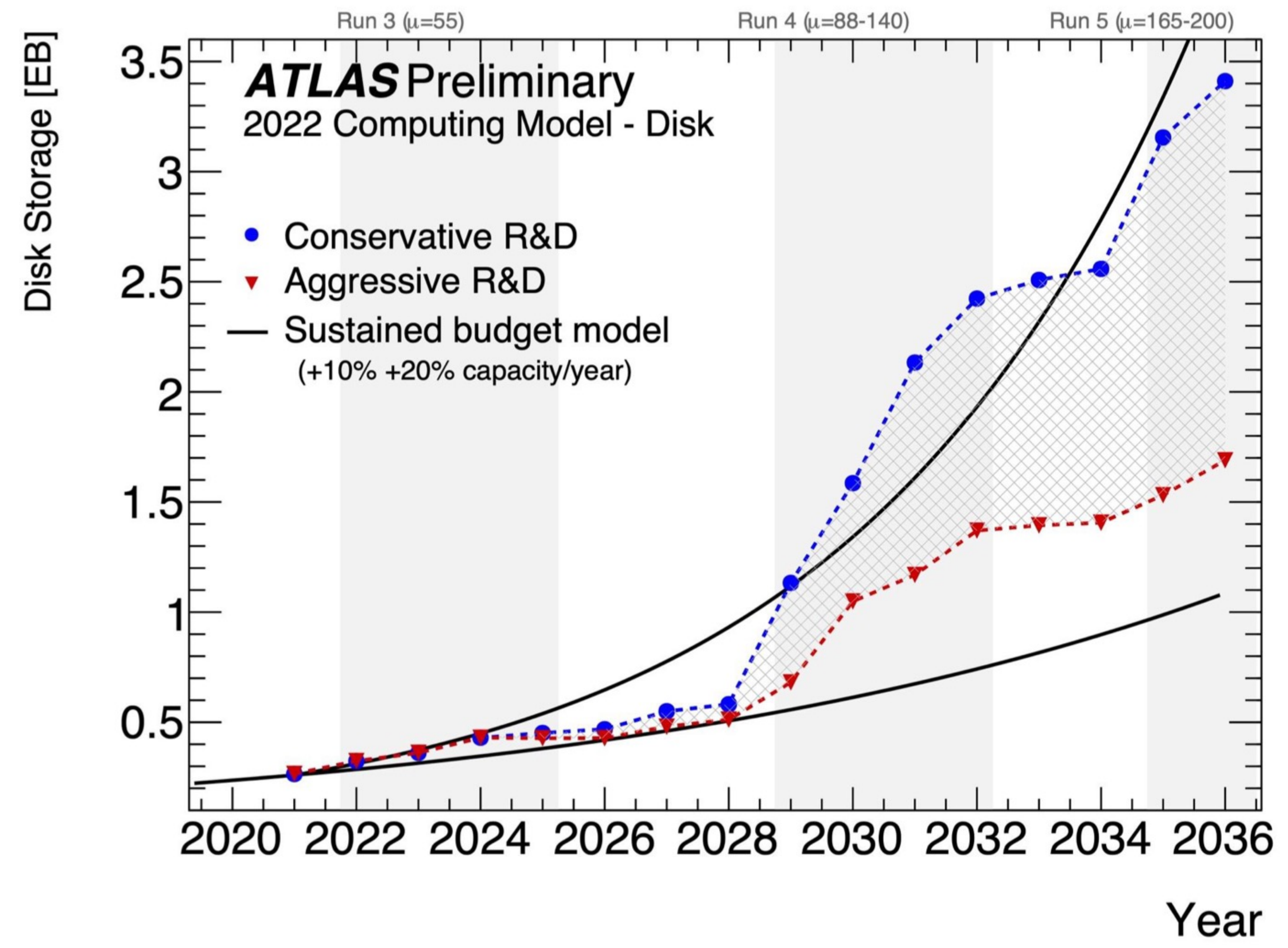
Preliminary schedule HL-LHC

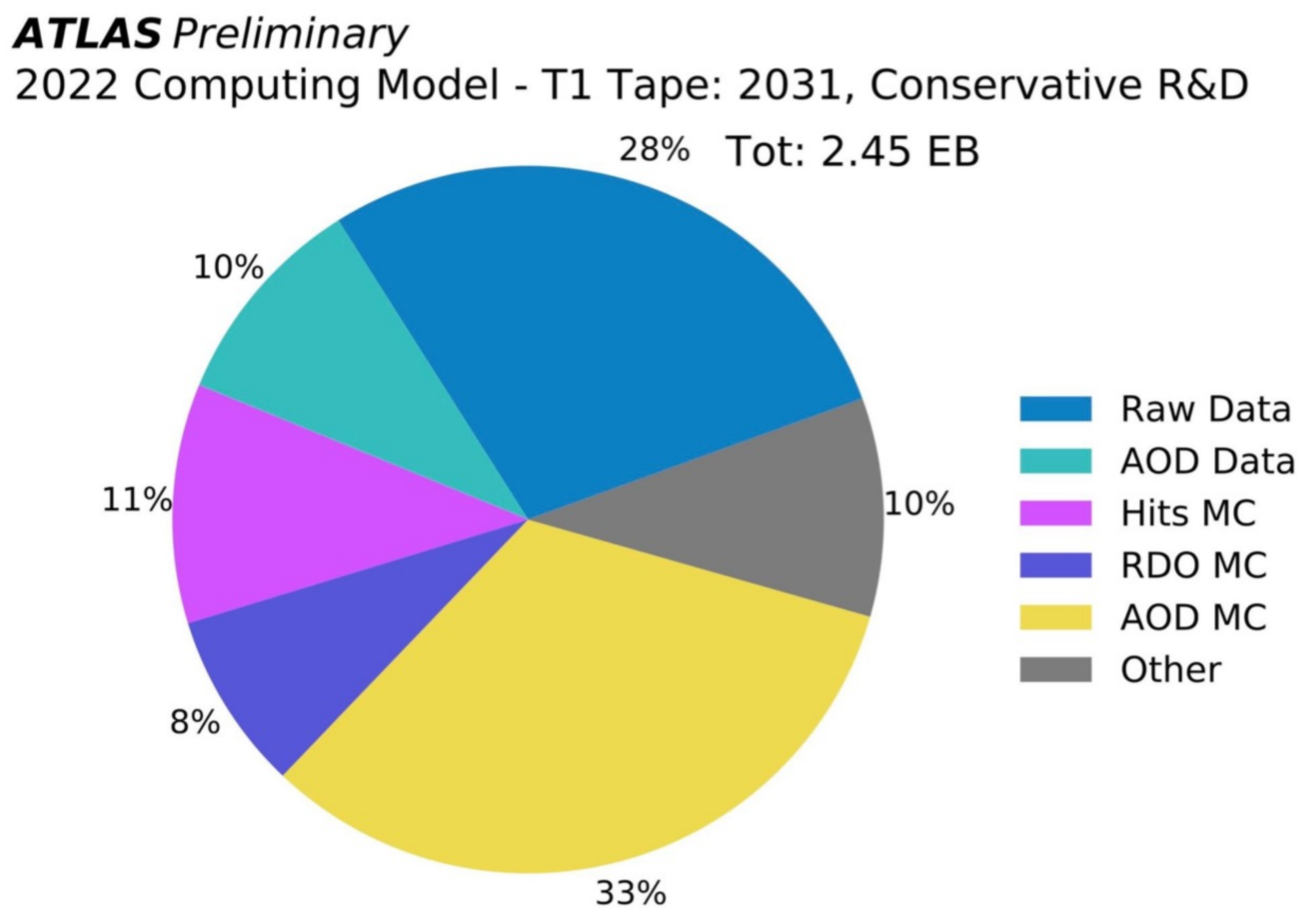


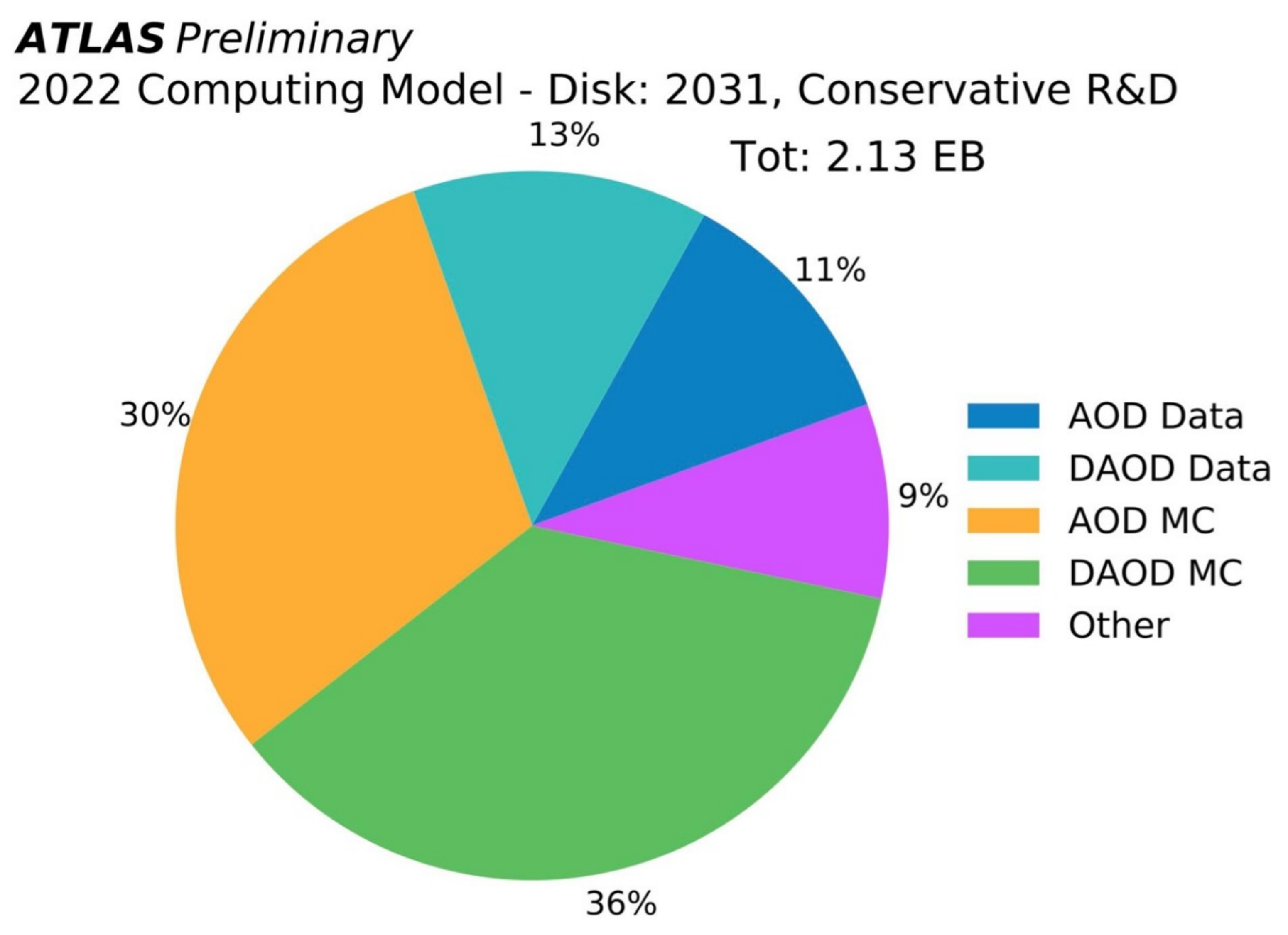


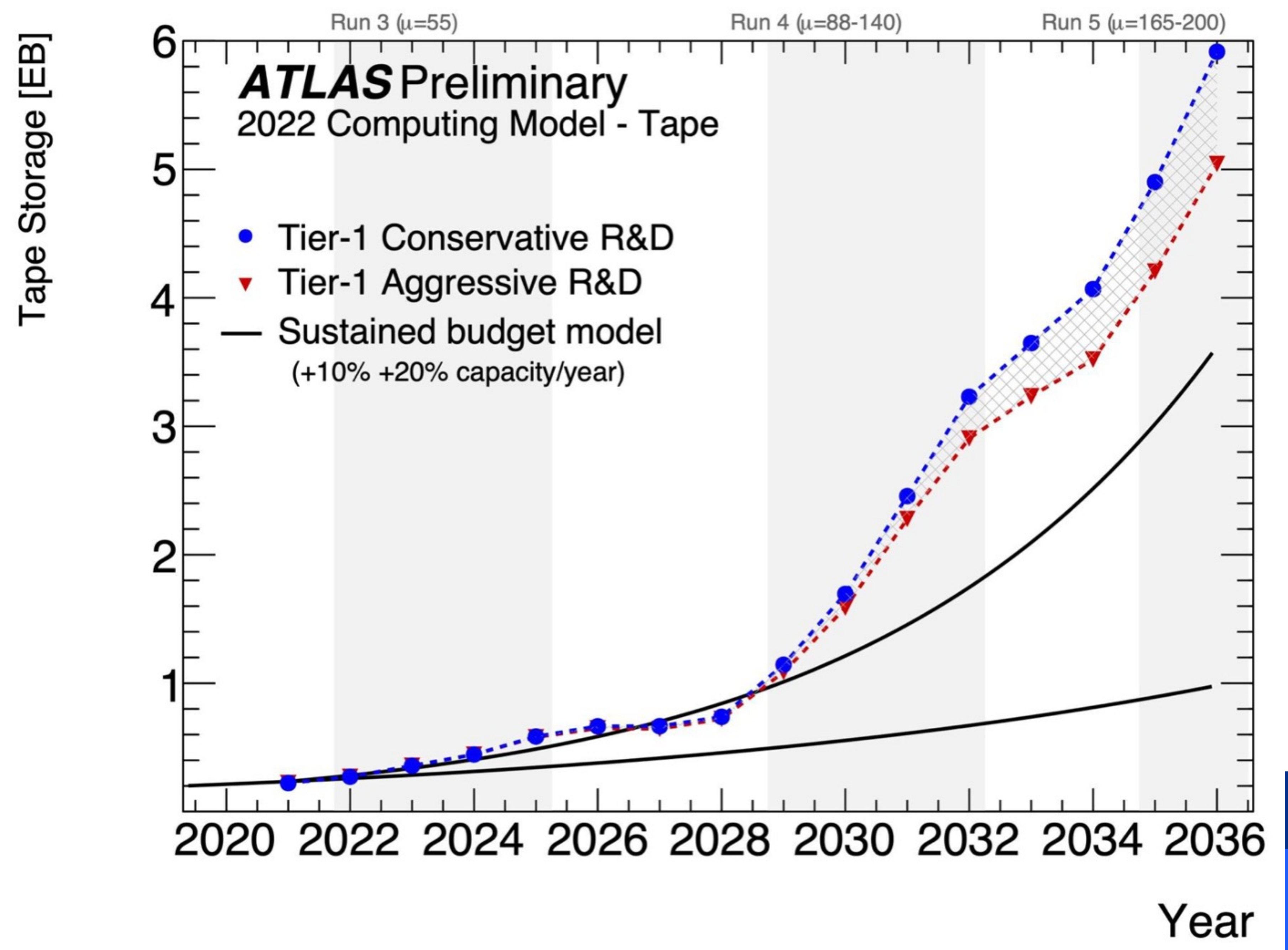
Storage

- CPU is not the only concern,
  - Disk space already at a premium,
  - Tape used for custodial data
    - and for less frequently used data needed in derivation steps
- Run-3 data taking - just starting;
  - New smaller data analysis formats for run-3, and prototypes for run-4



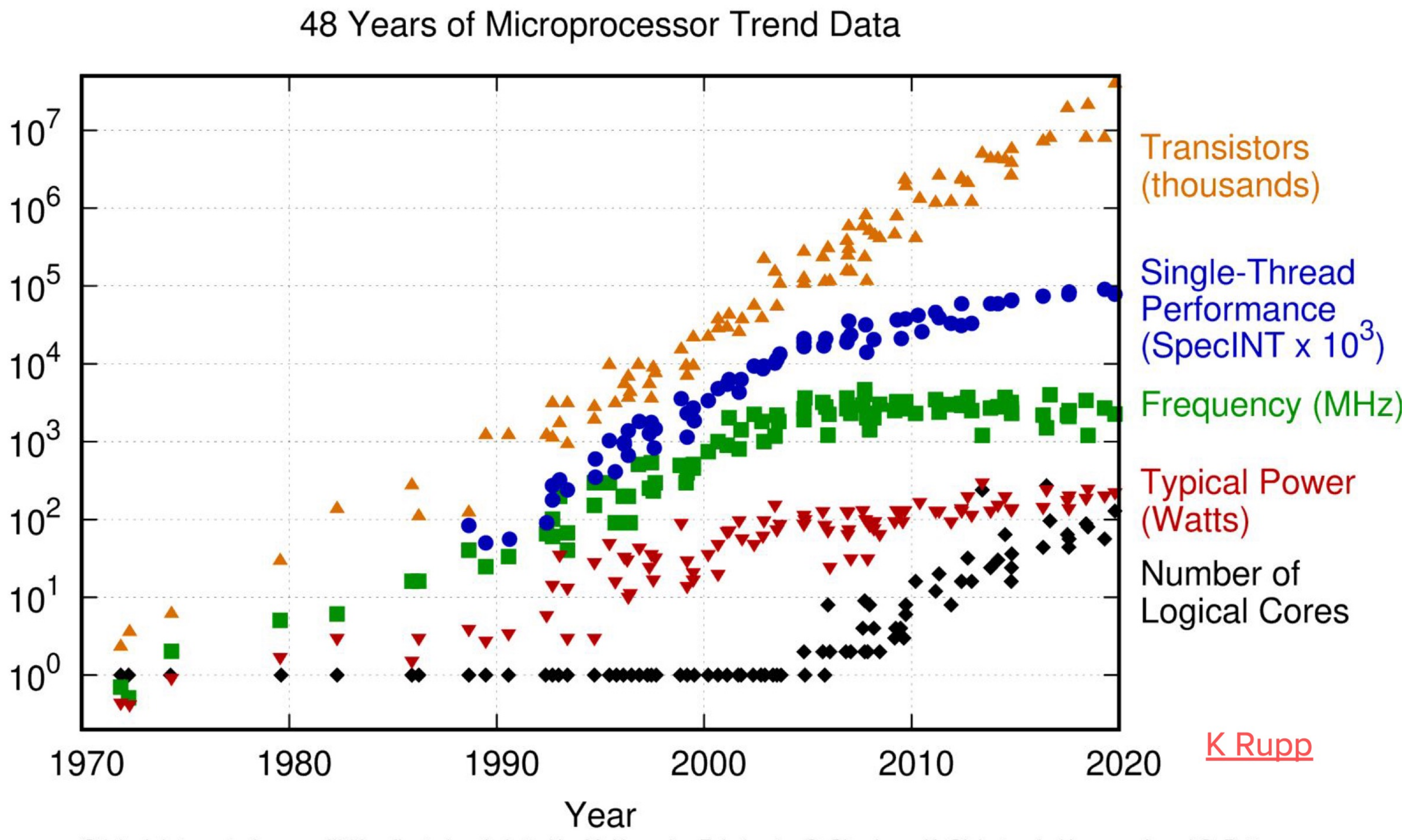






Utilising technologies

- As mentioned at start of the lecture - Clock speed scaling stopped following Moore's law around 2006
  - Limited serial processing options
- Transistor density does however appear to continue to follow the trend.
- Memory access can now take O(100)'s of clock cycles
- Facilities and capabilities / use cases shift towards GPU, FPGA, TPU processing:
  - Machine learning (and also towards differentiable programming)
  - Heterogeneity of systems
  - HPC sites vs more standard 'grid' architectures.
  - Analysis Facilities …

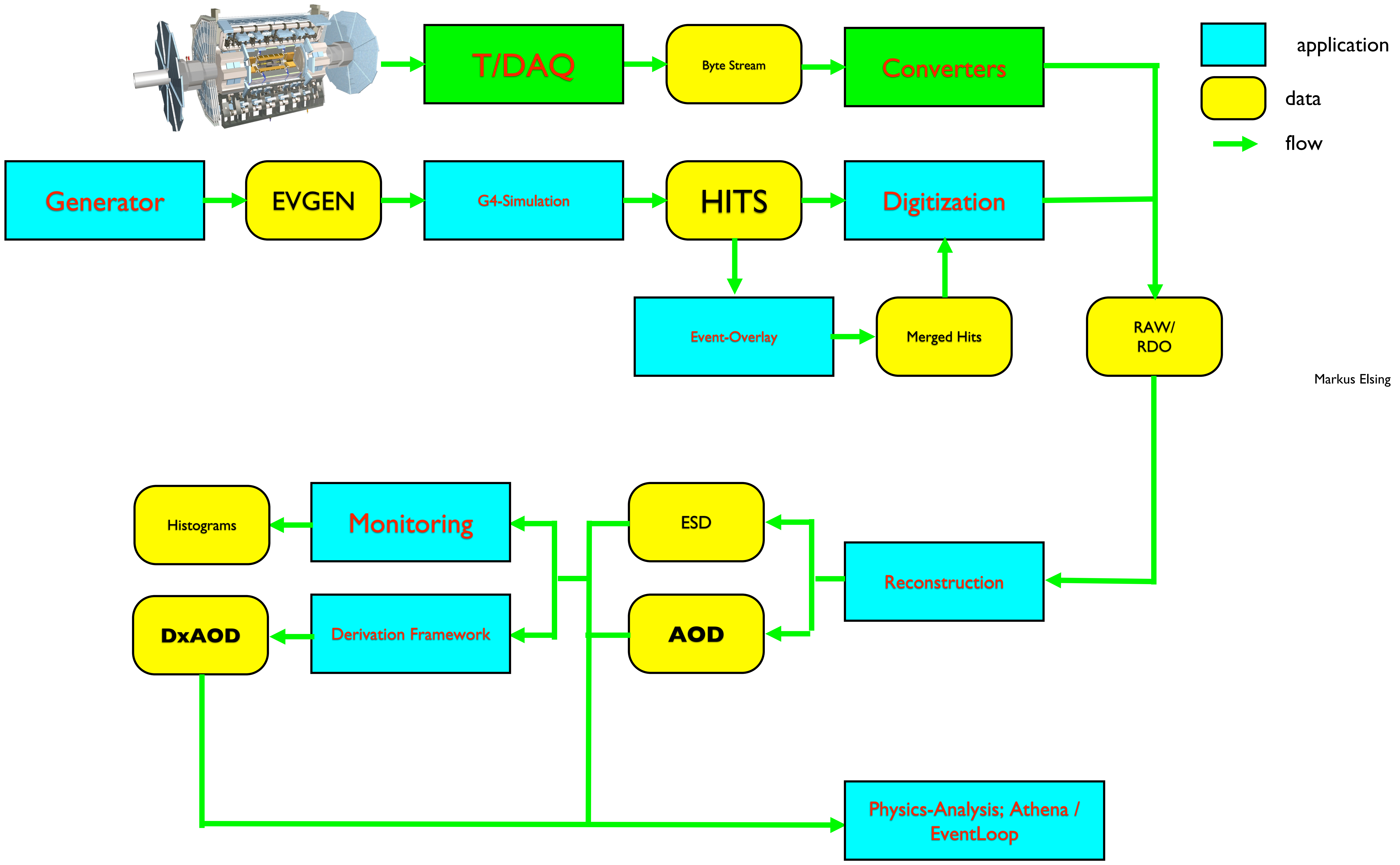


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Based on material from G. Stewart

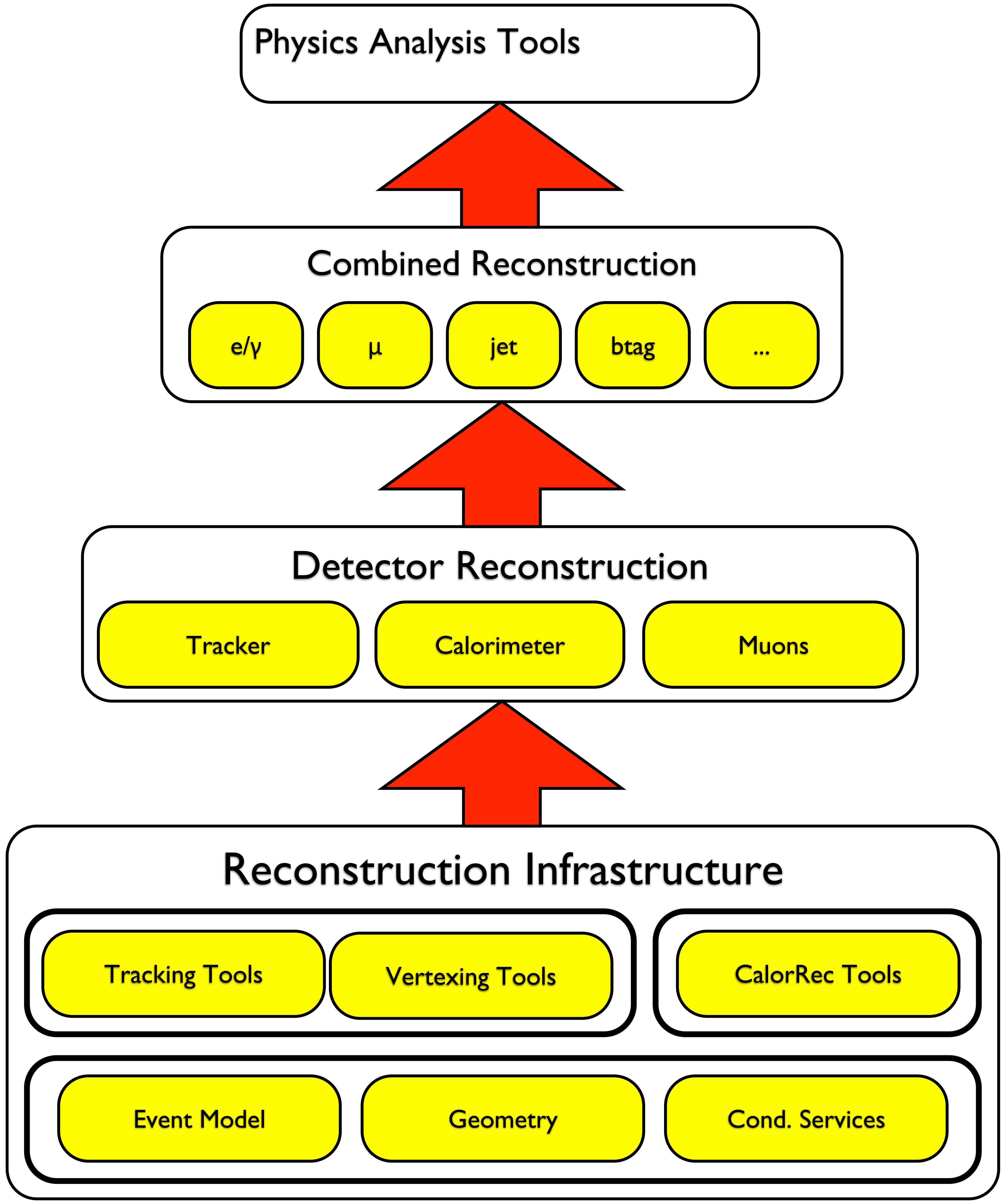
What do we need this code / data for?

- Simplified view of ATLAS processing chain for MC and data



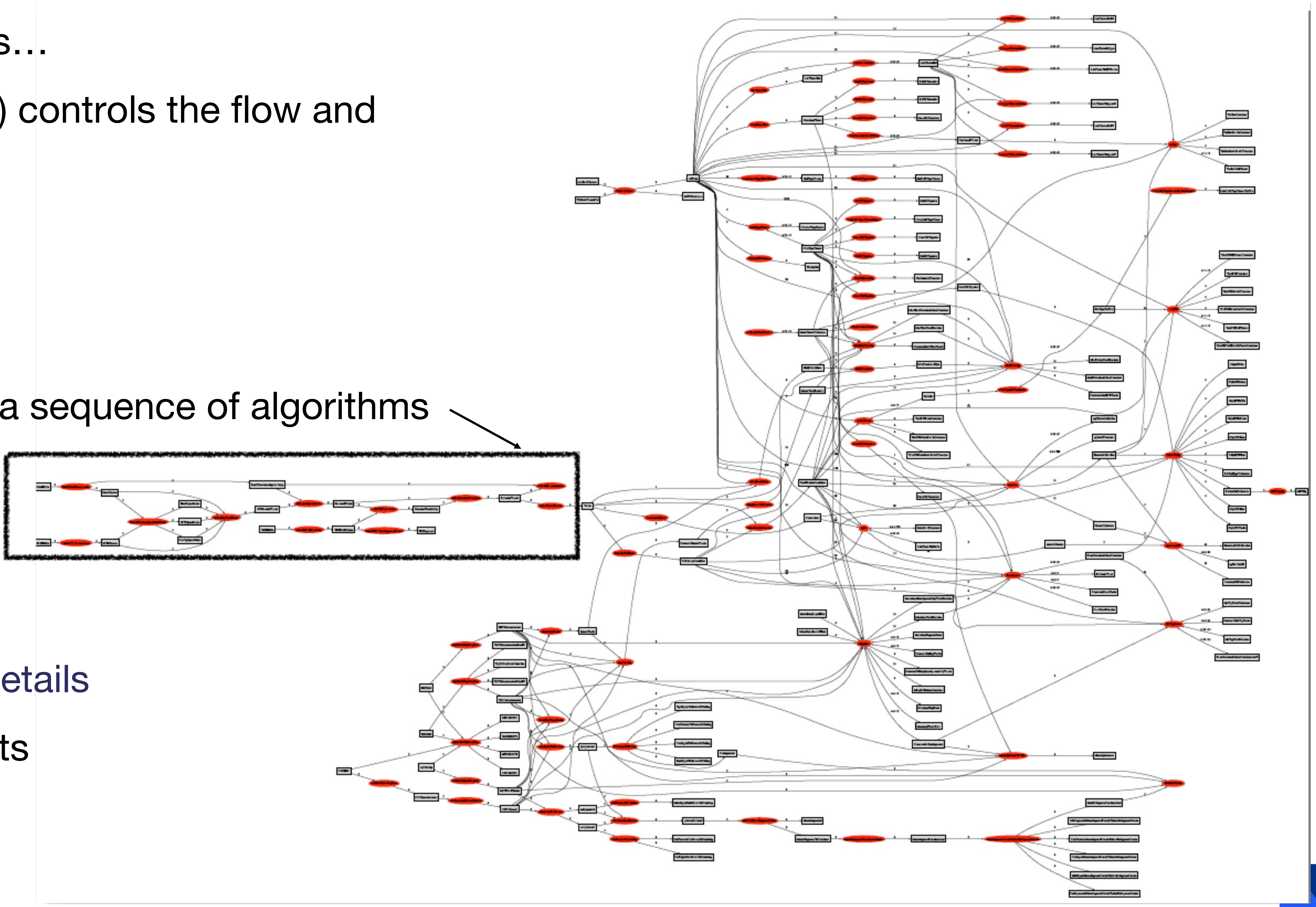
Reconstruction

- “natural” SW organization
    - first run detector specific reconstruction algorithms
        - for tracks and calorimeter clusters, ...
- then combined reconstruction
        - identify physics objects
- based on infrastructure
    - common Tools and Services
        - e.g. tracking and vertexing tools
        - fitting, propagation...
- used in all software layers above
- ~common code base with High Level Trigger
    - regional vs full event reconstruction



Reconstruction: Software flow

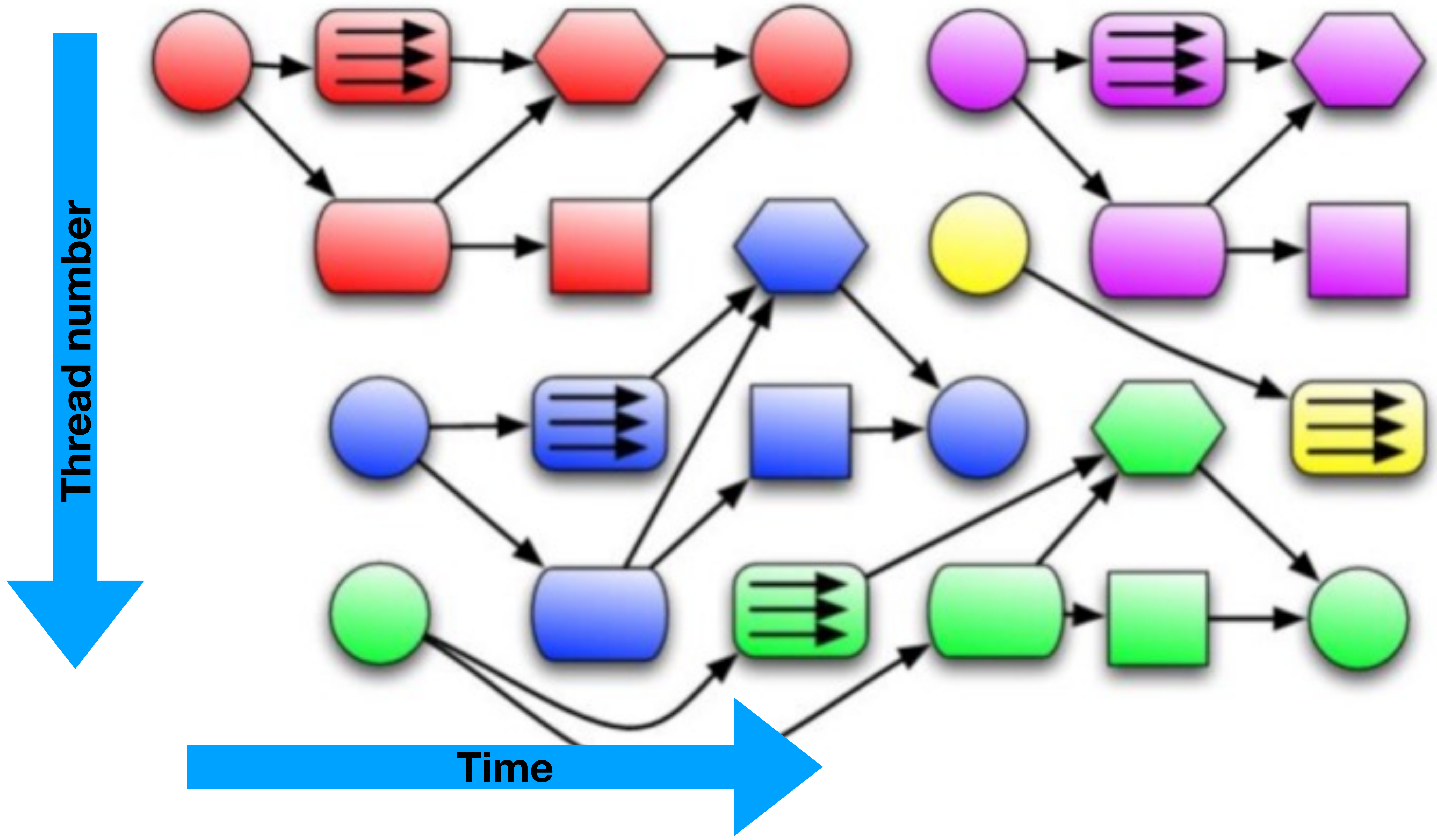
- chain of algorithms and data of full reconstruction
  - looks more complicated than it is…
  - Framework and steering (python) controls the flow and specific sets of algorithms
- can break it up in domains:
  - combined reconstruction
    - e.g. tracking broken down in a sequence of algorithms

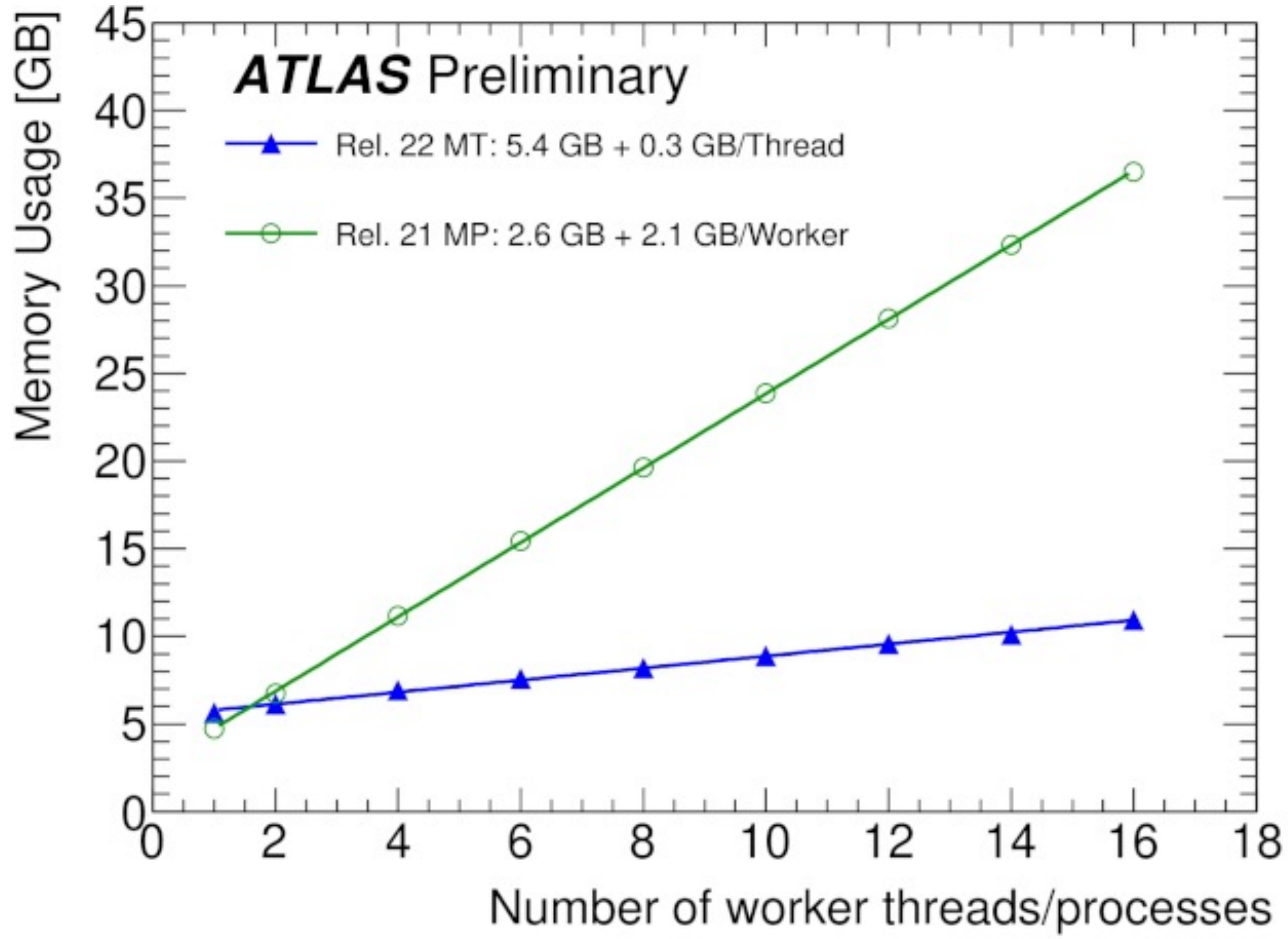


- one does not need to know all the details
  - usually work on a specific aspects

Improving framework software

- Previous framework diagram modelled from ATLAS Athena framework without multithreading available
  - athenaMT now in production. Schedular allocates algorithms from each event to run at correct moment once their corresponding inputs are available.
- Significant reduction of memory utilisations with very few compromises.

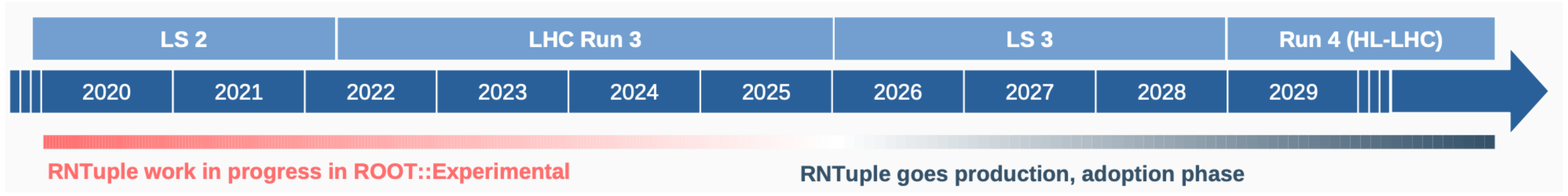




https://atlas.cern/updates/briefing/renovating-athena

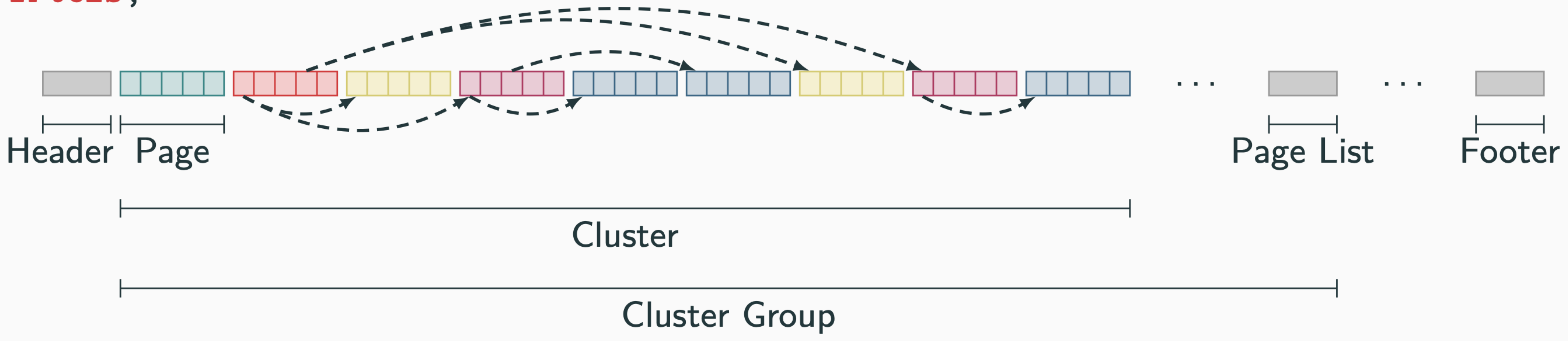
Data format evolutions

- ROOT TTree ~ 20 years of usage
  - RNtuple aiming as replacement by time of HL-LHC



- Estimations of ~ 3x faster, 10-20% smaller data size, together with better throughputs

```
struct Event {
    int fId;
    vector<Particle> fPtcls;
};
struct Particle {
    float fE;
    vector<int> fIds;
};
```



Cluster

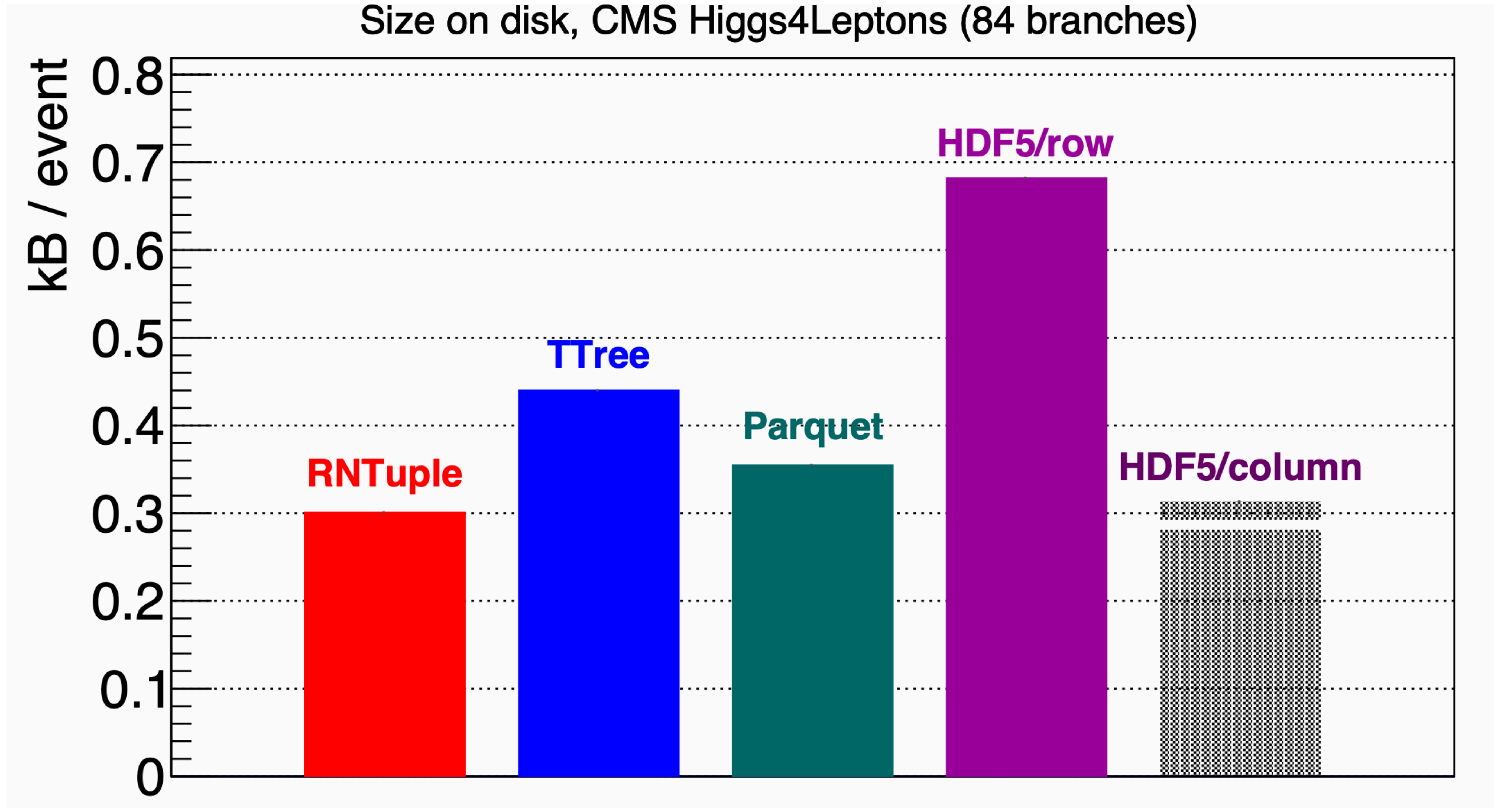
- Block of consecutive complete events
- Defaults to 50 MB compressed

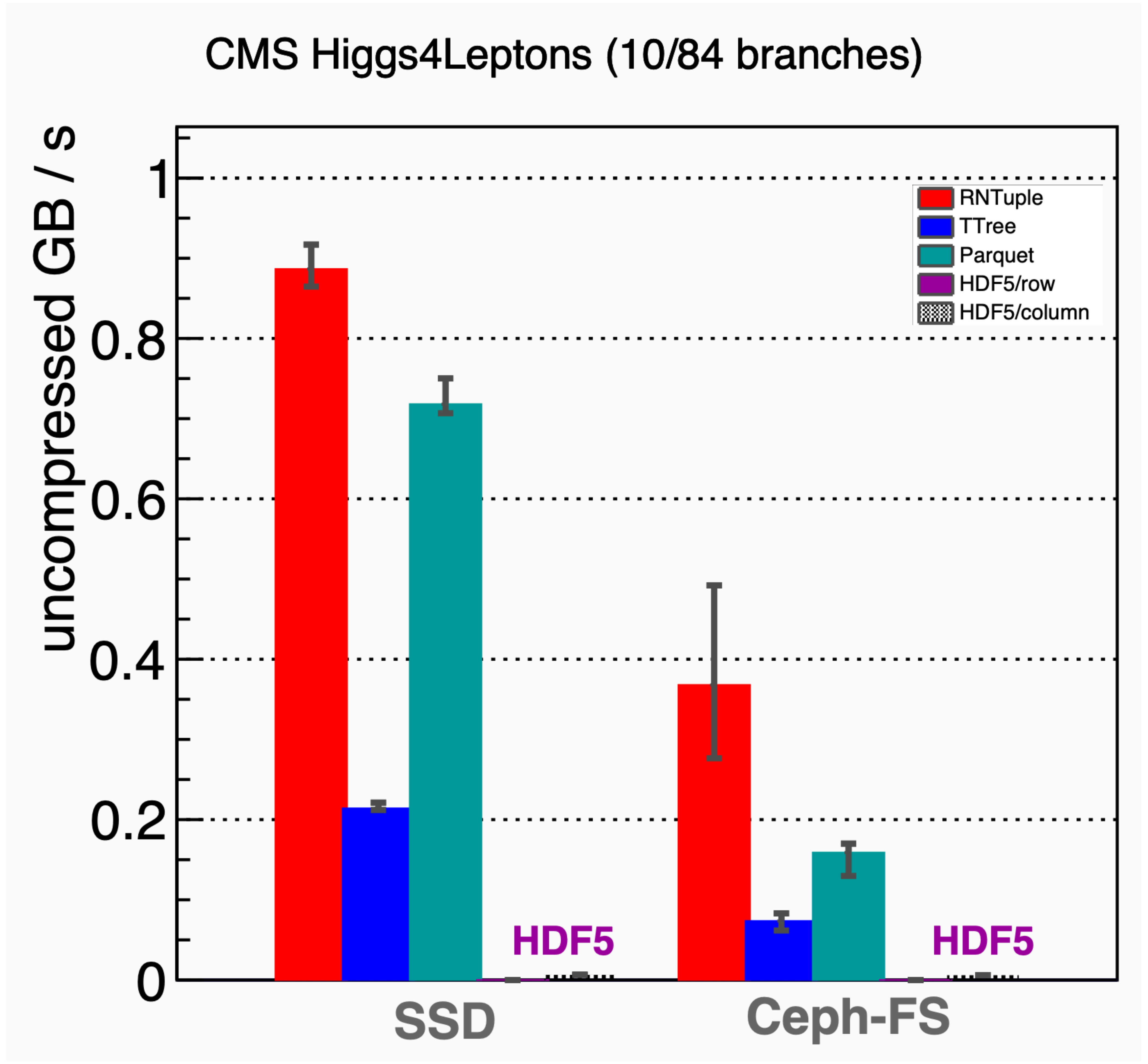
Page

- Unit of (de-)compression and (un-)packing
- Defaults to 64 kB uncompressed

ROOT2022-Workshop-Overview.pdf

RNTuple comparisons





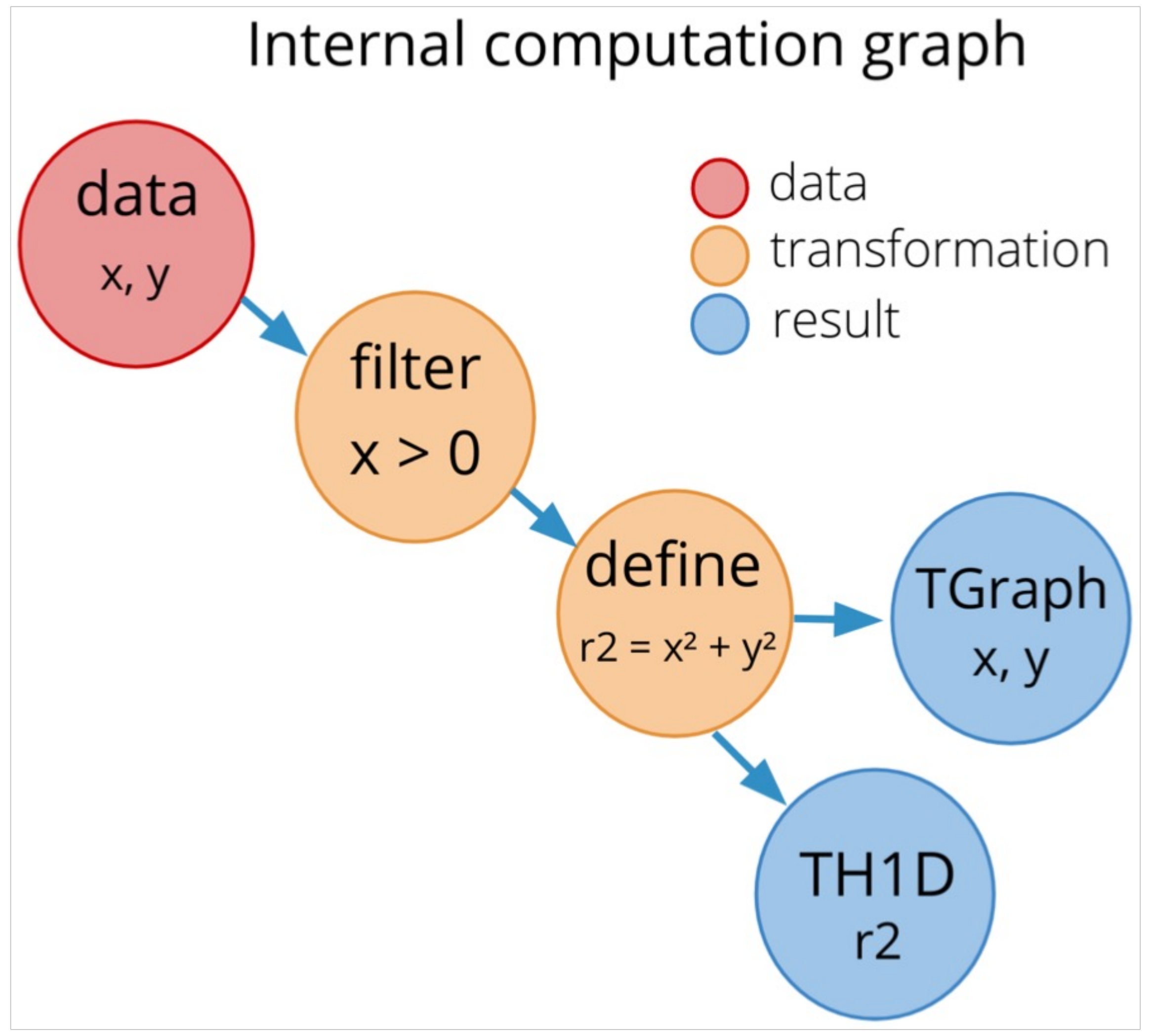
Declarative analysis

- Traditional analysis typically follows event-loop style processing;
  - Loop over each event;
  - Read in variables of interest
  - Make selections;
  - Augment data (e.g. new variables)
  - Fill histograms / TTrees
  - Save output for final analysis / statistical interpretations
- RDataFrame in ROOT allows to state what you want to happen, and worry less about how to make it happen.
- Multithreaded support
- Multiple file formats and backend capabilities
- Computational graph structure ensures data is only read as required
  - c.f. TTree::Draw() method for constructing several histograms.
- Latest versions of ROOT also to support including systematic variations within the computational graph structure of RDataFrame.

```
from ROOT import RDataFrame
df  = RDataFrame(dataset);
df2 = df.Filter("x > 0")
        .Define("r2", "x*x + y*y");

rHist = df2.Histo1D("r2");

g = df2.Graph("x","y")
```



EPJ Web of Conferences 245, 03009 (2020)
https://doi.org/10.1051/epjconf/202024503009

Summary

- Computing and data are fundamental to HEP
    - Current and future requirements continue to push needed development improvements
    - Mulit-threading / processing allows performance scaling beyond clock speed single-core improvements
        - Heterogeneous environments and resources
    - Machine learning increasingly used.

- Use tests to help:
    - Ensure correctness
    - Stop bugs reoccurring
    - Stop unwanted side effects
    - Help aid in the code design (e.g. with test-driven development).

- Use ‘Rubber duck’ philosophy to work through problems
- Profile your code and identify bottlenecks when performance is important:

Hands-on

Profiling in python

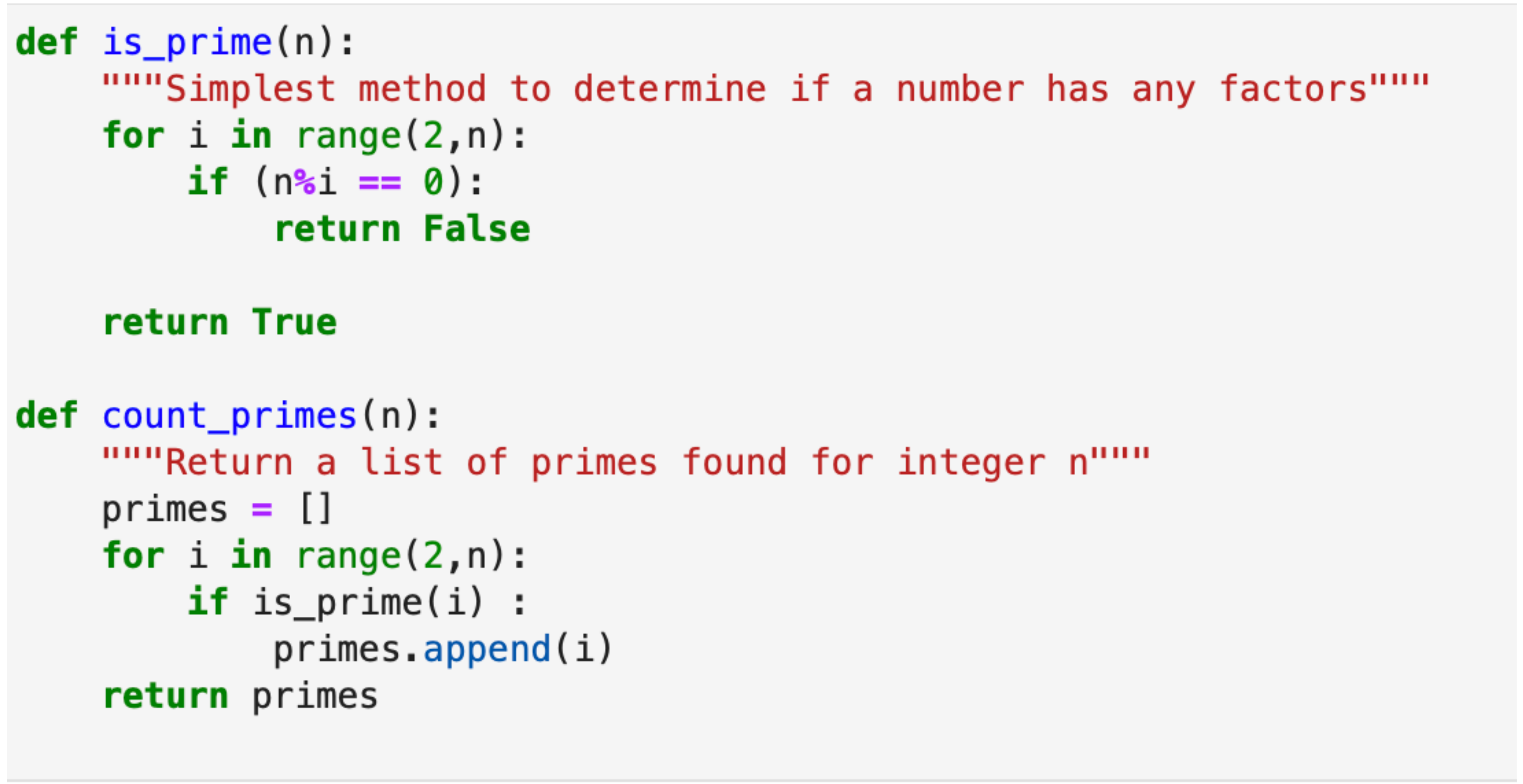
- https://github.com/snafus/OptimisationWorkshop.git
- Requires python 3 and Jupiter notebook
  - (Best to run with, e.g. anaconda and/or virtual env)
- Simplest way to run via mybinder:
  - mybinder.org link (or just http://cern.ch/go/p7tT)

- Take a simple algorithm to find the number of primes up to some number N.

- Use the ‘play button’ or command-enter (alt-enter) to run each cell.

▶ ■ ↻ ⏩

- Work through the cells and try to measure the algorithm’s performance and even see if you can improve it.



```python
def is_prime(n):
    """Simplest method to determine if a number has any factors"""
    for i in range(2,n):
        if (n%i == 0):
            return False

    return True

def count_primes(n):
    """Return a list of primes found for integer n"""
    primes = []
    for i in range(2,n):
        if is_prime(i) :
            primes.append(i)
    return primes
```

Profiling in python

- python -m cProfile Exercise_1.py
- 109596 function calls in 22.317 seconds

- Ordered by: standard name

- ncalls tottime percall cumtime percall filename:lineno(function)

C++ profiling

- Compare Exercise_1.cpp timing to Exercise_1.py
- Profile Exercise_2 and Exercise_3
    - Where can you optimize?