Improving code performance: an introduction

Practical examples from particle physics simulation

Sofia Vallecorsa

sofia.vallecorsa@cern.ch

2016 Openlab Summer Students lecture series

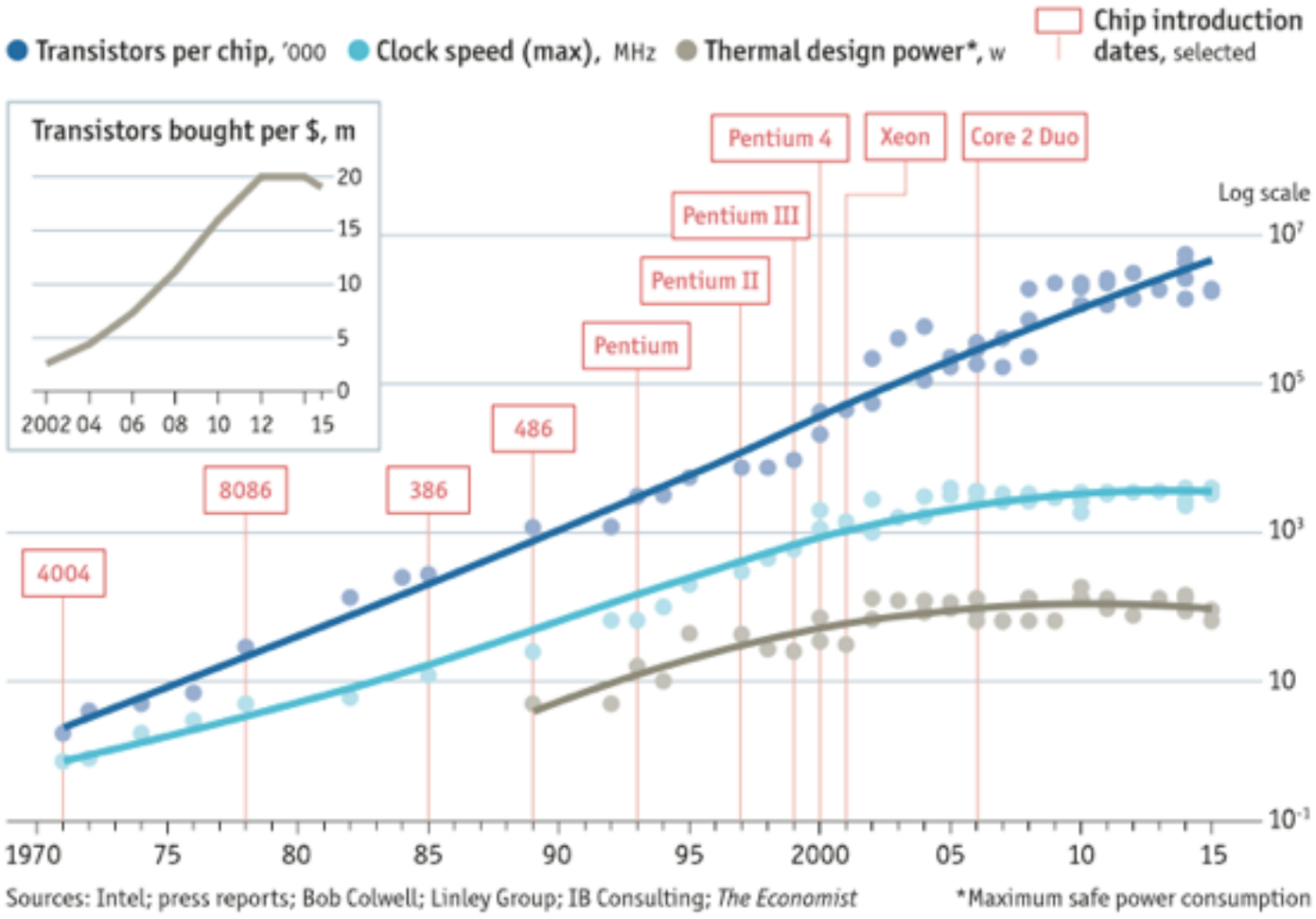
Outline

- 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 now!

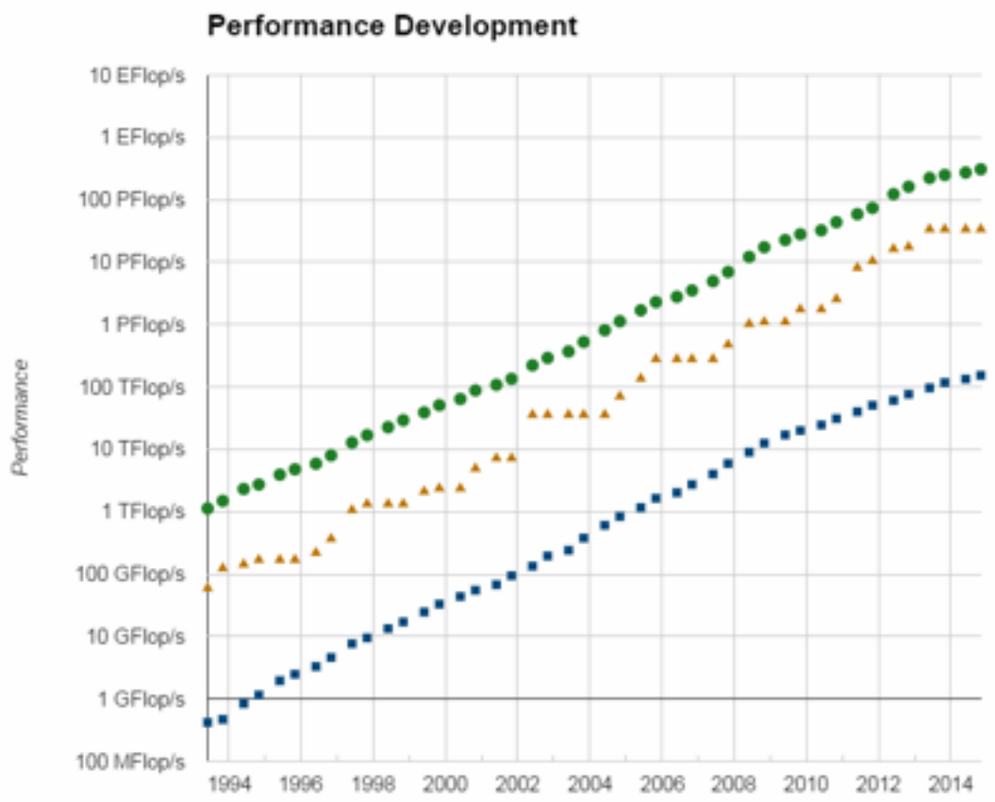
The era of supercomputers

Ever faster networks, distributed systems, and multi-processor architectures show that **parallelism is the future of 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



Need to think parallel!

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	Oak Ridge National Laboratory	Cray	Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x	USA	560,640	17.6	8.21

Performance

Is there A definition?

- **Timing**: faster execution
  - CPU time, latency,…
  - Speedup (parallel vs serial execution)
- **Amount of processed data**: throughput
- **Size**: smaller executable, smaller memory footprint
- …and “the holy grail”… **forward scalability**:
  - Maximum performance from today’s hardware should scale on future processors/accelerators
  - Automatically - with virtually no code rewriting
- **Good scaling** if:
  - x2 number of cores (or vector size) doubles performance

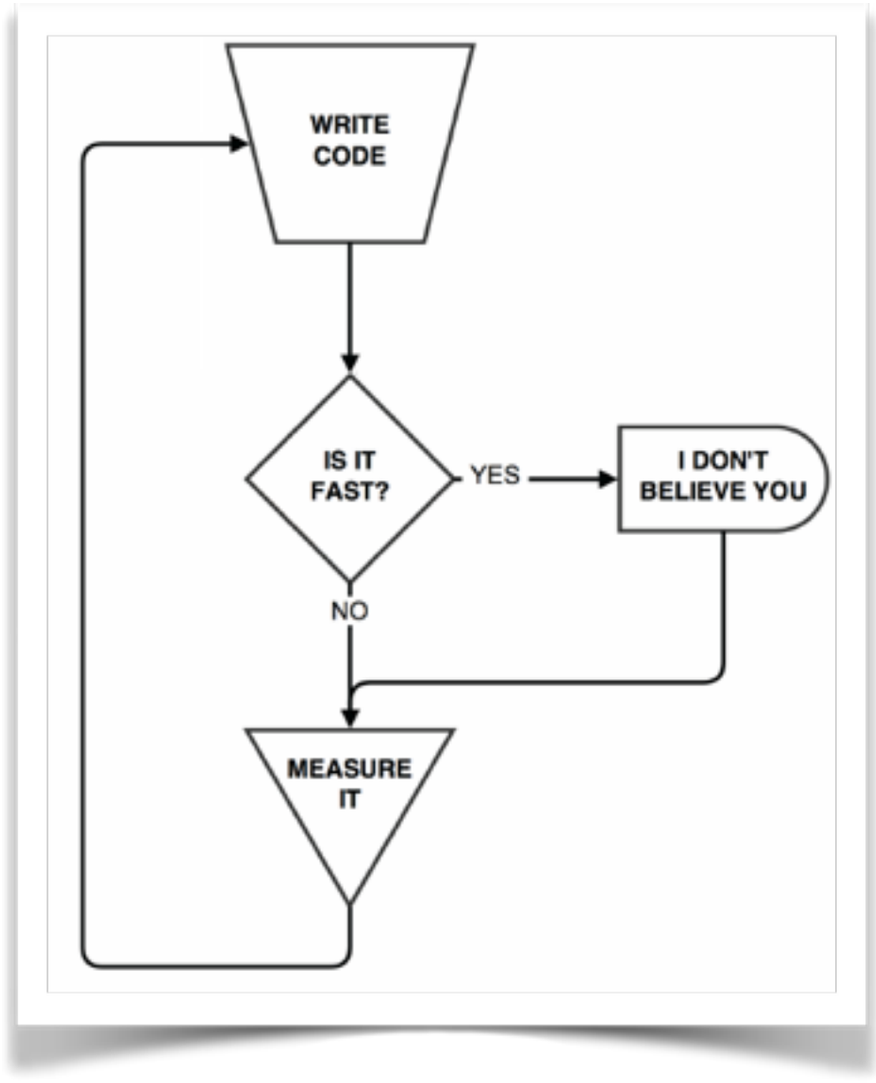
Improving performance is a tradeoff!!
- **Timing** vs. **Size**
- **Compilation speed** and **memory**
- **Latency** vs **throughput**

Measuring performance (I)

"First catch the rabbit"

a recipe for rabbit stew

- Before any optimisation we need a way to measure what we are optimising
- Before any measurement we need a clear, explicit statement of the problem to solve.



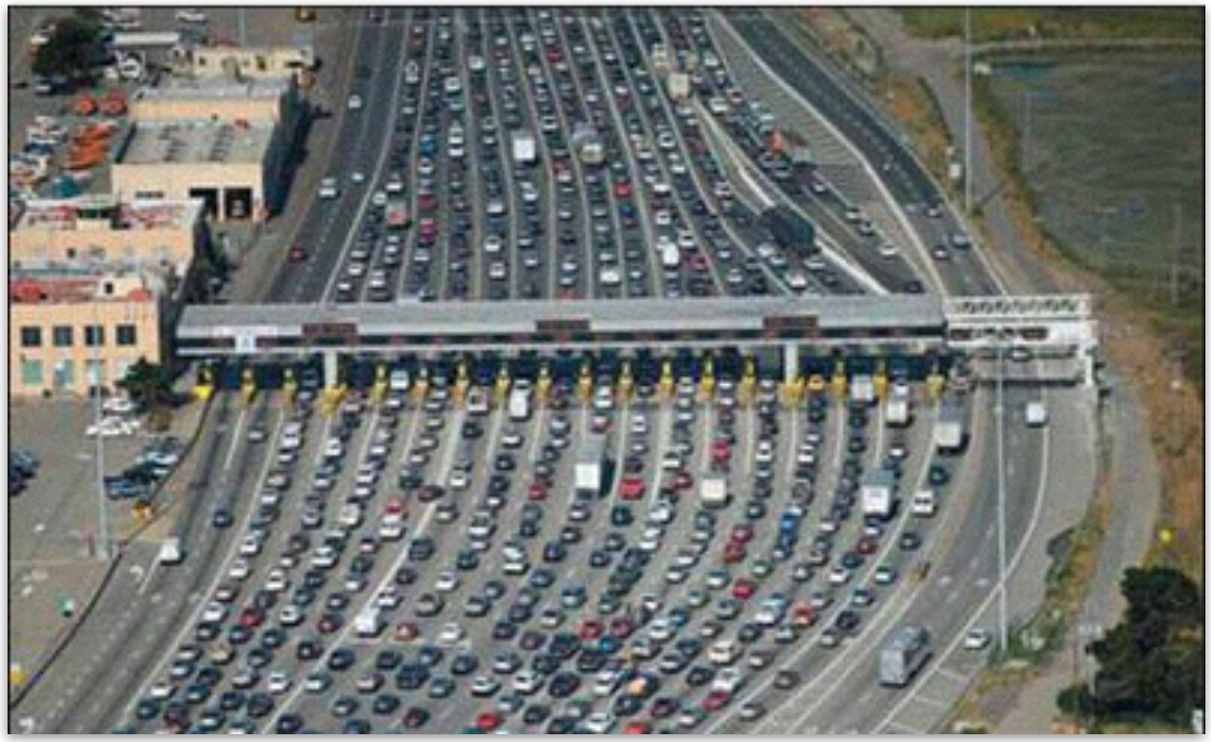
➥ A good understanding of the hardware

➥ Reproducible, representative benchmarks

➥ "The right" tool

➥ **Time (!):** Performance optimisation is a process that may require several iterations

Measuring performance (II)



Identify hotspots:

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

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

Identify bottlenecks:

Areas that are disproportionately slow, or cause parallelizable work to halt or be deferred (e.g. I/O)

Restructure or change algorithm to reduce or eliminate unnecessary slow areas

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"

Profiling techniques

Statistical Sampling:

Program flow is periodically interrupted, current program state is examined.

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

Code Instrumentation:

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

Profiling techniques

Statistical sampling advantages:

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

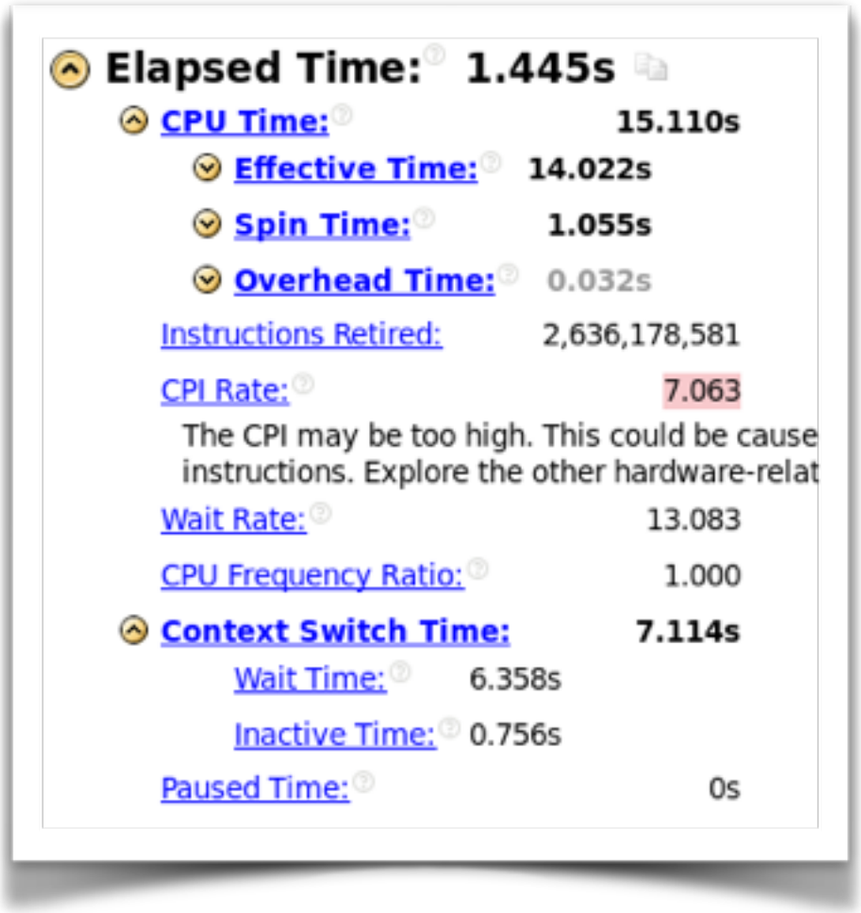
Statistical sampling limitations:

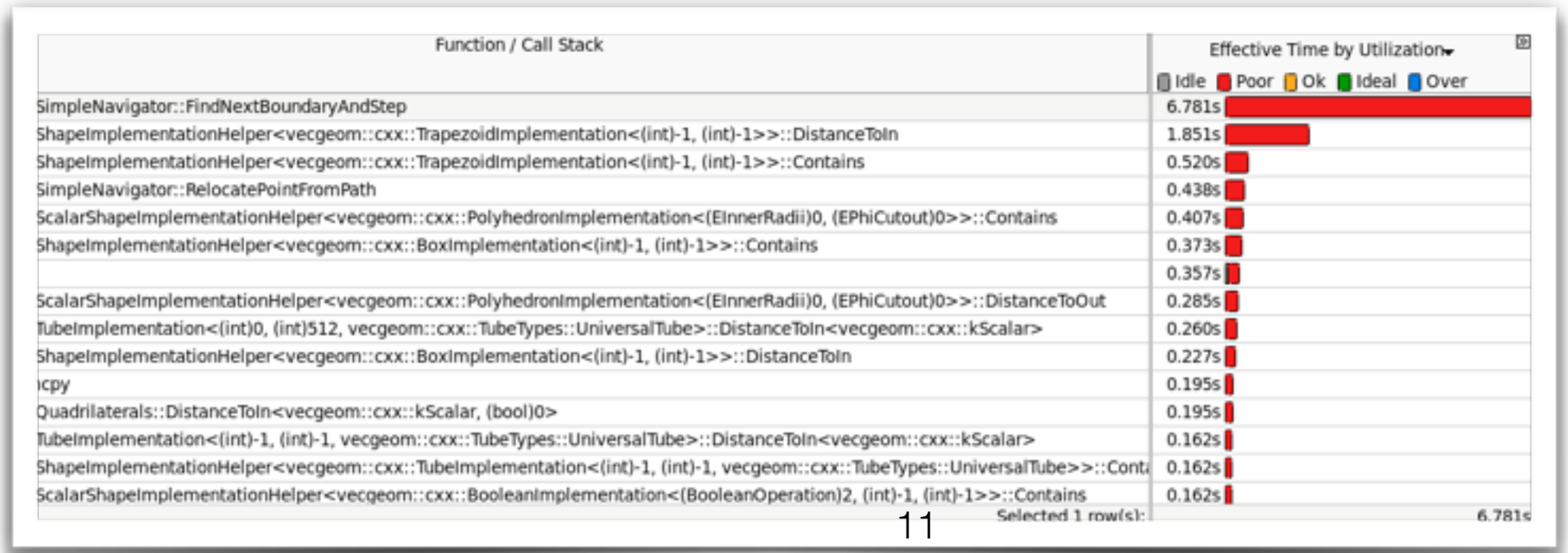
- Statistical sampling involves some degree of uncertainty
  - Information attributed to source lines may not be accurate
- Certain types of information not available:
  - Number of calls of a certain function
  - Average runtime per call of a certain function

Code instrumentation

Some profiling tools

- **VTune, Advisor** – Intel products, very powerful, include multi-threading analysis and vectorisation
- **gprof:** GNU, Flat profiles, call lists, Recompilation needed
- **PIN, Valgrind:** Instrumentation / Synthetic software CPU, Simulate such characteristics as 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
- Hardware threading
- Instruction Level Parallelism
  - Instruction pipelining
- Vector registers

Task/Process parallelism:
- split load into "**baskets of work**" consumed by a pool of resources
- need to check inter-dependency

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

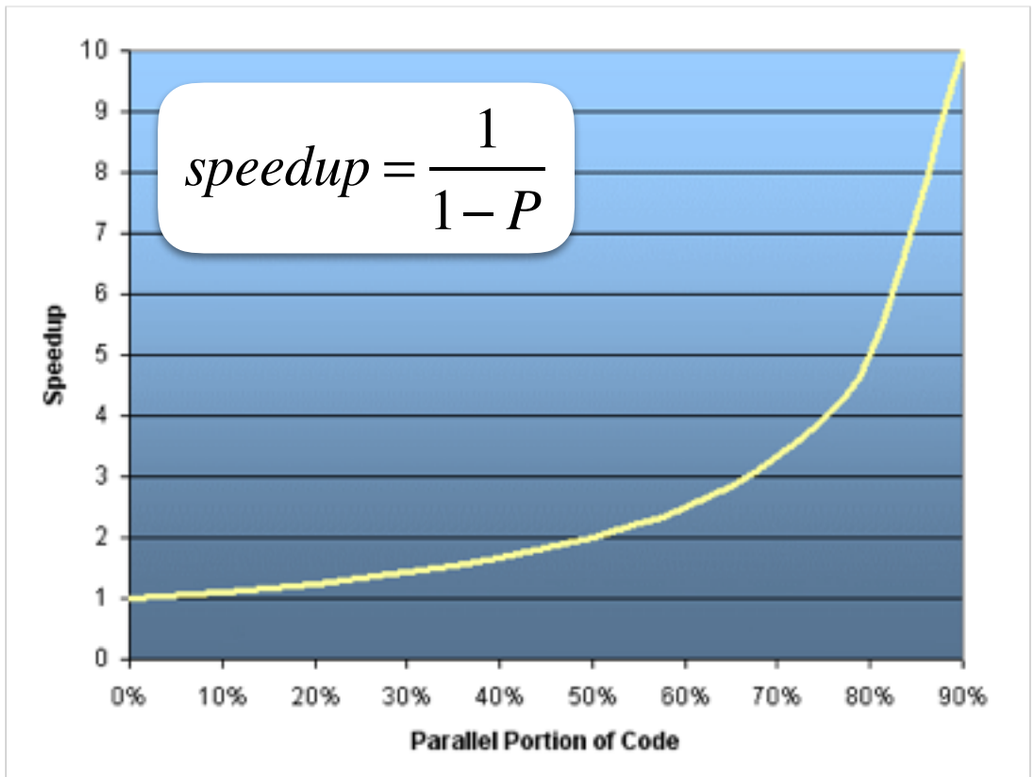
Multi-dimensional improvement

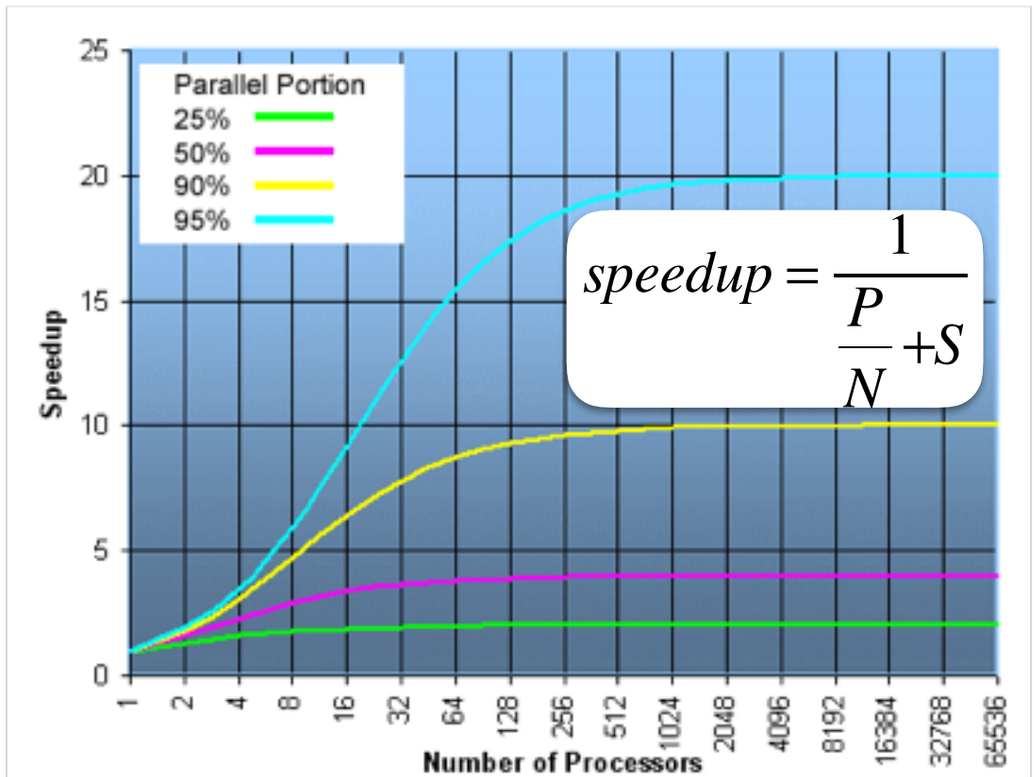
Which direction?

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

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.

Coming up next…

- Introduction to task parallelism
- Memory related programming models
- 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

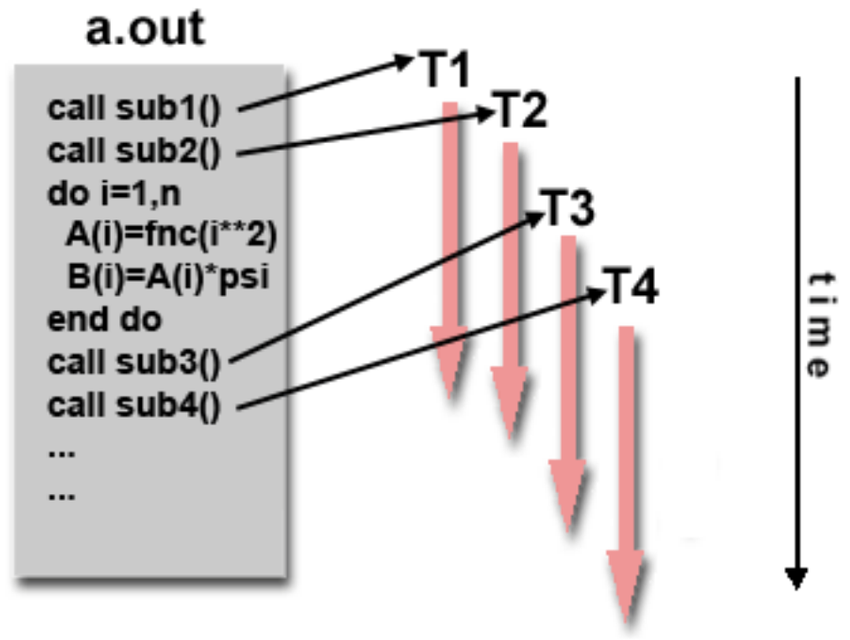
- **Thread**: light-weight process within process
  - share memory with other threads belonging to same process
  - Managed and scheduled by the kernel according to available resources
  - Many options available:
    - C++11 std::thread
    - OS: pthreads (linux)..
    - Libraries: OpenMP …

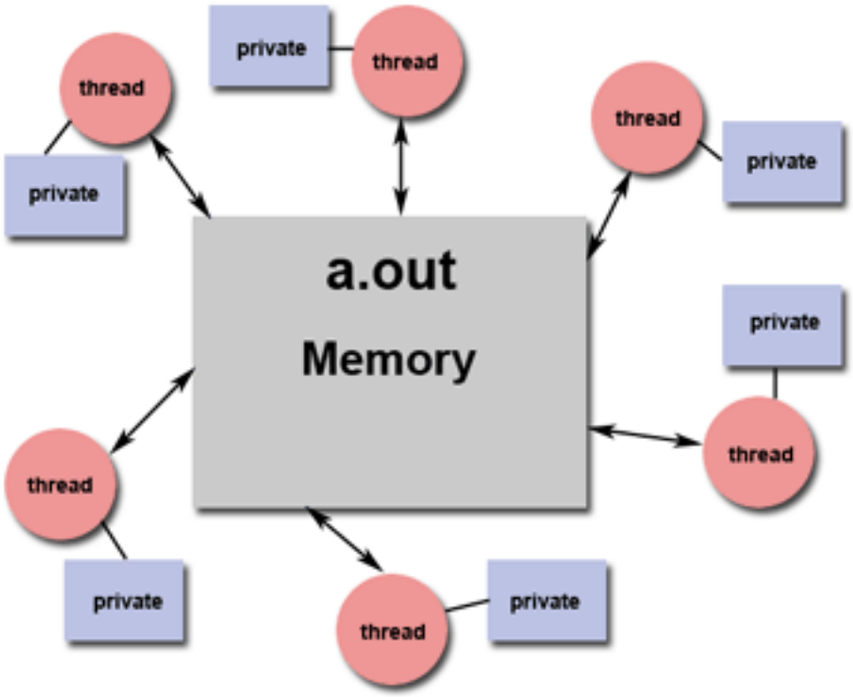
- **Task:** Logically discrete section of computational work. Typically a program-like set of instructions executed by a processor.

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
- Each thread has local data and shares the **common resources** (to avoid replication)
  - Threads communicate by updating global memory address locations
  - **Synchronisation** ensures that two threads do not update same global address at any time.
- Threads can come and go, but a.out remains present to provide necessary shared resources until the completion.

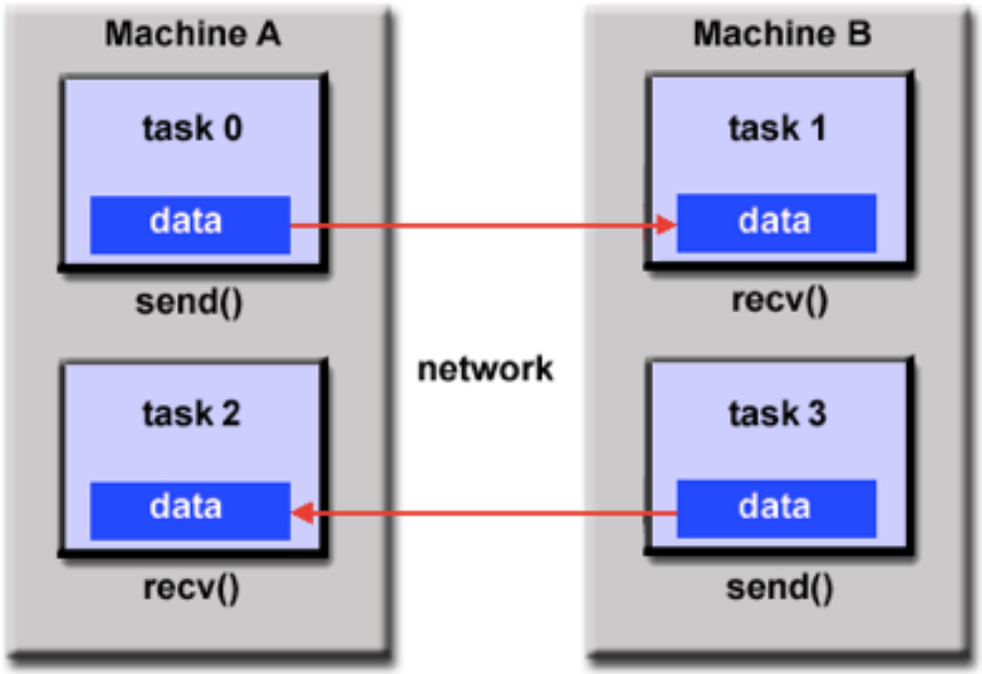




Distributed/Hybrid memory models

Distributed memory: Tasks use own local memory (same and/or across many physical machines)

- Tasks exchange data through communications by sending and receiving messages
- Message passing through a library.e.g. Message Passing Interface (MPI)



Hybrid memory: combines more than one programming model. e.g: message passing model (MPI) & threads model (OpenMP).

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

Here the underlying hardware network communication speed & bandwidth do matter!

Designing parallel code (I)

- **Understand the problem**: can it actually be parallelised?
  - Identify inhibitors to parallelism (e.g. data dependence)
  - Change the algorithm, check external libraries
- **Partitioning:** 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
```

Designing parallel code (II)

- **Synchronisation:** Managing the sequence of work is critical!
  - **Barriers**: Each task works until the barrier, then stops.
    - When the last task reaches the barrier, all are synchronised.
  - **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**

traffic deadlock in Tel Aviv, 2011

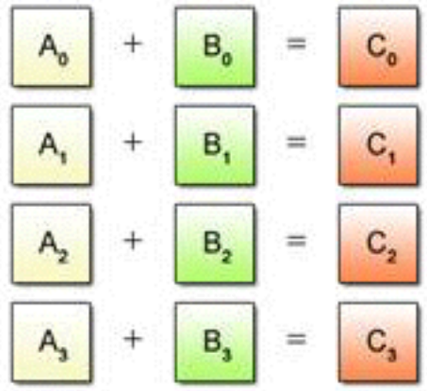
The bottom line is that there is no silver bullet!

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

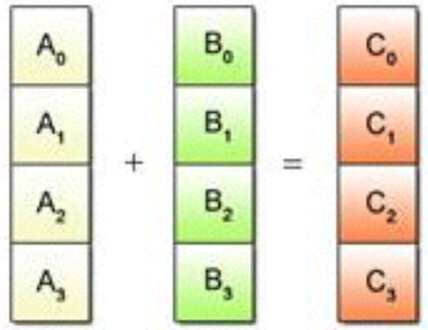
Vectorisation

Vectorised data is a prerequisite to make efficient use of modern CPU vector instruction sets

scalar operation



vector operation

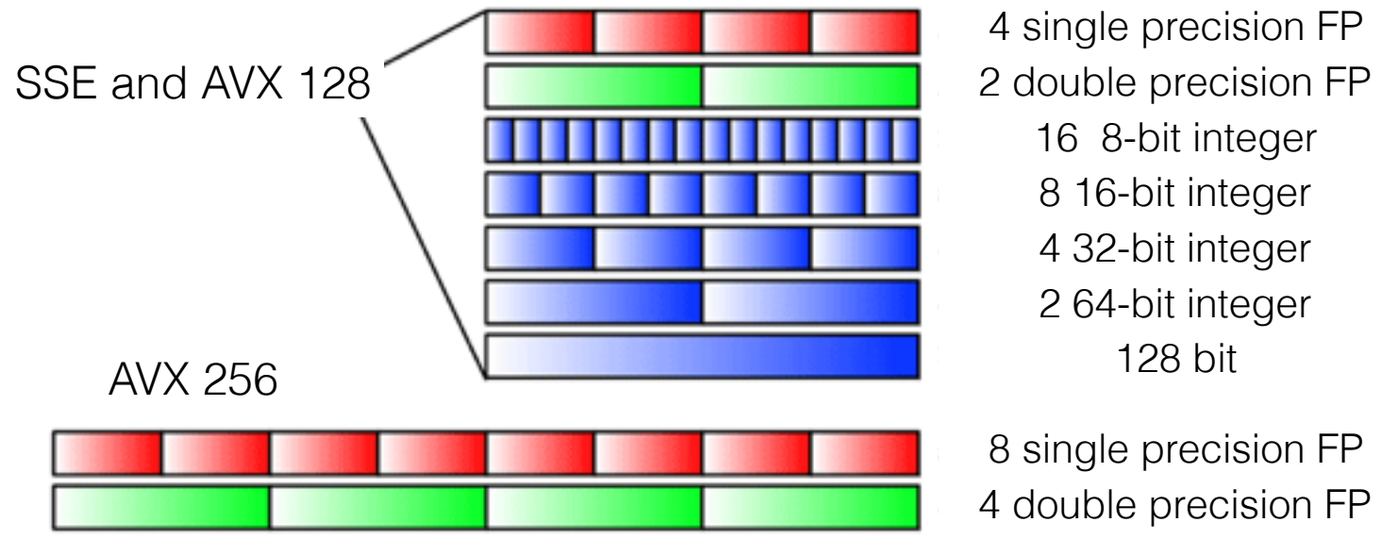


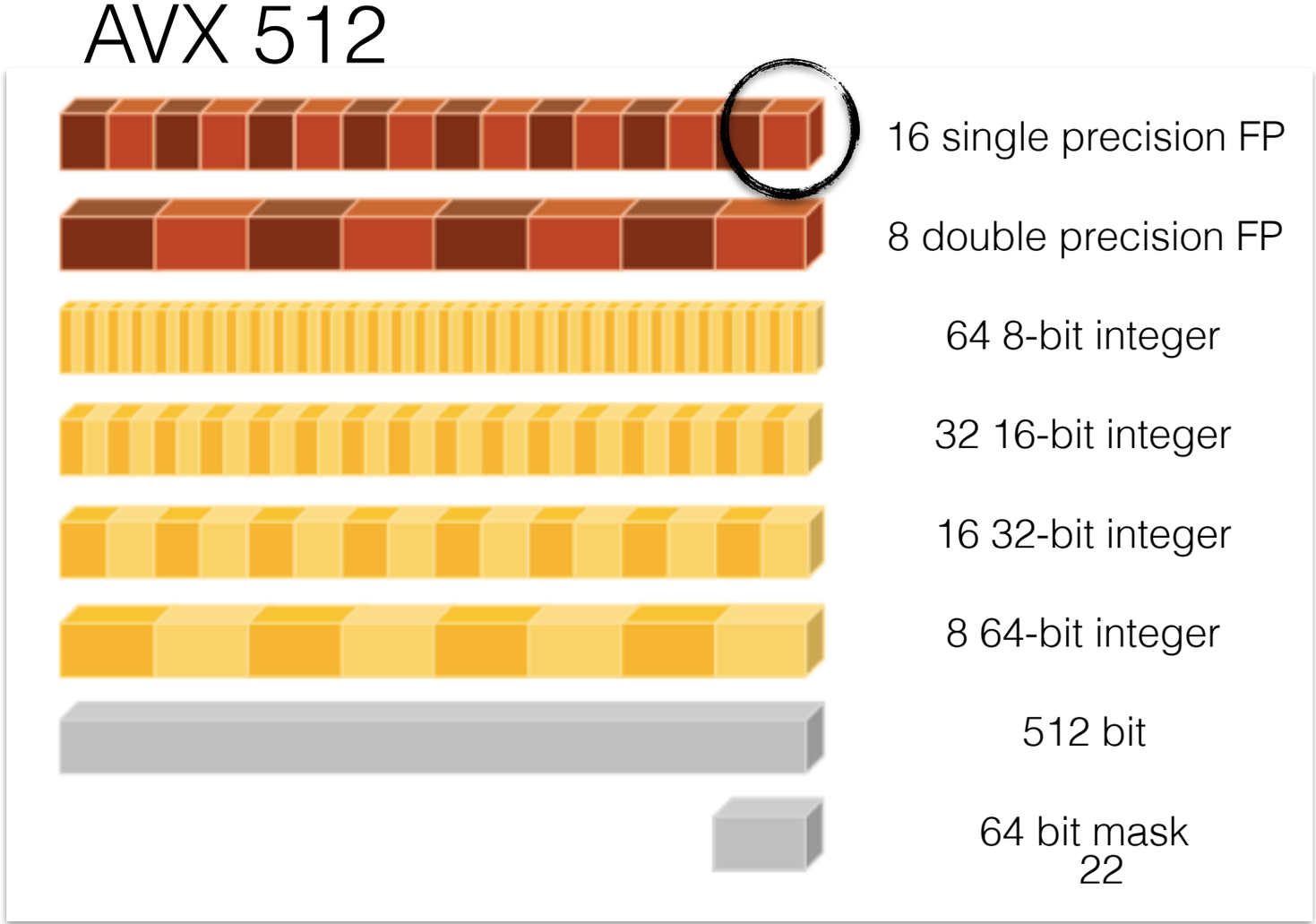
Ex: Intel

Year	Register	Corresponding Instruction set	
~1997	80 bit	MMX	P5 Pentium
~1999	128bit	SSE1	Pentium III
~2001	128 bit	SSE2	Pentium IV
…	128 bit	SSEx	Pentium - Nehalem core i7
2008	128 bit	AVX	Sandy Bridge
~2010-2011	256 bit	AVX2	Haswell
2013	512 bit	IMCI	Xeon Phi (Knights Corner)
2015	512 bit	AVX512	Xeon Phi (Knights Landing)

Vectorisation

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!

Compiler optimisations

- Compiler optimisation are 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
- Differences among compilers and target architectures can be large
- You might be compromising accuracy and precision

Rice (1953): For every compiler there is a modified compiler that generates shorter code.

Need to run tests!

- **Instruction selection:** e.g. Multiplication*2 can be done by addition, bit-shift
- **Constant elimination**
- **Algebraic simplification:** Use algebraic properties to simplify expressions
- **Dead code removal**
- **Loop Optimisations:** often executed, large payoff!
- **Inlining:** improves time at the cost of space (larger code); allows for further optimisation;
- **Auto-Parallelisation, Auto-vectorization..**

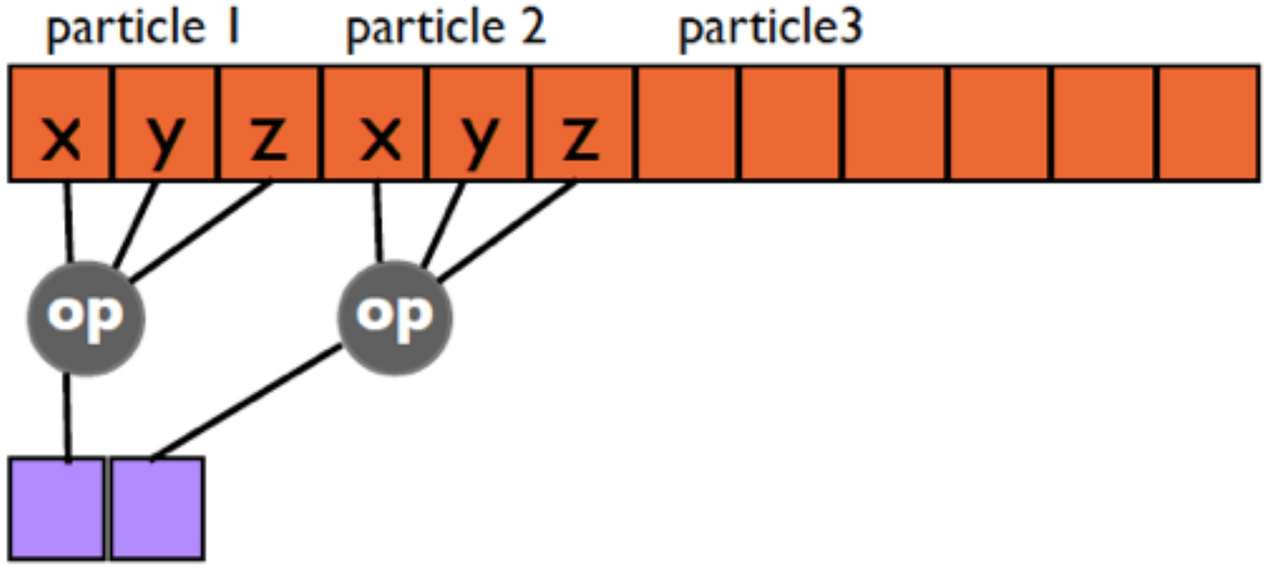
Auto-Vectorisation

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
- **Use efficient memory accesses**
- **Favour inner loops with unit stride**
- **Align data.** Data is memory aligned when the data to be operated upon as an n-byte chunk is stored on an n-byte memory boundary
- **Prefer Structure of Arrays** (SoA) over **Array of Structures** (AoS)

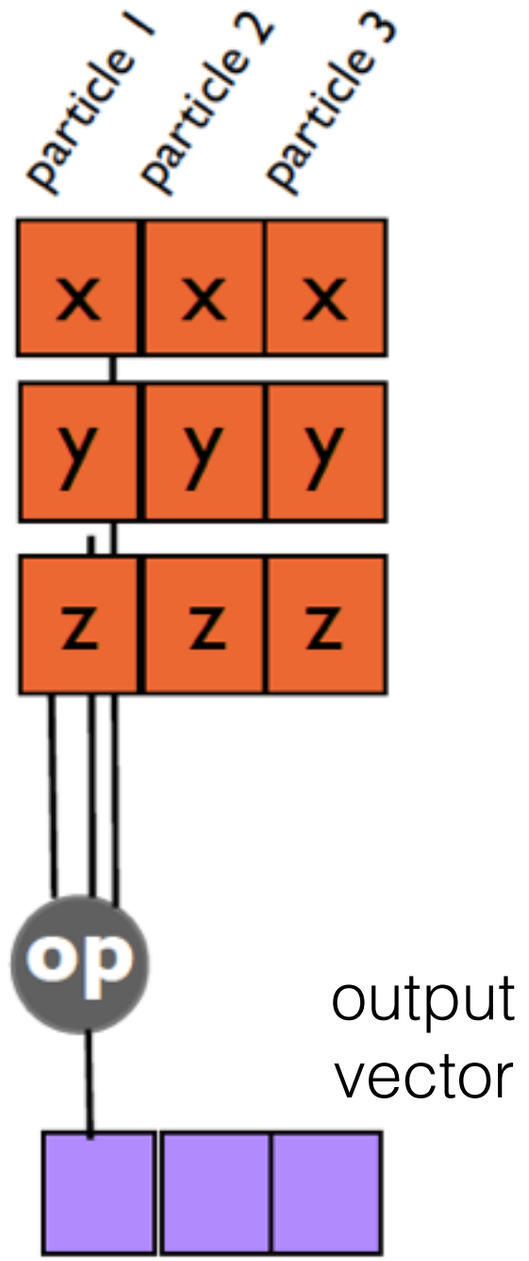
Memory access pattern

AOS



output vector

SOA



output vector

- **AOS approach seems the natural way** to do vector processing of particles
  - 3-to-1 typical memory access pattern
- **SOA approach is better vectorised by the compiler**
  - Memory access in SOA pattern also more efficient

Need to take possible overheads into account!!

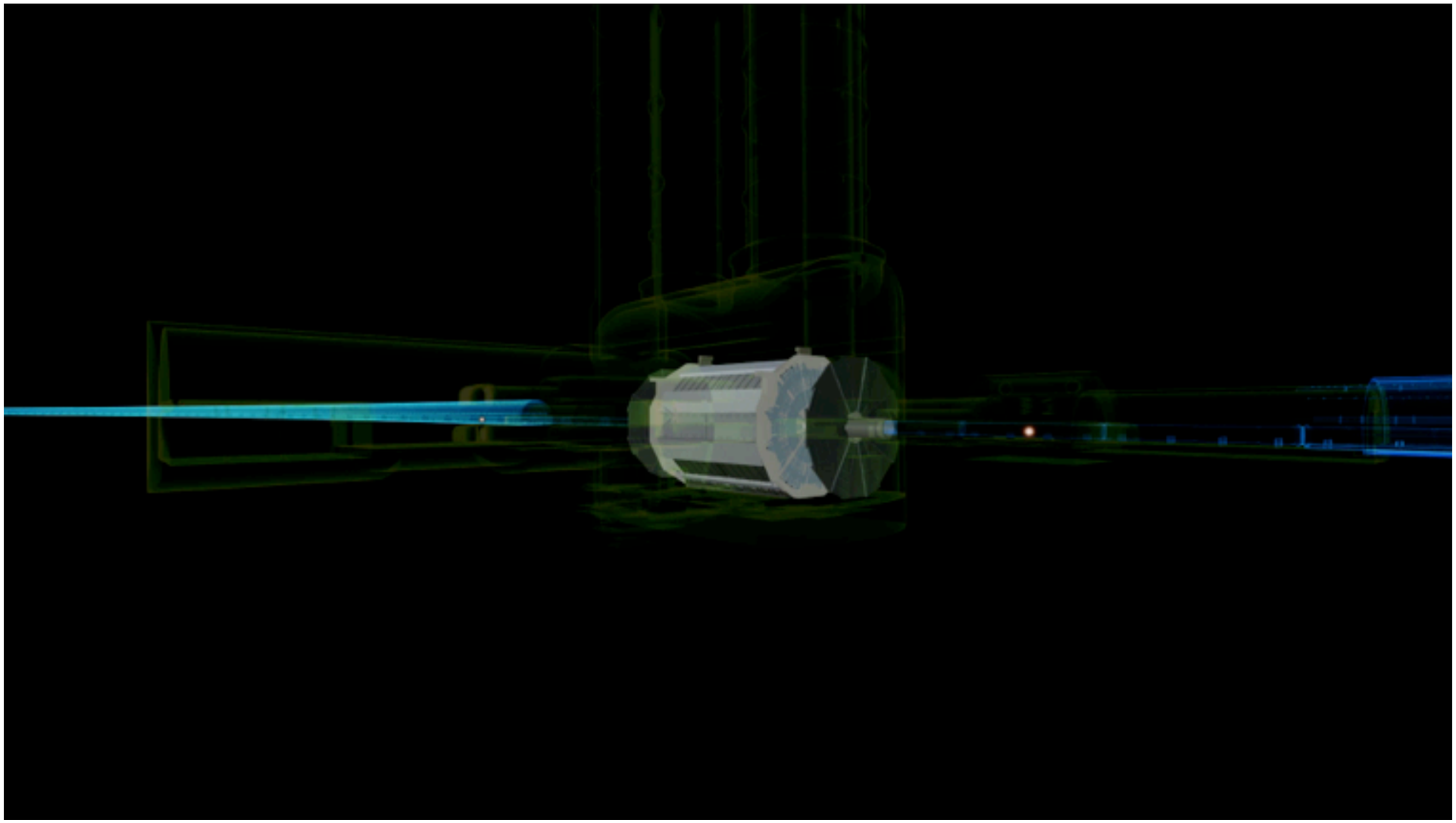
Our case study

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!

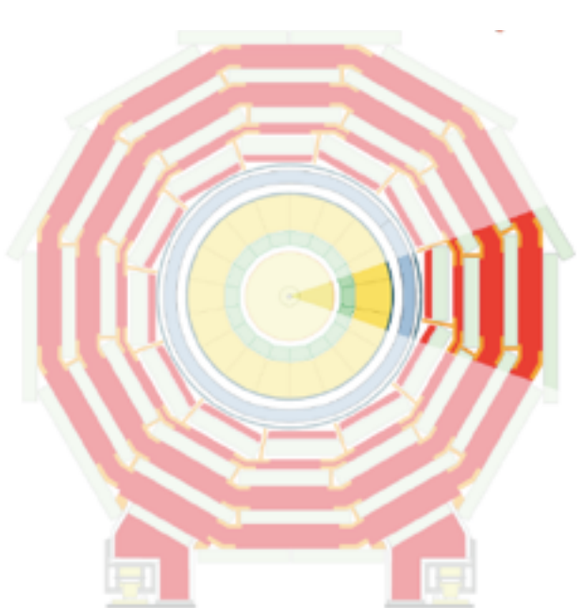


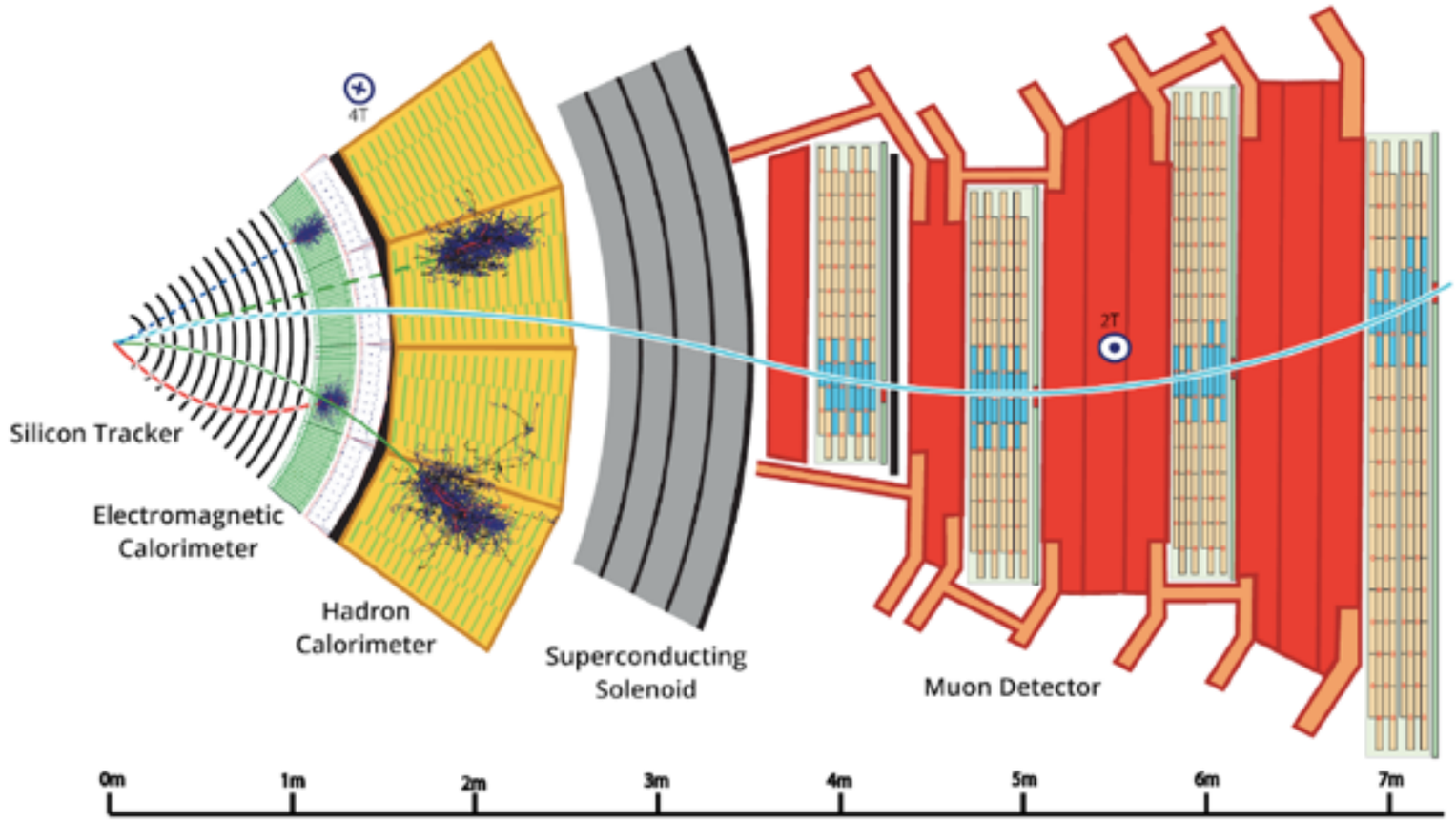
https://cds.cern.ch/record/1309872

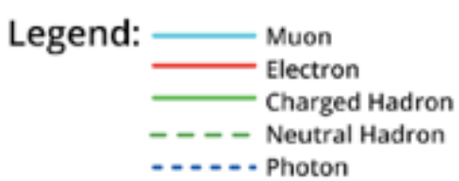
Simulation in HEP

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)





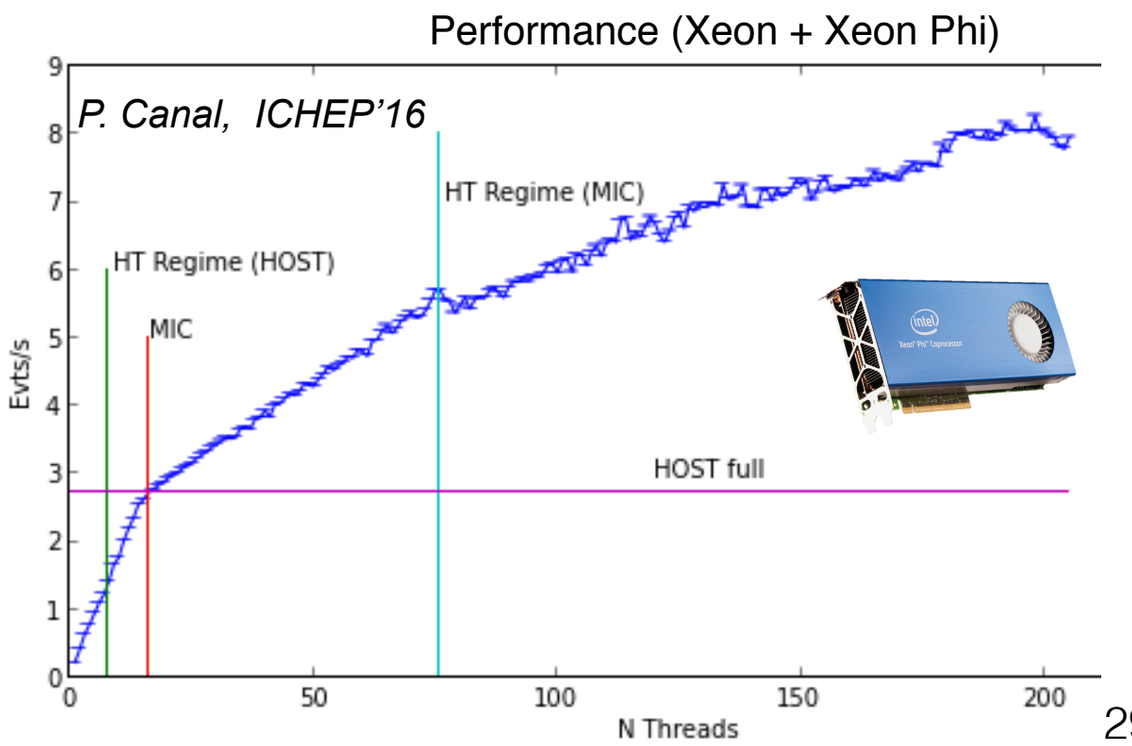


cms.web.cern.ch

Geant4 (GEometry ANd Tracking)

current standard within HEP

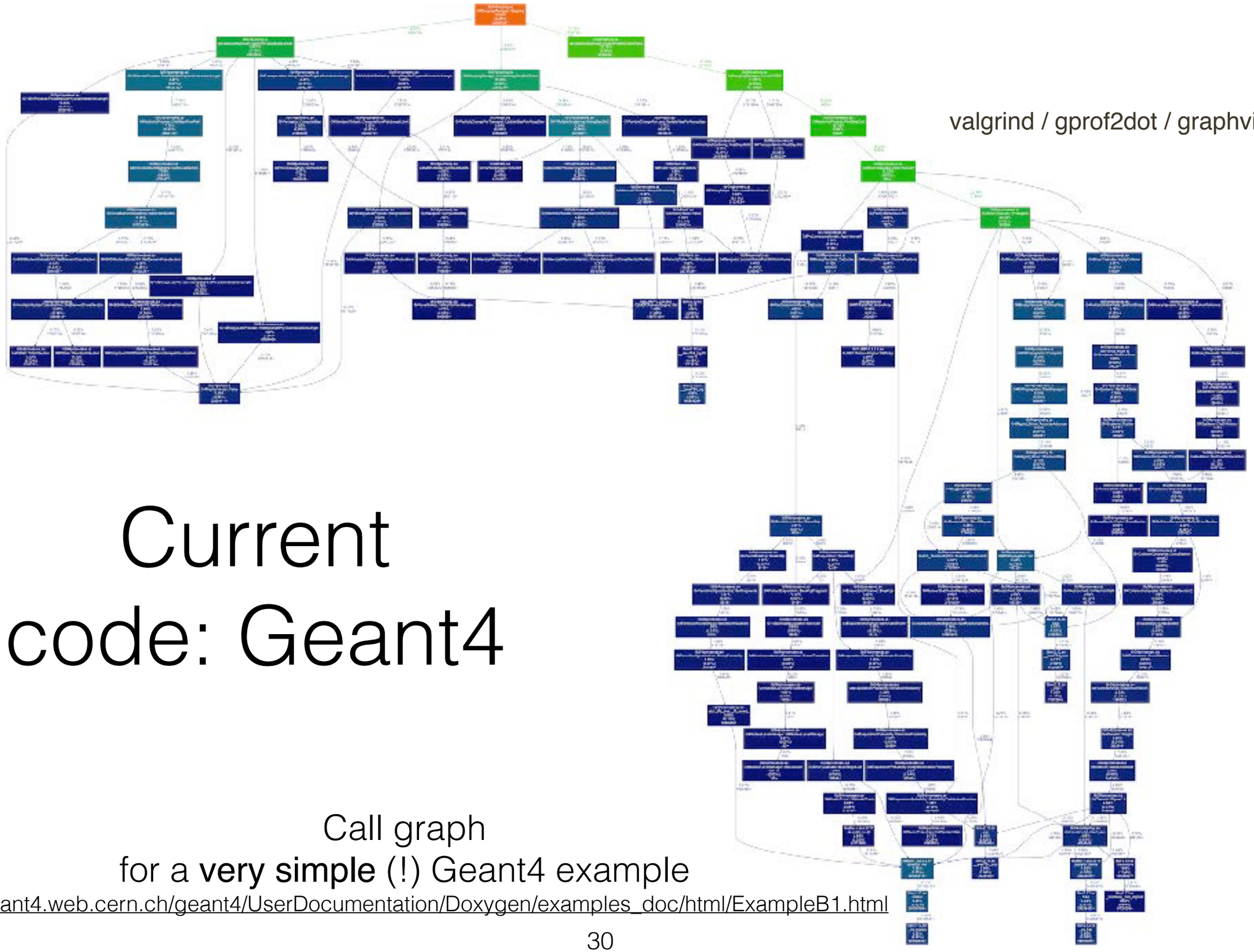
- **Major international collaboration**, ~2M lines of code, hundreds of users worldwide
- **Large variety of applications ..beyond HEP:** Medical applications, materials & space science
- **Scalar processing**: Each particle is simulated and followed through its whole life one by one.
- **Event level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

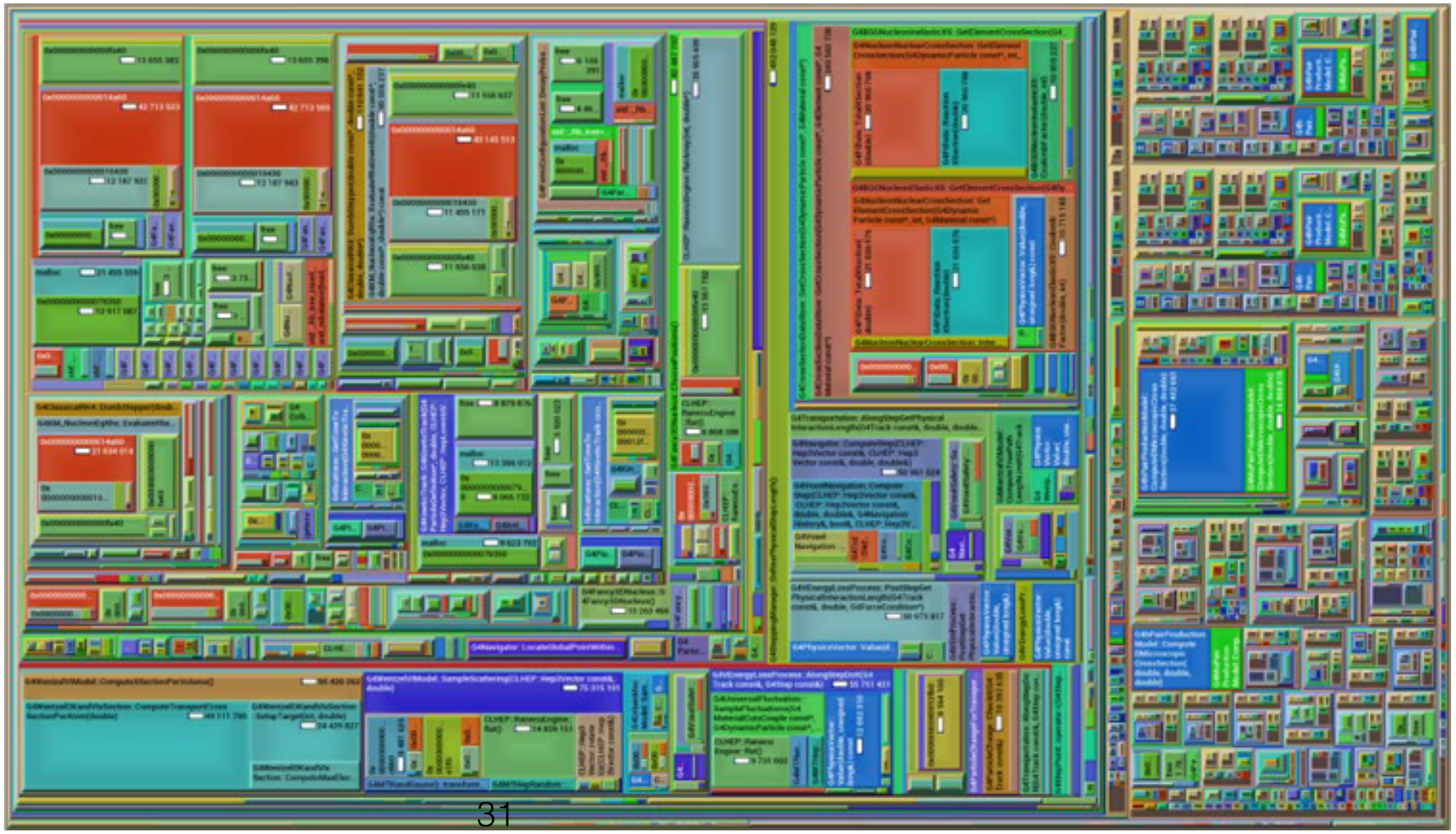


Call graph
for a **very simple** (!) Geant4 example
http://geant4.web.cern.ch/geant4/UserDocumentation/Doxygen/examples_doc/html/ExampleB1.html

Current code: Geant4

- Codebase very large and non-homogenous
- Very deep call stack (IC misses) and virtual table structure
- Hotspots practically inexistent
- Each rectangle represents a function
- Its size is proportional to the cost spent therein

Valgrind/kCachegrind



so .. how do we optimise?

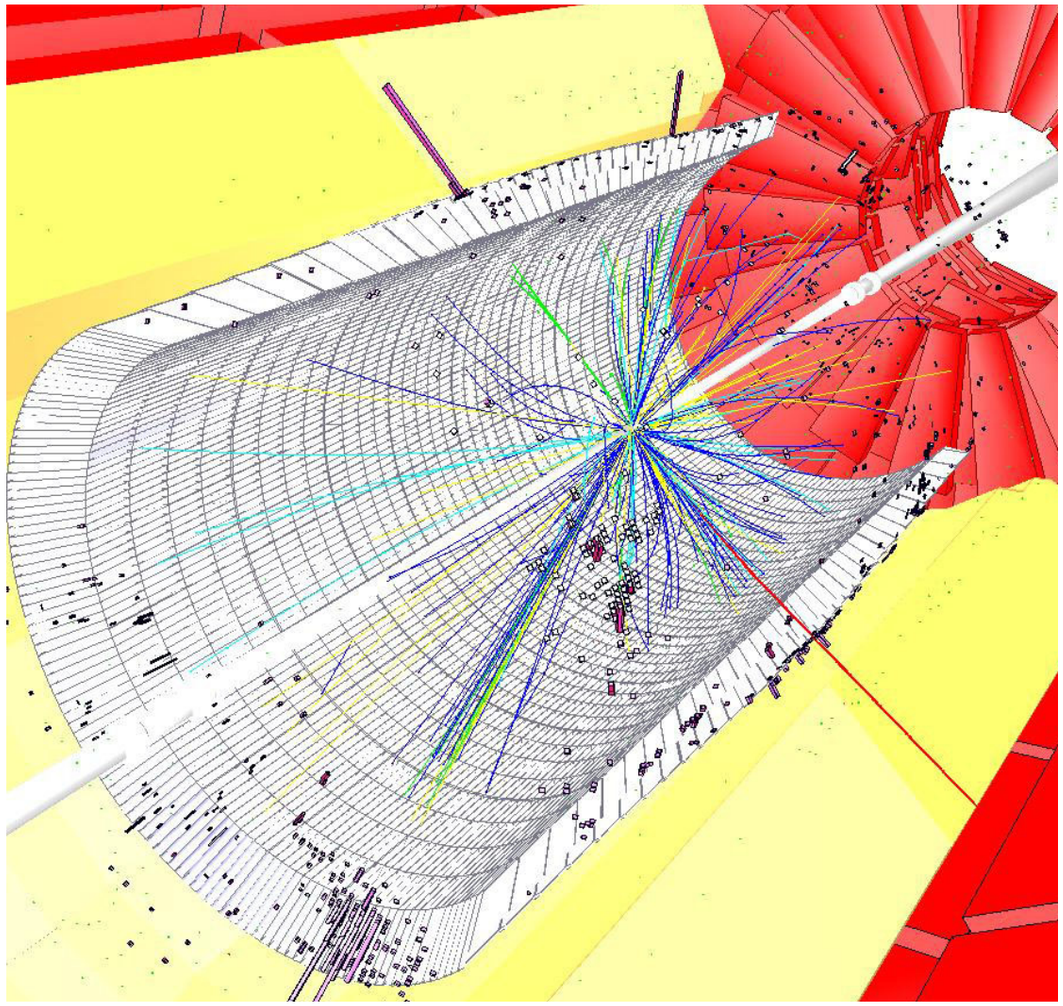
..a hint..

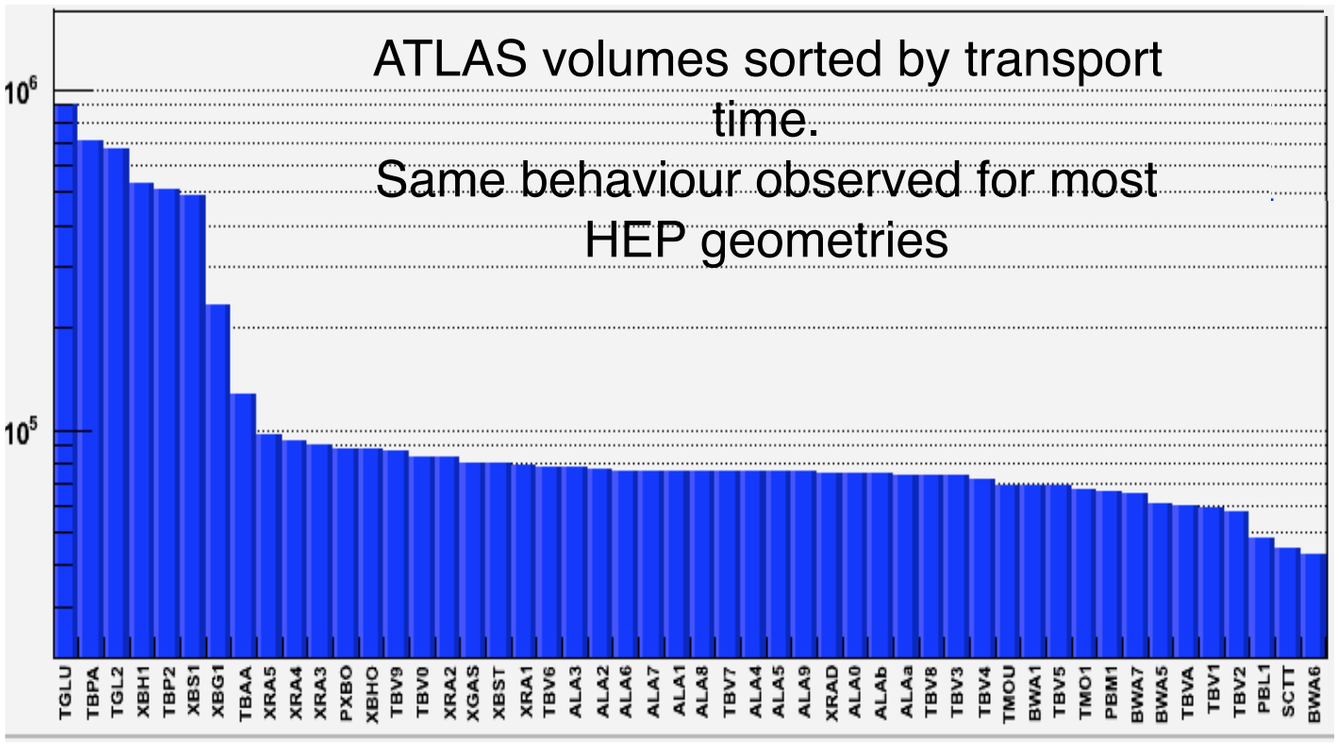
Level	Potential gains	Estimate
Algorithm	Major	~10x-1000x
Source code	Medium	~1x-10x
Compiler level	Medium-Low	~10%-20% (more possible with autovec or parallelization)
Operating system	Low	~5-20%
Hardware	Medium	~10%-30%

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





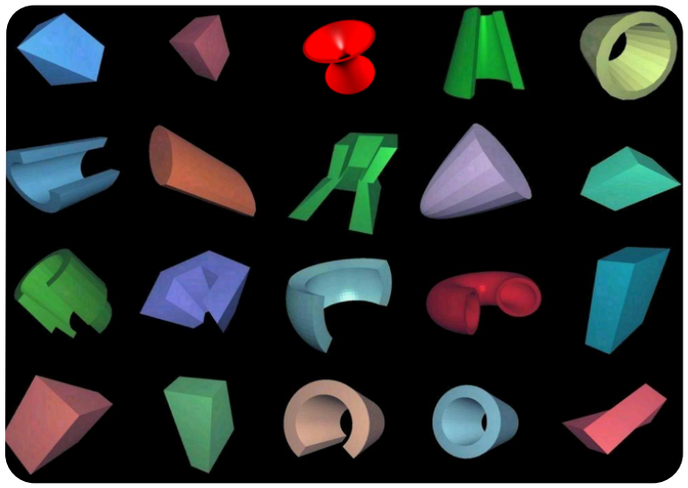
- Locality not exploited by classical transport code
- Existing code inefficient
- Cache misses due to fragmented code

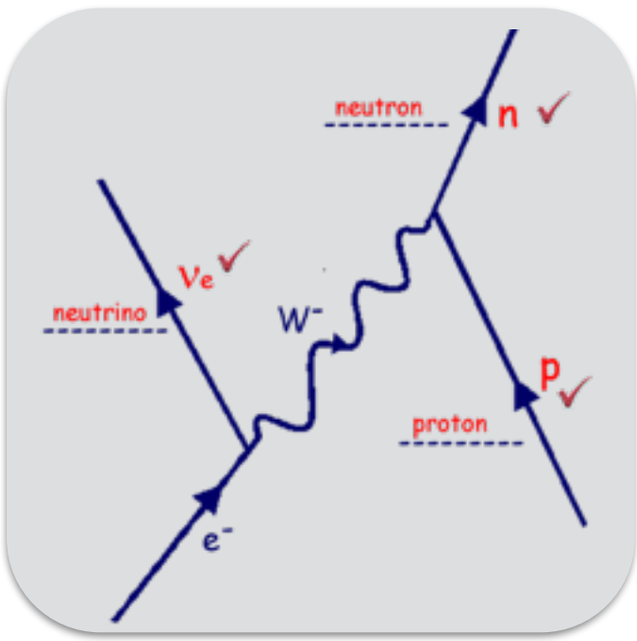
GeantV: introducing parallelism

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

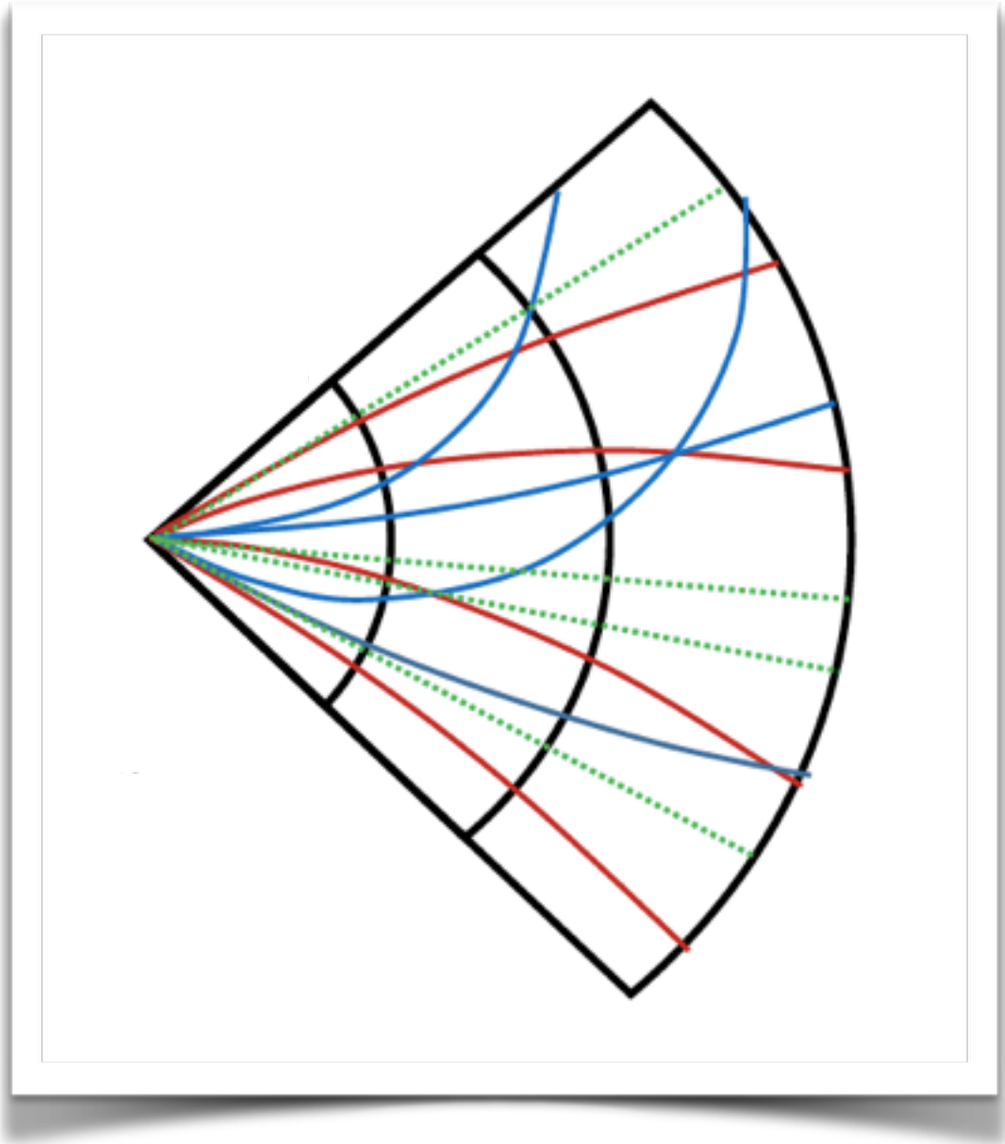




GeantV: introducing parallelism

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…

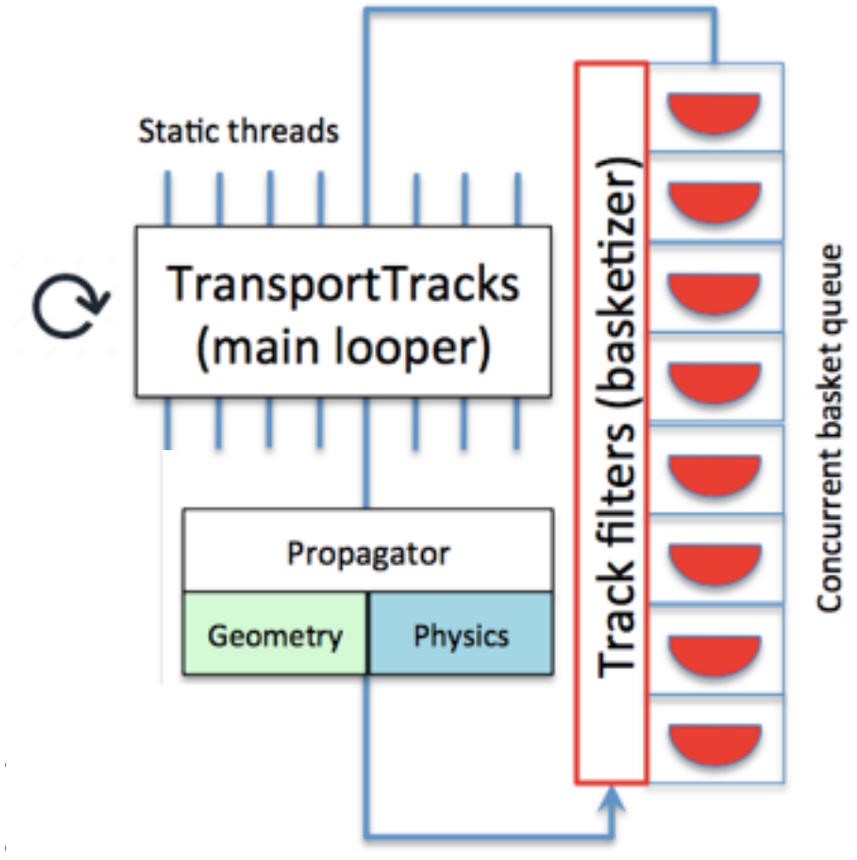
- How we've implemented concurrency
- An example on removing bottlenecks
- Introducing vectorisation (geometry)
- Performance improvement!

Concurrency in GeantV

Investigated different ways of scheduling & sharing work

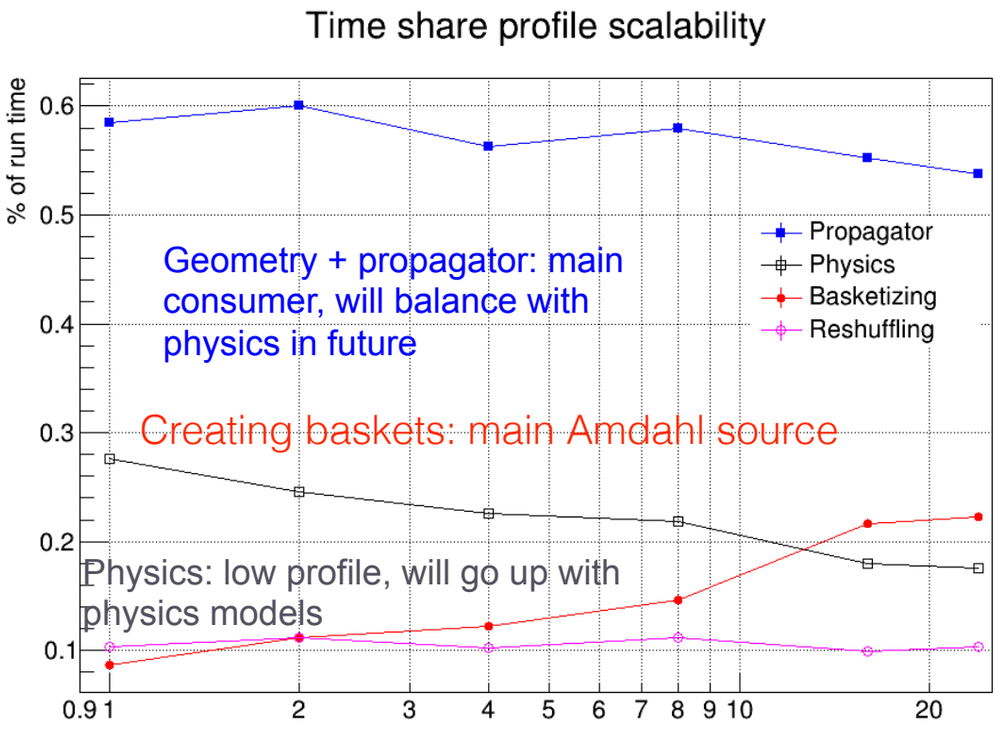
Current model: static allocation of workers

- Main thread method as infinite looper
- Any thread can 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)



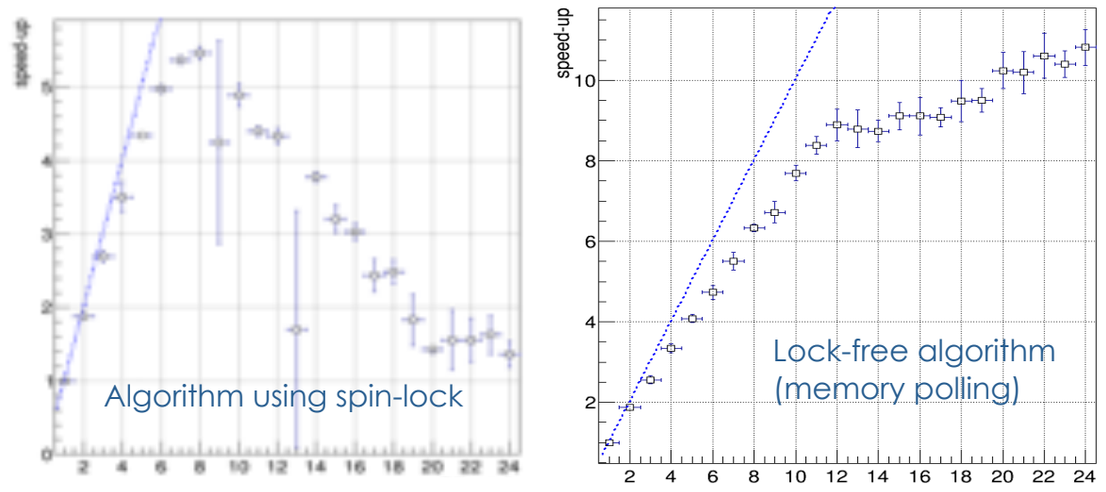
Concurrency in GeantV

Watch for overheads!!



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

scalability with number of threads



Fine grain MT prevents scaling to high number of threads

Issue for many cores architectures!

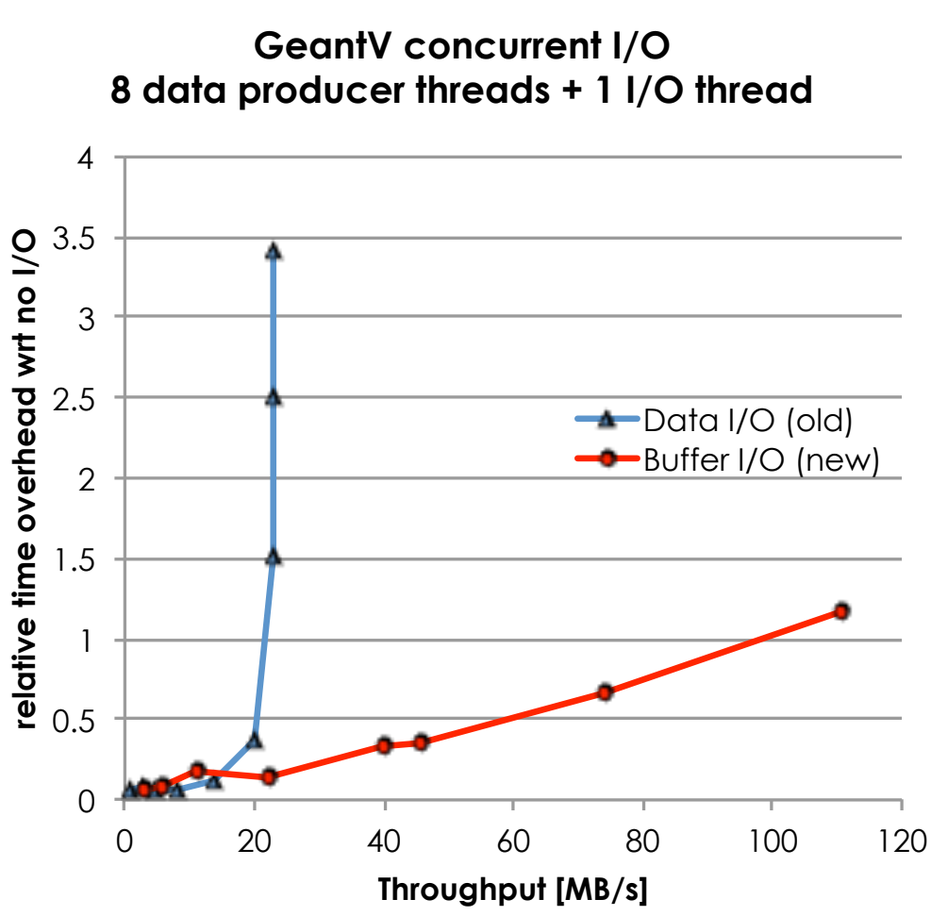
..On-going work

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

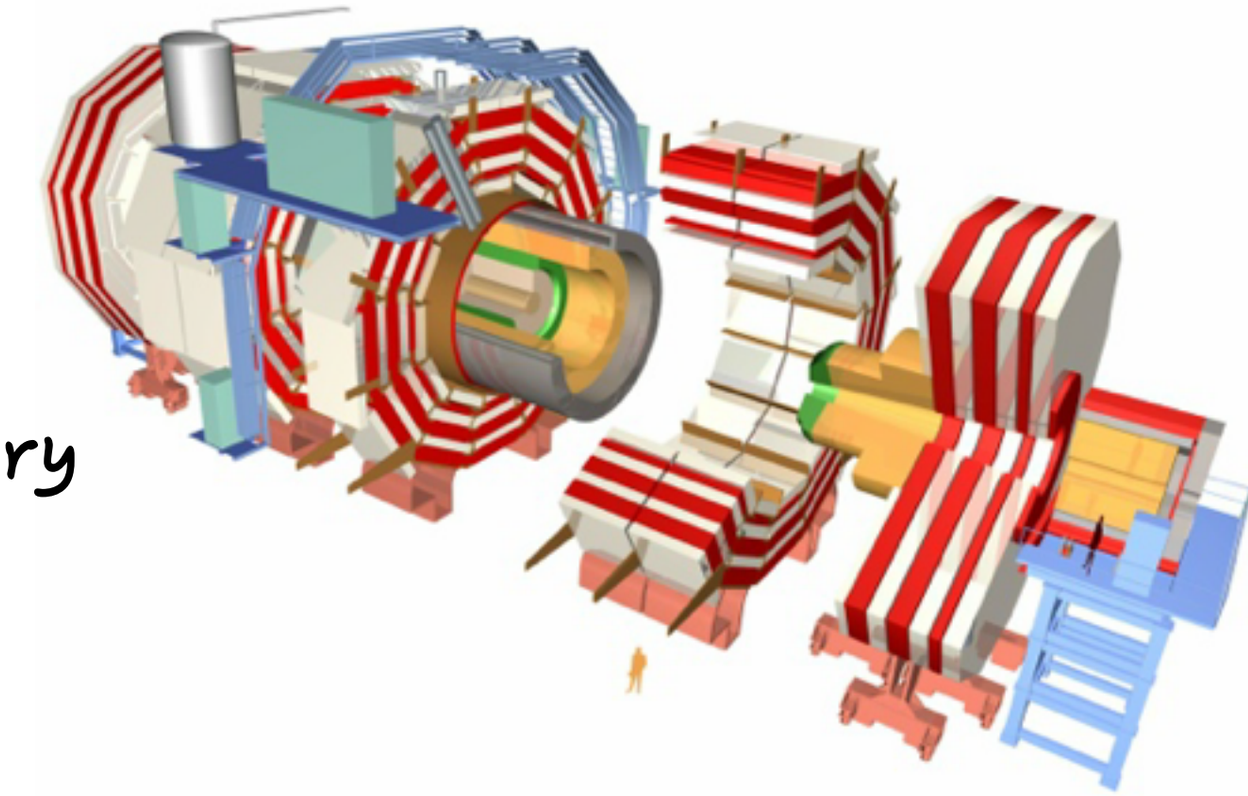
- **Buffer mode**: Send concurrently local hits connected to memory files produced by workers to one thread dealing only with final merging/writing to disk



Geometry…

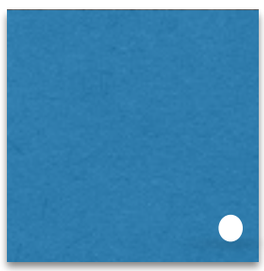
It sums up to more than 30% of processing time

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



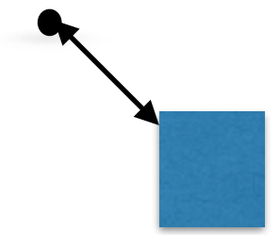
A geometry library provides APIs to:

In or out?

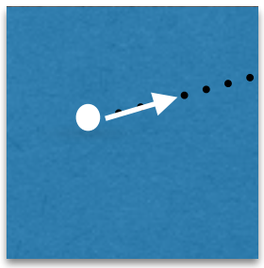


collision detection and distance to enter de object

minimal safe distance to object

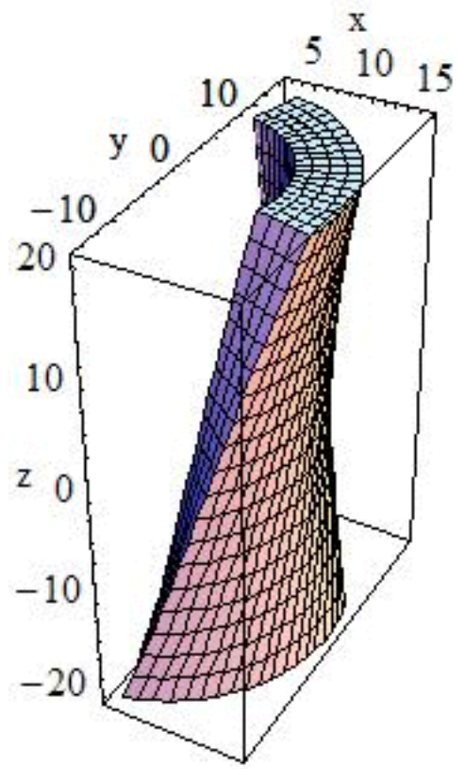


distance to leave object

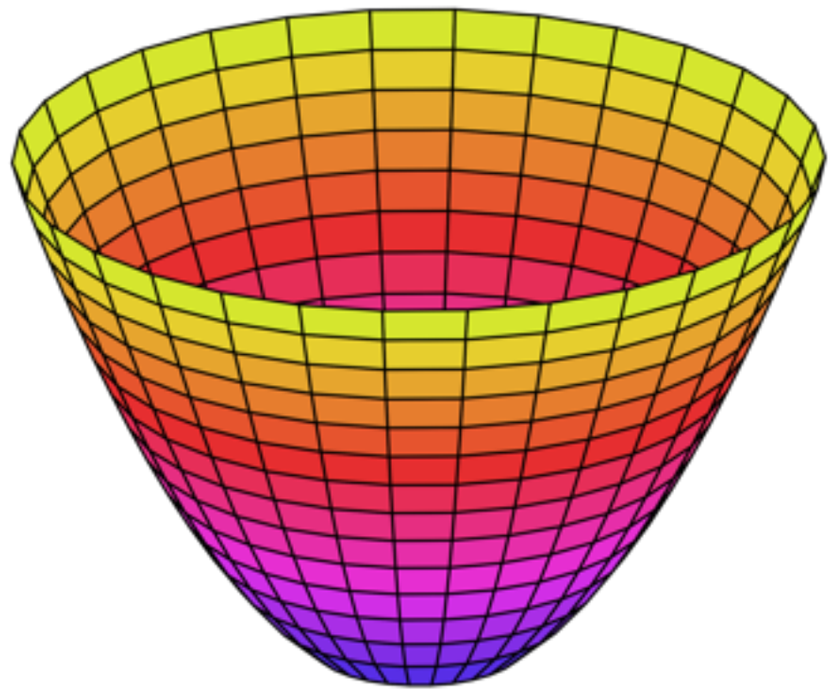


VectorizedGeometry

High performance geometry library for next generation simulation frameworks



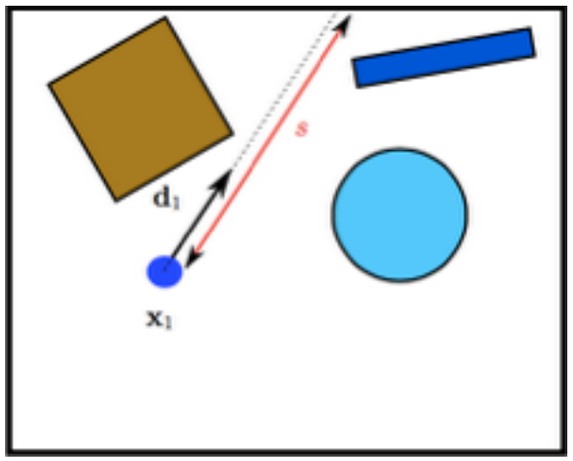
- Optimised library of primitive and composite solids
  - Reduce virtual function calls and avoid code multiplication
  - Use template code
- Introduce data parallelism
  - Explicit vectorisation (external libraries)
    - APIs for single & many-track navigation
    - “Inner” vectorisation of complex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



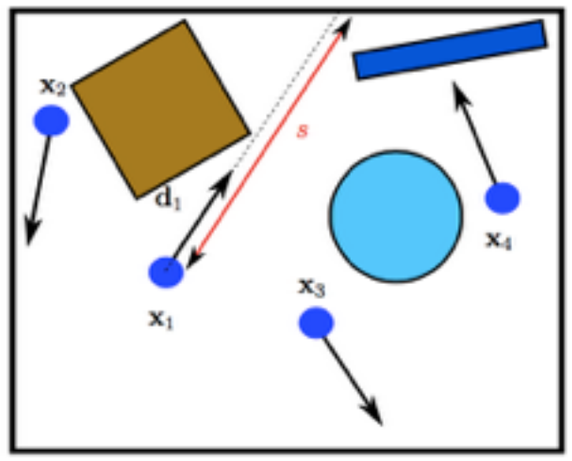
Vectorising Geometry

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 do the work works in only few cases

Option B (“convince the compiler”):

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

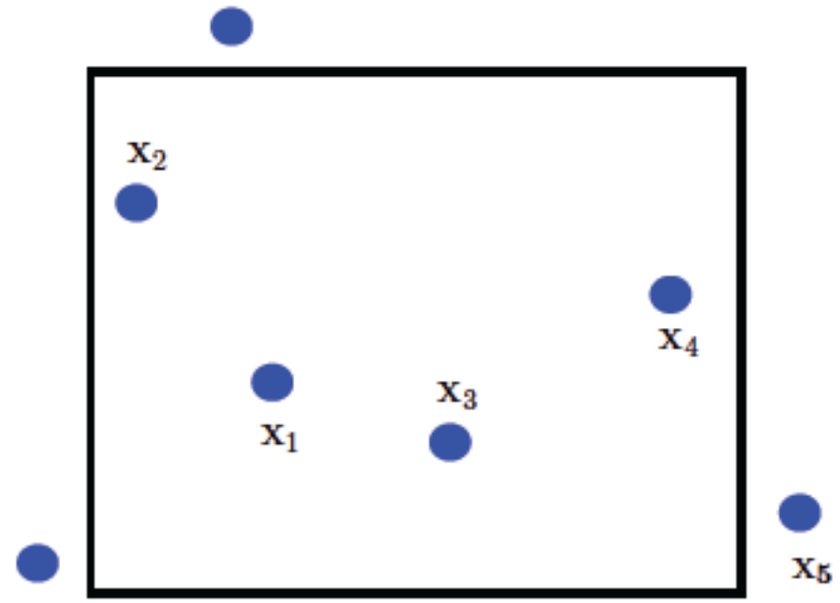
Option C (“use SIMD library”):

refactor the code and perform explicit vectorization using external libraries library compiler independent

Example

Some existing (C++) code 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;
}
```

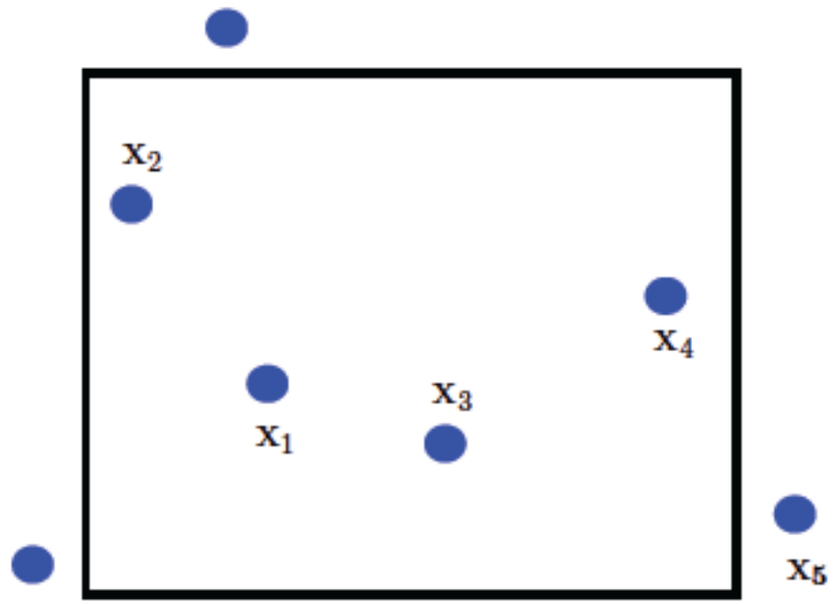


positions/dimensions vectors (x,y,z)

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…)

The struggle to autovectorisation (I)

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];
}}
```

The struggle to autovectorisation (II)

AOS to SOA

not enough! gcc 4.8 vectorises but not icc 13

```
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];
}}
```

Option B: “convince the compiler”

massage/refactor original code to make the compiler autovectorize
1. copy scalar code to new function ("manual inline")
2.change the data layout (se 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 auto-vectorises 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)
- given that we have to refactor code anyway, this is our implementation
- choice

https://github.com/VcDevel

Improving vectorisation

"branches are the enemy of vectorization..."

Many branches just distinguish between "static" properties of class instances

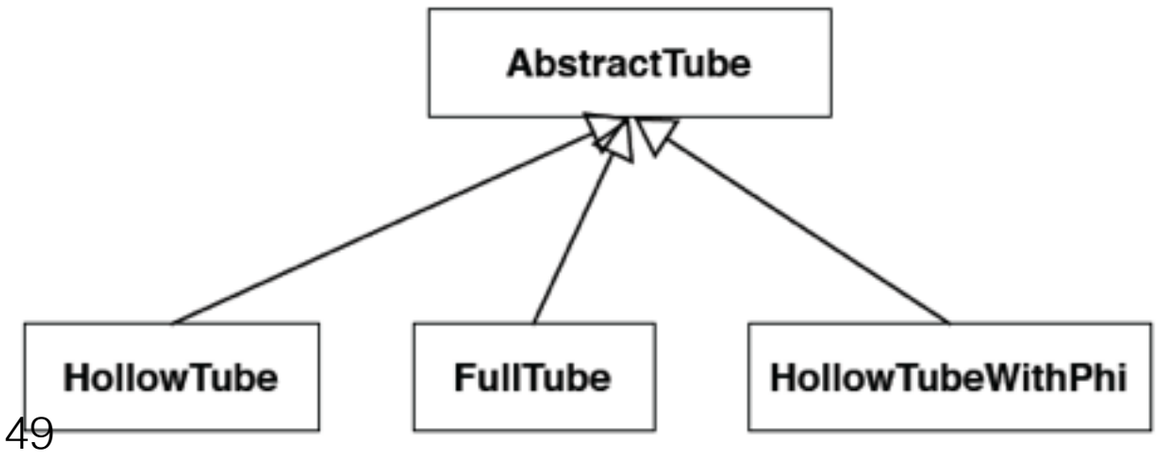
general "tube" class distinguishes at runtime between "FullTube", "Hollow Tube" ...

Tube | HollowTube | HollowTubePhi | FullTubePhi | HalfHollowTube

To get rid of many branches we could introduce a separate class for each important tube realisation

canonical approach:
solution with handwritten separate classes

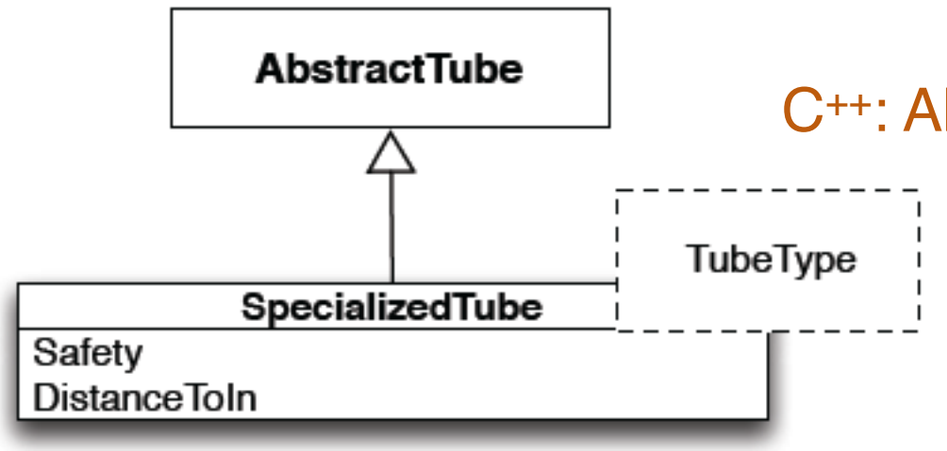
C++: AbstractTube *t = new FullTube();



Reducing branches: templates

Alternative idea: use 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!

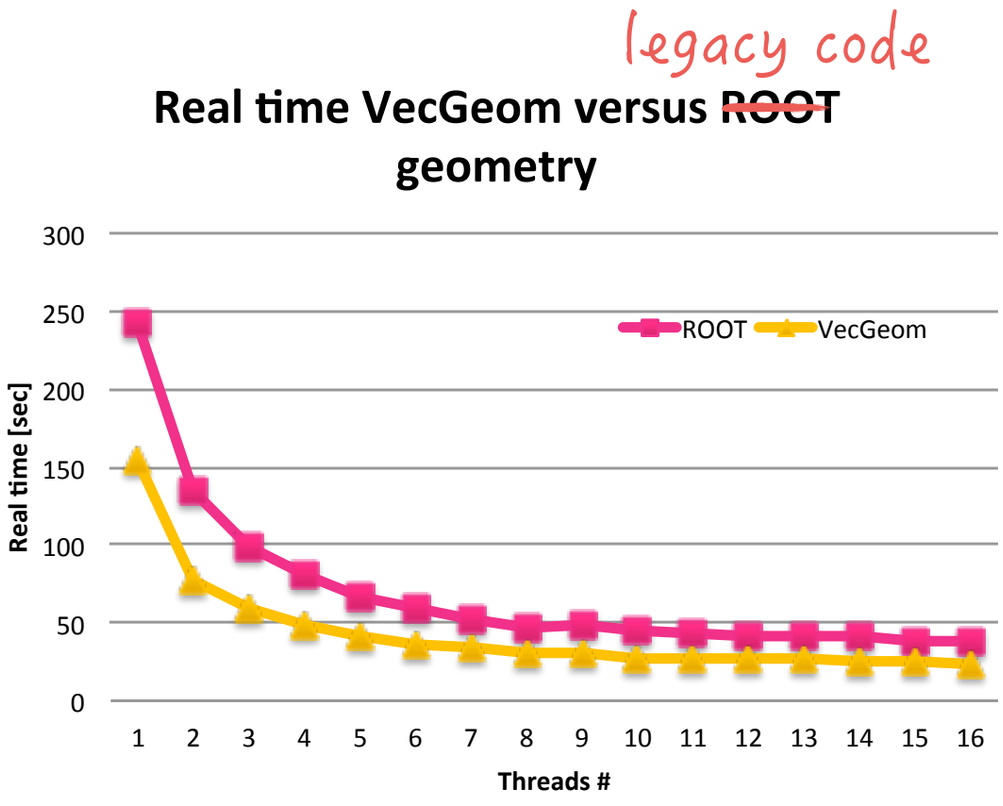
- ➡ vectorisation is efficient
- ➡ better compiler optimisations in scalar code
- ➡ less virtual functions means less calls to virtual tables
- ➡ embrace “generic programming” philosophy :-)
- ➡ Use the same approach to insure portability (..but this is another story..)

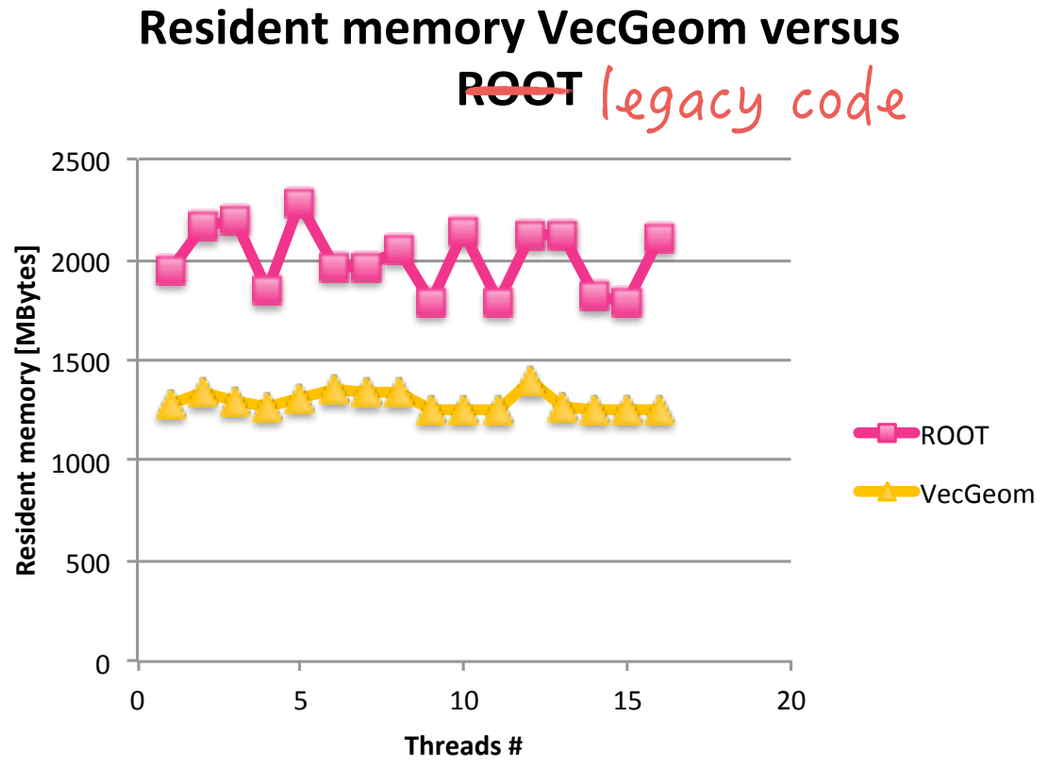
scalar

VecGeom performance

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

legacy code





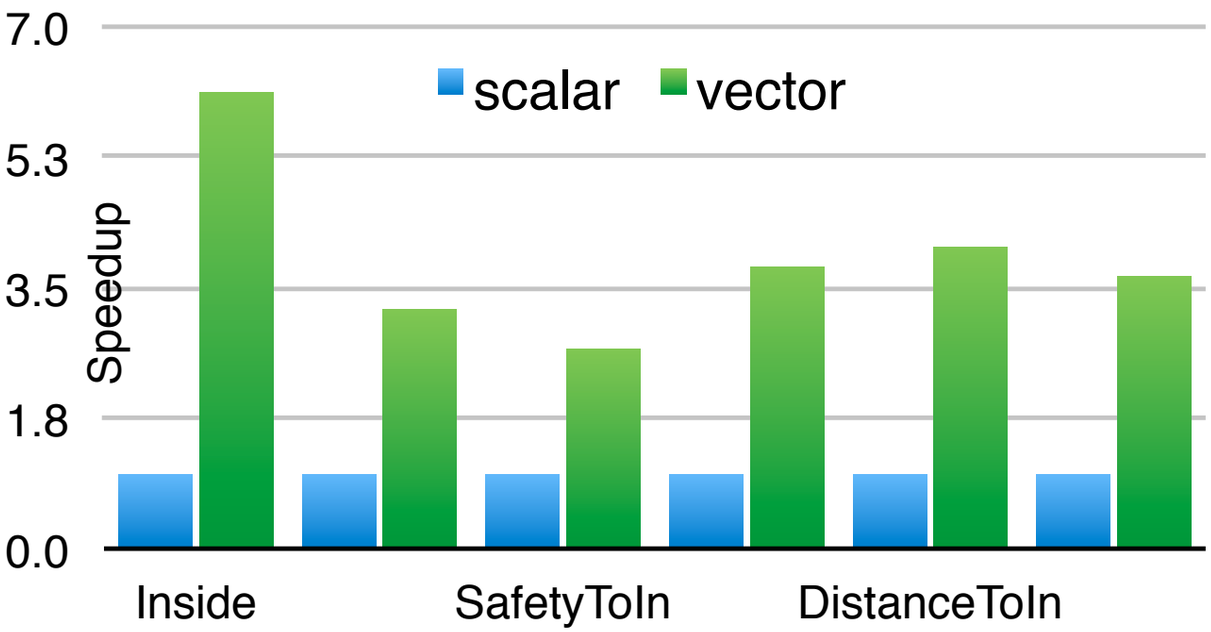
- GeantV runs VecGeom scalar navigation in full CMS geometry
  - first realistic estimate of overall impact on simulation time: ~1.6 improvement
  - so far using only scalar navigation mode

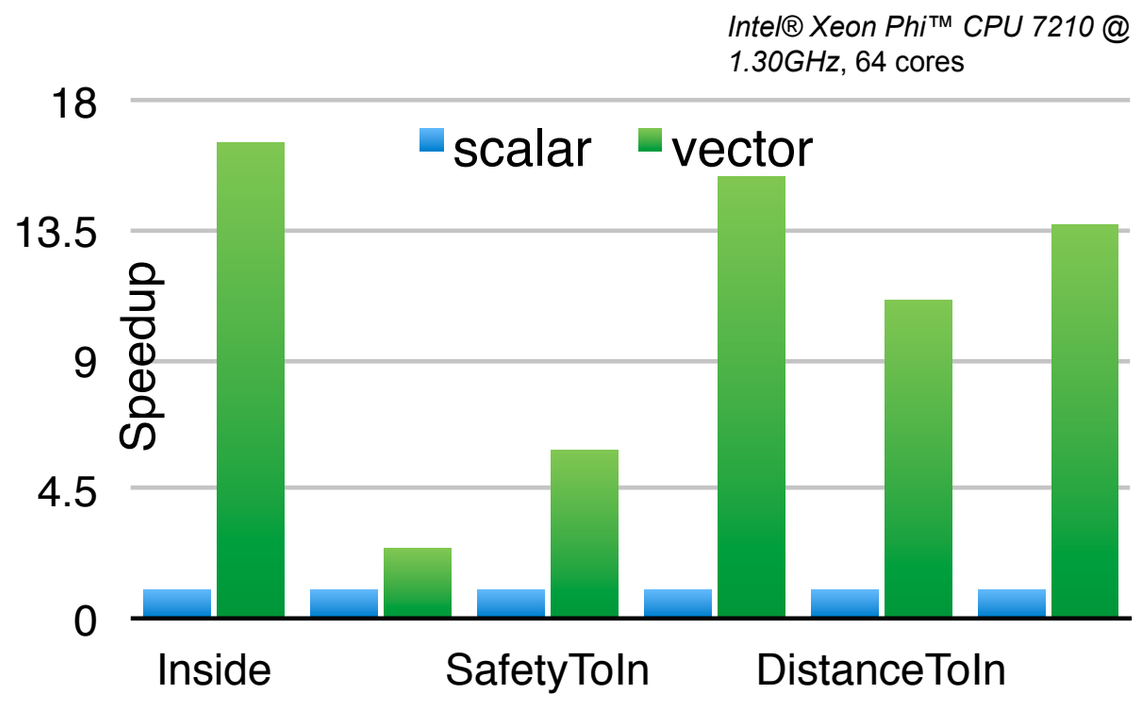
VecGeom performance

A set of CPU-intensive navigation methods:

Measure wall time for **vector and scalar implementations:**

Calculate vector speed-up (scalar time is reference =1) using AVX2 and AVX512





Quiz: assign the correct label!
(all our code uses double precision…)

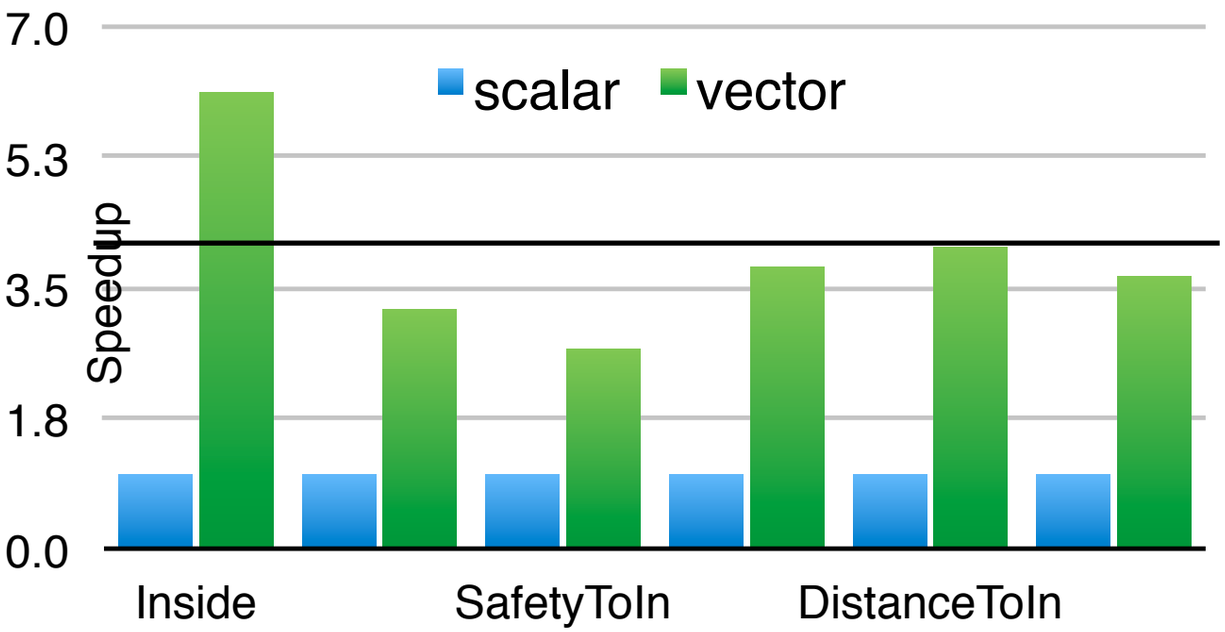
AVX2 **AVX512**

VecGeom performance

A set of CPU-intensive navigation methods:

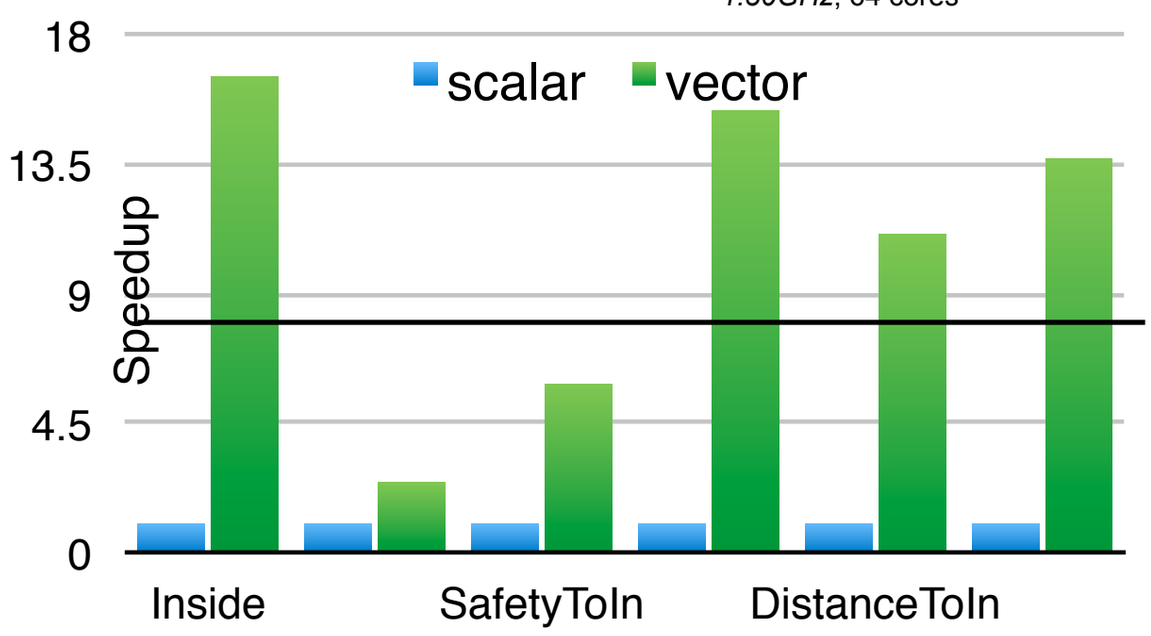
Measure wall time for **vector and scalar implementations:**

Calculate vector speed-up (scalar time is reference =1) using AVX2 and AVX512



AVX2

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



AVX512

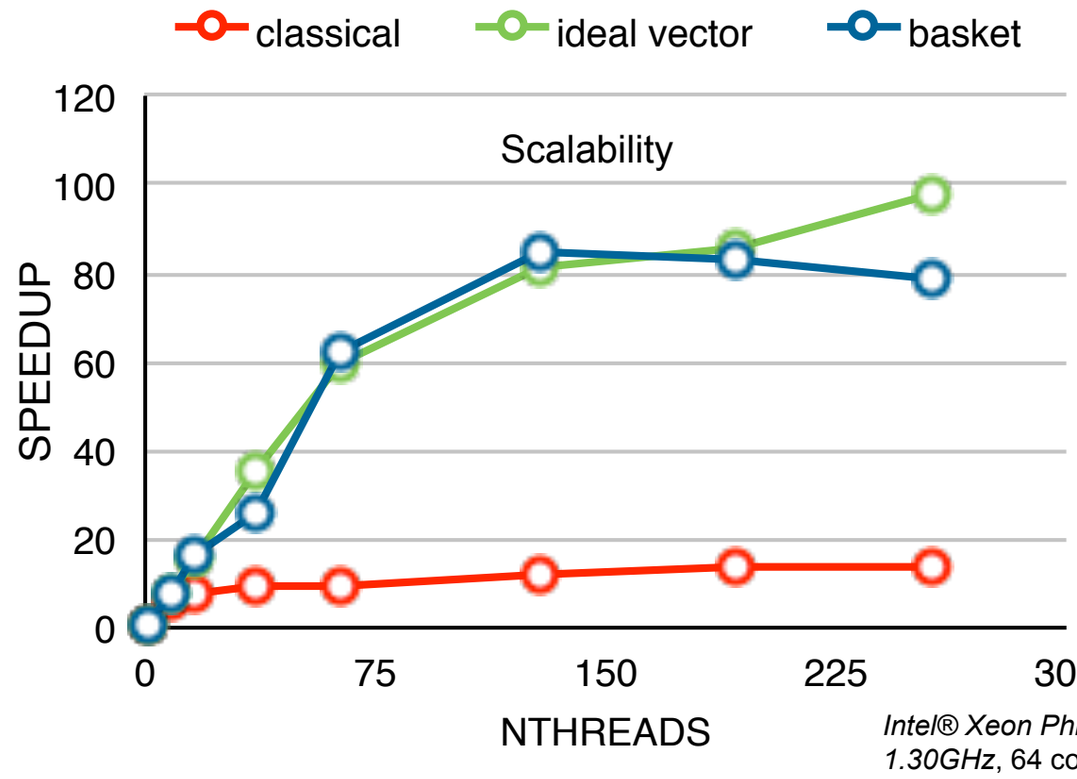
Super-linear speedup for some of the methods !

Scalability

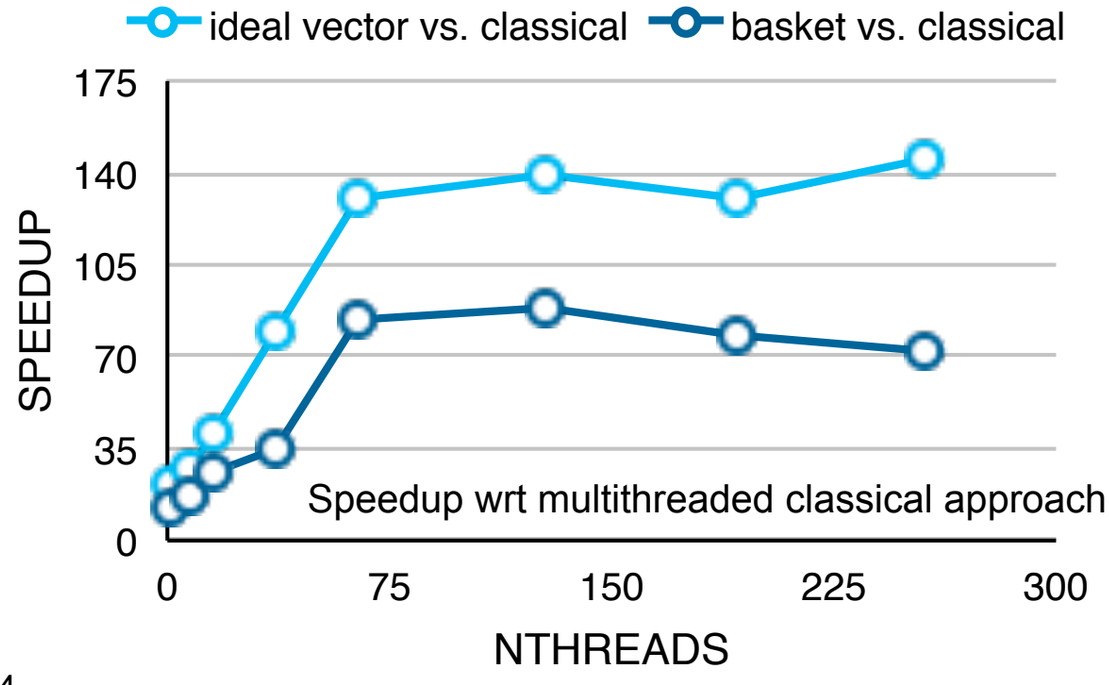
To test our **concurrency model** we setup a simplified testbed:

- **a toy detector** (typical tracker geometry)
- **a "ideal vector" transport** scenario in which particles are transported in bunches without any overhead due to particle re-shuffling to use as the "theoretical" best case
- **we also compare to the classical code navigation method**
- **Measure speedup wrt $N_{threads}$**

benchmarks are run on new Intel Xeon Phi systems recently released!

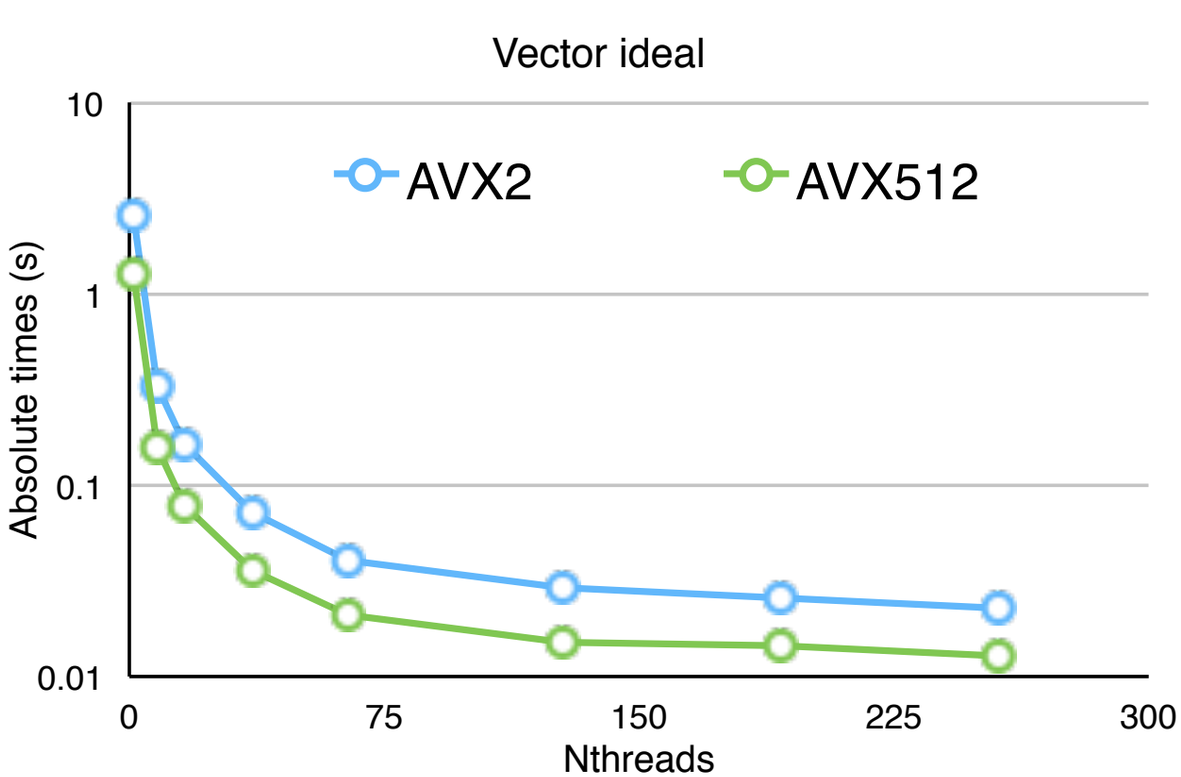


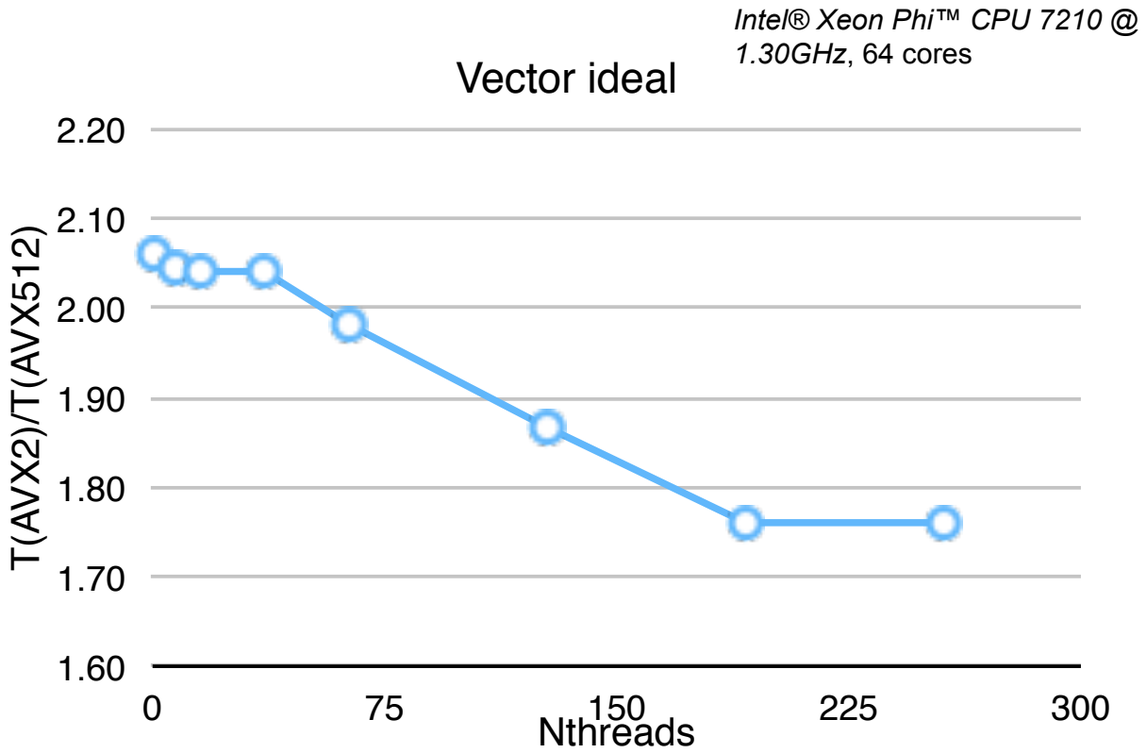
Intel® Xeon Phi™ CPU 7210 @ 1.30GHz, 64 cores



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!

The end

Summary (I)

What we have done

We started the GeantV project aiming at a x5-10 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
  - Caching & Memory management
  - Going multi-process
  - …

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

Conclusions

Improving code performance is an “epic fight”

There is no pre-defined “improving performance algorithm”

There is a large variety of methods, strategies, “handles” to use so..

..use your brain!

THE END

Thank you!

Have a nice weekend

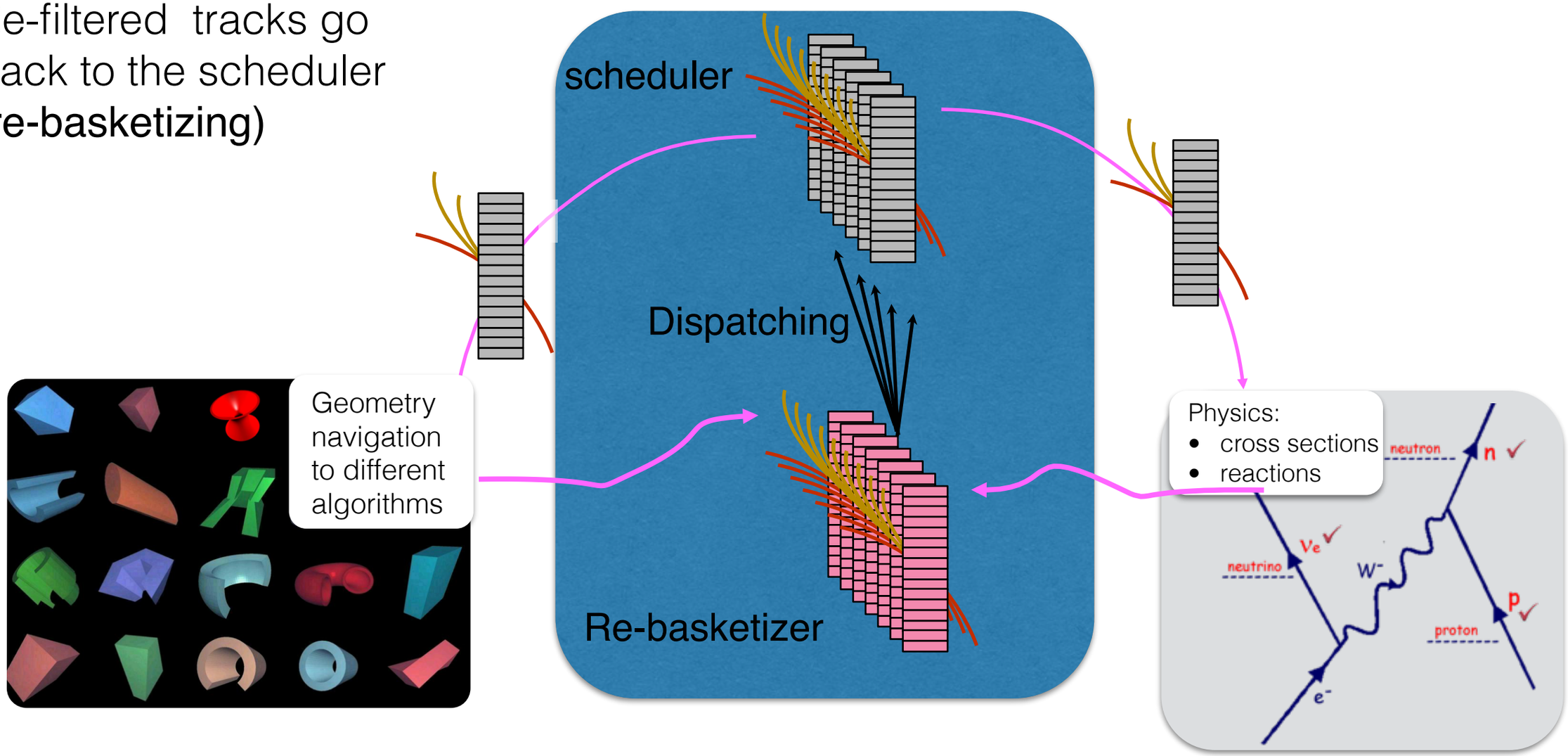
Profiling tools

VTune: https://software.intel.com/en-us/intel-vtune-amplifier-xe
Advisor: https://software.intel.com/en-us/intel-advisor-xe
Valgrind: http://valgrind.org
PIN: https://software.intel.com/sites/landingpage/pintool/docs/65163/Pin/html/
gprof: https://sourceware.org/binutils/docs/gprof/
perfmon2: http://perfmon2.sourceforge.net

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

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 …
};
```

Xray benchmark