Improving code performance: an introduction

Practical examples from particle physics simulation

Sofia Vallecorsa

sofia.vallecorsa@cern.ch

2017 Openlab Summer Students lecture series

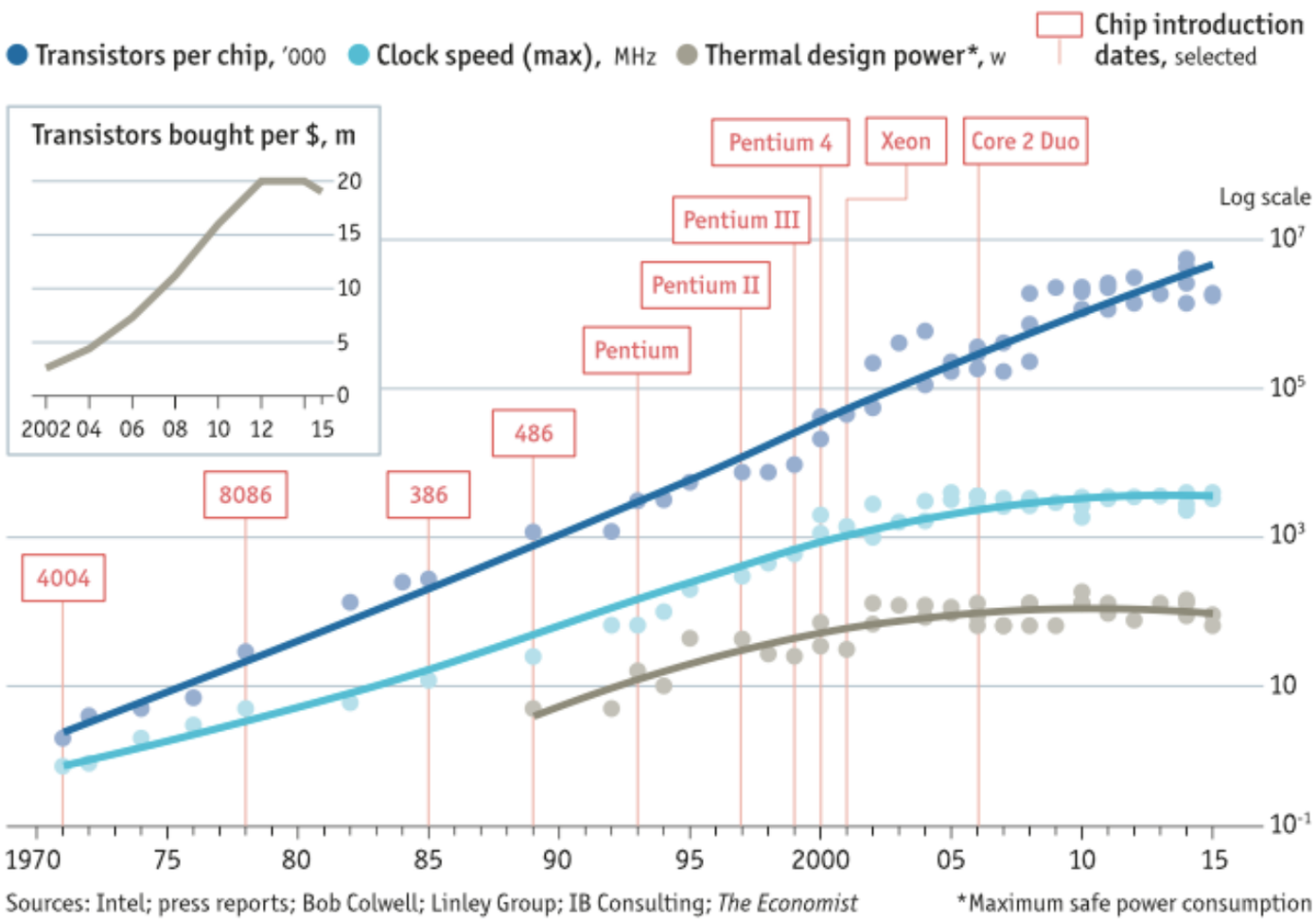
Outline

- Introduction
  - Why performance is important?
- Performance
  - Can we define it?
  - How do we measure it?
- Improving performance
- Use case: Simulating particle interactions through matter
  - Current status: Geant4 performance
  - The GeantV prototype
- The end: Summary & Conclusions

Moore’s law and power wall

- In 1965 G. Moore noted that the number of electronic components which could be crammed into an integrated circuit was doubling every year.

- Moore’s law is not a “Law”, it’s more of a self-fulfilling prophecy..



Number of transistors per chip is going up
The clock speed is not
The amount of energy dissipated per chip is the limiting factor (power wall)

Why do we care?

Bottom line…
Massive data processing, modelling, simulation from fundamental research and beyond!

For years we have relied on the increase of clock speed to simply see our code running faster on more performant hardware.. it’s over!

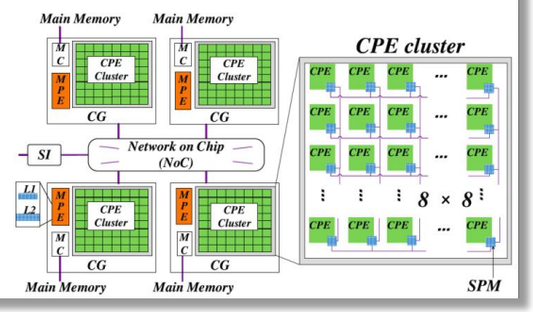
Technology is moving fast: ever faster networks, performant storage, distributed systems, multi-processor architectures, supercomputers, Cloud, Grid and heterogeneous architectures

An example

Need to think parallel!

SUNWAY TAIHULIGHT

- SW26010 processor (Chinese design, ISA, & fab)
- 1.45 GHz
- Node = 260 Cores (1 socket)
  - 4 – core groups
  - 32 GB memory
- 40,960 nodes in the system
- 10,649,600 cores total
- 1.31 PB of primary memory (DDR3).
- 125.4 Pflop/s theoretical peak
- 93 Pflop/s HPL, 74% peak
- 15.3 Mwatts water cooled
- 3 of the 6 finalists for Gordon Bell Award@SC16



High Performance Computing

- > x500,000 increase in supercomputer performance in past 20 years
- The race is already on for Exascale computing!

  ExaFLOP = 10^{18} calculations per second

Top500 @ISC2017

[top500.org](top500.org)

#	Site	Manufacturer	Computer	Country	Cores	Rmax [Pflops]	Power [MW]
1	National Supercomputing Center in Wuxi	NRCPC	Sunway TaihuLight NRCPC Sunway SW26010, 260C 1.45GHz	China	10,649,600	93.0	15.4
2	National University of Defense Technology	NUDT	Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi	China	3,120,000	33.9	17.8
3	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC50, Xeon E5 12C 2.6GHz, Aries, NVIDIA Tesla P100	Switzerland	361,760	19.6	2.27

Performance

- Timing: faster execution
  - CPU time, latency,…
  - Speedup (parallel vs serial execution)
- Throughput: Amount of processed data
- Size: smaller executable, smaller memory footprint
- Scaling: x2 in number of cores (or vector size) doubles performance

Is there A definition?

…and of course ... forward scalability:

- Maximum performance today should scale on future hardware "automatically"

Improving performance is a tradeoff!!

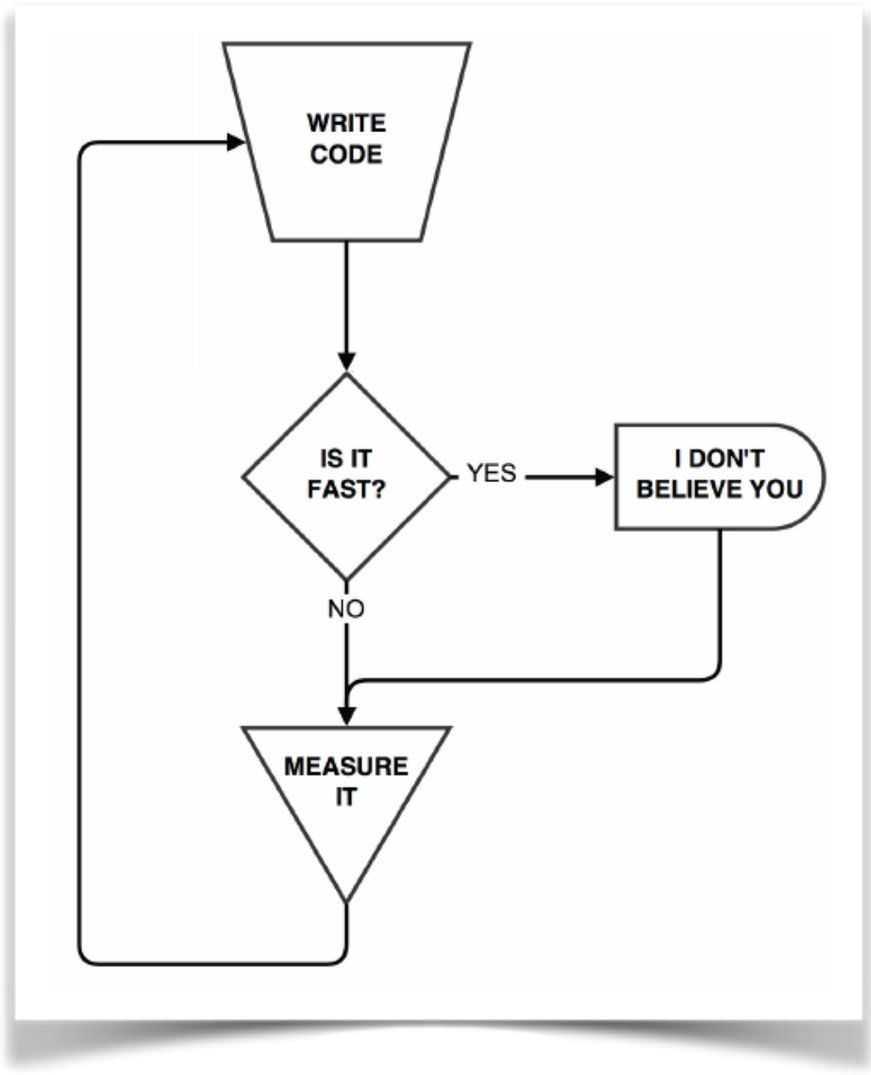
- Timing vs. Size
- Compilation speed and memory
- Latency vs throughput

Measuring performance (I)

“First catch the rabbit”

a recipe for rabbit stew

- Before any optimisation, need a way to measure what to optimise
- Before any measurement, need an explicit statement of the problem to solve.



- ➥ A good understanding of the hardware
- ➥ Reproducible, representative benchmarks
- ➥ “The right” tool
- ➥ Time (!): Performance optimisation is an iterative process

Measuring performance (II)

Identify hotspots:

Focus on hotspots and ignore sections that account for little CPU usage.

Identify bottlenecks:

Disproportionately slow parts

Places where parallelizable work halts or is deferred (e.g. I/O)

Change algorithm to reduce unnecessary slow areas

Majority of scientific and technical programs accomplish most of their work in a few places!

Profiling techniques

Statistical Sampling:

Program flow is periodically interrupted, to examine state

- Asynchronous sampling:
  - Timers
  - Hardware counters (CPU cycles, L3 cache misses, etc.)
- Synchronous sampling:
  - Calls to certain library functions are intercepted (malloc, fread, …)

Code Instrumentation:

- Code for collecting information is inserted into original program
- Approaches:
  - Manual (measurement APIs)
  - Automatic source level
  - Compiler assisted (e.g. gprof)
  - Binary translation
  - Runtime instrumentation

Pros & Cons

Statistical sampling

Advantages:

- No changes to program or build process
  - Recommended: Debugging symbols
- No blind spots:
  - Library functions
  - Functions with unavailable source code
- Low overhead (typically 3-5%)

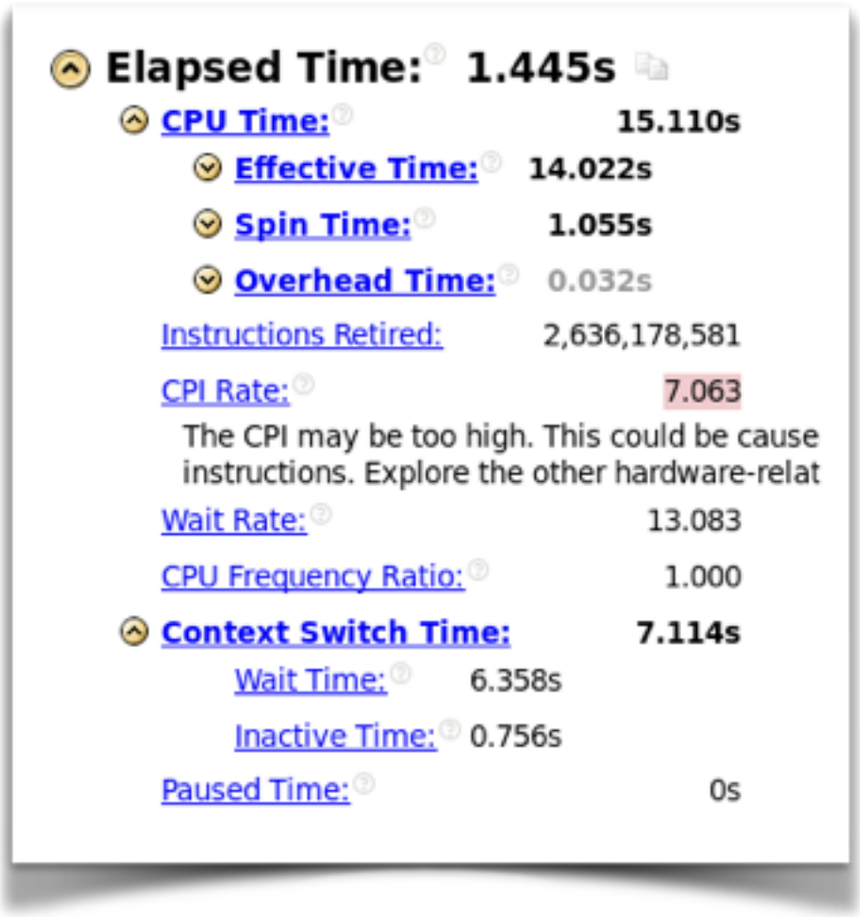
Limitations:

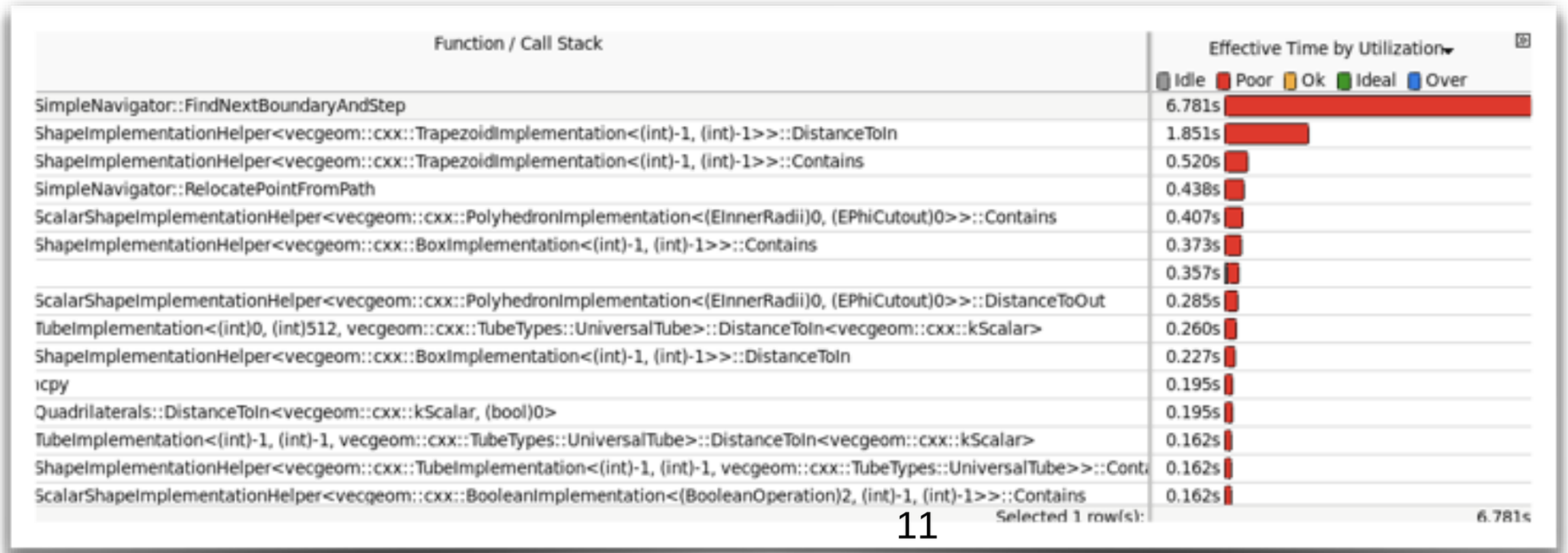
- Some degree of uncertainty
  - Information attributed to source lines may not be accurate
- No access to some information:
  - Number of calls of a certain function
  - Average runtime per call of a certain function

Reverse everything for Code instrumentation

Some profiling tools

- VTune, Advisor – Intel products, powerful, multi-threading analysis and vectorisation
- gprof: GNU, Flat profiles, call lists, Recompilation needed
- PIN, Valgrind: Instrumentation / Synthetic software CPU: cache misses and branch mispredictions, memory space usage, function call relationships
- perfmon2: Low level access to counters, No recompilation needed





Examples from Intel VTune

Multi-dimensional improvement

- Multiple computing nodes
- Multi-socket
- Multi-core
- Multi-threading
- Instruction Level Parallelism
  - Instruction pipelining
- Vector registers

Task/Process parallelism:
- split load into “baskets of work” through a pool of resources
- Check inter-dependency

Data parallelism:
- same transformation to multiple pieces of data
- wise design of data structures

Multi-dimensional improvement

- Multiple computing nodes
- Multi-socket
- Multi-core
- Multi-threading
- Instruction Level Parallelism
  - Instruction pipelining
- Vector registers

Which direction?

Coming up next…

- Introduction to concurrency
- Suggestions to design parallel code
- Vectorisation
- Compiler optimisation and auto-vectorisation

Introducing concurrency

Processes-threads-tasks

- Process: isolated instance of a program, with its own space in memory
  - Can have multiple threads
  - Easy to manage
  - Communication/switching between them possible but pricey

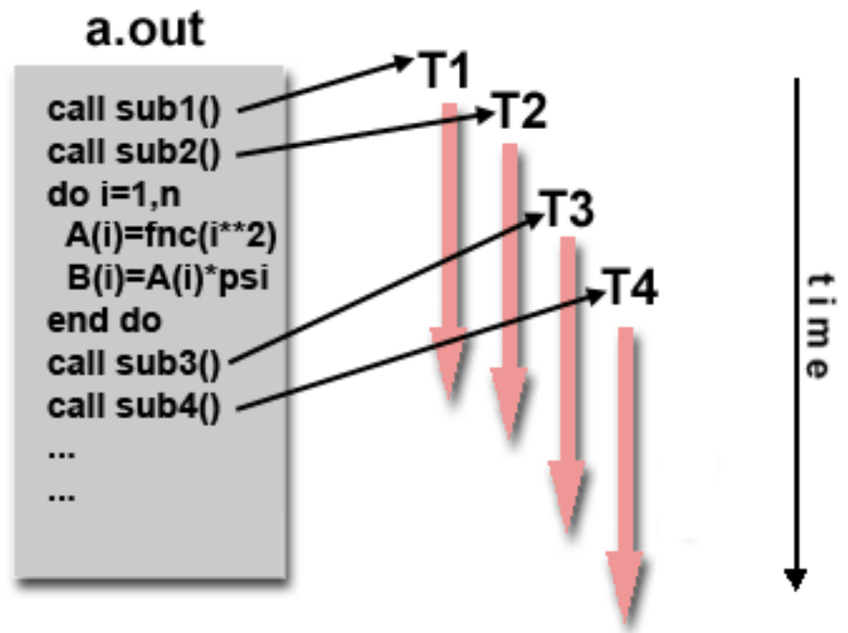
- Thread: light-weight process within process
  - share memory with other threads belonging to same process
  - Managed and scheduled by the kernel according to available resources
  - Many options available:
    - C++11 std::thread
    - OS: pthreads (linux)..
    - Libraries: OpenMP …

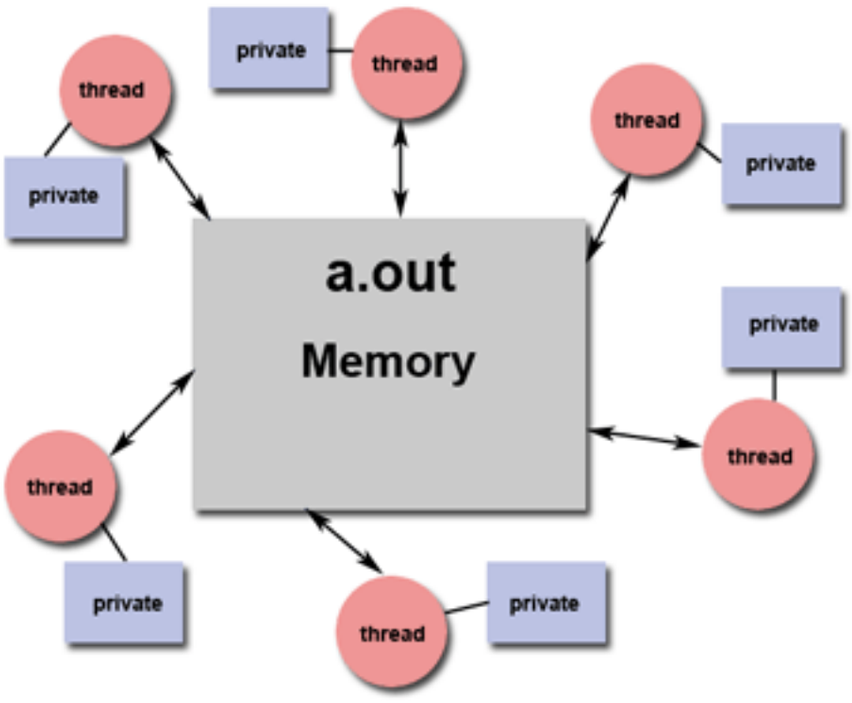
- Task: Logically discrete section of computational work. Typically a program-like set of instructions executed by a processor.

and what about memory!?!

Shared memory (thread) model

- Main program loads necessary system and user resources
  - Performs serial work and creates threads, scheduled and run by OS
- Threads have local data and share common resources
  - Communicate by updating global memory address locations
  - Synchronisation ensures that two threads do not update same global address

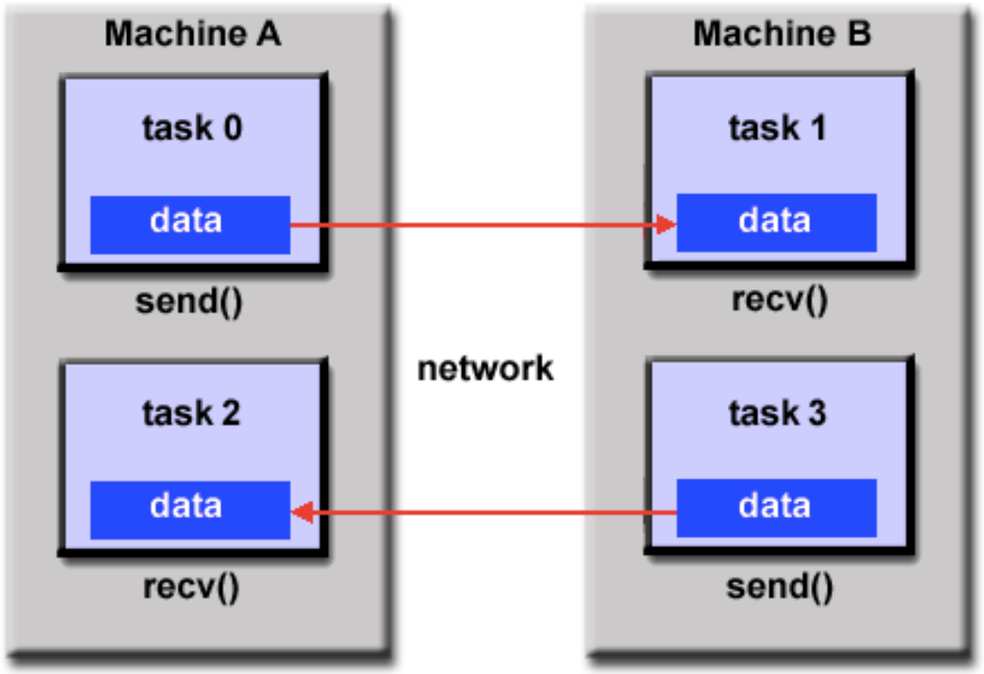




Distributed/Hybrid memory models

Distributed memory: Tasks use own local memory

- Exchange data by sending and receiving messages
- Typically use libraries e.g. Message Passing Interface (MPI)



Hybrid memory: combines more than one programming model e.g: MPI + OpenMP

- Threads perform computationally intensive kernels using local, on-node data
- Communication between processes on different nodes occurs over the network using MPI

Underlying hardware network speed & bandwidth do matter!

Designing parallel code

- Understand the problem: can it actually be parallelised?
  - Identify inhibitors to parallelism (e.g. data dependence)
  - Change the algorithm, check external libraries
- Partition: break the problem in discrete chunks
- Communication: what is needed? (e.g. visibility and scope, synchronous or asynchronous…)
  - Consider cost in terms of overhead, latency and bandwidth

Loop carried dependency:

```
DO 500 J = MYSTART,MYEND
   A(J) = A(J-1) * 2.0
500 CONTINUE
```

Loop independent dependency:

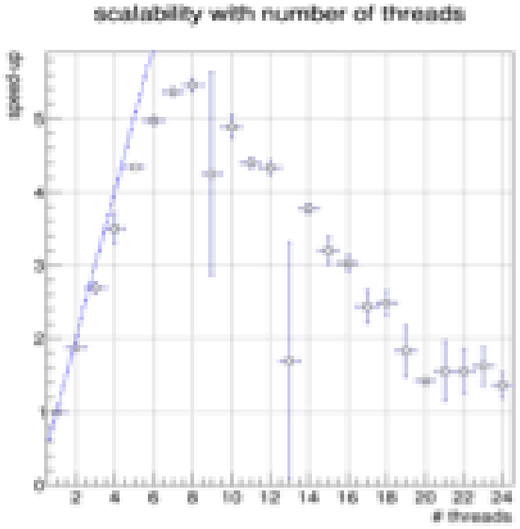
```
task 1          task 2
------          ------

X = 2           X = 4
  .               .
  .               .
Y = X**2        Y = X**3
```

Synchronisation

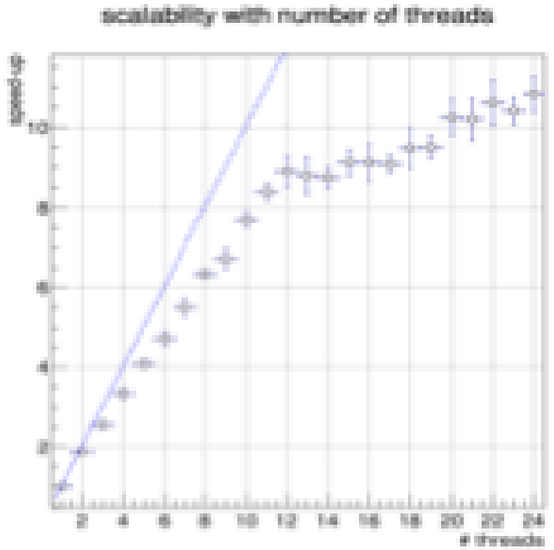
Managing the sequence of work is critical!

- Barriers: Each task works until the barrier, then stops.
  - Synchro when last task reaches the barrier.
- Locks and semaphores: protect access to global data or a code section.
  - One task at a time may own it
  - The first task to acquire the lock "sets" it. Others wait until the owner releases the lock
- Load balancing/granularity

Algorithm using spin-lock



Lock-free algorithm (memory polling)



2x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz

Best solution?

traffic deadlock in Tel Aviv, 2011

There is no silver bullet!

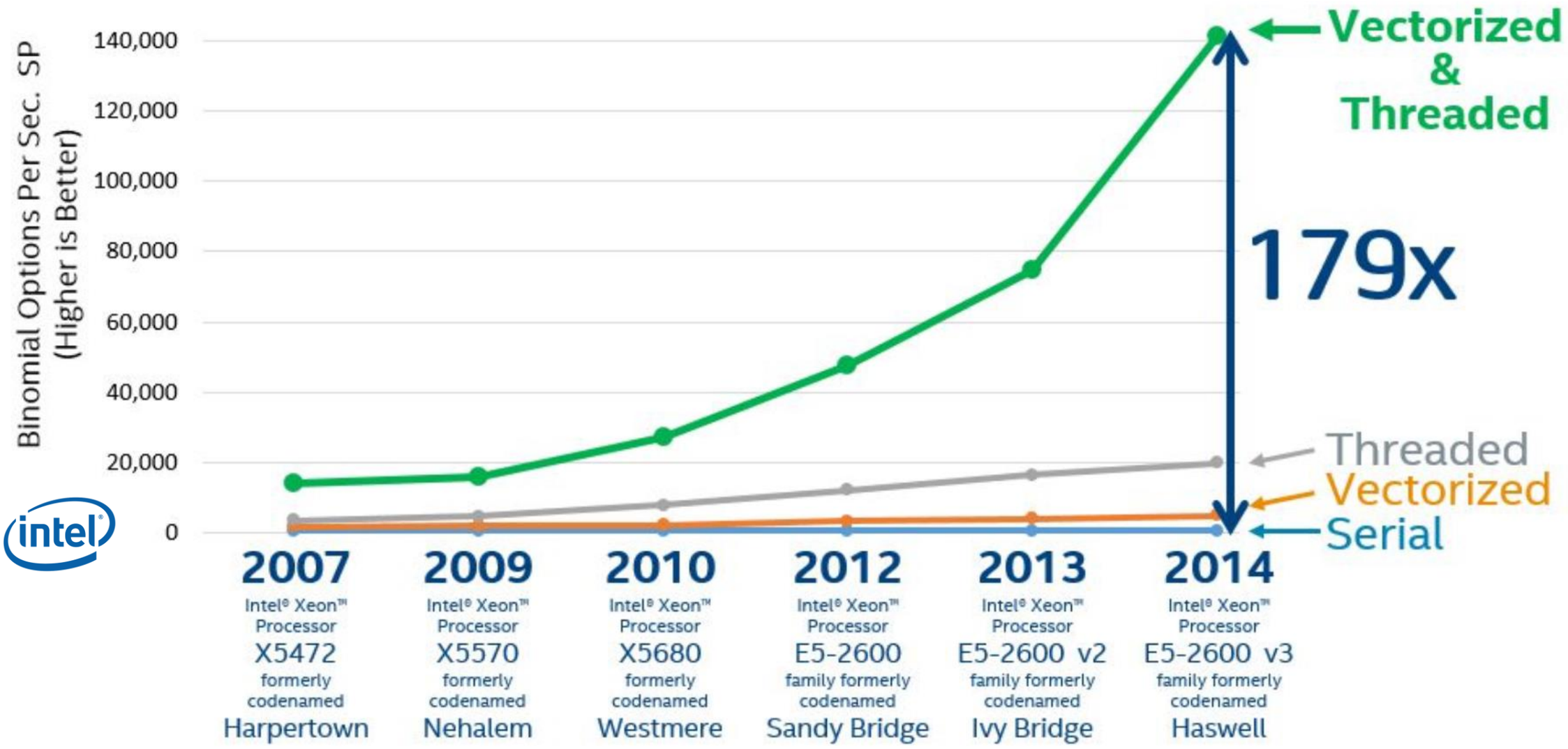
- Case by case investigation needed
- Best solution: often a trade-off

Coming up next…

- Introduction to concurrency
- Suggestions to design parallel code
- Vectorisation
- Compiler optimisation and auto-vectorisation

Vectorisation: Why?

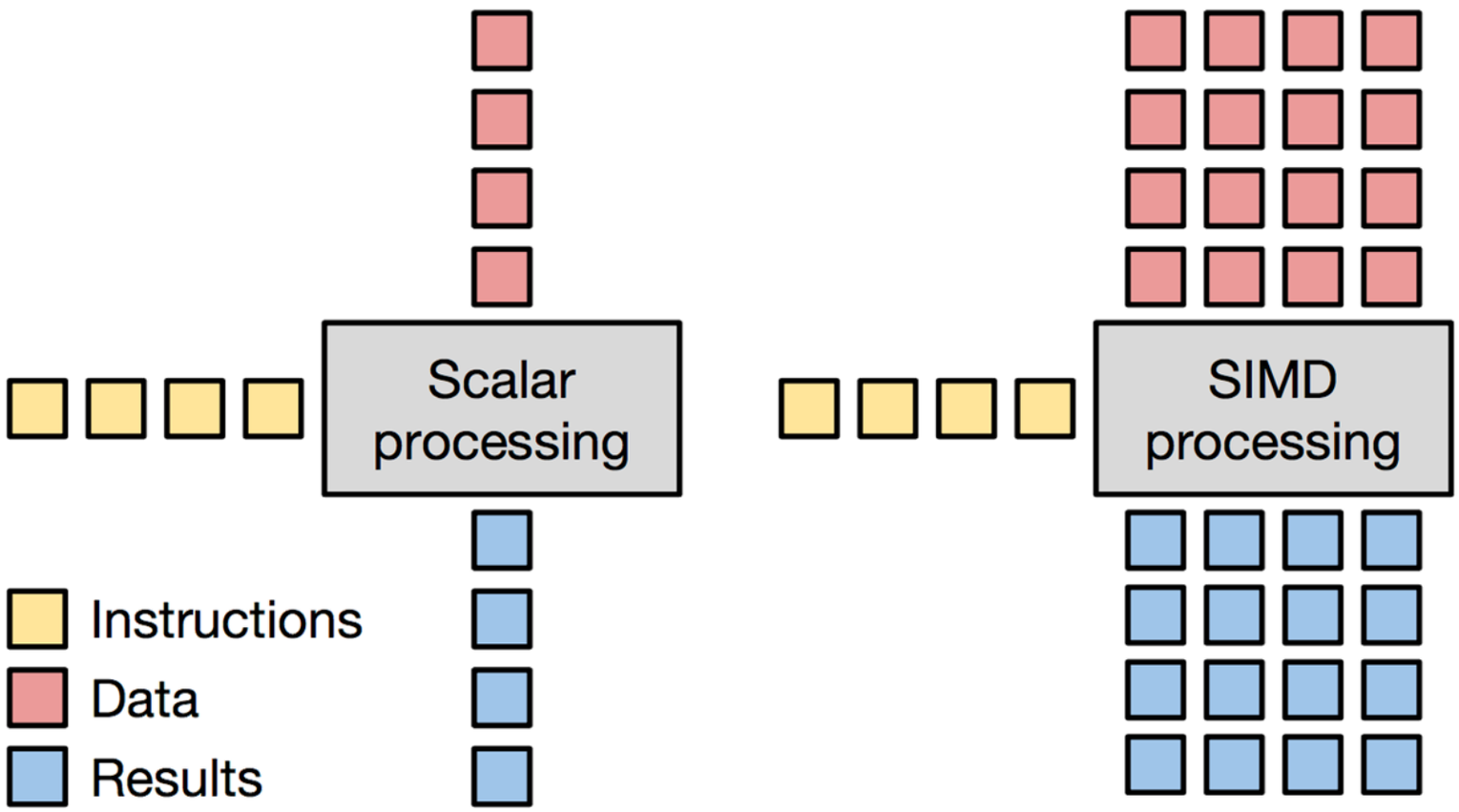
Vectorised data is a prerequisite to make efficient use of modern hardware

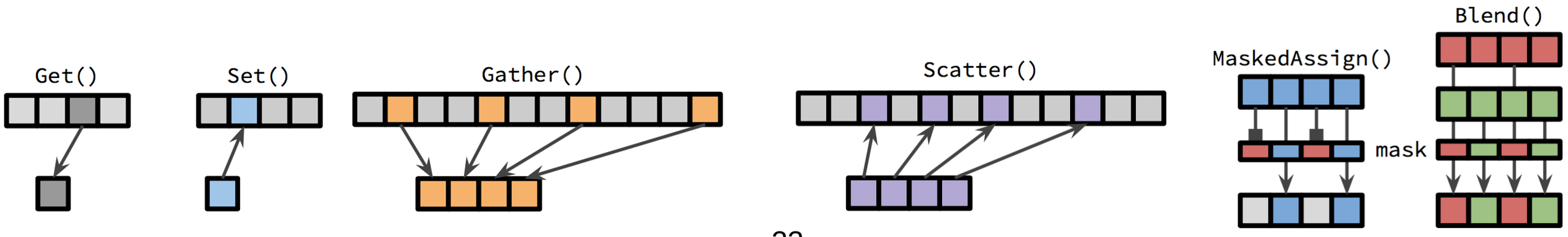


source: Intel

Single Instruction Multiple Data

SIMD





Vectorisation: some history

Year	Register	Corresponding Instruction set
~1997	80 bit	MMX
~1999	128bit	SSE1
~2001	128 bit	SSE2
…	128 bit	SSEx
2008	128 bit	AVX
~2010-2011	256 bit	AVX2
2013	512 bit	IMCI
2015	512 bit	AVX512

P5 Pentium

Pentium III

Pentium IV

Pentium - Nehalem core i7

Sandy Bridge

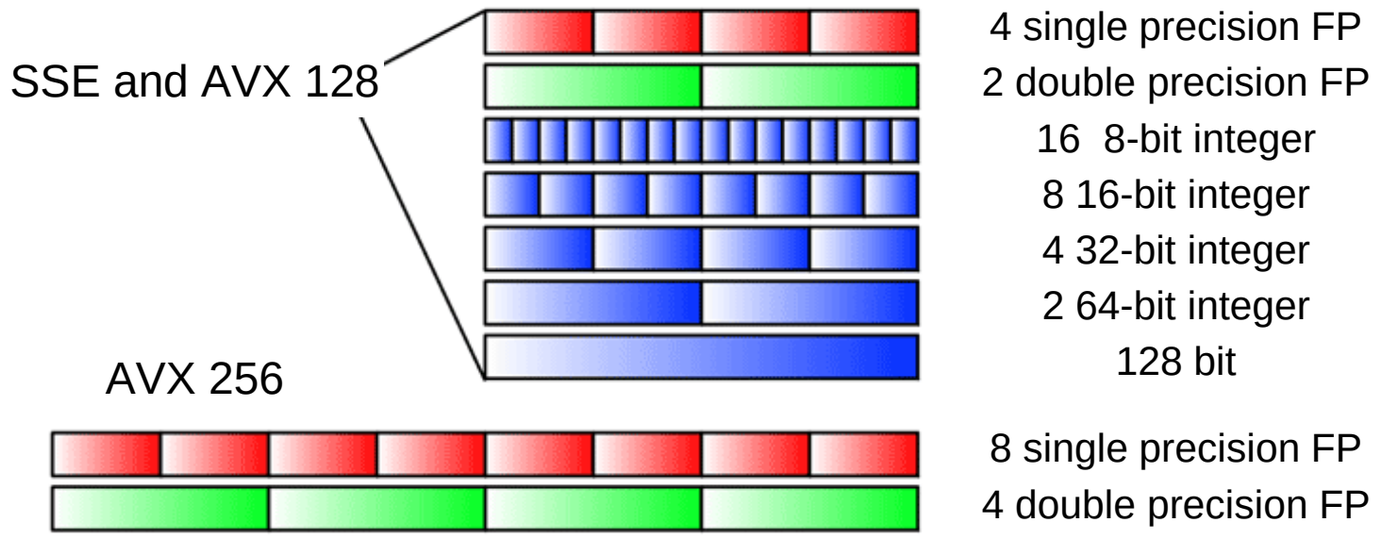
Haswell

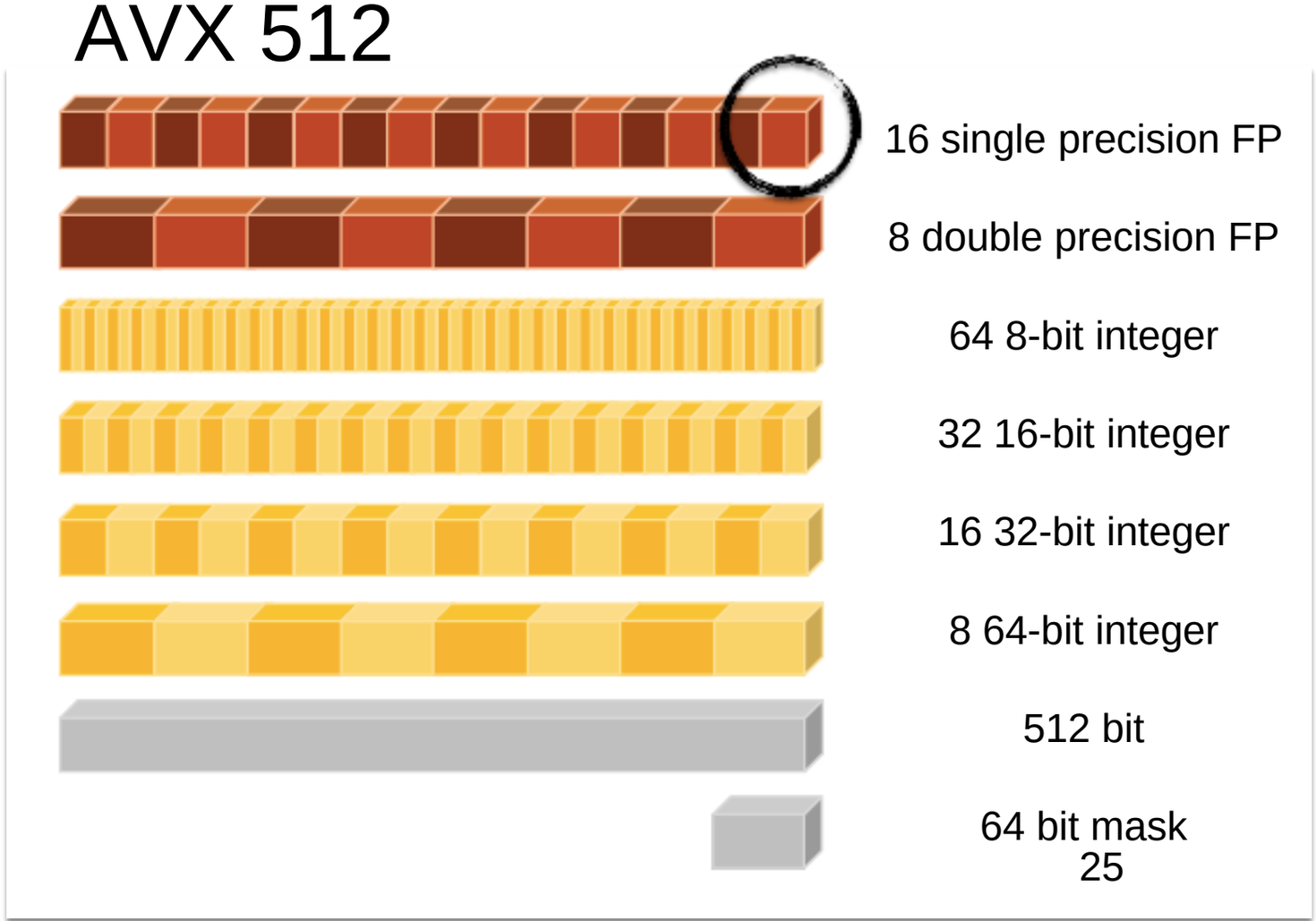
Xeon Phi (Knights Corner)

Xeon Phi (Knights Landing)

Vector registers

Reminder:
Single Precision Floating Point (FP) : 32 bit
Double Precision FP : 64 bit

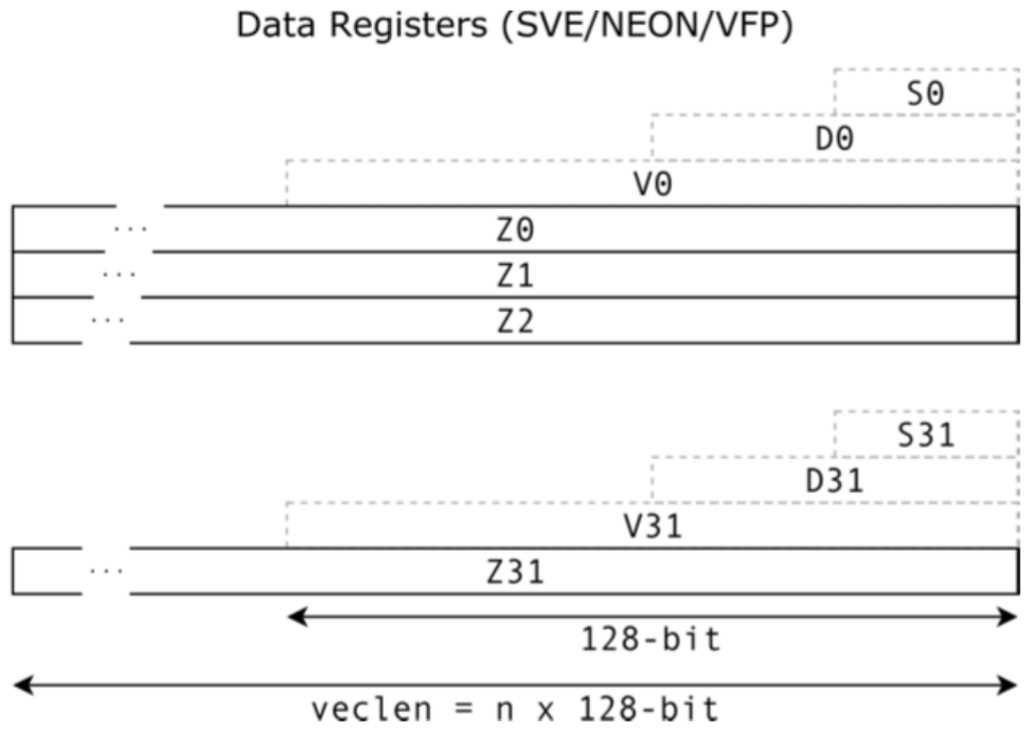




Using today one FP (single precision) means wasting 15 slots in a register!

Scalable vector extension

ARM



- Flexible register size: from 128 bits up to 2048 bits per register
- Supports vector-length-agnostic programming model that can adapt to available registers
- Compile or hand-code programs for SVE once, then run at different implementation performance points

No need to recompile or rewrite when longer vectors appear!

https://developer.arm.com/hpc/a-sneak-peek-into-sve-and-vla-programming

SIMD programming models

- Autovectorization
- External / compiler pragmas
- SIMD libraries
- Compiler Intrinsics
- Inline Assembly

```
float a[N], b[N], c[N];

for (int i = 0; i < N; i++)
  a[i] = b[i] * c[i];
```

```
float a[N], b[N], c[N];

#pragma omp simd
#pragma ivdep
for (int i = 0; i < N; i++)
  a[i] = b[i] * c[i];
```

```
#include <Vc/Vc>
Vc::SimdArray<float, N> a, b, c;

a = b * c;
```

```
#include <x86intrin.h>
__m256 a, b, c;

a = _mm256_mul_ps(b, c);
```

```
asm volatile("vmulps %ymm1, %ymm0");
```

Auto-Vectorisation

Good practices to “convince the compiler”

- Prefer countable single entry and single exit “for” loops.
- Write straight line code, reducing branches (switches, goto or return statements)
- Avoid dependencies between loop iterations
- Prefer array notation to pointers.
- Use the loop index directly in array subscripts where possible
- Favour inner loops with unit stride
- Align data (Data to be operated upon as an n-byte chunk is stored on an n-byte memory boundary)
- Use efficient memory accesses
- Prefer Structure of Arrays (SoA) over Array of Structures (AoS)

Code example: quadratic solver

By Guilherme Amadio

Simple Implementation

```
template <typename T> int QuadSolve(T a, T b, T c, T &x1, T &x2)
{
  T delta = b * b - 4.0 * a * c;

  if (delta < 0.0) return 0;

  if (delta < std::numeric_limits<T>::epsilon()) {
    x1 = x2 = -0.5 * b / a;
    return 1;
  }

  if (b >= 0.0) {
    x1 = -0.5 * (b + std::sqrt(delta)) / a;
    x2 = c / (a * x1);
  } else {
    x2 = -0.5 * (b - std::sqrt(delta)) / a;
    x1 = c / (a * x2);
  }

  return 2;
}
```


VecCore library. https://github.com/root-project/veccore

Code example: quadratic solver

By Guilherme Amadio

Optimized Implementation

```
template <typename T> void QuadSolve(const T &a, const T &b, const T &c, T &x1, T &x2, int &roots)
{
  T a_inv = T(1.0) / a;
  T delta = b * b - T(4.0) * a * c;
  T s     = (b >= 0) ? T(1.0) : T(-1.0);

  roots = delta > numeric_limits<T>::epsilon() ? 2 : delta < T(0.0) ? 0 : 1;

  switch (roots) {
  case 2:
    x1 = T(-0.5) * (b + s * std::sqrt(delta));
    x2 = c / x1;
    x1 *= a_inv;
    return;

  case 0:
    return;

  case 1:
    x1 = x2 = T(-0.5) * b * a_inv;
    return;

  default:
    return;
  }
}
```

Code example: quadratic solver

AVX2 Intrinsics Implementation

By Guilherme Amadio

```
void QuadSolveAVX(const float *__restrict__ a, const float *__restrict__ b, const float *__restrict__ c,
                  float *__restrict__ x1, float *__restrict__ x2, int *__restrict__ roots)
{
  __m256 one    = _mm256_set1_ps(1.0f);
  __m256 va     = _mm256_load_ps(a);
  __m256 vb     = _mm256_load_ps(b);
  __m256 zero   = _mm256_set1_ps(0.0f);
  __m256 a_inv  = _mm256_div_ps(one, va);
  __m256 b2     = _mm256_mul_ps(vb, vb);
  __m256 eps    = _mm256_set1_ps(std::numeric_limits<float>::epsilon());
  __m256 vc     = _mm256_load_ps(c);
  __m256 negone = _mm256_set1_ps(-1.0f);
  __m256 ac     = _mm256_mul_ps(va, vc);
  __m256 sign   = _mm256_blendv_ps(negone, one, _mm256_cmp_ps(vb, zero, _CMP_GE_OS));
#if defined(__FMA__)
  __m256 delta = _mm256_fmadd_ps(_mm256_set1_ps(-4.0f), ac, b2);
  __m256 r1    = _mm256_fmadd_ps(sign, _mm256_sqrt_ps(delta), vb);
#else
  __m256 delta = _mm256_sub_ps(b2, __256_mul_ps(_mm256_set1_ps(-4.0f), ac));
  __m256 r1    = _mm256_add_ps(vb, _mm256_mul_ps(sign, _mm256_sqrt_ps(delta)));
#endif
  __m256 mask0 = _mm256_cmp_ps(delta, zero, _CMP_LT_OS);
  __m256 mask2 = _mm256_cmp_ps(delta, eps, _CMP_GE_OS);
  r1           = _mm256_mul_ps(_mm256_set1_ps(-0.5f), r1);
  __m256 r2    = _mm256_div_ps(vc, r1);
  r1           = _mm256_mul_ps(a_inv, r1);
  __m256 r3    = _mm256_mul_ps(_mm256_set1_ps(-0.5f), _mm256_mul_ps(vb, a_inv));
  __m256 nr    = _mm256_blendv_ps(one, _mm256_set1_ps(2), mask2);
  nr           = _mm256_blendv_ps(nr, _mm256_set1_ps(0), mask0);
  r3           = _mm256_blendv_ps(r3, zero, mask0);
  r1           = _mm256_blendv_ps(r3, r1, mask2);
  r2           = _mm256_blendv_ps(r3, r2, mask2);
  _mm256_store_si256((__m256i *)roots, _mm256_cvtps_epi32(nr));
  _mm256_store_ps(x1, r1);
  _mm256_store_ps(x2, r2);
}
```


VecCore library: https://github.com/root-project/veccore

Code example: quadratic solver

VecCore API Implementation

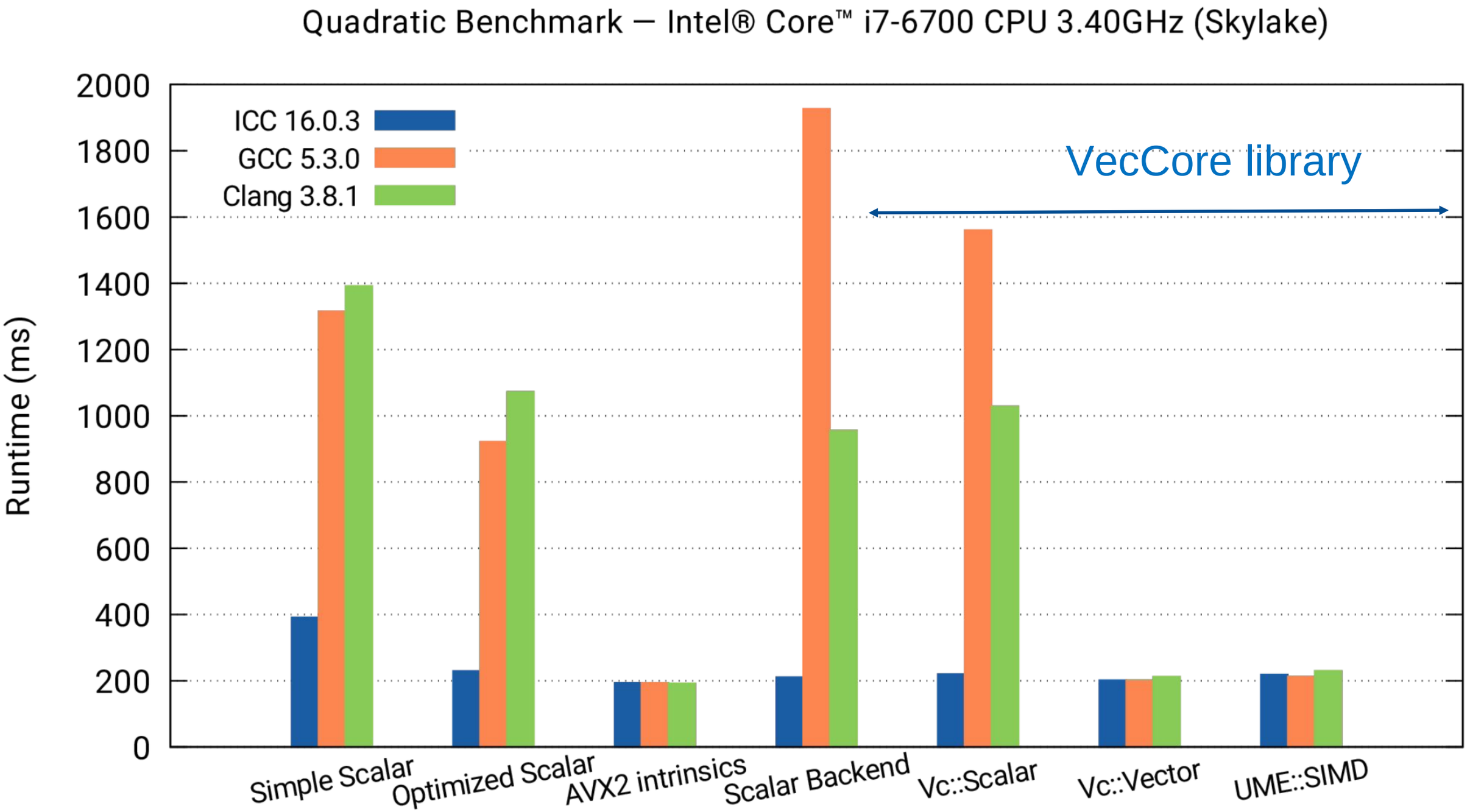
By Guilherme Amadio

```
template <typename Float_v, typename Int32_v>
void QuadSolveVecCore(Float_v const &a, Float_v const &b, Float_v const &c,
                      Float_v &x1, Float_v &x2, Int32_v &roots)
{
  Float_v a_inv = Float_v(1.0f) / a;
  Float_v delta = b * b - Float_v(4.0f) * a * c;

  Mask<Float_v> mask0(delta < Float_v(0.0f));
  Mask<Float_v> mask2(delta >= NumericLimits<Float_v>::Epsilon());

  Float_v root1 = Float_v(-0.5f) * (b + math::Sign(b) * math::Sqrt(delta));
  Float_v root2 = c / root1;
  root1         = root1 * a_inv;

  MaskedAssign(x1, mask2, root1);
  MaskedAssign(x2, mask2, root2);
  roots = Blend(Mask<Int32_v>(mask2), Int32_v(2), Int32_v(0));

  Mask<Float_v> mask1 = !(mask2 || mask0);

  if (MaskEmpty(mask1)) return;

  root1 = Float_v(-0.5f) * b * a_inv;
  MaskedAssign(roots, Mask<Int32_v>(mask1), Int32_v(1));
  MaskedAssign(x1, mask1, root1);
  MaskedAssign(x2, mask1, root1);
}
```


VecCore library: https://github.com/root-project/veccore

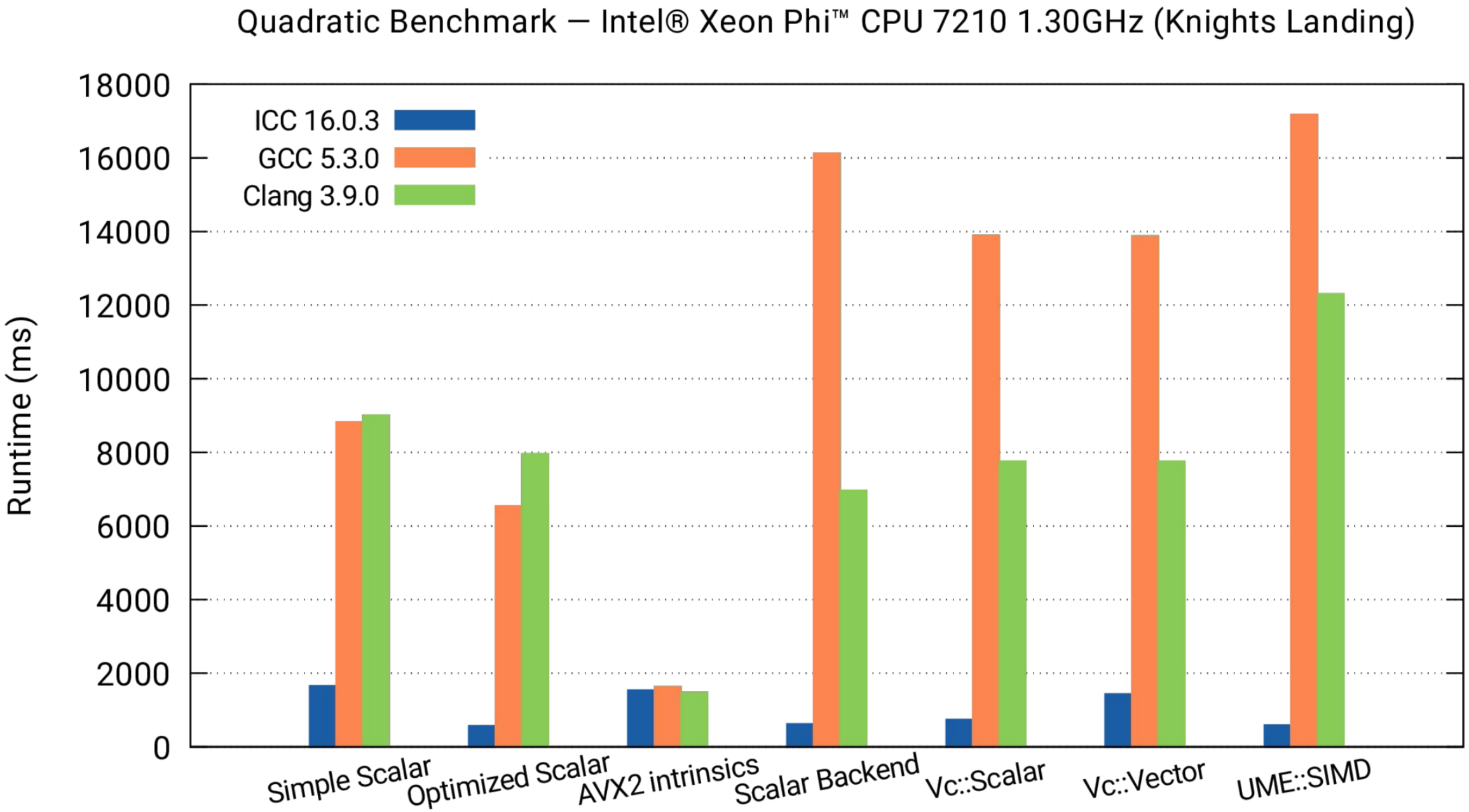
Performance comparison

By Guilherme Amadio



Performance comparison

By Guilherme Amadio



VecCore library: https://github.com/root-project/veccore

Other compiler optimisations

For every compiler there is a modified compiler that generates shorter code

Rice, 1953

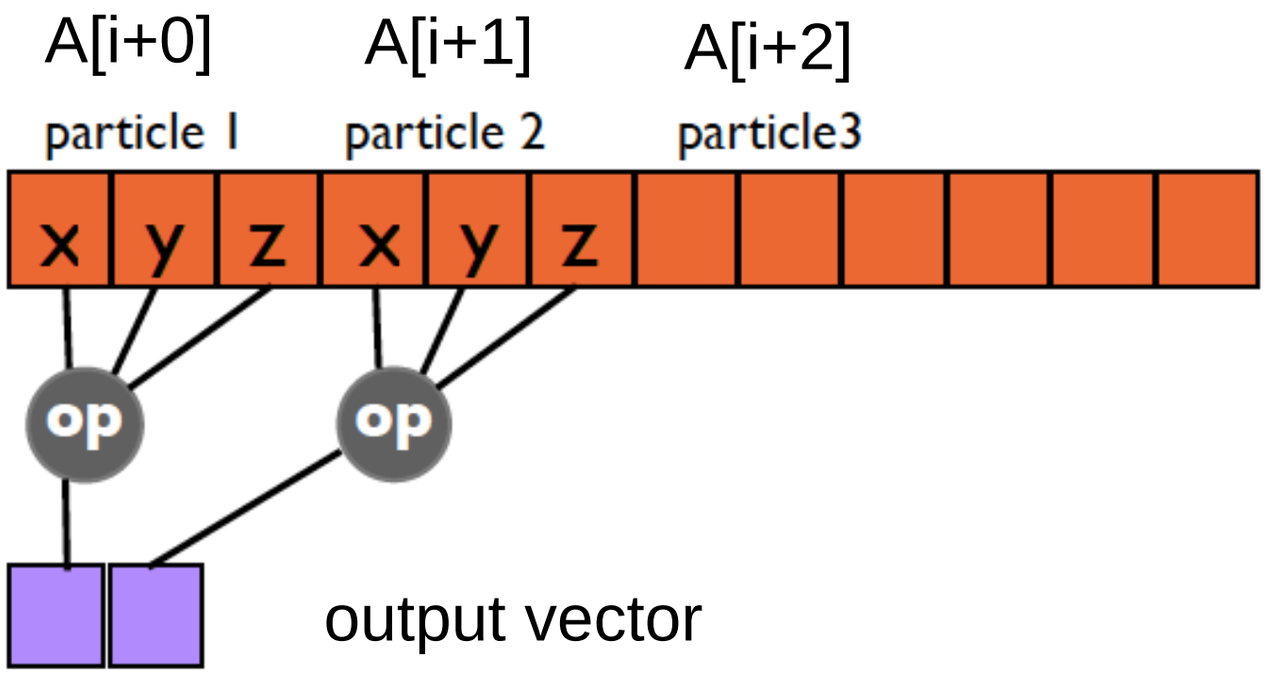
- Instruction selection: e.g. *2 multiplication done by addition, bit-shift
- Constant elimination
- Algebraic simplification
- Dead code removal
- Loop Optimisations: often executed, large payoff!
- Inlining: improves time at the cost of space (larger code); allows for further optimisation;

Differences among compilers and target architectures

Trade off on accuracy and precision

- Controlled by flags and pragmas
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
  - https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

Memory access pattern: AOS



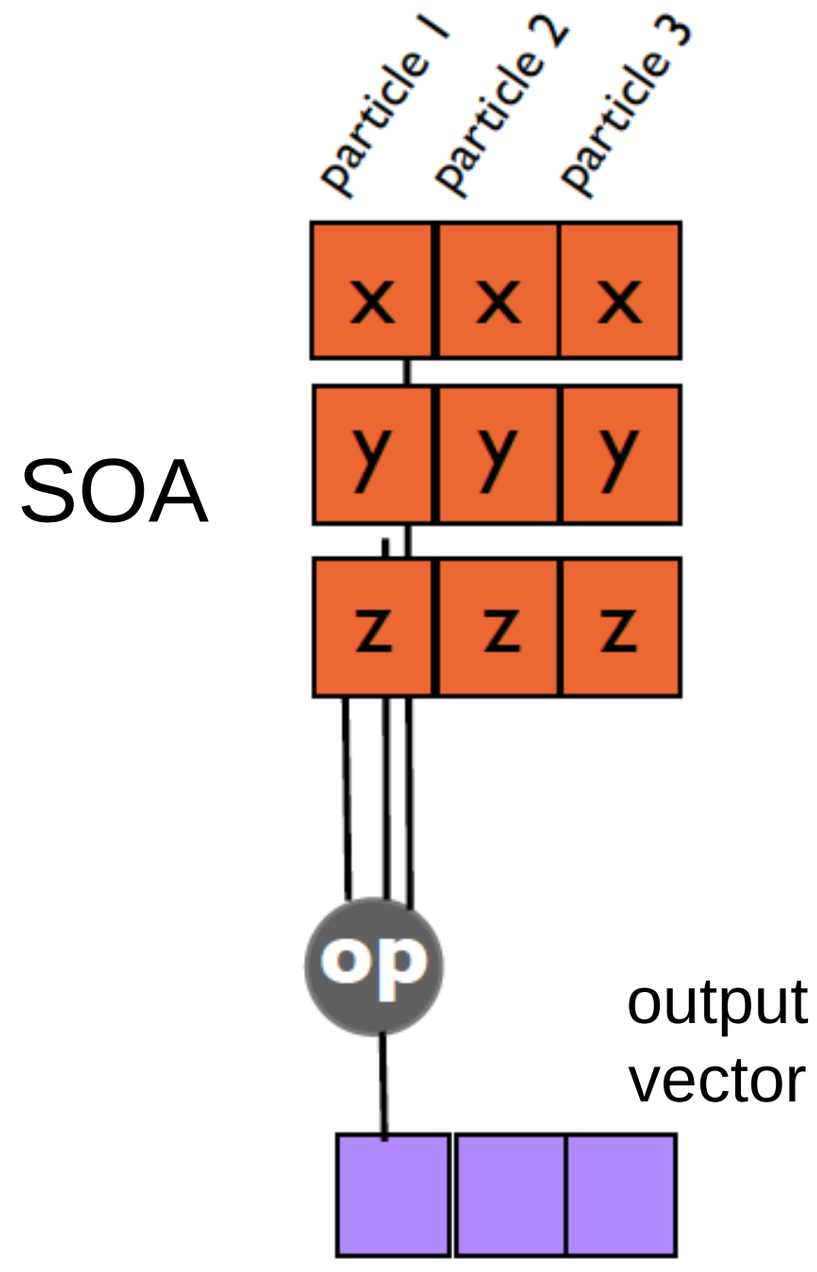
Loading AOS in a vector register is a strided load/store operation

- Multiple load/shuffle/insert or gather operations
- As vector register lenght increases so does the number of ops required to fill it
- Need large compute part in the algorithm to amortize the AOS overhead

Memory access pattern: SOA

SOA approach is better vectorised by compilers

- Memory access is more efficient if memory layout has multiple instances of a data member adjacent in memory and aligned
- Single load/store to move data in/out of registers



Need to take overheads to write data as SOA into account!!

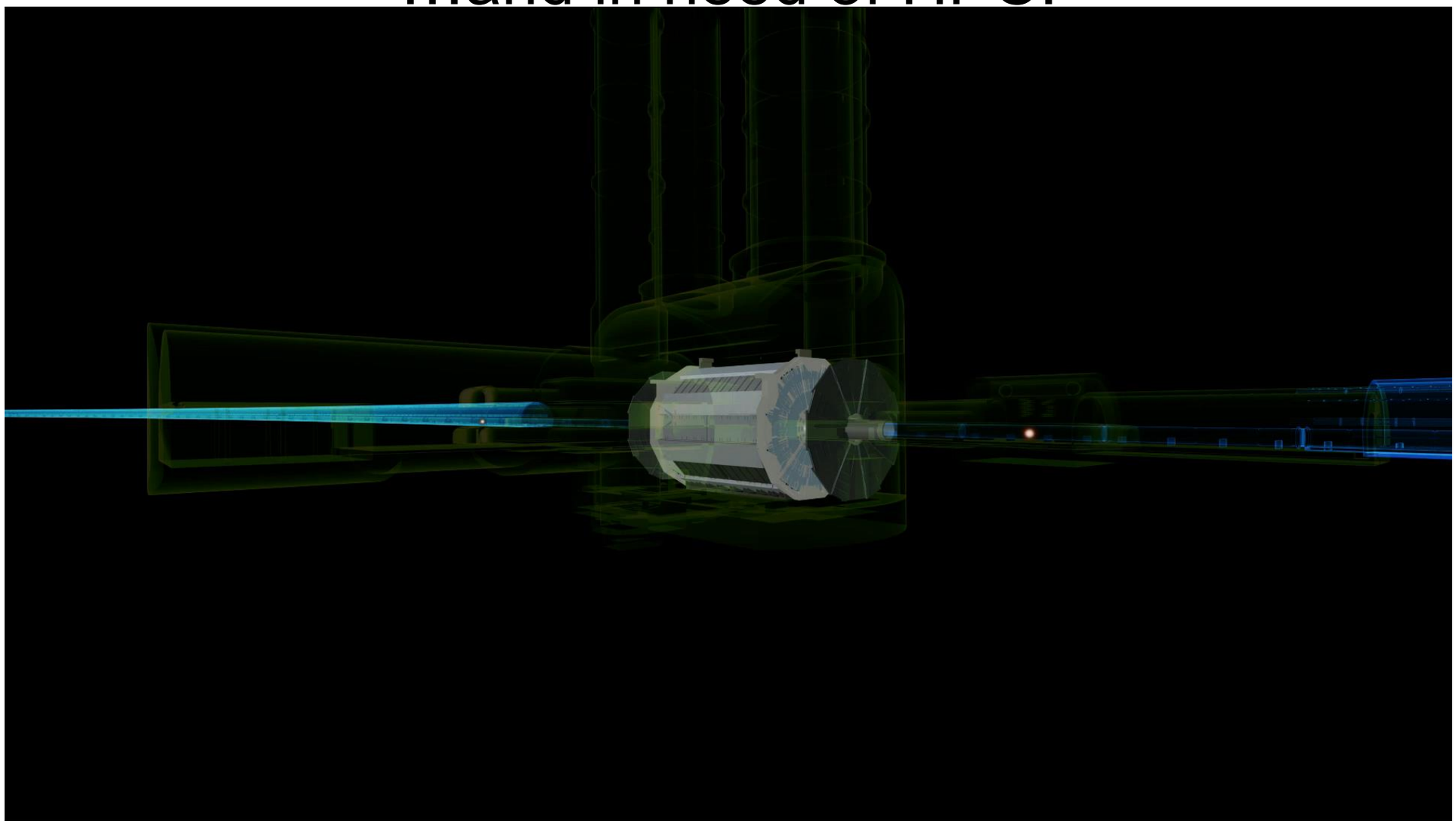
Our case study

Simulation in High Energy Physics

simulating the passage of particles through matter

Essential for detector design and data-theory comparison

…and in need of HPC!

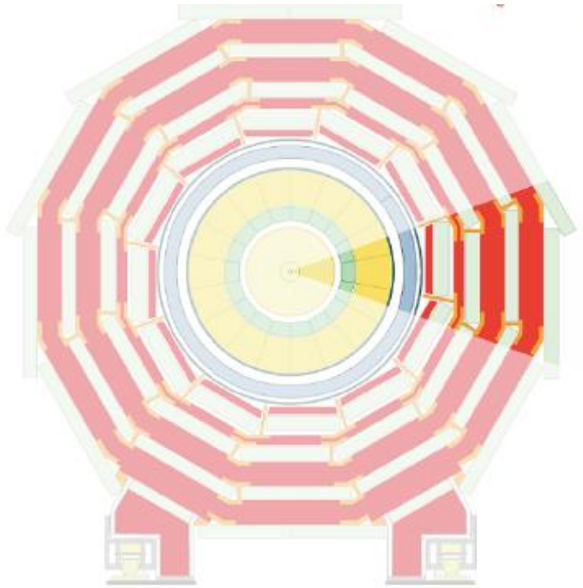


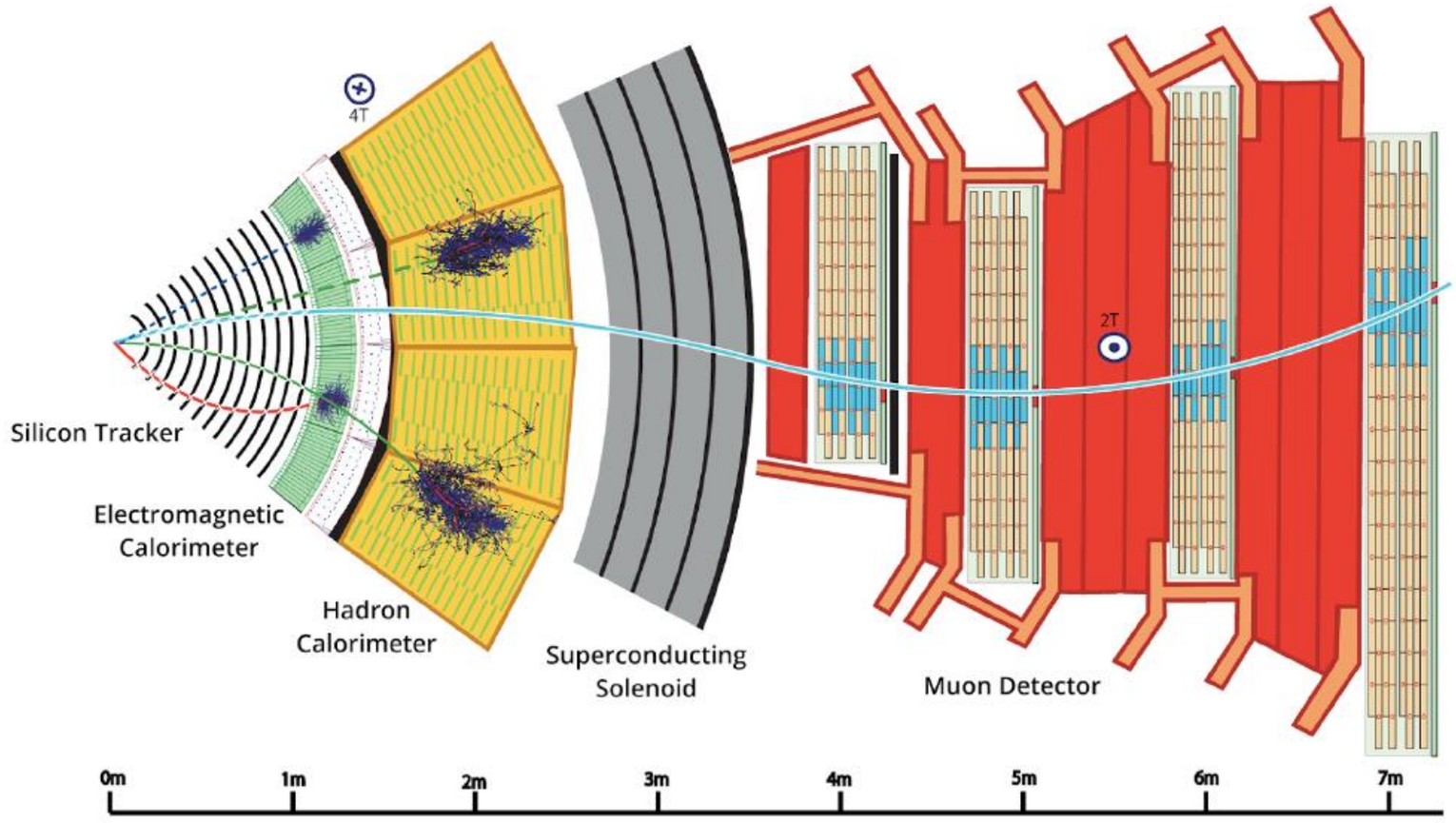
https://cds.cern.ch/record/1309872

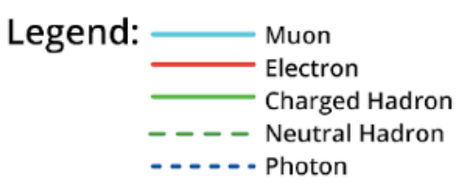
Simulation in HEP

Heavy computation requirements, massively CPU-bound

The LHC uses more than 50% of its distributed GRID power for detector simulations (~250.000 CPU years equivalent so far)





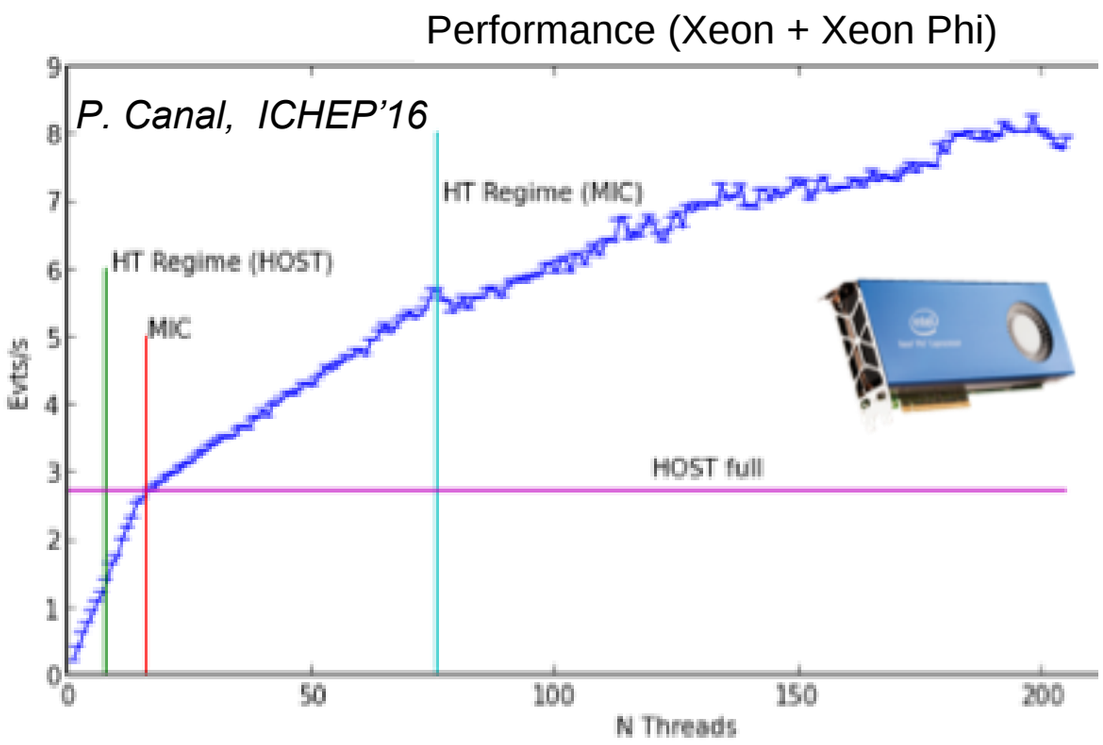


cms.web.cern.ch

Geant4 (GEometry ANd Tracking)

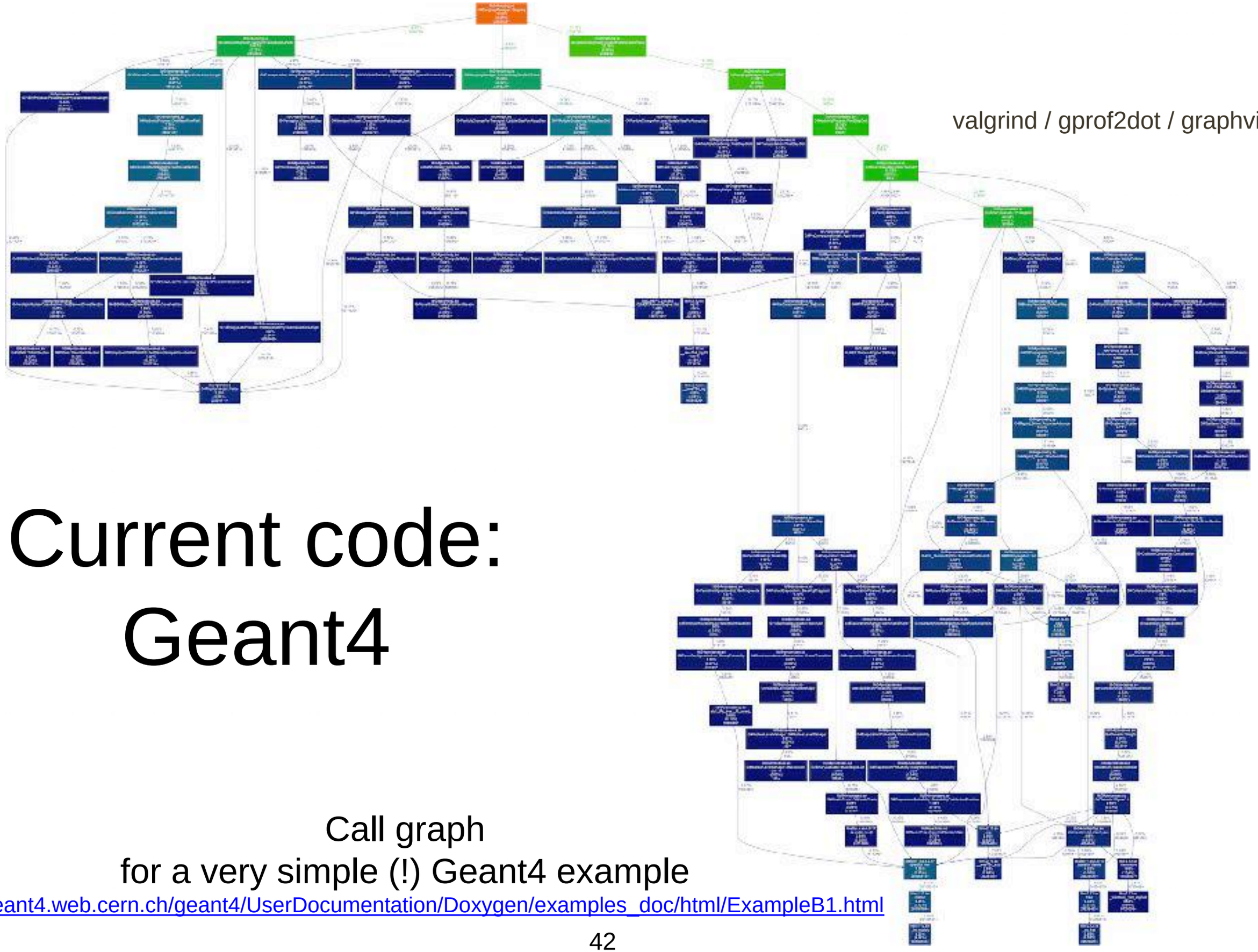
current standard within HEP

- Major international collaboration, ~2M lines of code, hundreds of users worldwide
- Large variety of applications ..beyond HEP: Medical applications, materials & space science
- Scalar processing: Each particle is simulated and followed through its whole life one by one.
- Event level parallelism: each thread processes one event exclusively



- Linear scaling of throughput with number of threads
- Large savings in memory: 9MB extra memory per thread
- No Performance/Throughput increase

Current code: Geant4



valgrind / gprof2dot / graphviz

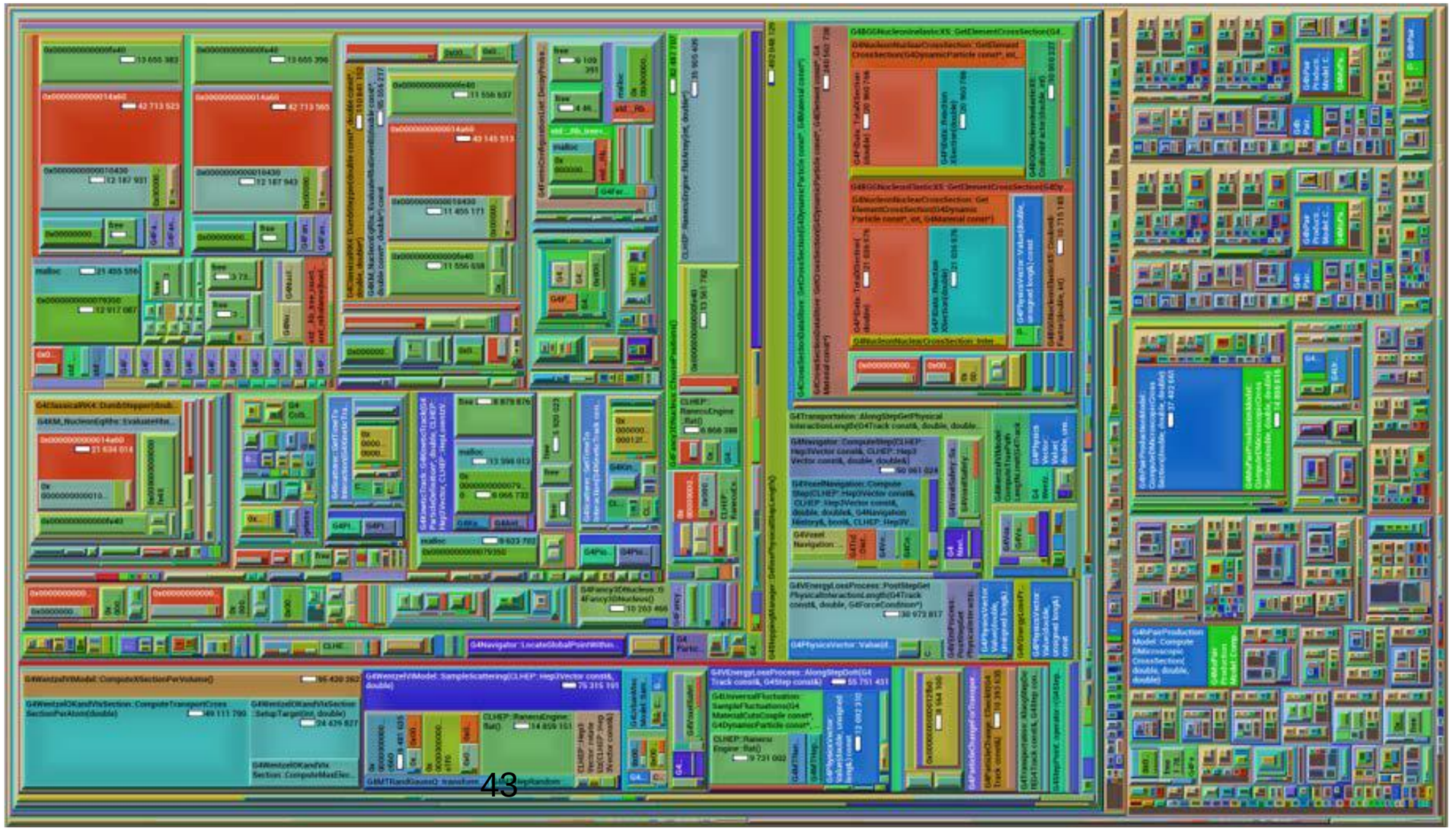
Call graph
for a very simple (!) Geant4 example

http://geant4.web.cern.ch/geant4/UserDocumentation/Doxygen/examples_doc/html/ExampleB1.html

Current code: Geant4

- Codebase very large and non-homogenous
- Very deep call stack (IC misses) and virtual table structure
- Hotspots practically inexistent

Valgrind/kCachegrind



Each rectangle represents a function

so .. how do we optimise?

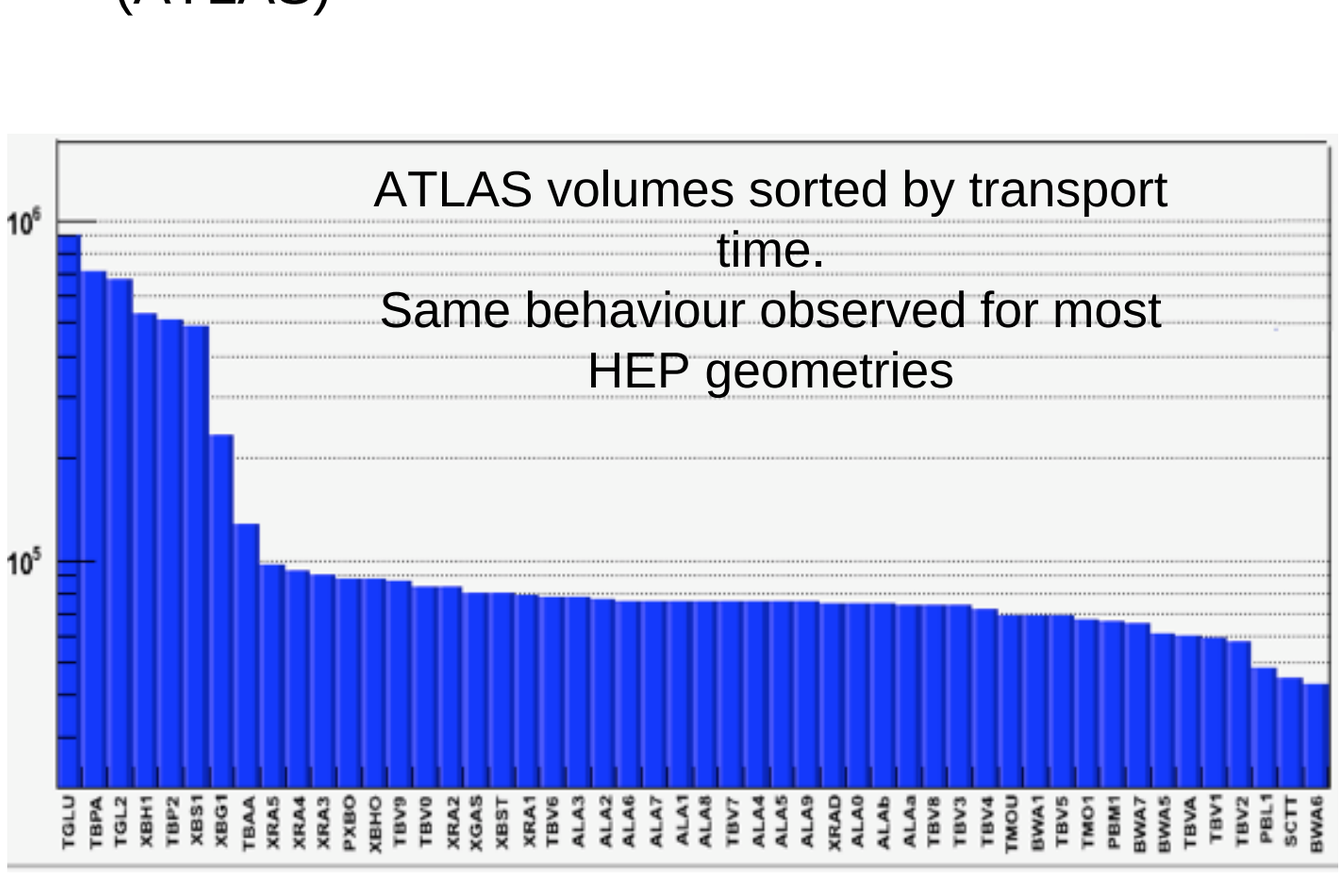
..a hint..

Level	Potential gains	Estimate
Algorithm	Major	~10x-1000x
Source code	Medium	~1x-10x
Compiler level	Medium-Low	~10%-20% (more possible with autovec or parallelization)
Operating system	Low	~5-20%
Hardware	Medium	~10%-30%

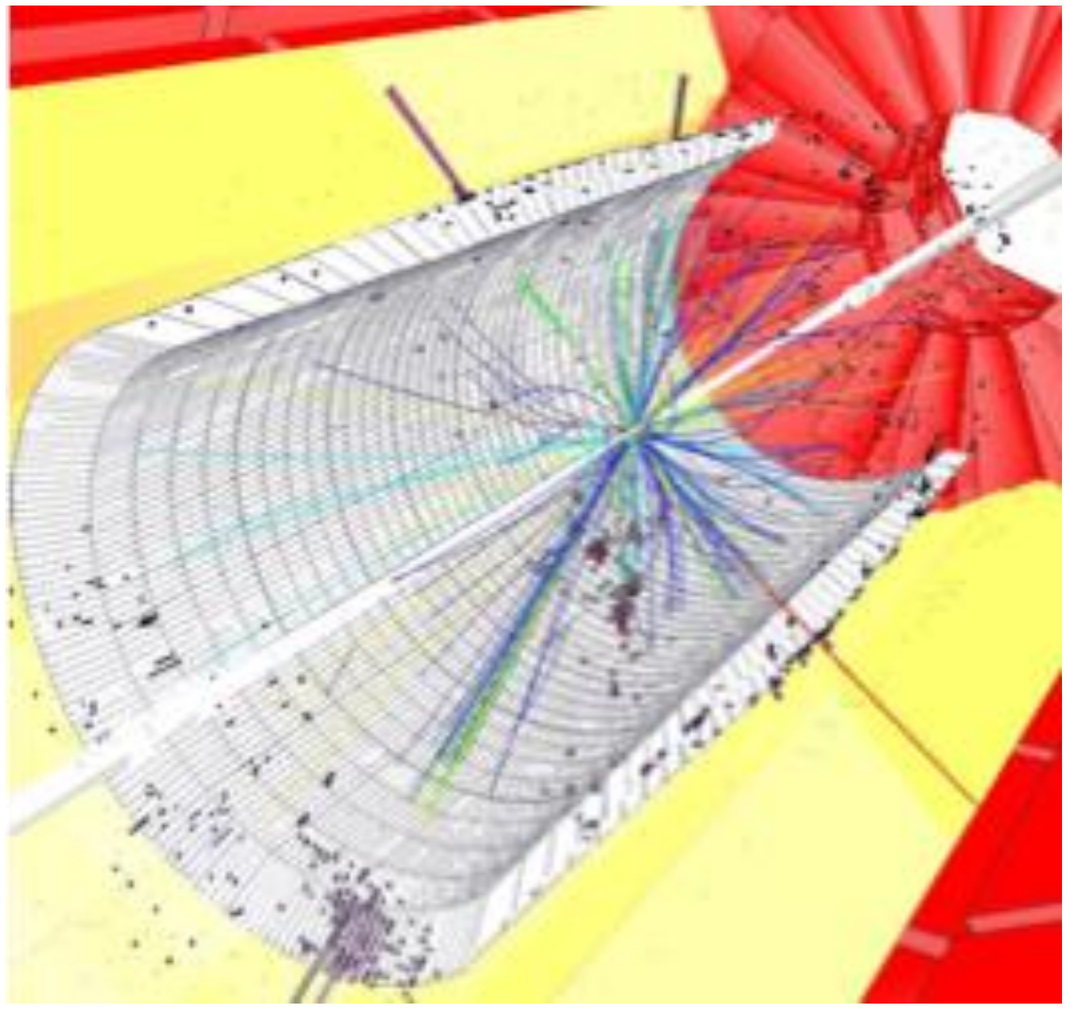
Tuning levels
“a reality check by A.Nowak”

Let's see..

- Physics is "naturally parallel"
  - Events, particle trajectories, energy depositions
- Particle transport is mostly local:
  - 50% of the time spent in 50/7100 volumes (ATLAS)



cms.cern.ch



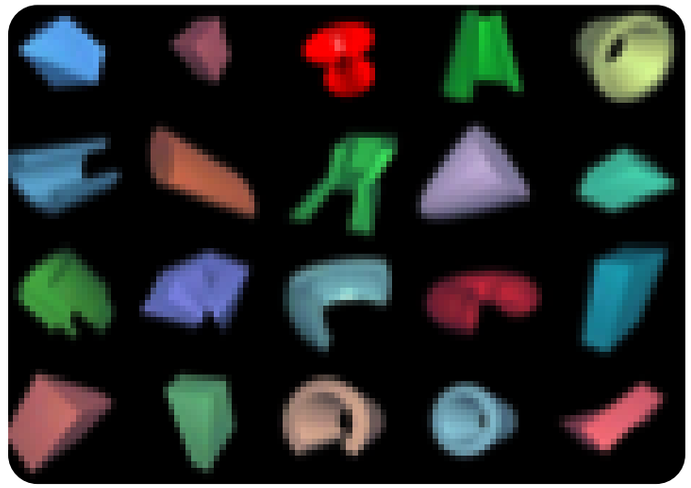
- Locality not exploited by classical transport code
- Cache misses due to fragmented code

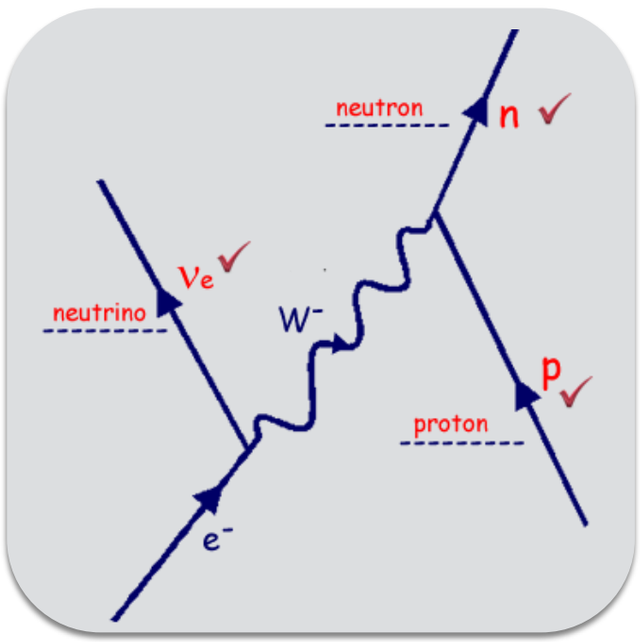
GeantV: introducing parallelism

Restructuring simulation code in a new prototype

An algorithm to transport particles through matter has “few” key ingredients:

- Geometrical shapes that describe detector volumes
- Physics algorithms that describe particle interactions with detector materials
- “Navigation” framework that organises particles and transports them “through” geometry and physics

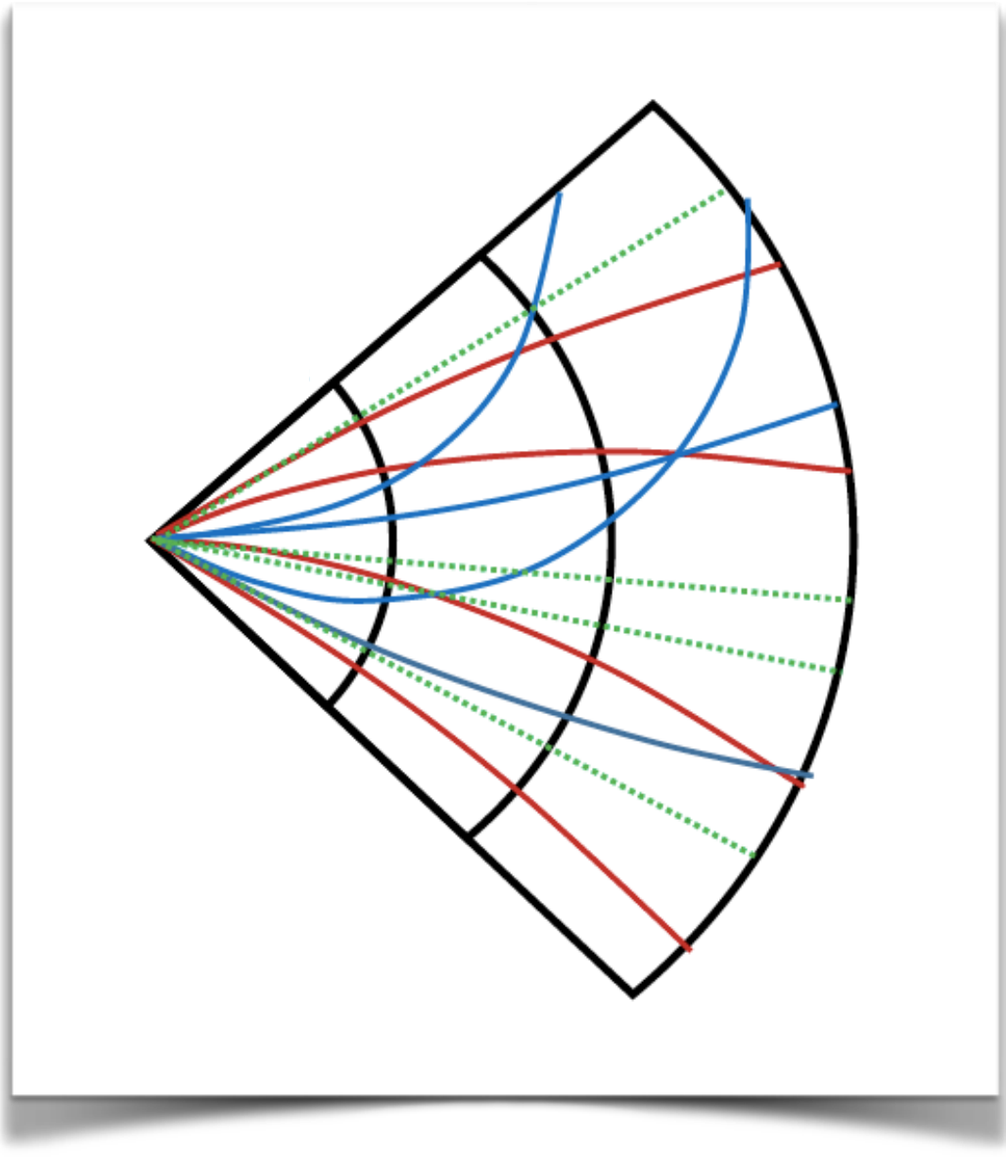




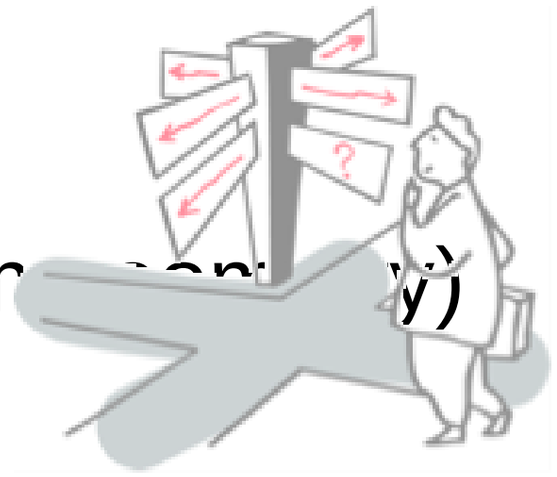
GeantV: introducing parallelism

Restructuring simulation code in a new prototype

- Introduce data parallelism: transport particles in groups
  - Group them according to geometrical volumes they cross and/or physics processes
  - Keep overhead under control!
- Introduce concurrency: split the whole flow in different tasks and/or threads to run simultaneously
- Portable on different architectures (CPUs, GPUs and accelerators)



Moving on to…

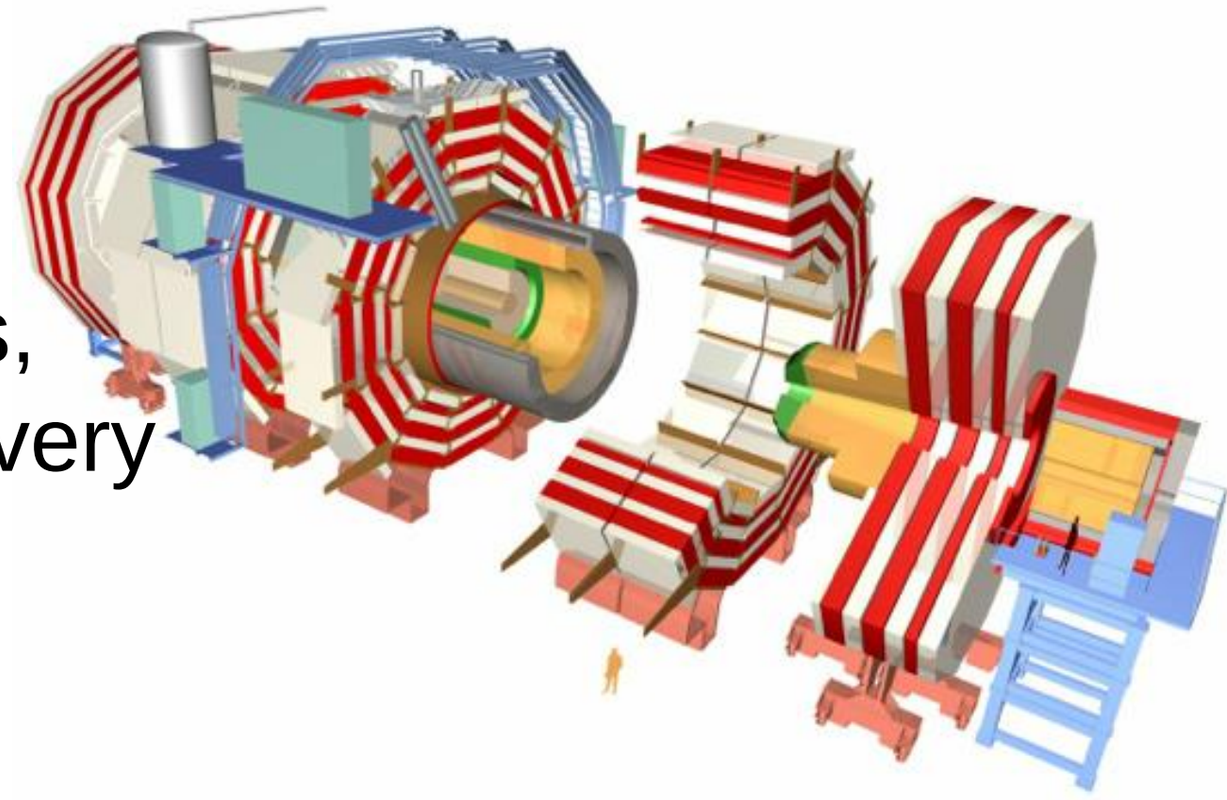


- Introducing vectorization (examples from [illegible])
- How we've implemented concurrency
- Memory management
- Performance improvement!

Geometry…

It sums up to more than 30% of processing time

The CMS detector:
boxes, trapezoids, tubes, cones, polycones millions of volumes, very complex hierarchy…

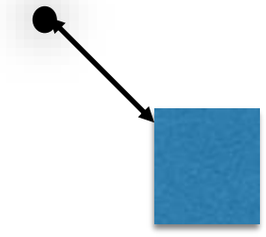


A geometry library provides APIs to:

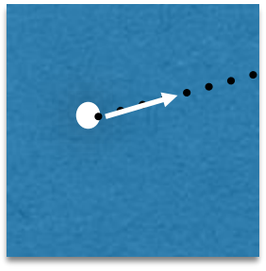
In or out?

collision detection and distance to enter de object

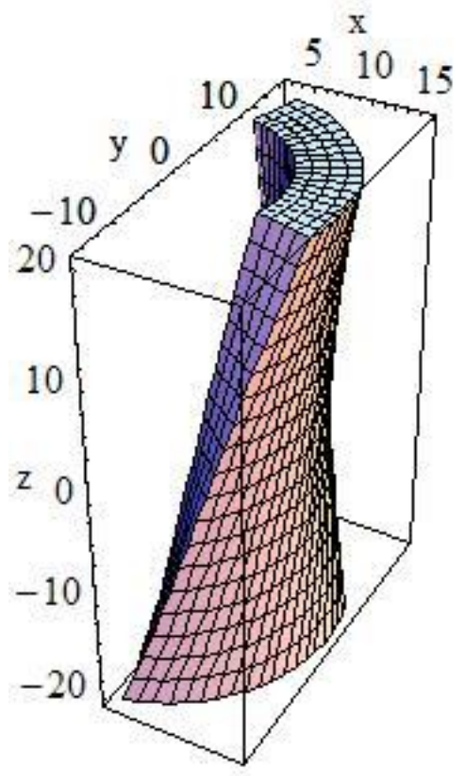
minimal safe distance to object



distance to leave object

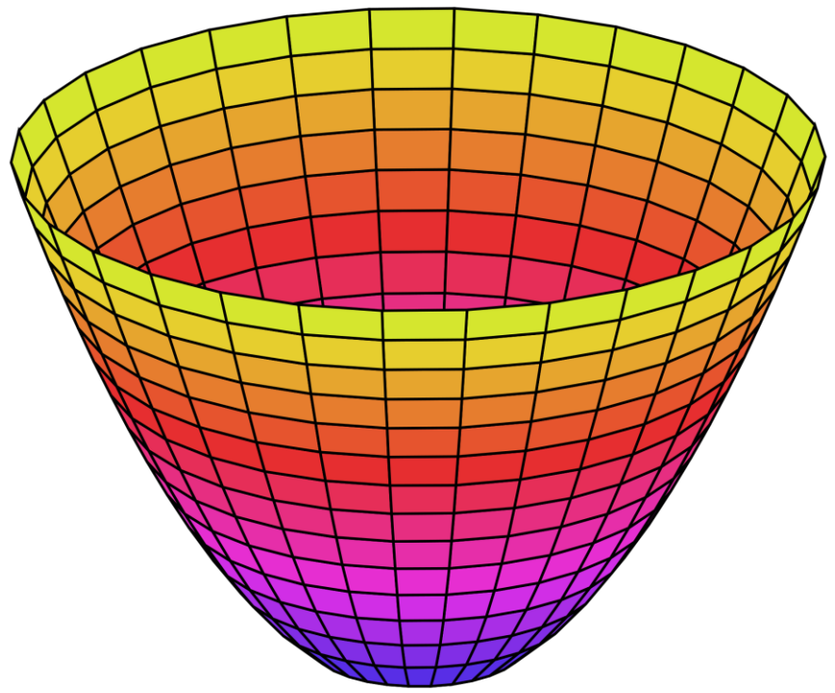


VecctorizedGeommetry



High performance geometry library for next generation simulation frameworks

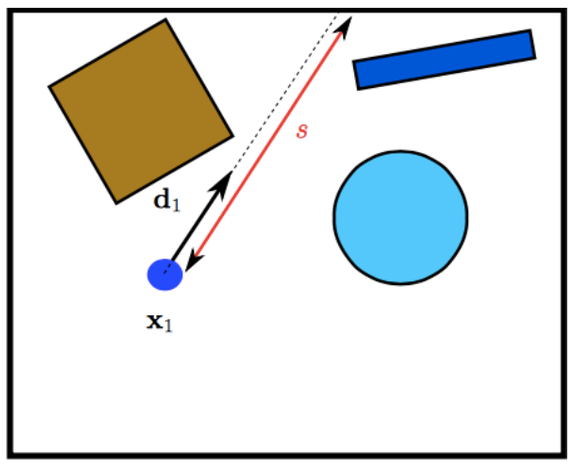
- Optimised library of primitive and composite solids
  - Reduce virtual function calls and avoid code multiplication
    - Use template code
- Introduce data parallelism
  - Explicit vectorisation (SIMD external libraries + VecCore abstraction)
    - APIs for single & many-track navigation
    - “Inner” vectorisation of complex shapes
  - Compiler autovectorisation



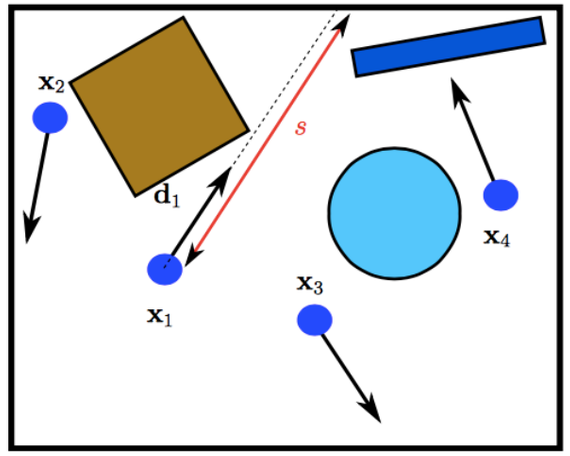
Vectorising Geometry

typical geometry task in particle tracking:
find next hitting boundary and get distance to it

1 particle -> 1 result



N particles -> N results



Option A (“free lunch”):

put code into a loop and let the compiler vectorize it works only in few cases

Option B (“convince the compiler”):

refactor the code to make it “auto-vectorizer” friendly might work but strongly compiler dependent

Option C (“use SIMD library”):

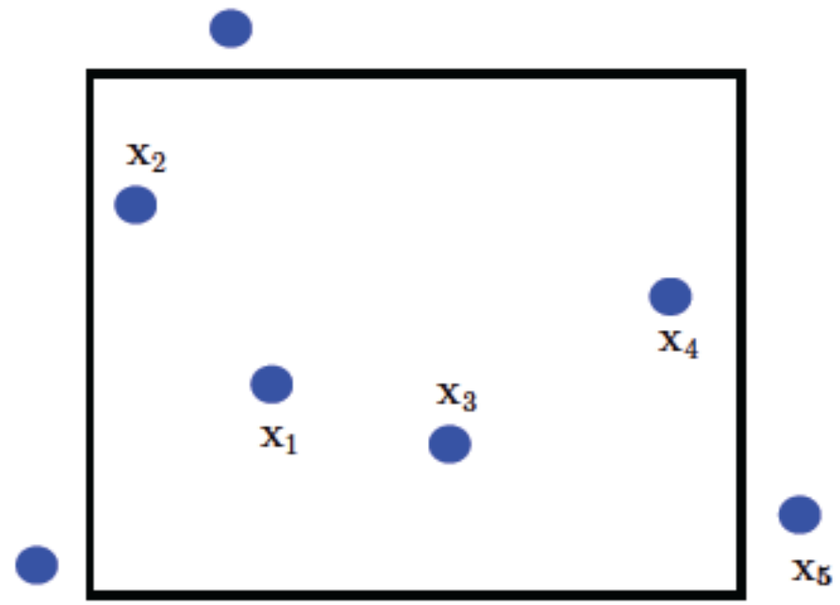
refactor the code and perform explicit vectorization using external libraries library

compiler independent

Example

A (C++) code fragment to tell whether a particle is inside a volume

```
bool contains( const double * point ){
    for( unsigned int dir=0; dir < 3; ++dir ){
        if( fabs (point[dir]-origin[dir]) > boxsize[dir] )
        return false;
    }
    return true;
}
```

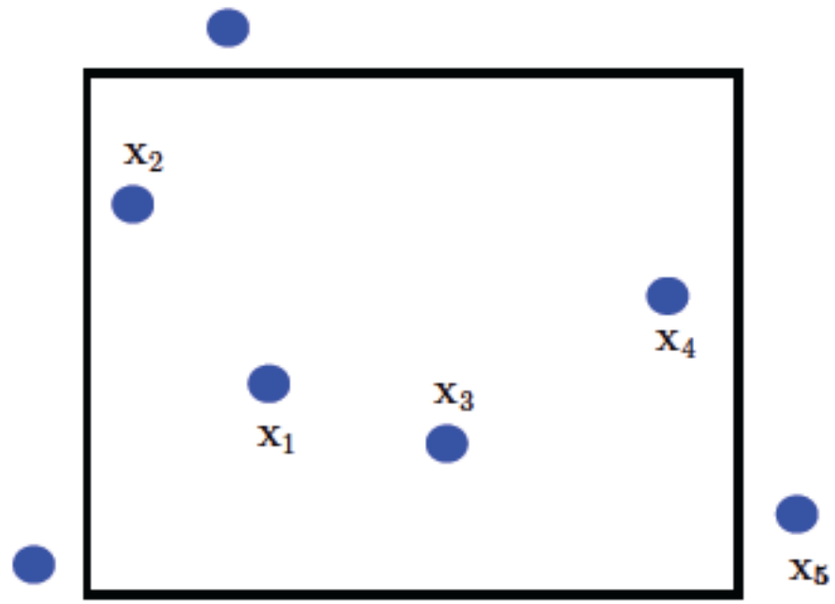


positions/dimensions vectors (x,y,z)

Option A: “free lunch”

Start from some existing code

```
bool contains( const double * point ){
    for( unsigned int dir=0; dir < 3; ++dir ){
        if( fabs (point[dir]-origin[dir]) > boxsize[dir] )
        return false;
    }
    return true;
}
```



Provide a vector interface and .. hope that compiler vectorise

```
void contains_v( const double * point, bool * isin, int np ) {
    for( unsigned int k=0; k < np; ++k) {
        isin[k]=contains( &point[3*k] );
}}
```

It doesn’t vectorise!

positions/dimensions AOS: (x,y,z,x,y,z…)

Option B: “convince the compiler”

1. copy scalar code to new function ("manual inline")
2. change the data layout (see SOA)
3. remove early - returns
4. manually unroll loops

```
void contains_v_autovec( const P & points, bool *  isin, int np ){
  for (int k=0; k < np; ++k)
  {
      bool resultx=(fabs (point.coord[0][k]-origin[0]) > boxsize[0]);
      bool resulty=(fabs (point.coord[1][k]-origin[1]) > boxsize[1]);
      bool resultz=(fabs (point.coord[2][k]-origin[2]) > boxsize[2]);
      isin[k]=resultx & resulty & resultz;
}}
```

It works but results depend on compilers choice and choice of optimisation flags

Option C: “use external library”

```
void contains_v_Vc( const P & points, bool *  isin, int np )
{
  for( int k=0; k < np; k+=Vc::double_v::Size)
  {
     Vc::double_m inside;
      inside  = (abs (Vc::double_v(point.coord[0][k])-origin[0]) < boxsize[0]);
      inside&= (abs (Vc::double_v(point.coord[1][k])-origin[1]) < boxsize[1]);
      inside&= (abs (Vc::double_v(point.coord[2][k])-origin[2]) < boxsize[2]);
      // write mask as boolean result
      for (int j=0;j<Vc::double_v::Size;++j){
          isin[k+j]=inside[j];
      }
}
```

Always vectorizes …don’t have to convince the compiler!

- excellent performance (automatically uses aligned data)
- can mix vector context and scalar context (code)

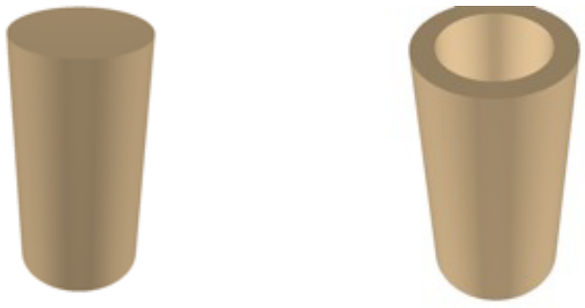
https://github.com/VcDevel

Improving vectorisation

branches are the enemy of vectorization...

Branches distinguish between “static” properties of class instances:

general “tube” class distinguishes at runtime between “FullTube”, “Hollow Tube” ...



Tube

HollowTube

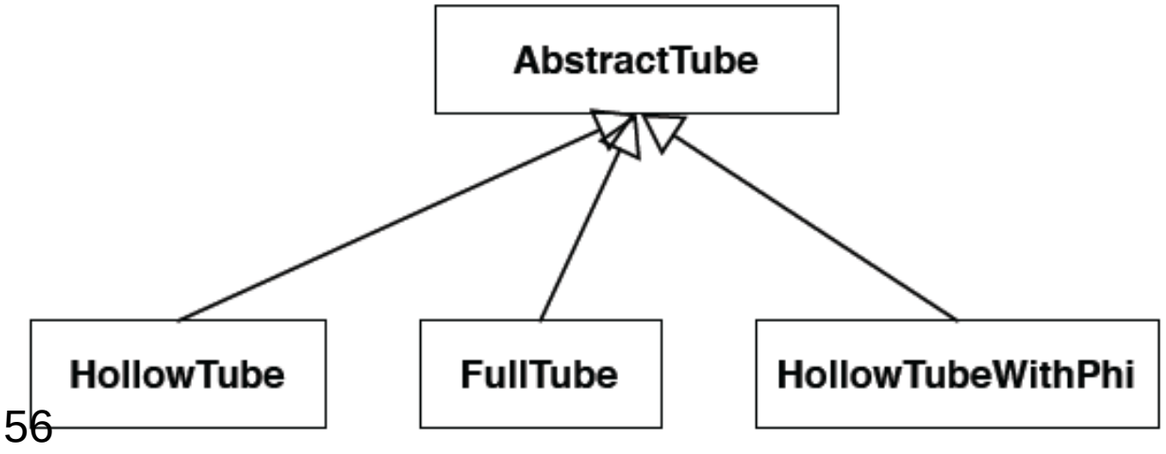
HollowTubePhi

FullTubePhi

HalfHollowTube

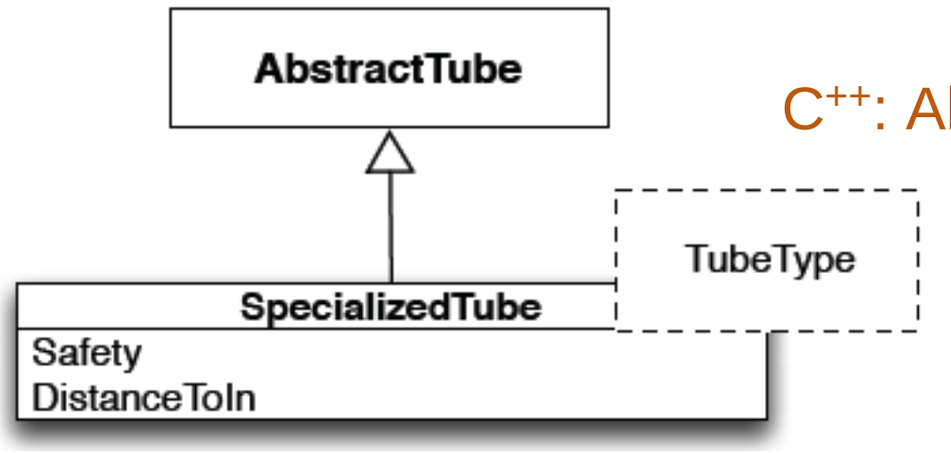
Remove branches introducing a separate class for each tube realisation

C++: AbstractTube *t = new FullTube();



Reducing branches: C++ templates

- evaluate and reduce “static” branches at compile time
- generate binary code specialised to concrete solid instances



C++: AbstractTube *t = new SpecializedTube<FullTube>();

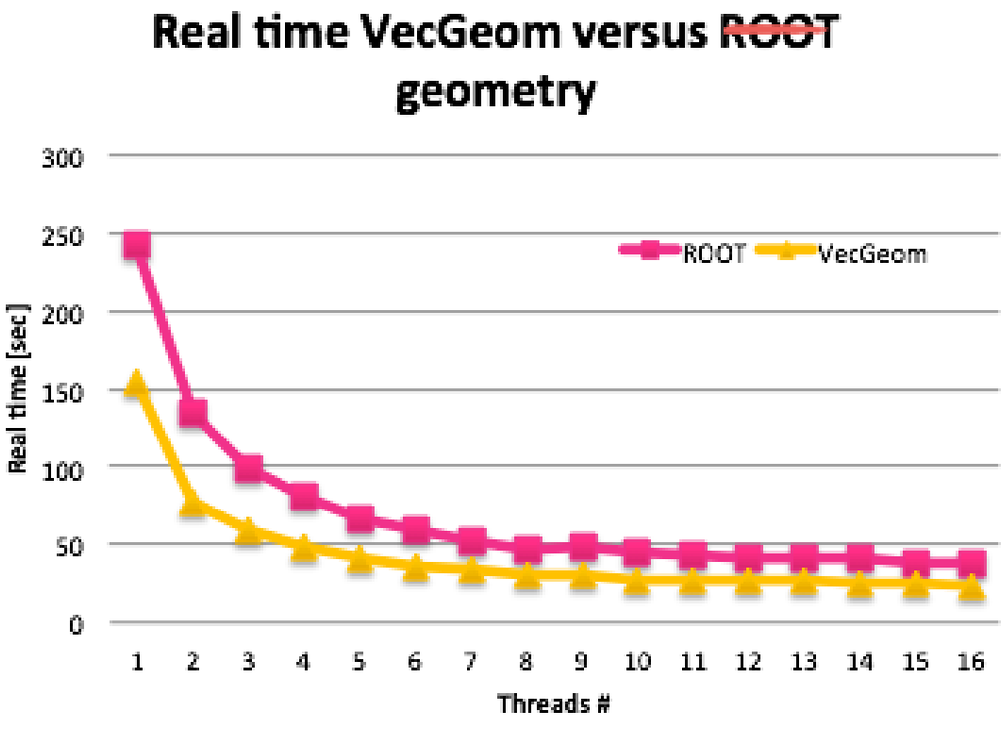
Performance
and no code duplication!

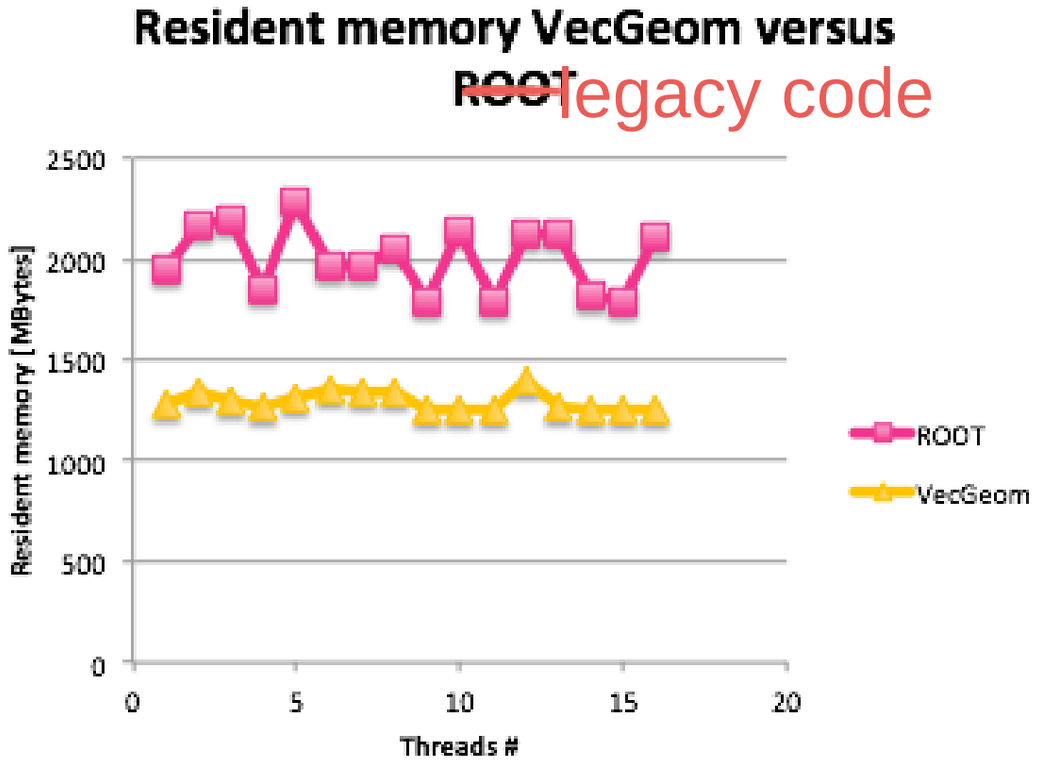
- ➞ vectorisation is efficient
- ➞ better compiler optimisations of scalar code
- ➞ less virtual functions (less calls to virtual tables)
- ➞ “generic programming” philosophy :-)
- ➞ Usage of SIMD external libraries is straightforward (VecCore abstraction layer)
- ➞ Can be used to insure portability

Scalar VecGeom performance

Simulation of 10 pp events at 7TeV in the CMS detector

legacy code





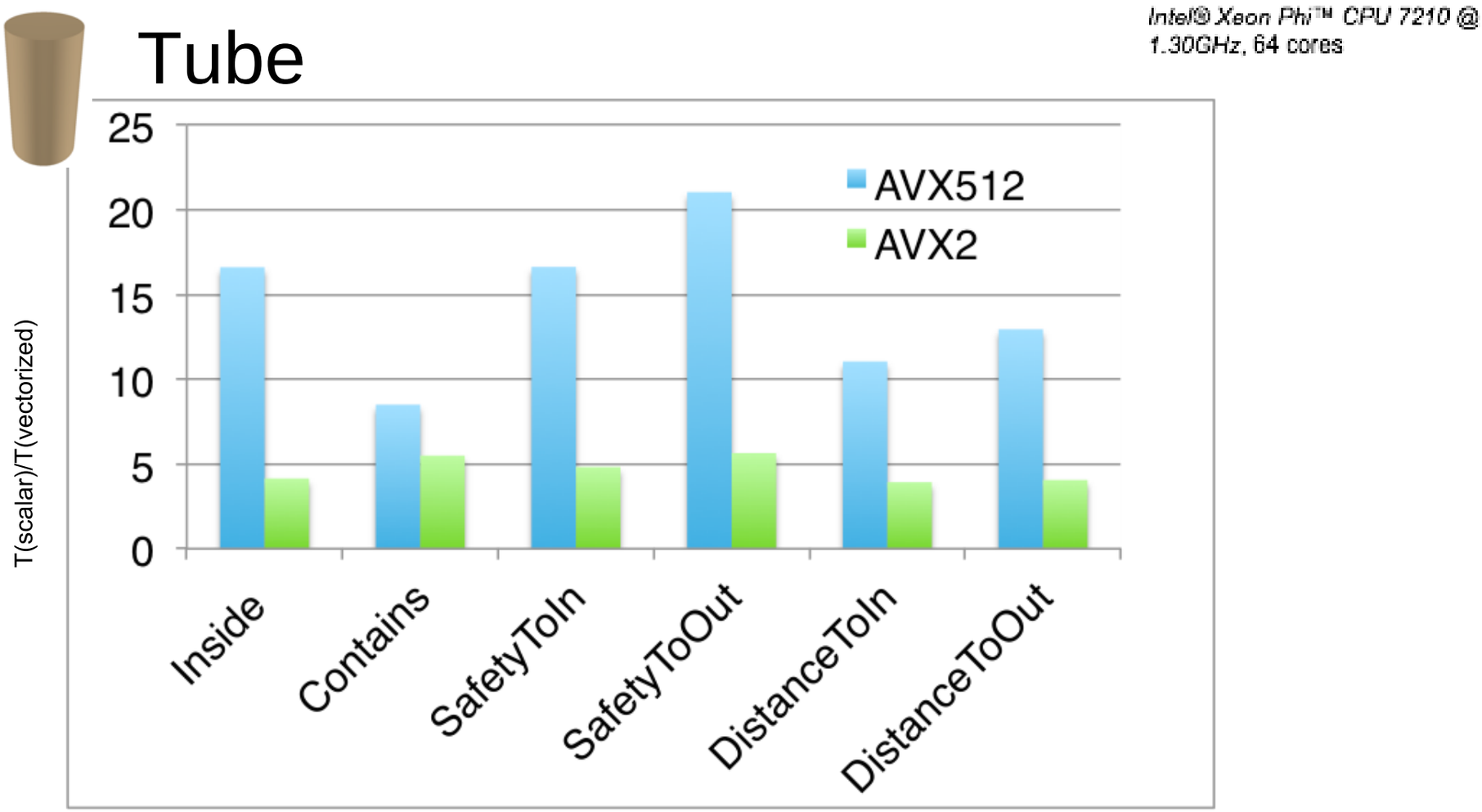
- GeantV runs VecGeom scalar navigation in full CMS geometry
  - first realistic estimate of overall impact on simulation time: ~1.6 improvement using only scalar navigation mode

VecGeom performance

A set of CPU-intensive navigation methods:

Measure wall time for vector and scalar implementations:

Calculate vector speed-up (wrt scalar) using AVX2 and AVX512

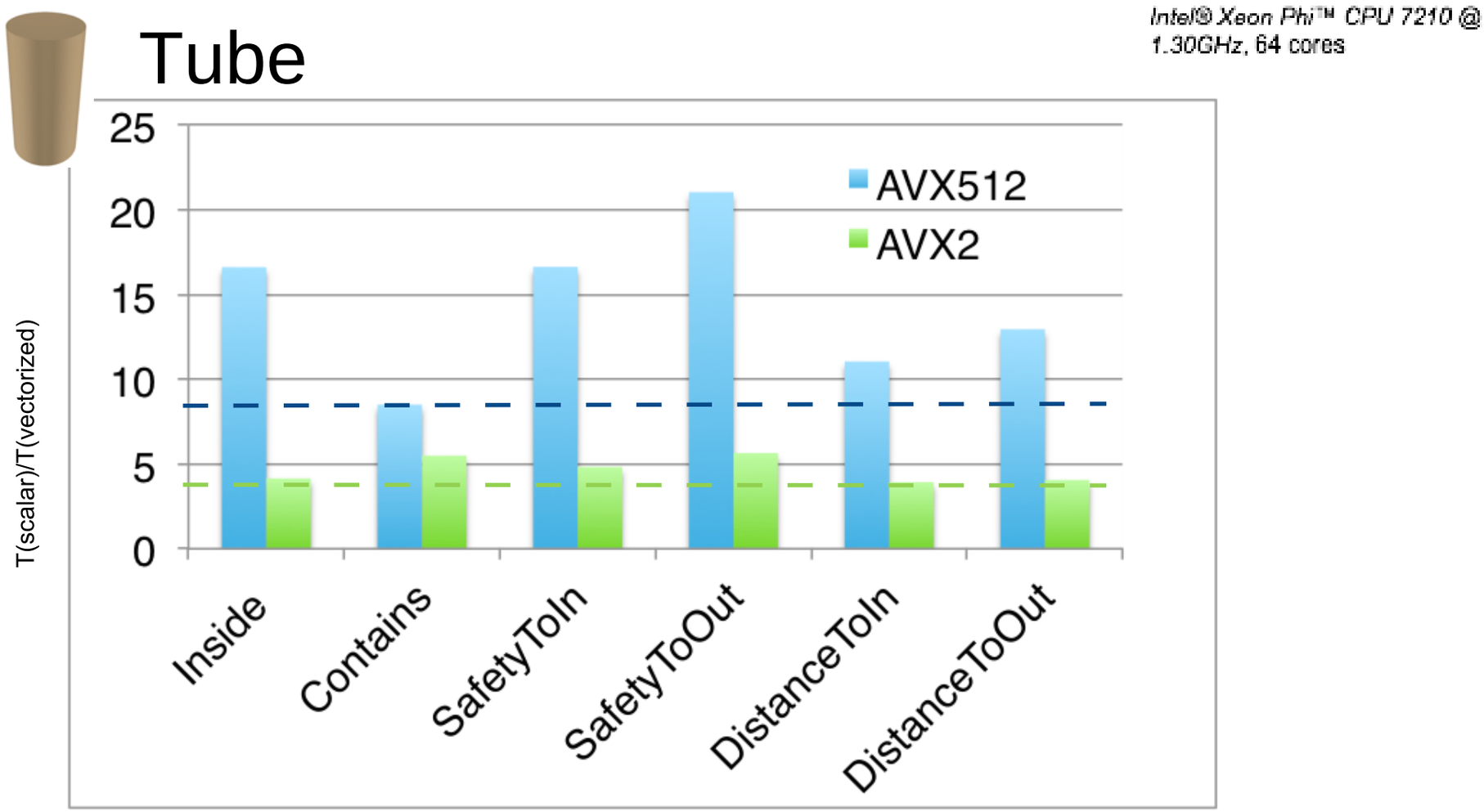


VecGeom performance

A set of CPU-intensive navigation methods:

Measure wall time for vector and scalar implementations:

Calculate vector speed-up (wrt scalar) using AVX2 and AVX512

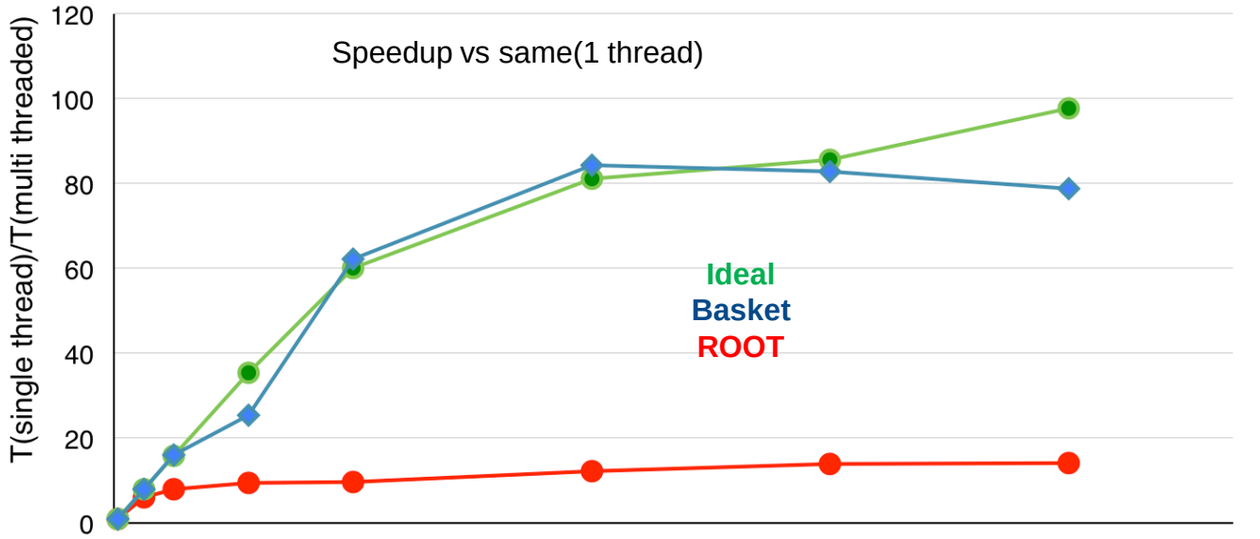


Super-linear speedup for some of the methods !

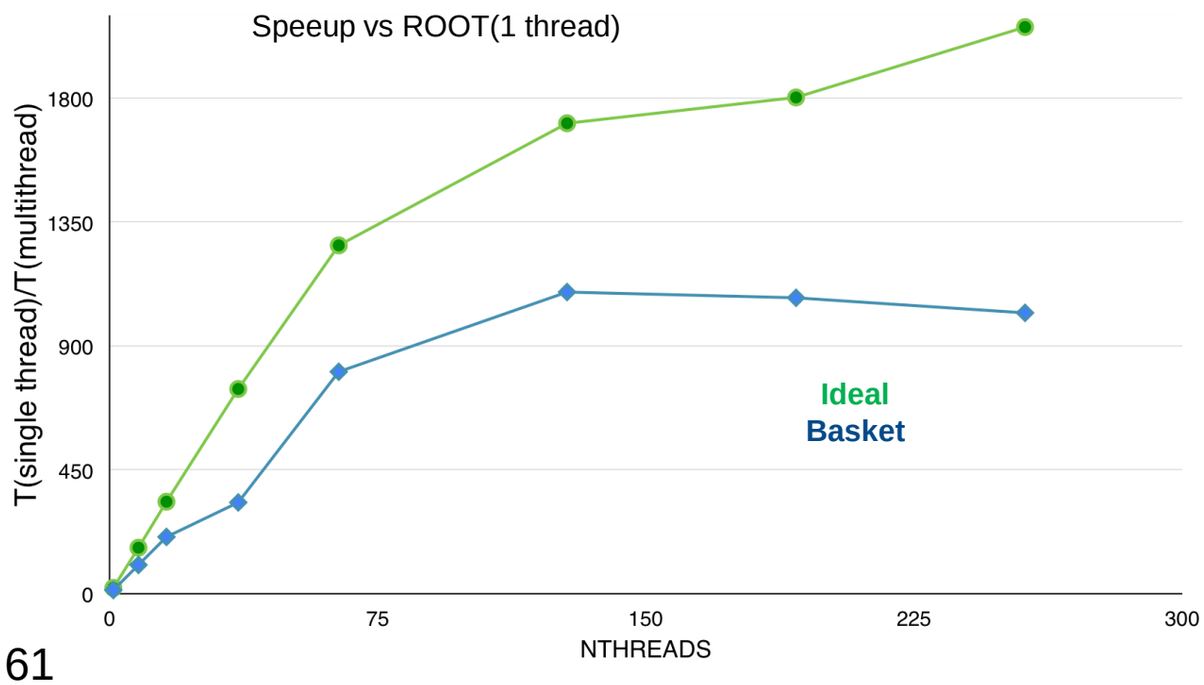
Scalability

A simplified testbed for geometry navigation:

- Toy detector (typical tracker geometry)
- Basket approach: particles are processed in bunches reshuffled after each step
- “ideal vector” transport: particles are processed in bunches without any re-shuffling (“theoretical” best case)
- Comparison to classical navigation
- Measure speedup wrt $N_{threads}$

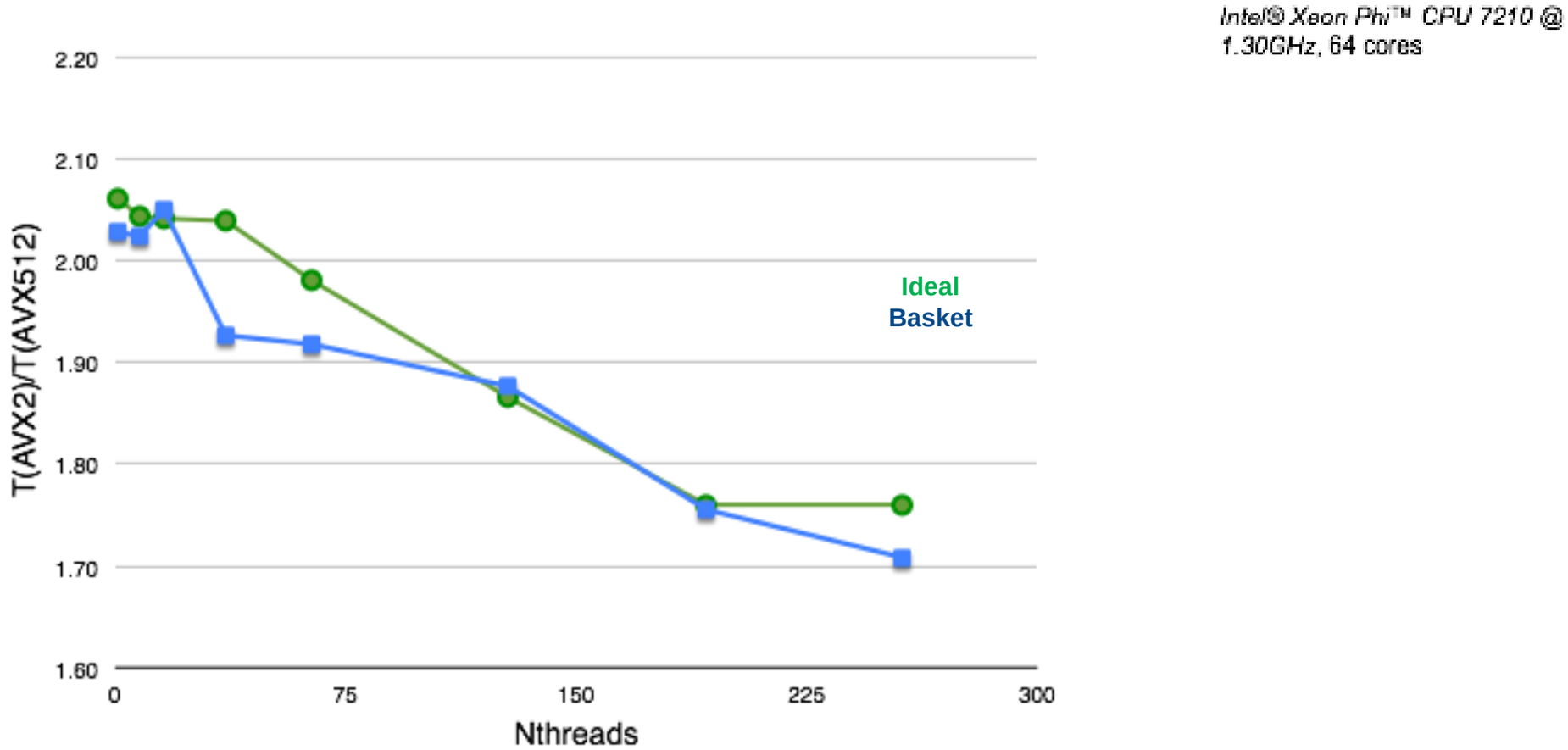


Intel® Xeon Phi™ CPU 7210 @ 1.30GHz, 64 cores



Scalability (II)

High vectorization intensity achieved for both ideal and basketized cases
AVX-512 brings an extra factor of ~2 to our benchmark



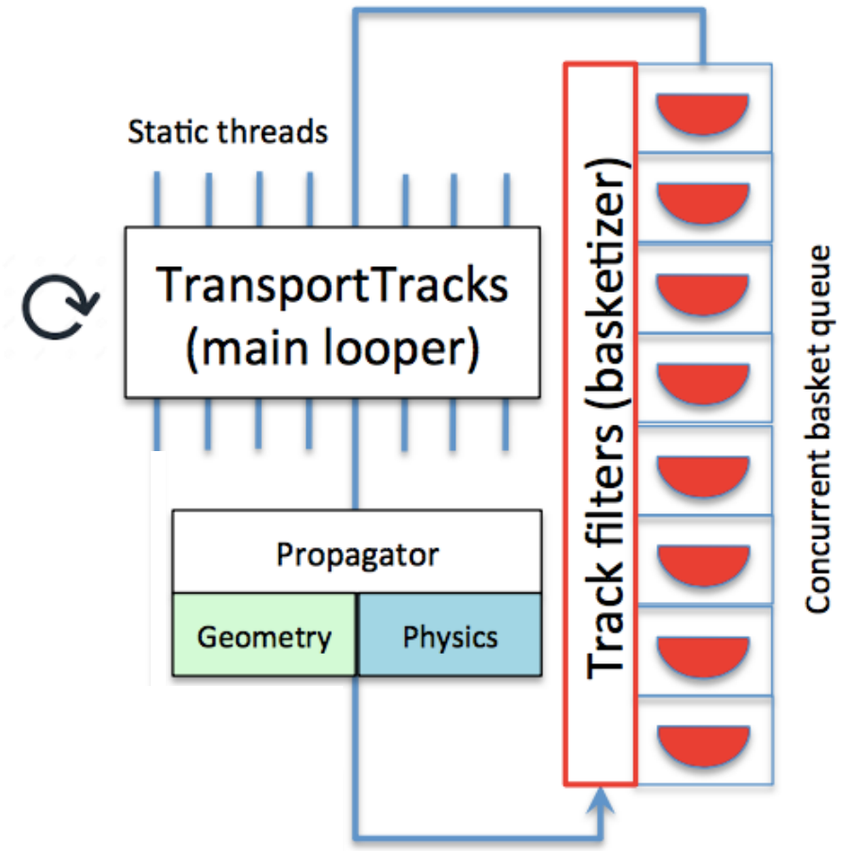
we do understand vectorisation!

Concurrency in GeantV

Investigated different ways of scheduling & sharing work

Initial approach: static allocation of workers

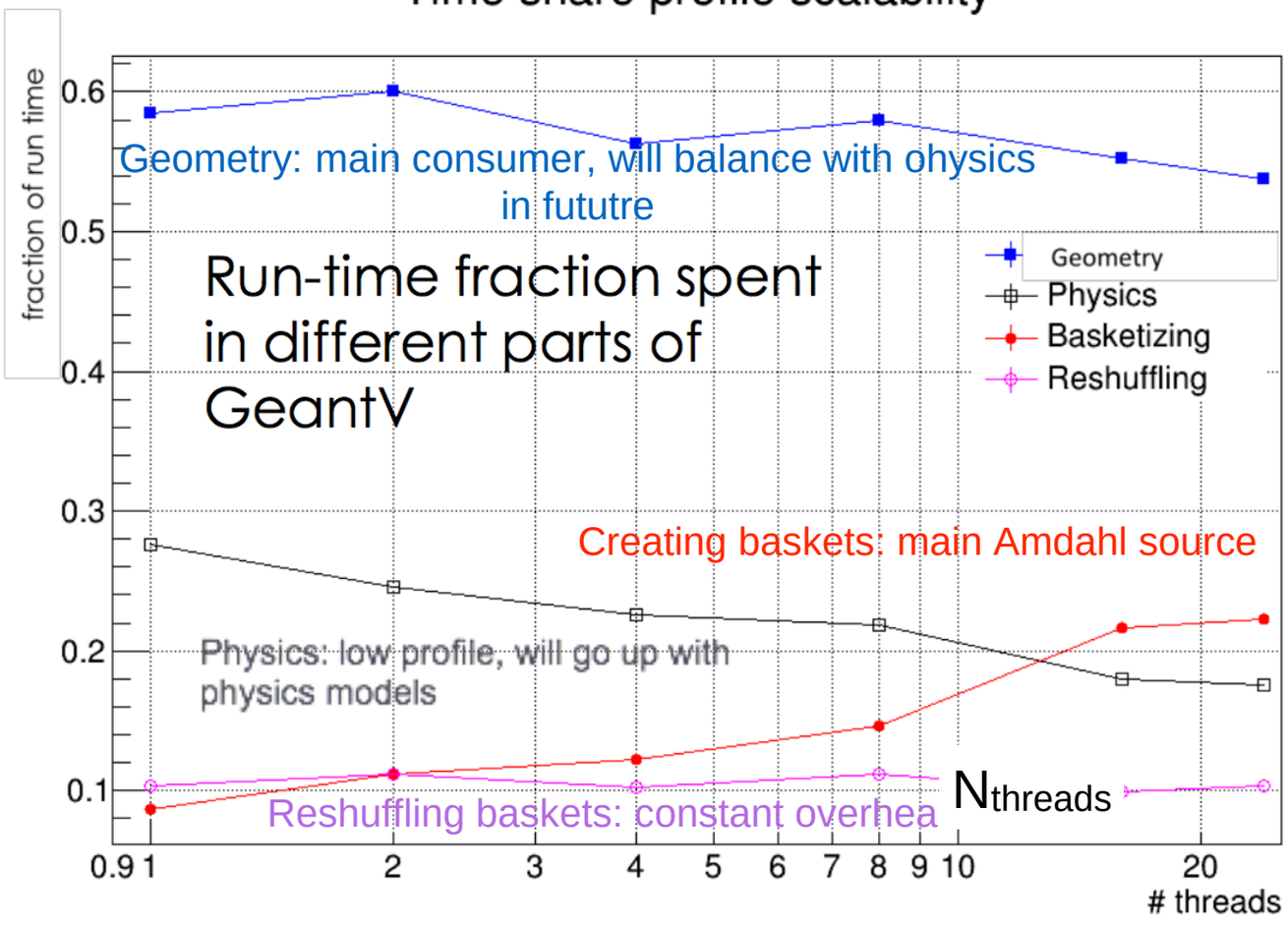
- Main thread method as infinite looper
- Worker threads execute a set of chained tasks (geometry navigation, propagation in the magnetic field, physics processes..)
- Data communication by concurrent queues
  - Main queue of baskets of tracks
  - Secondary queues of transport byproducts (I/O, files, final products)
  - Use SOA



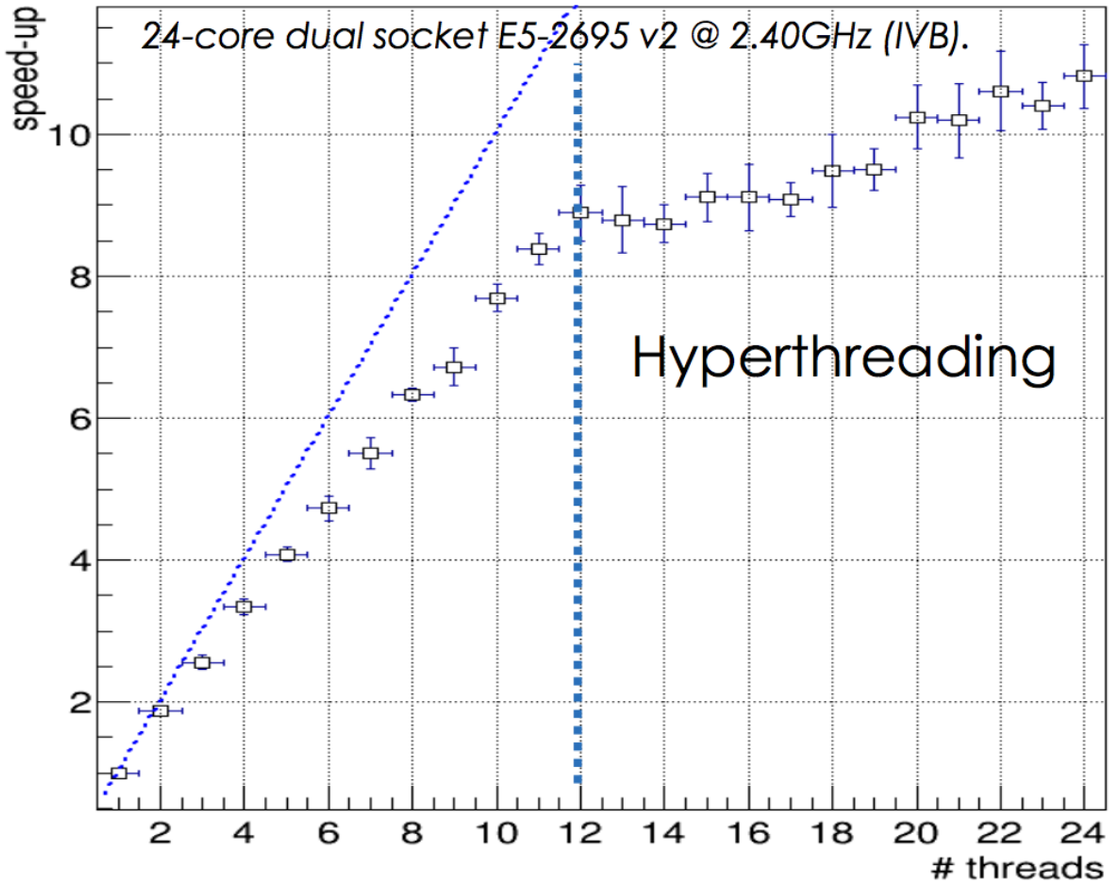
Concurrency in GeantV

Geometry + propagator: main consumer, will balance with physics in future

Time share profile scalability



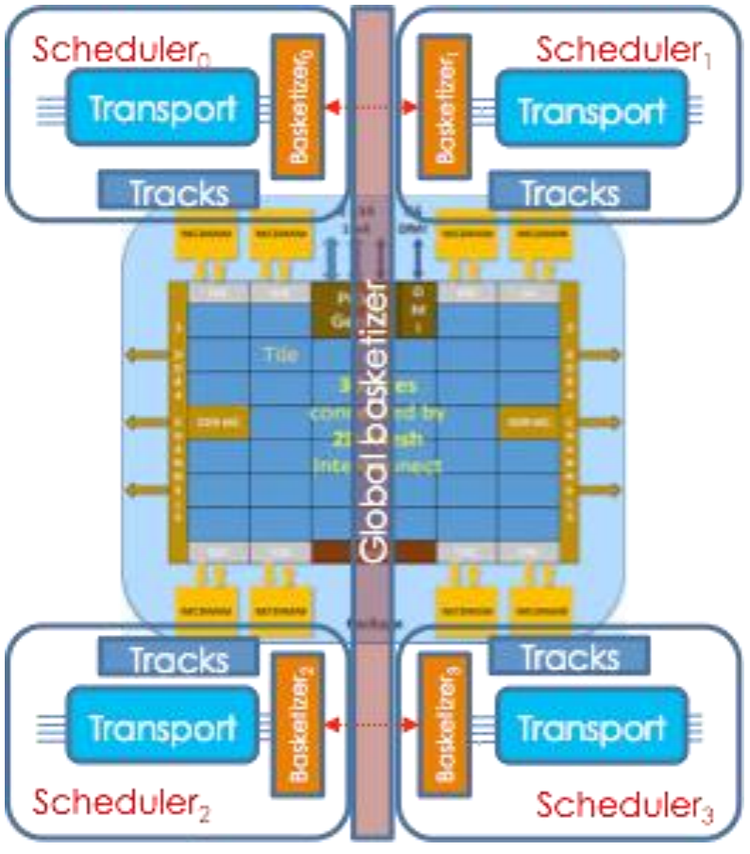
scalability with number of threads



Contention prevents scaling to high number of threads
Issue for many cores architectures!

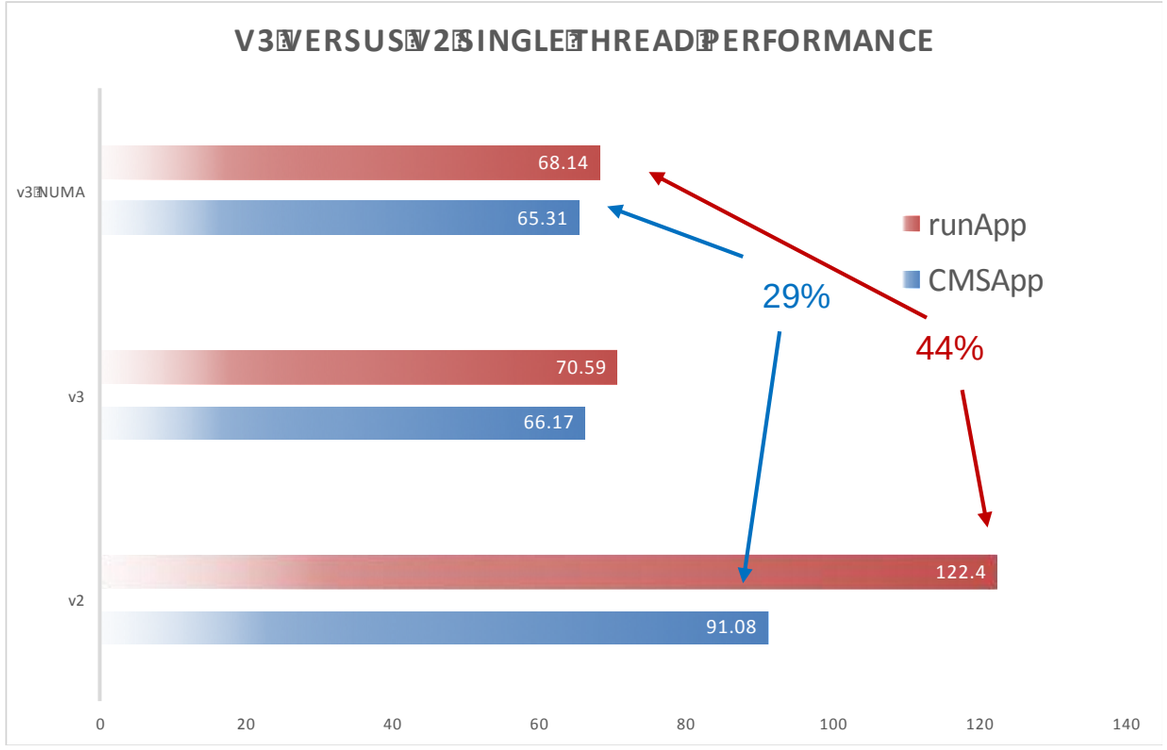
New GeantV workflow

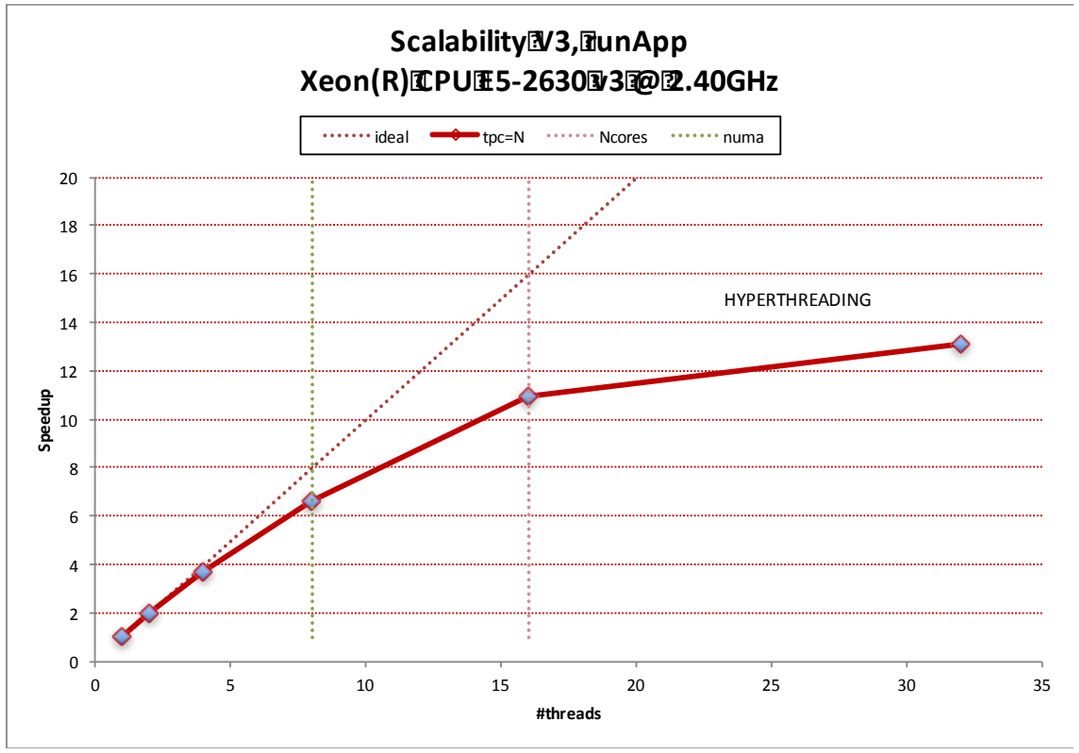
Initial version	New workflow
SOA container: overheads for reshuffling, concurrency	Hybrid system: • AOS handling in basketization, • SOA for dispatching to vector code
Main basket queue: non-local, adding contention points	Thread-local data and containers, relying less on common concurrency services
System-driven allocation of resources (threads, memory)	NUMA-aware allocation of resources, relying on topology discovery
“Avalanche” memory behavior: tracks are never released, the full shower is kept in memory	Smart stack-like behavior, favoring transporting secondaries/low energy tracks with priority



New GeantV workflow

Performance tests





A change of perspective

Many parameters and multiple layers of parallelism: a complex system to tune

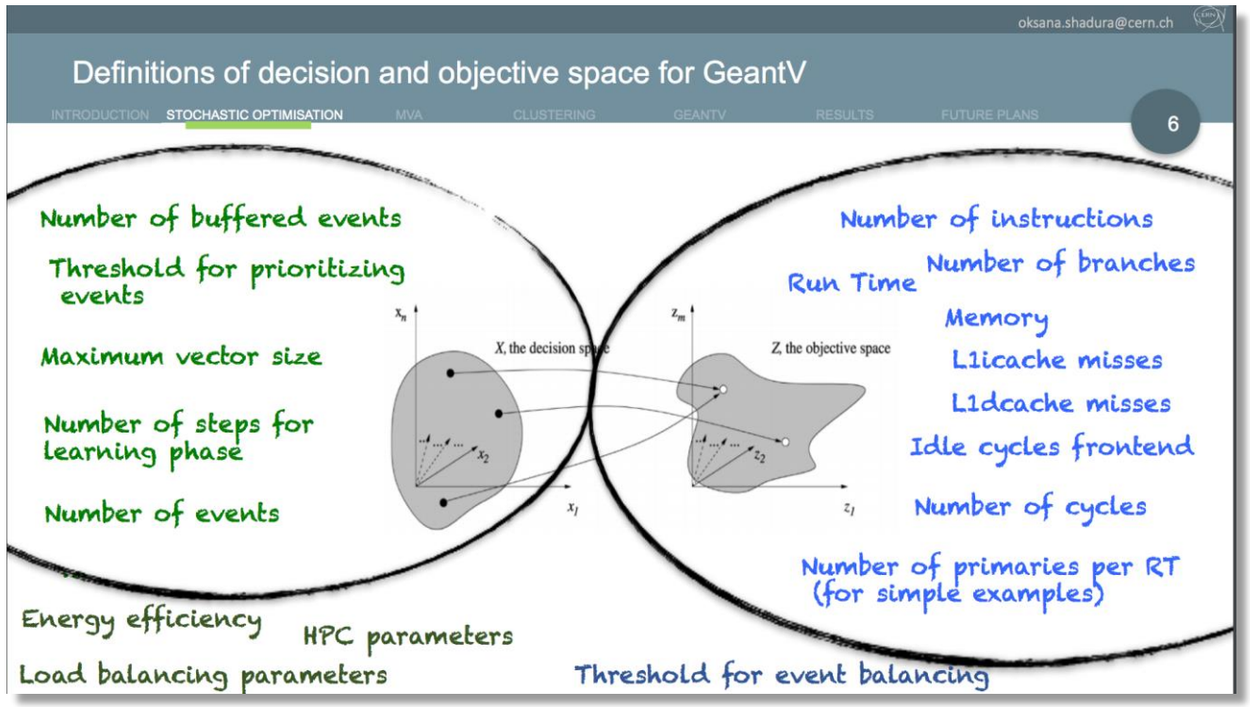
Stochastic optimization of GeantV code by use of genetic algorithms

CHEP 2016

Oksana Shadura

CERN

Performance tuning as a multi-objective optimisation problem



https://indico.cern.ch/event/505613/contributions/2230828/

The end

Summary (I)

What we have done

We started the GeantV project aiming at a x5 speedup wrt current simulation software

- Relied on several techniques leveraging compiler and C++ features
  - Compiler optimisation (& inlining)
  - c++ templating
- Introduced data parallelism and concurrency to profit from the latest advancements in terms of architecture
- Results in terms of vectorisation and scalability are encouraging and call for further optimisation
  - Multi-node
  - …

Summary (II)

What you should know now..

- Why we worry about performance
- How to approach the problem of improving performance
- Basic concepts of data and task parallelism
  - Concurrency, Memory related programming models, Vectorisation
- A real life example

Summary (III)

What I did not talk about..

- Details on memory management, bandwidh, cache usage
- Scaling through many nodes (messaging, resource sharing, I/O)
- Portability vs performance

Conclusions

Improving code performance is an “epic fight”

There is no pre-defined “improving performance algorithm”

There is a large variety of methods and strategies (including machine learning and genetic algorithms) to use so..

..use your brain!

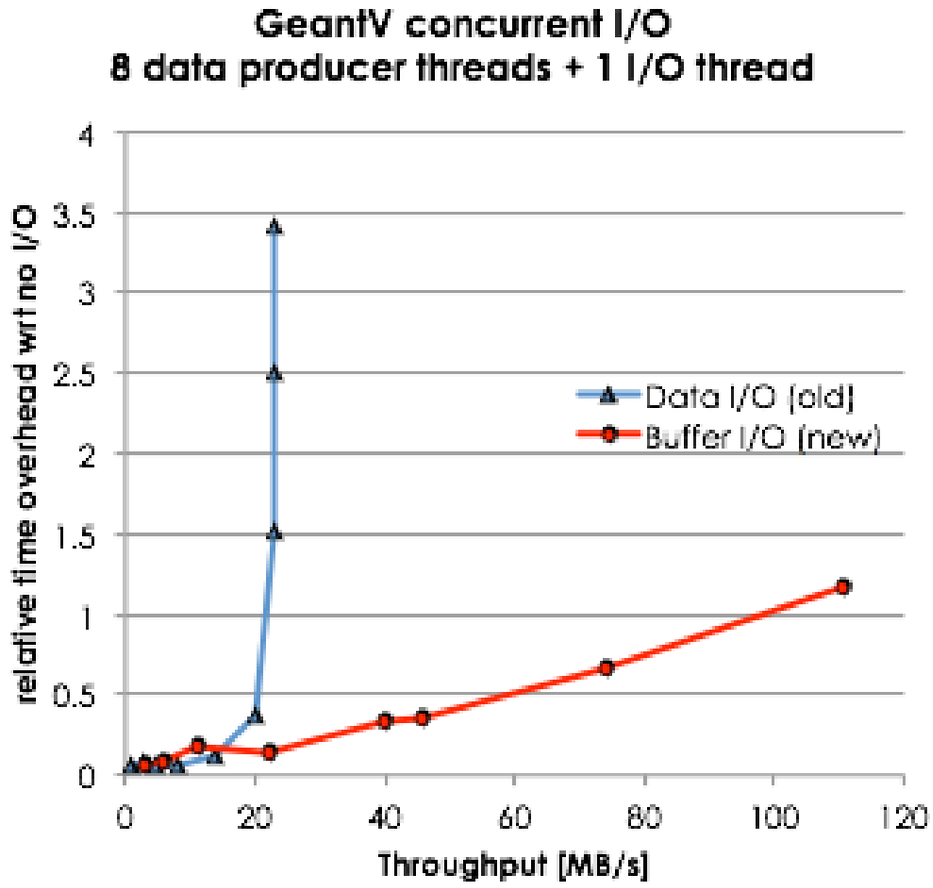
THE END

Thank you!

Removing bottlenecks: I/O

- Physics simulation produces ‘hits’ i.e. energy depositions in detector sensitive parts
- Hits are produced concurrently by all the simulation threads
- Thread-safe queues handle asynchronous generation of hits by several threads
- Dedicated output thread transfers the data to storage

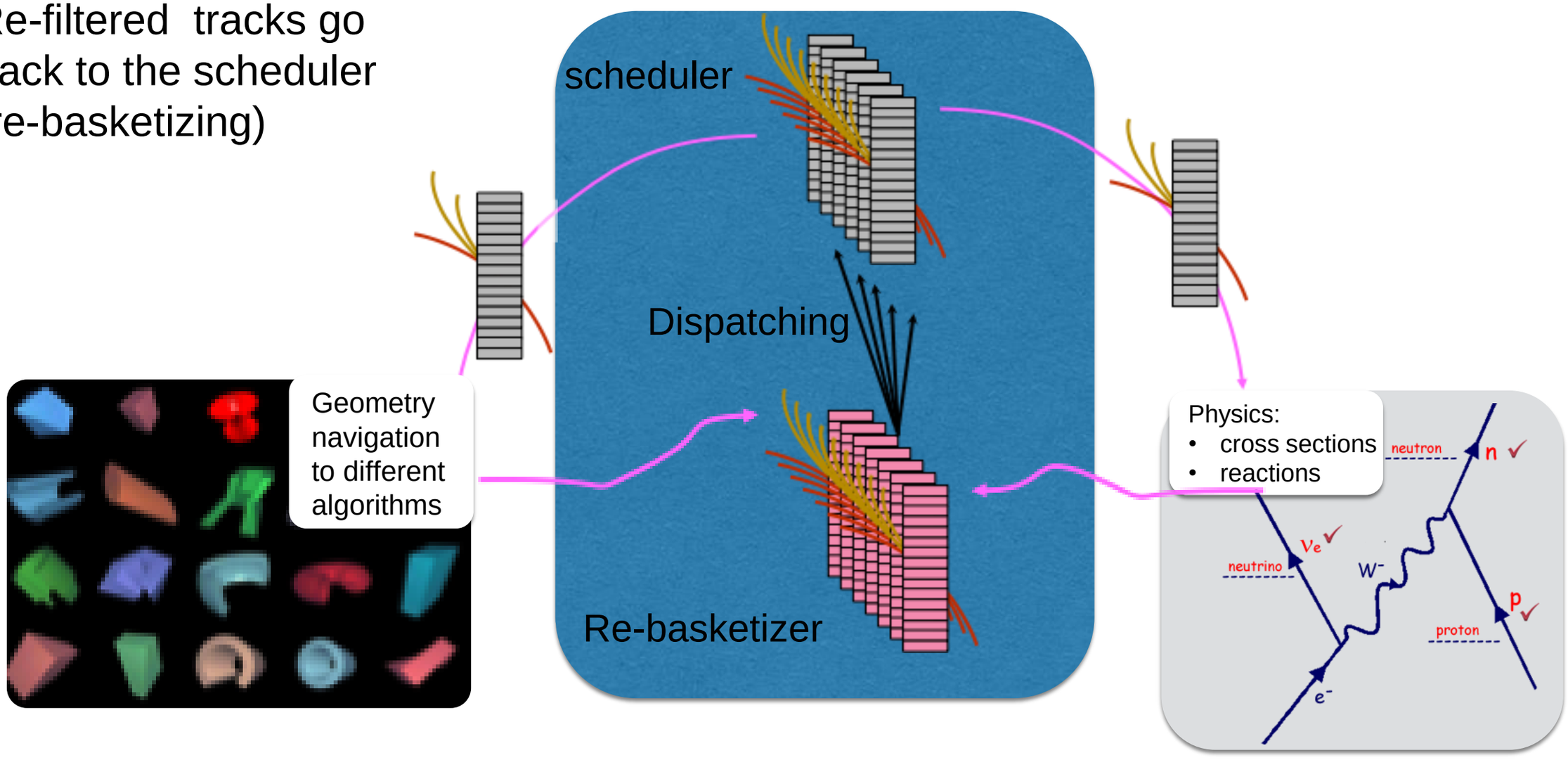
- First implementation:Send concurrently data to one thread dealing with full I/O
- Buffer mode: Send concurrently local hits connected to memory files produced by workers to one thread dealing only with final merging/writing to disk



GeantV: scheduler

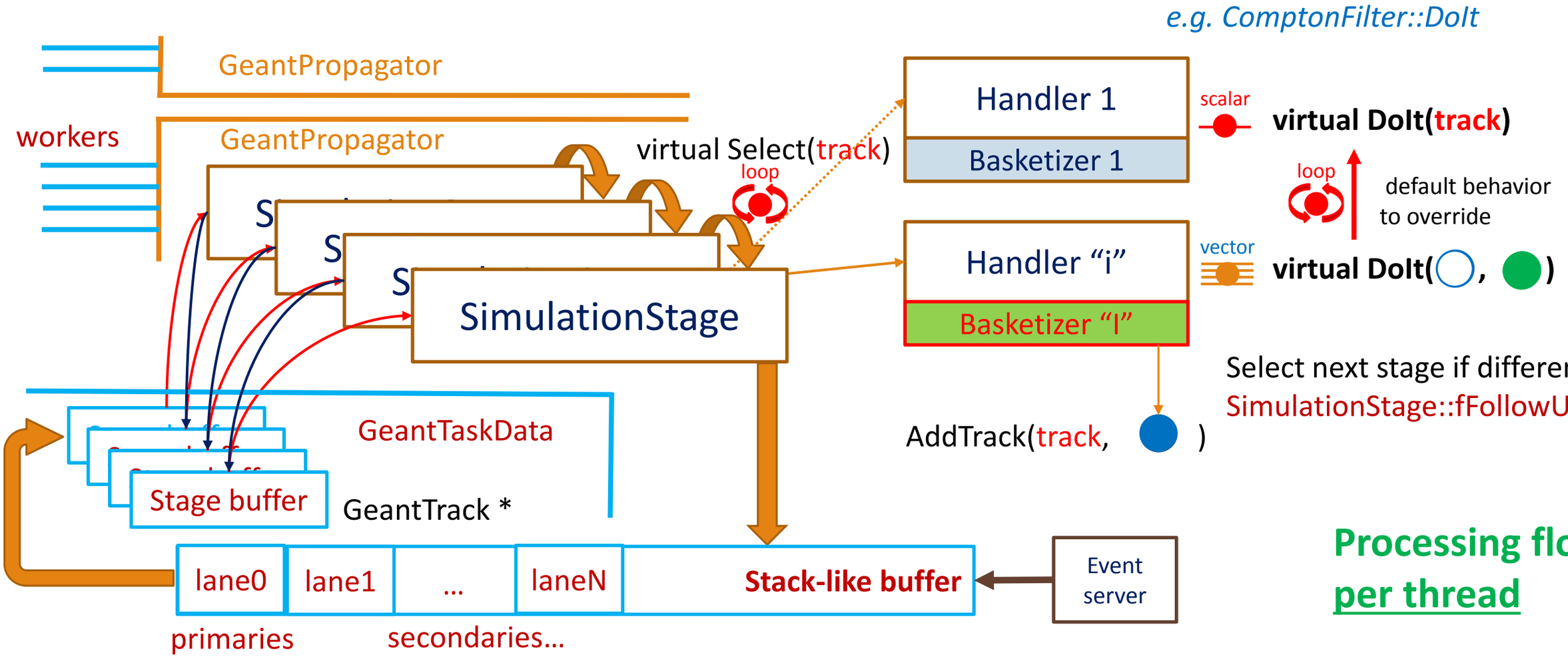
After each step particles move on to different fates → need re-filtering!

Re-filtered tracks go back to the scheduler (re-basketizing)



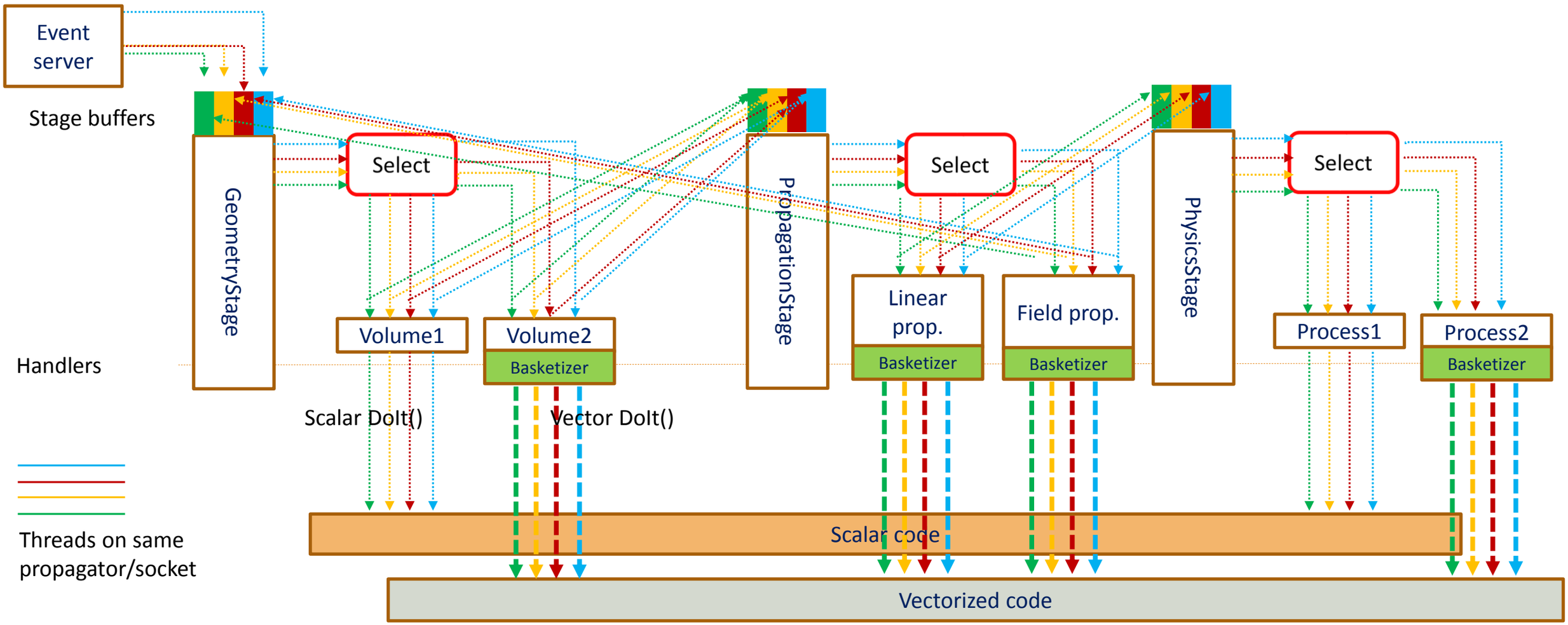
- Overhead should be much smaller than locality/SIMD gains
- portable without hindering performance

GeantV version 3: A generic vector flow approach

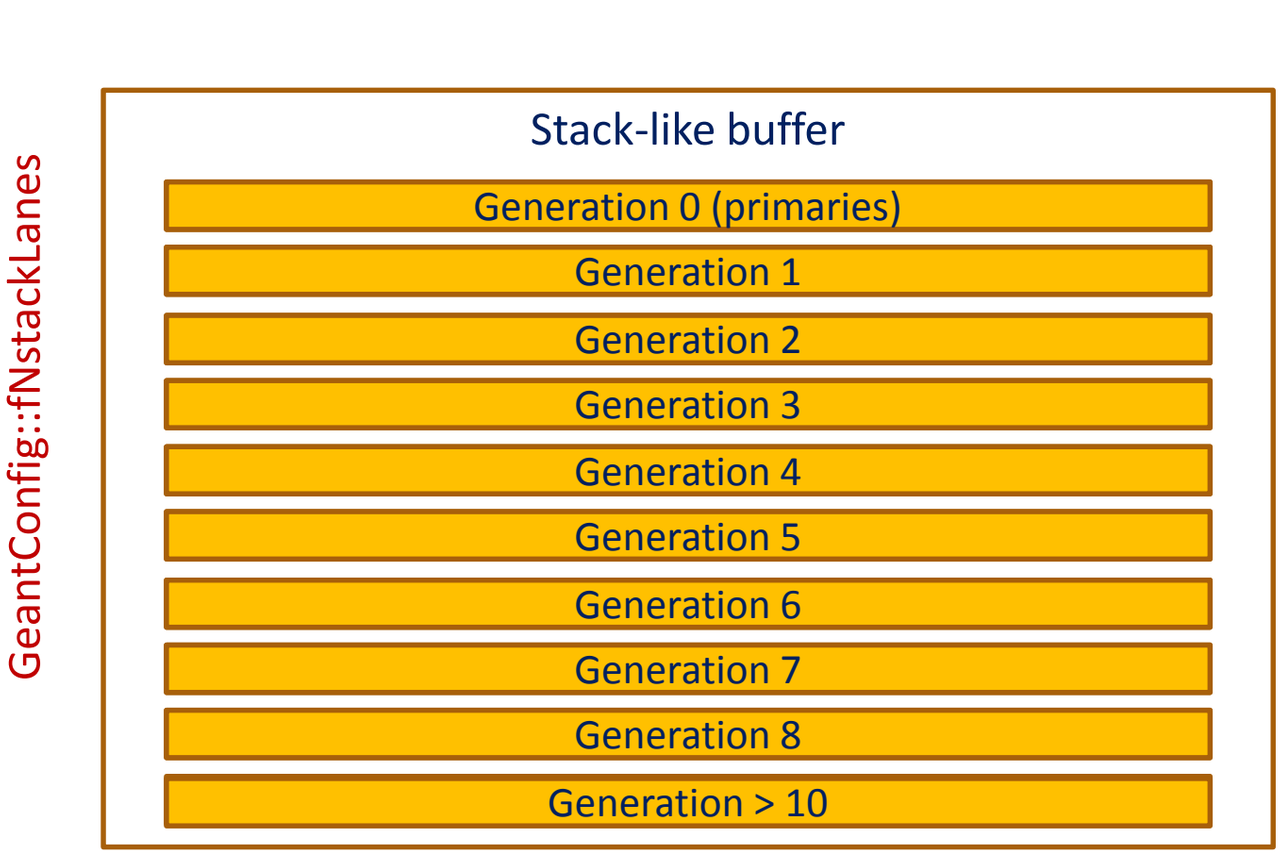


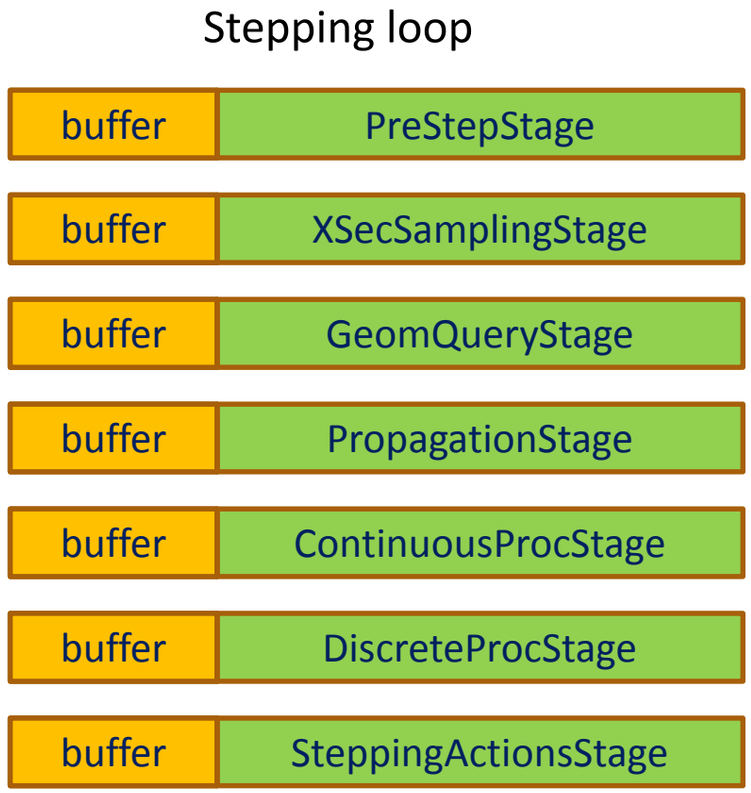
Processing flow per thread

Processing flow per propagator/NUMA node



Stack-like handling of tracks





Number of lanes flushed into the stepping loop controlled by: GeantConfig::fNmaxBuffSpill

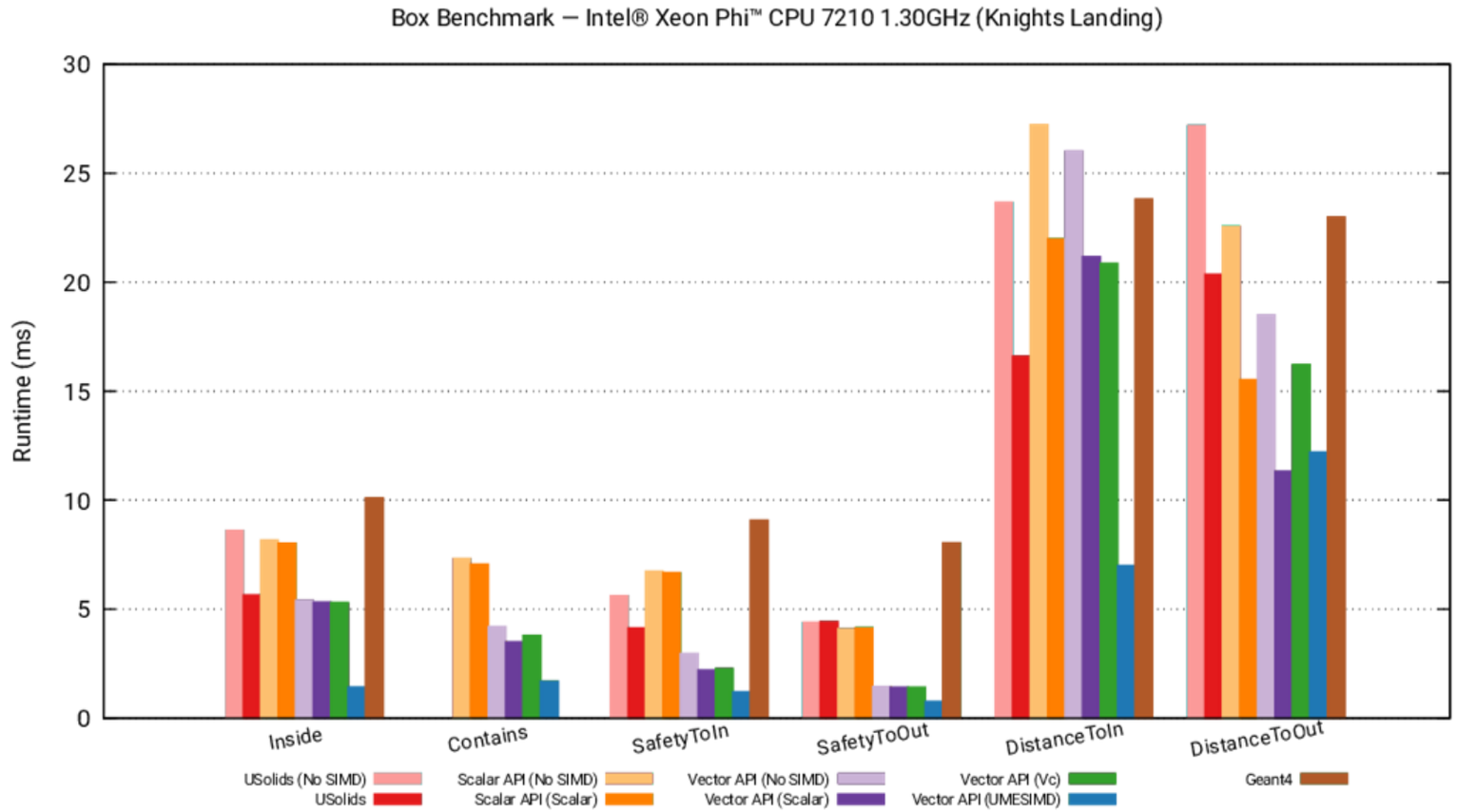
VecGeom Benchmarks on Intel® Xeon Phi™ (KNL)

Everything was compiled with Intel C/C++ compiler 16.0.3

Used “-O3 -xMIC-AVX512”

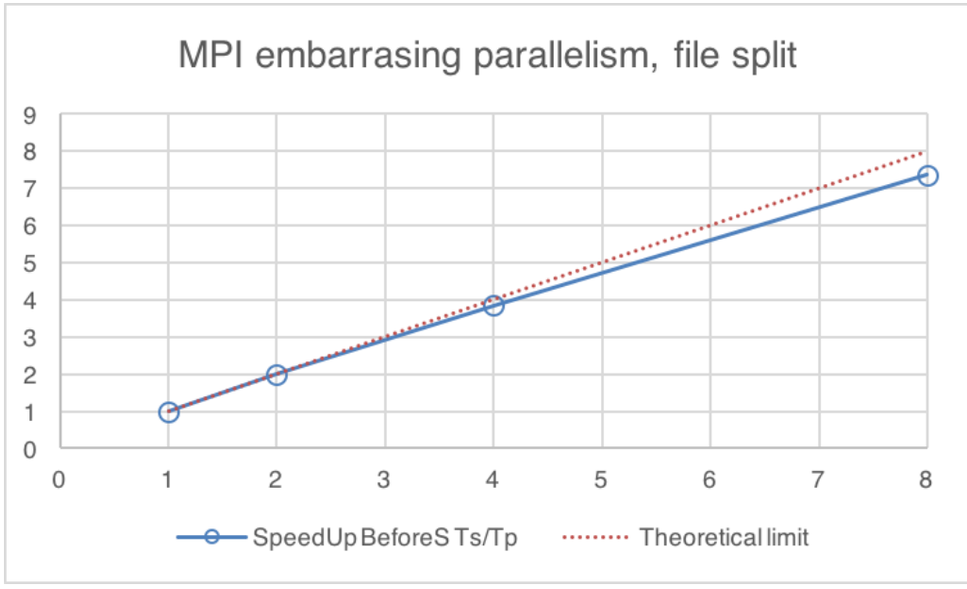
Contrary to AVX2 benchmarks on Skylake, UME::SIMD gives best performance on Knights Landing

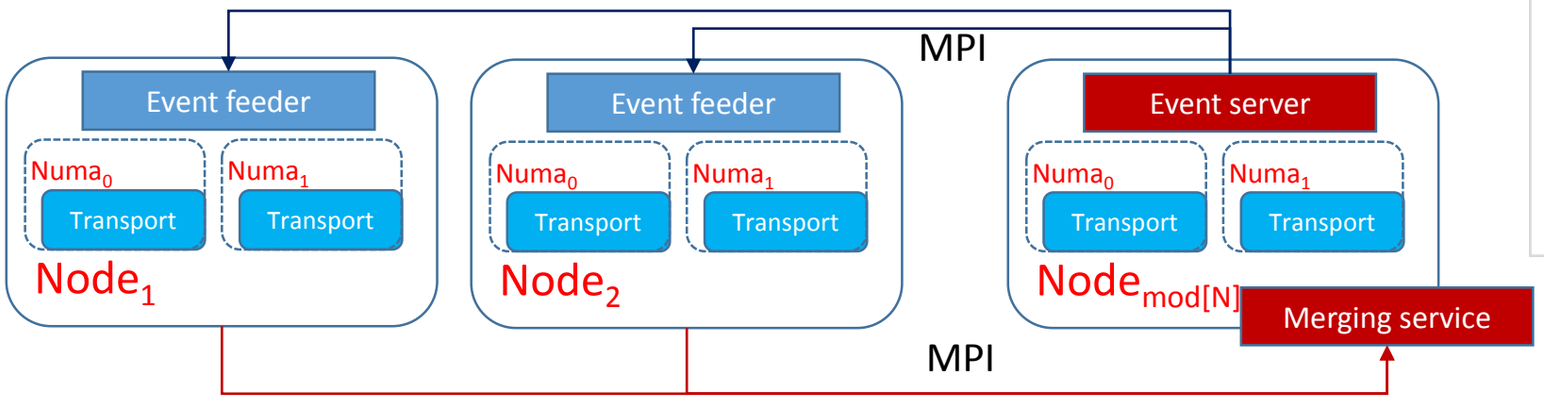
Scalar code under Vector API shows auto-vectorization in many cases

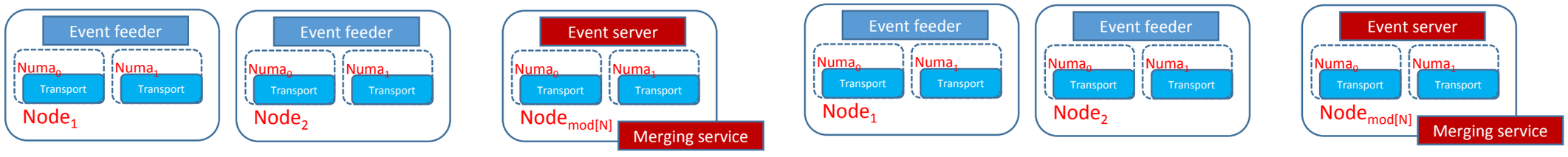


GeantV plans for HPC environment

- Standard mode (1 independent process per node)
  - Always possible, no-brainer
  - Possible issues with work balancing (events take different time)
  - Possible issues with output granularity (merging may be required)
- Multi-tier mode (event servers)
  - Useful to work with events from file, to handle merging and workload balancing
  - Communication with event servers via MPI to get event id's in common files







Virtual vs template

Virtual inheritance: one of the most powerful features of C++
Allow for maximum flexibility
Separation of interface and implementations: clean code
Unified treatment of components behind the same interface
Comply to interfaces: easy mixing of components
E.g. Library developer provides interfaces, user complies to them when writing implementations

```
class ISolid{
public:
virtual bool IsInside(const Particle&) = 0;
virtual double DistanceToBoundary (const Particle&) = 0;
};
```

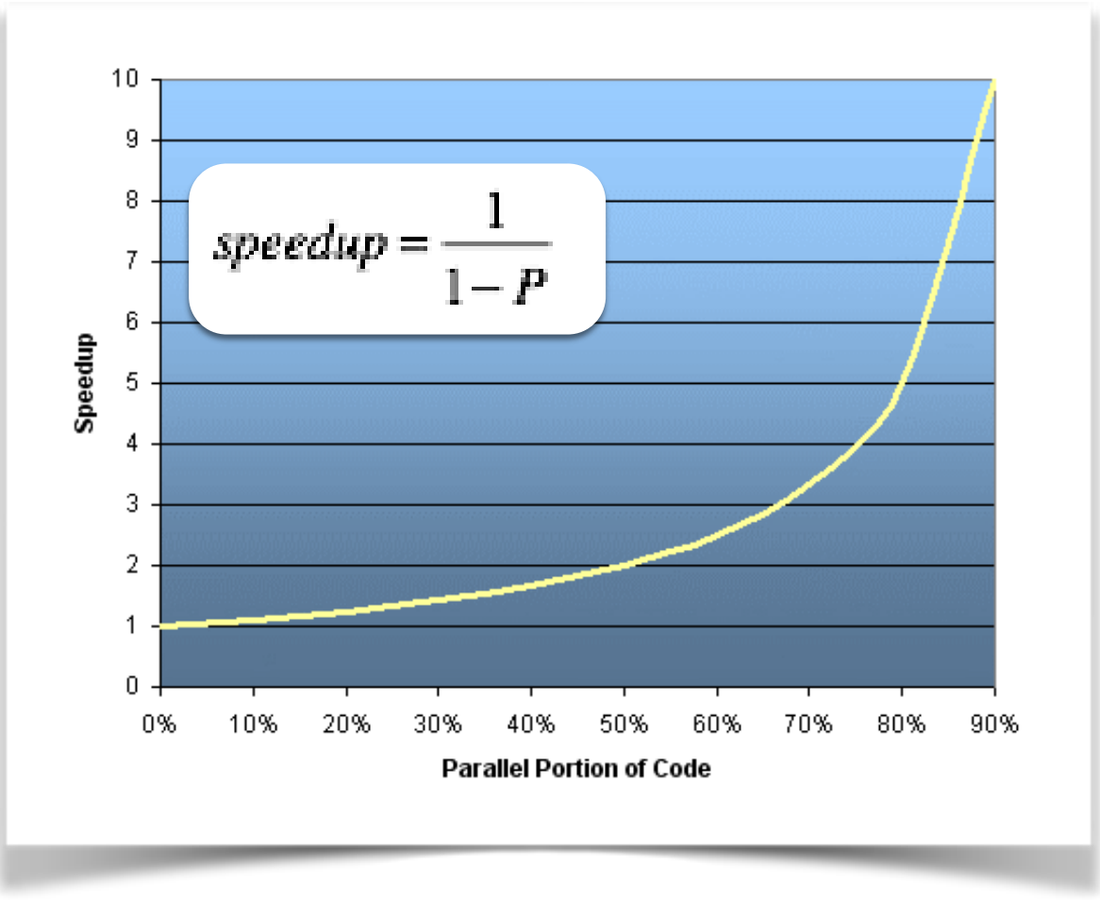
```
Class Cube: public ISolid {
public:
bool IsInside(const Particle&){…};
double DistanceToBoundary (const Particle&){…}
};
```

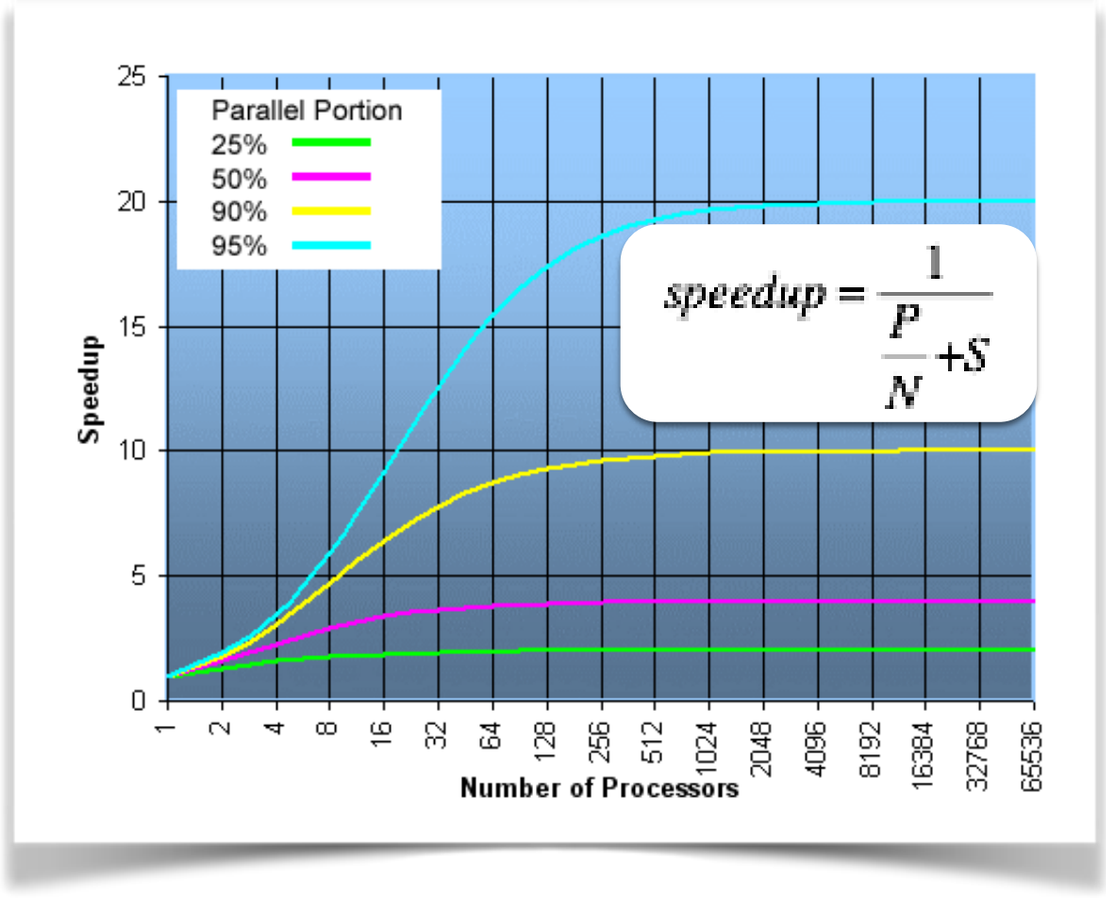
```
Class Sphere: public ISolid {
```

```
Class Cylinder: public ISolid {
public:
bool IsInside…
double DistanceToBoundary …
};
```

Amdahl's law

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





It tells us something about parallel execution: It states the maximum speed up achievable given a certain problem of FIXED size and serial portion of the program.

Option B: “convince the compiler”

inline and remove early returns

not enough!
no vectorisation

```
void contains_v3( const double * point, bool * isin, int np ){
  for( unsigned int k=0; k < np; ++k){
      for( unsigned int dir=0; dir < 3; ++dir ){
      if ( fabs ( point[3*k+dir]-origin[dir] ) > boxsize[dir] ) isin[k]=false;
    }
     isin[k]=true;
}}
```

Intermediate local variables
+ if conversion

not enough!
no vectorisation

```
void contains_v4( const double * point, bool * isin, int np){
 for( unsigned int k=0; k < np; ++k){
    bool tmp[3]={true, true, true};
    for( unsigned int dir=0; dir < 3; ++dir ){
     tmp[dir] = fabs ( point[3*k+dir]-origin[dir] ) > boxsize[dir]; }
     isin[k]=tmp[0] & tmp[1] & tmp[2];
}}
```

Option B… continued

AOS to SOA

```
typedef struct {
  double *coord[3];
} P;

void contains_v6( const P &  point, bool * isin, int np ){
 for( unsigned int k=0; k < np; ++k){
     bool tmp[3];
     for( unsigned int dir=0; dir < 3; ++dir ){
       tmp[dir] = (fabs (point.coord[dir][k]-origin[dir]) > boxsize[dir]);
     }
     isin[k]=tmp[0] & tmp[1] & tmp[2];
}}
```

manually unroll loops

```
void contains_v_autovec( const P & points, bool *  isin, int np ){
  for (int k=0; k < np; ++k)
  {
      bool resultx=(fabs (point.coord[0][k]-origin[0]) > boxsize[0]);
      bool resulty=(fabs (point.coord[1][k]-origin[1]) > boxsize[1]);
      bool resultz=(fabs (point.coord[2][k]-origin[2]) > boxsize[2]);
      isin[k]=resultx & resulty & resultz;
}}
```