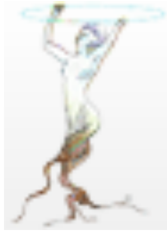


Vectorizing the detector geometry to optimize particle transport

J.Apostolakis, R.Brun, F.Carminati, A.Gheata, S.Wenzel

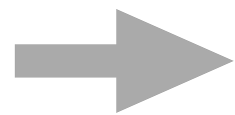
CHEP 2013, Amsterdam

Motivation

Explore possibilities to recast particle simulation so that it takes advantage from all performance dimensions/technologies

- Geometry navigation takes a large part of the particle transport budget (40-60% in ALICE)

Dimension 1 ("sharing data") : multithreading/multicore



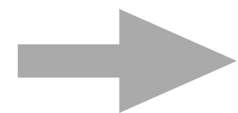
In HEP, mainly to reduce memory footprint

Motivation

Explore possibilities to recast particle simulation so that it takes advantage from all performance dimensions/technologies

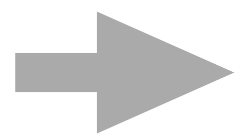
- Geometry navigation takes a large part of the particle transport budget (40-60% in ALICE)

Dimension 1 ("sharing data") : multithreading/multicore



In HEP, mainly to reduce memory footprint

Dimension 2 ("throughput increase") : in-core instruction-level parallelism and vectorization, re-usage of caches by increasing locality



Currently not exploited because requires "parallel data" to work on

Motivation

Explore possibilities to recast particle simulation so that it takes advantage from all performance dimensions/technologies

- Geometry navigation takes a large part of the particle transport budget (40-60% in ALICE)

Dimension 1 ("sharing data") : multithreading/multicore

→ In HEP, mainly to reduce memory footprint

Dimension 2 ("throughput increase") : in-core instruction-level parallelism and vectorization, re-usage of caches by increasing locality

→ Currently not exploited because requires "parallel data" to work on

Research projects: GPU prototype , Simulation Vector Prototype (talk from F. Carminati on Thursday) have started targeting beyond dimension 1

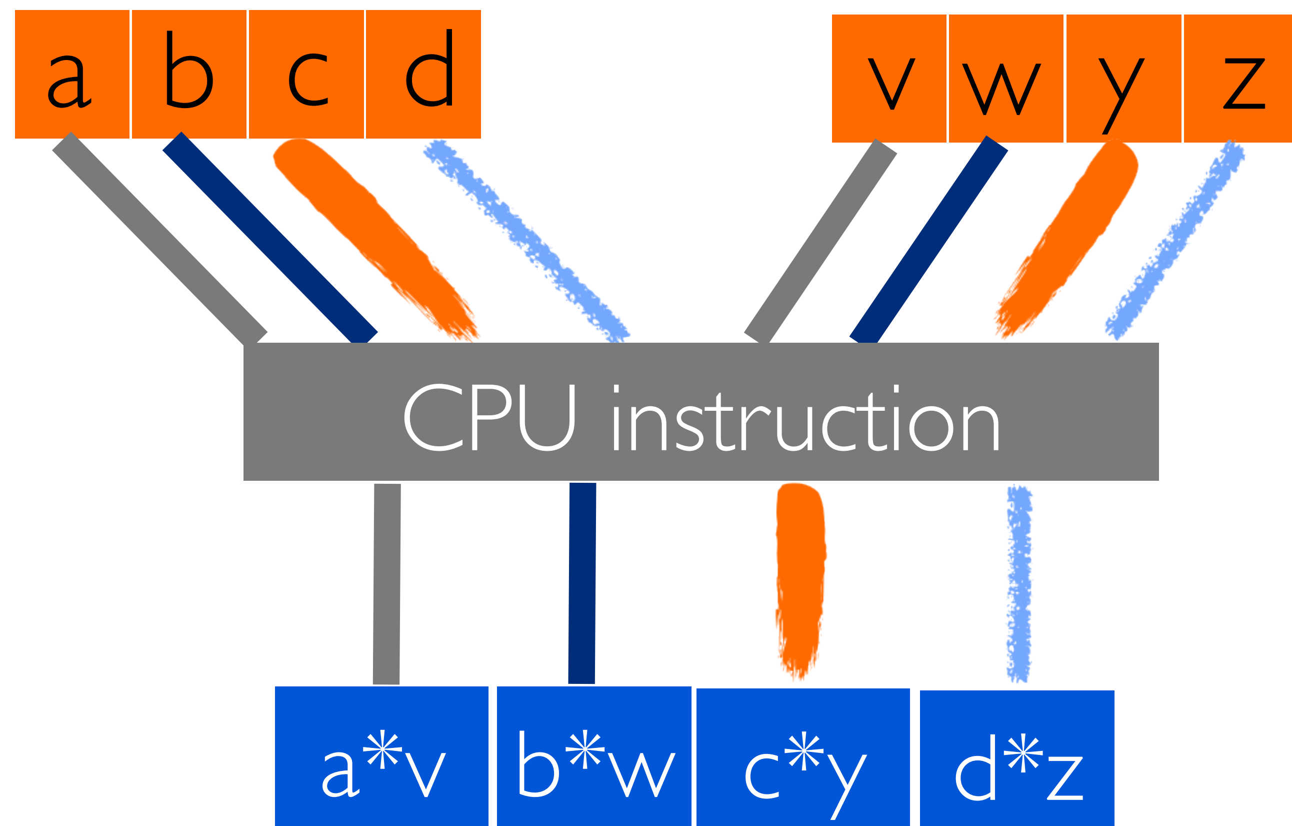
parallel data ("baskets") = particles from different events grouped by logical volumes

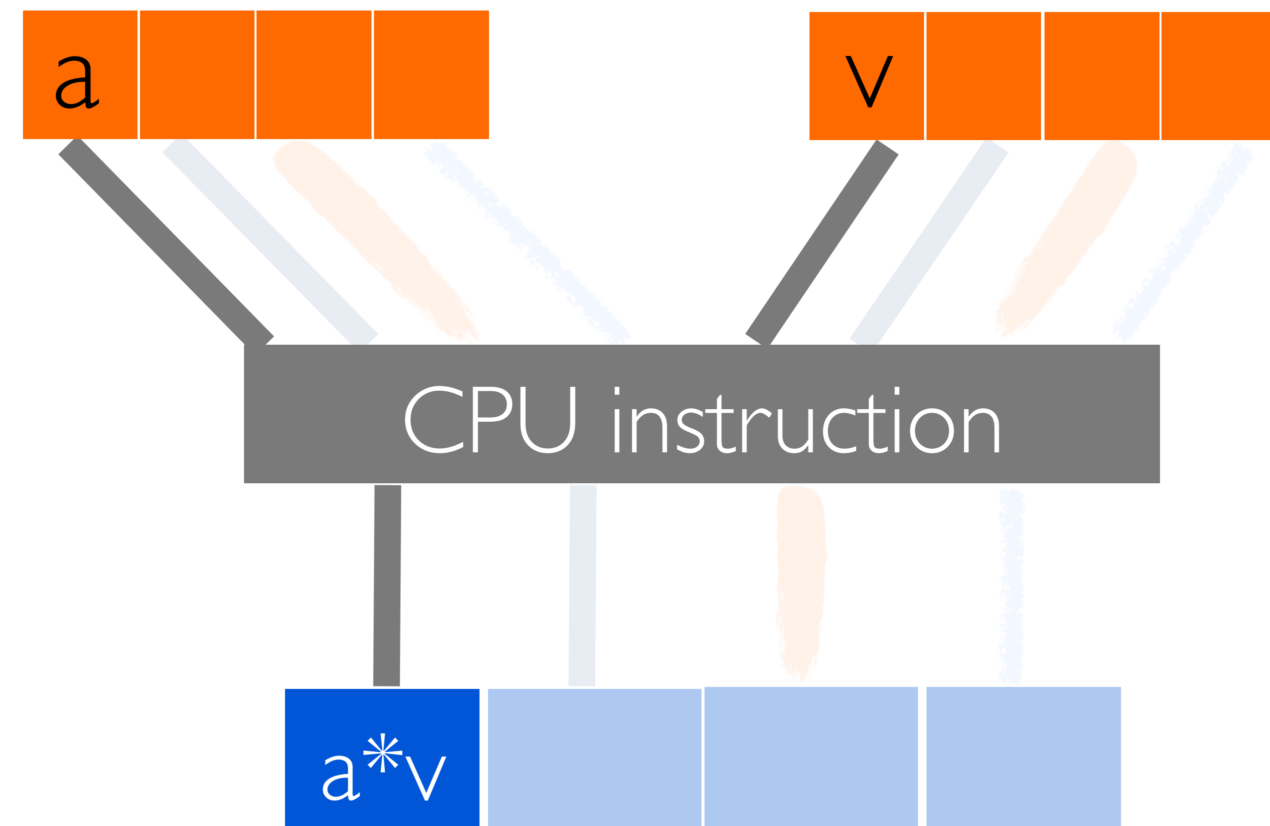
Reminder of vector micro-parallelism

- Commodity processors have **vector registers** on whose components (single) instruction can be performed in parallel (**micro-parallelism**)

single instruction multiple data = SIMD

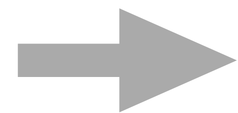
- Examples of SIMD architectures: MMX, SSE, AVX, ...





losing factors…

1st Goal: Vector processing for a single solid



Goal: Enable geometry components to process baskets/vectors of data and study performance opportunities

1 particle

distFromInside mothervolume

1 result

I. Milestone

vector of N particles

SIMD

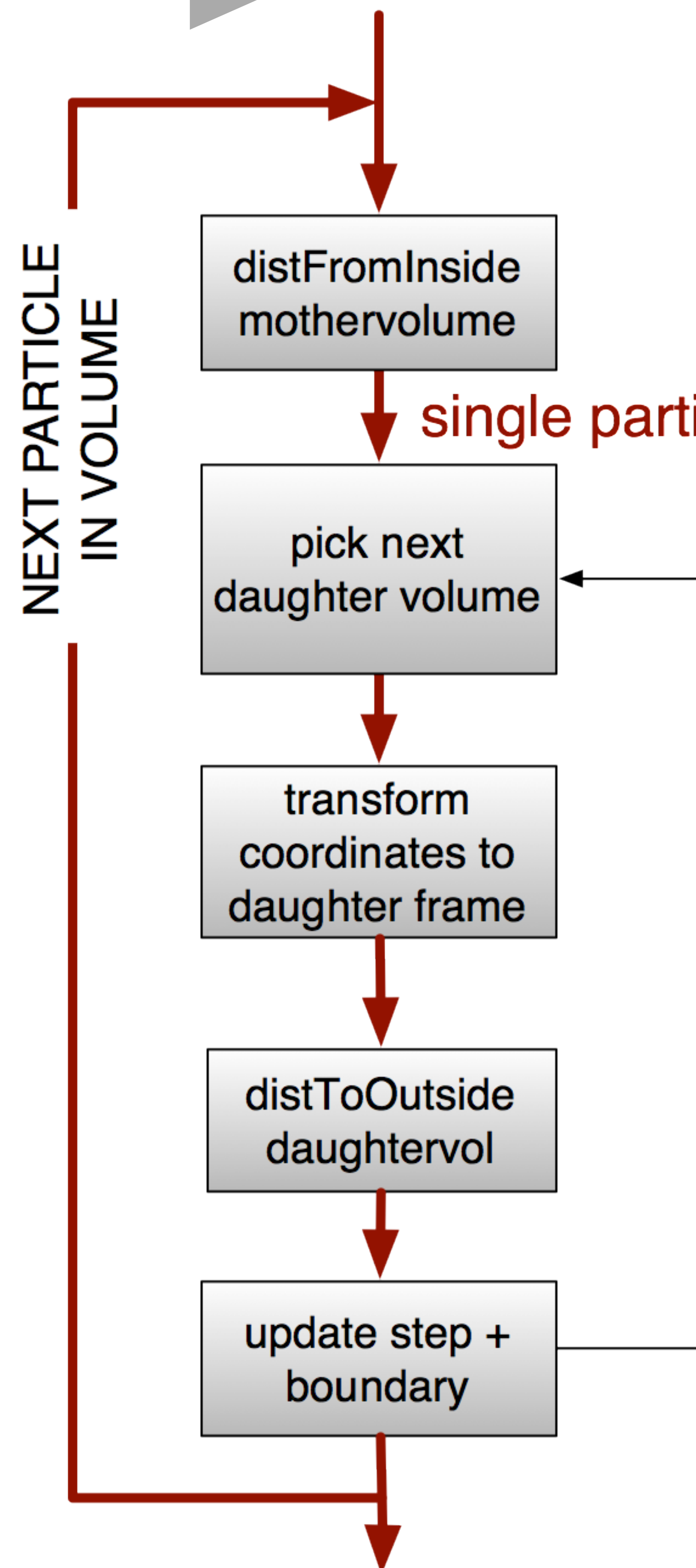
distFromInside mothervolume

N results

- Provide **new interfaces** to process vectors in basic geometry algorithms
- Make efficient use of baskets and try to use SIMD vector instructions wherever possible (**throughput optimization**)

2nd goal: vector processing within a volume

Scalar (simple) navigation versus vector navigation



single particle flow

Each particle undergoes a series of basic algorithms (with outer loop over particles)

2nd goal: vector processing within a volume

Scalar (simple) navigation versus vector navigation

NEXT PARTICLE IN VOLUME

distFromInside mothervolume

single particle flow

pick next daughter volume

transform coordinates to daughter frame

distToOutside daughtervol

update step + boundary

Each particle undergoes a series of basic algorithms (with outer loop over particles)

2. Milestone

Each algorithm takes a basket of particles and spits out vectors to the next algorithms

- fewer function calls!
- SIMD (SSE, AVX) instructions
- better code locality (icache)

SIMD distFromInside mothervolume

vector flow

pick next daughter volume

SIMD transform coordinates to daughter frame

SIMD distToOutside daughtervol

SIMD update step + boundary

The programming model

In order to use SIMD CPU capabilities, need to provide special assembly instructions (e.g. "vaddp" versus "add") to the hardware. Multiple options exist:

- **"Autovectorization":** Let the compiler figure this out himself (without code changes).
  Pro: best option for portability and maintenance
  Cons: This seldom works (but in a few cases)....
- **Explicit vector oriented programming via intrinsics:** Manually instruct the compiler to use vector instructions:
  - At the lowest level: *intrinsics*, assembler code, hard to write/read/maintain
  - At higher level: template based APIs that hide low level details like the **Vc library**

  Pro: good performance, portability, only little platform dependency (templates!)
  Cons: requires some code changes, refactoring of code
- **Language extensions**, such as Intel Cilk Plus Array notation
  - Similiar to point 2, investigated but not covered in this talk

The programming model

In order to use SIMD CPU capabilities, need to provide special assembly instructions (e.g. "vaddp" versus "add") to the hardware. Multiple options exist:

- **"Autovectorization":** Let the compiler figure this out himself (without code changes).
  Pro: best option for portability and maintenance
  Cons: This seldom works (but in a few cases)....
- **Explicit vector oriented programming via intrinsics:** Manually instruct the compiler to use vector instructions:
  - At the lowest level: *intrinsics*, assembler code, hard to write/read/maintain
  - At higher level: template based APIs that hide low level details like the **Vc library**
  Pro: good performance, portability, only little platform dependency (templates!)
  Cons: requires some code changes, refactoring of code
  code.compeng.uni-frankfurt.de/projects/Vc
- **Language extensions**, such as Intel Cilk Plus Array notation
  - Similiar to point 2, investigated but not covered in this talk

Status of simple shape/algorithm investigations

Provided vector interfaces to all shapes and optimized code to simple ROOT shapes

- **"DistFromInside"**, **"DistFromOutside", "Safety", "Contains"**
- good experience and results using the Vc programming model

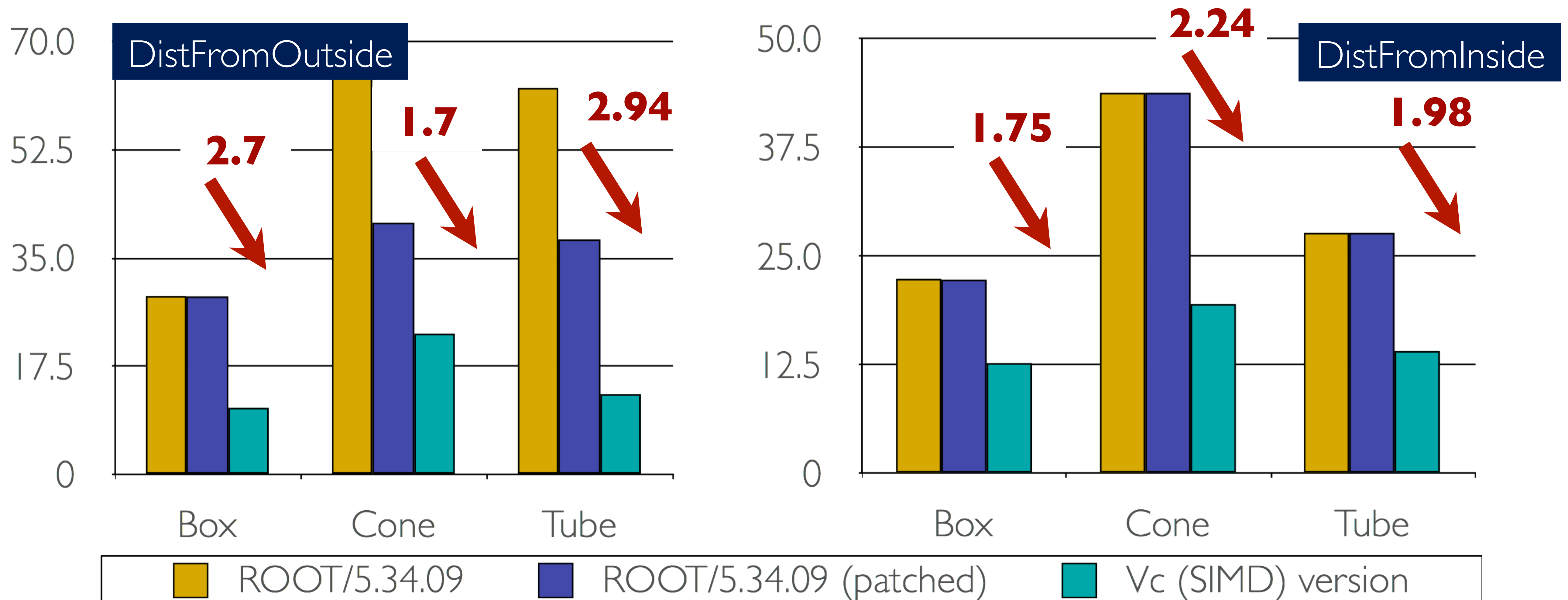
For simple shapes the **performance gains match our expectations**

Status of simple shape/algorithm investigations

Provided vector interfaces to all shapes and optimized code to simple ROOT shapes

- **"DistFromInside"**, **"DistFromOutside", "Safety", "Contains"**
- good experience and results using the Vc programming model

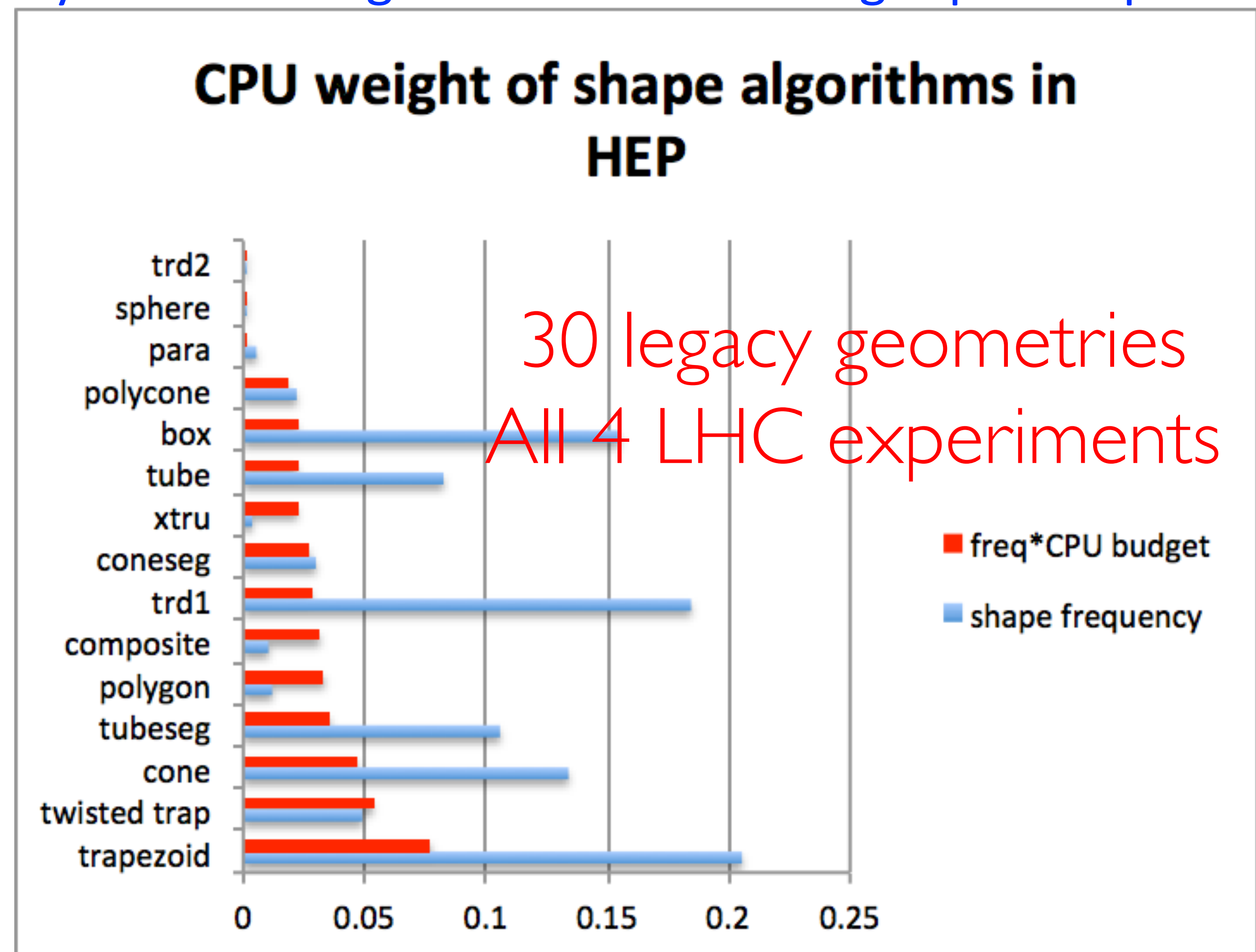
For simple shapes the **performance gains match our expectations**



comparison of processing times for 1024 particles (AVX instructions), times in microseconds

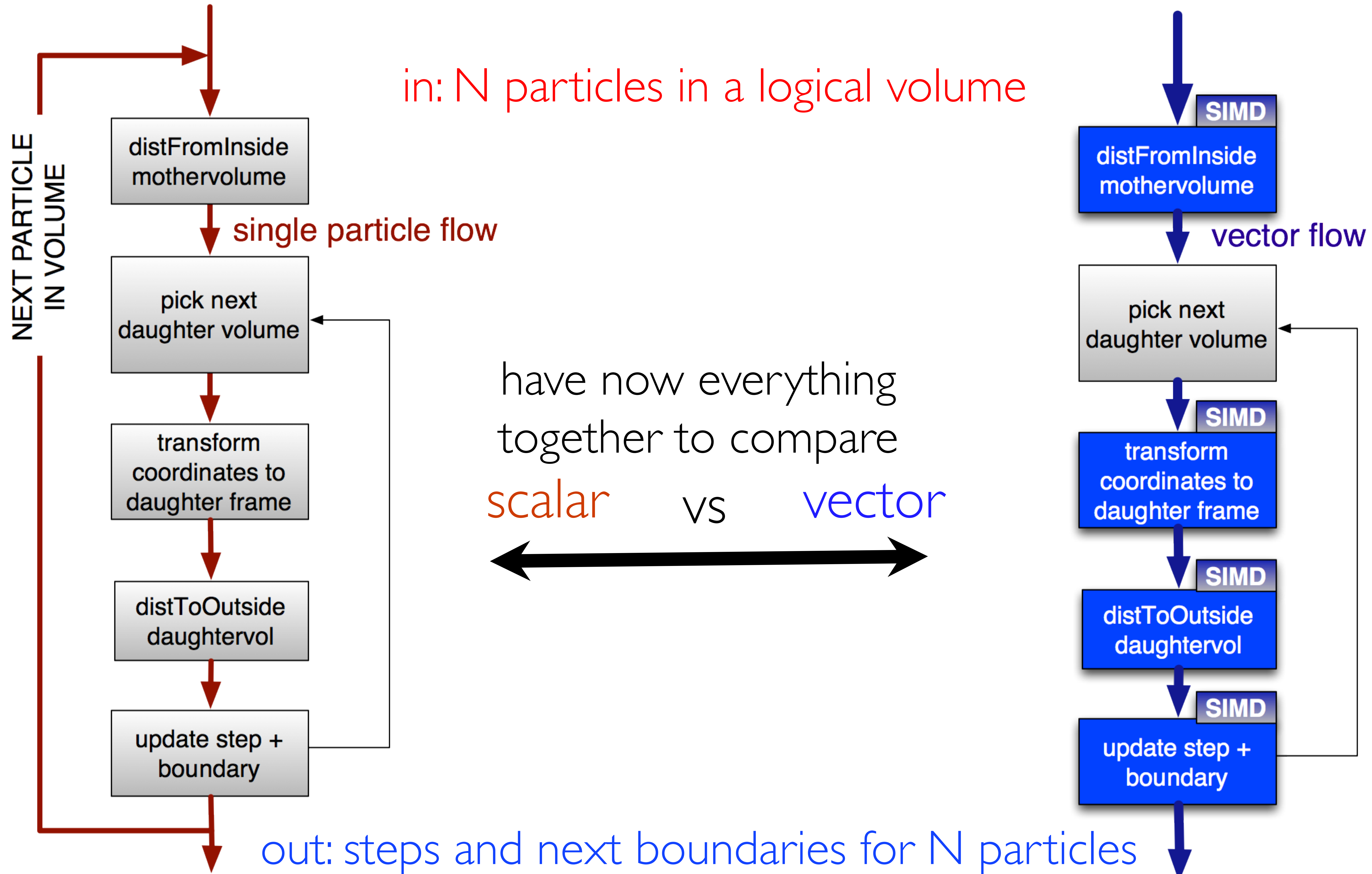
CPU budget – some first estimate

- A lot of work still to do in SIMD-optimizing more complicated shapes; preliminary results available for Polycone (backup)
  - Evaluate "CPU budget" per shape type
  - First estimate: shape frequency times average estimate CPU budget per shape



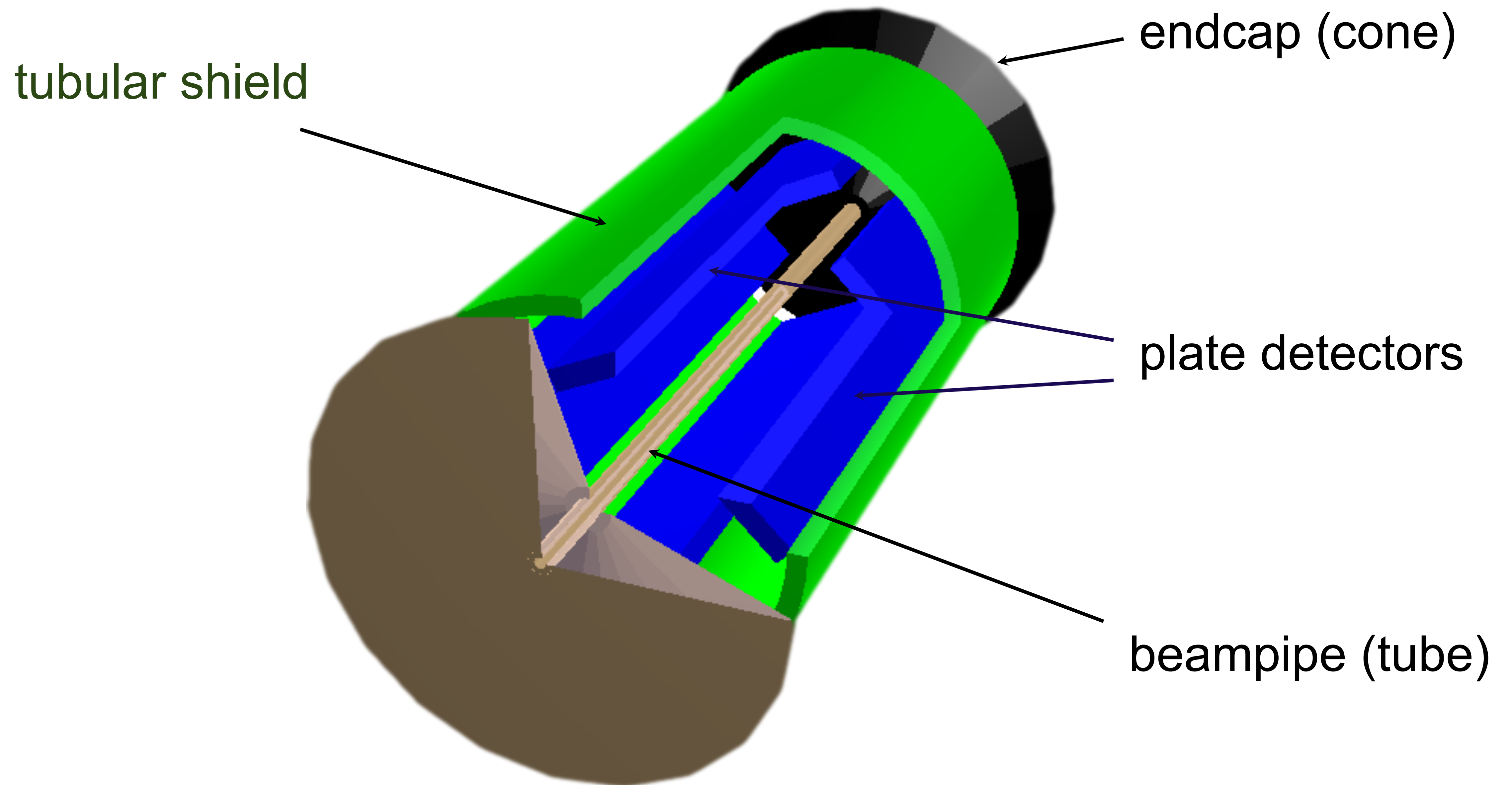
- Besides shapes, vector-optimize other simple algorithmic blocks:
  - coordinate and vector transformations ("master-to-local")
  - min, max algorithms, ...

Benchmarking the Vector Navigation



Testing a simple navigation algorithm

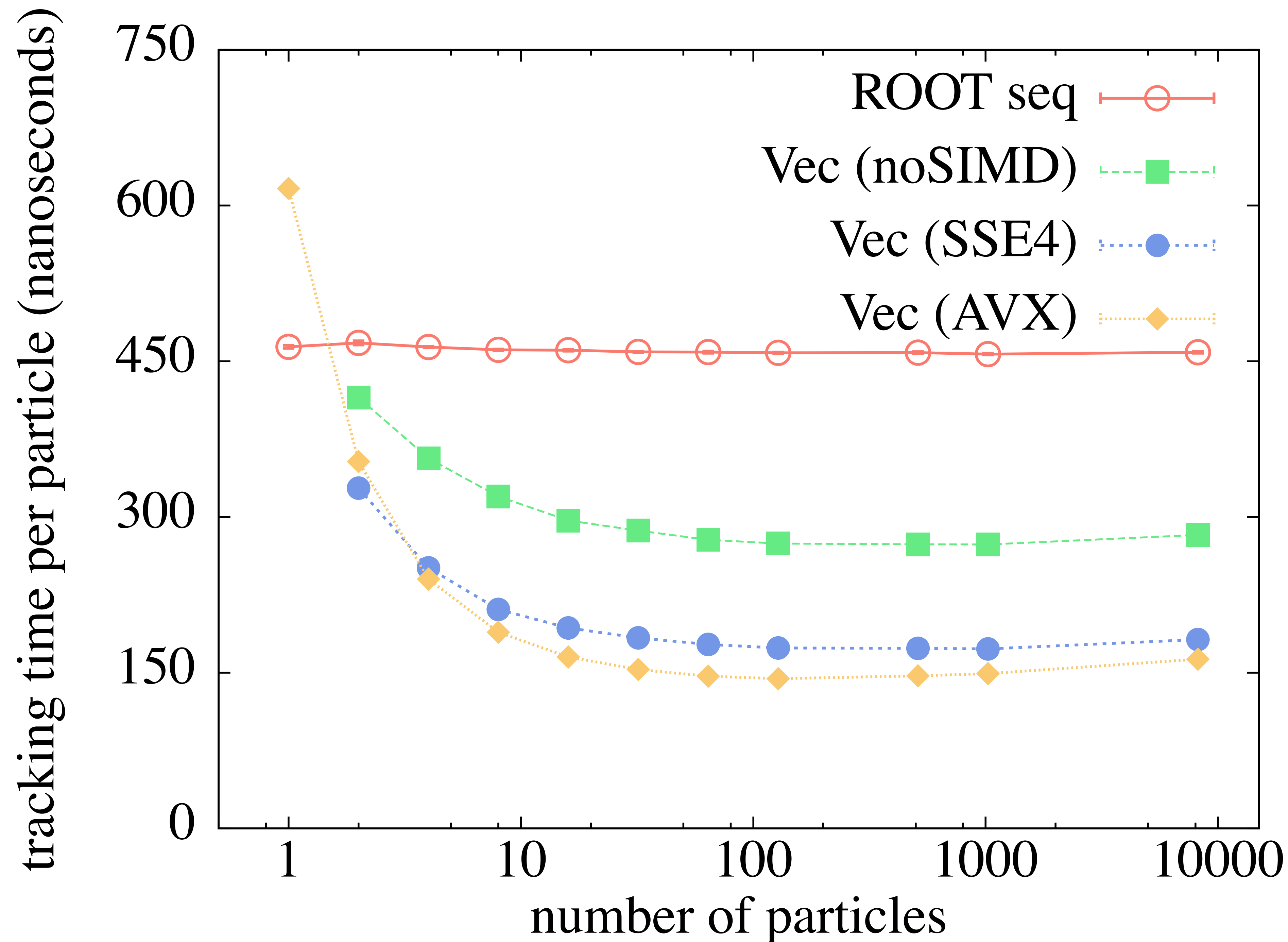
Implemented a toy detector for a benchmark: 2 tubes, 4 plate detectors, 2 endcaps (cones), 1 tubular mother volume



Logical volume filled with test particle pool (random position and random direction) from which we use a subset N for benchmarks (P repetitions)

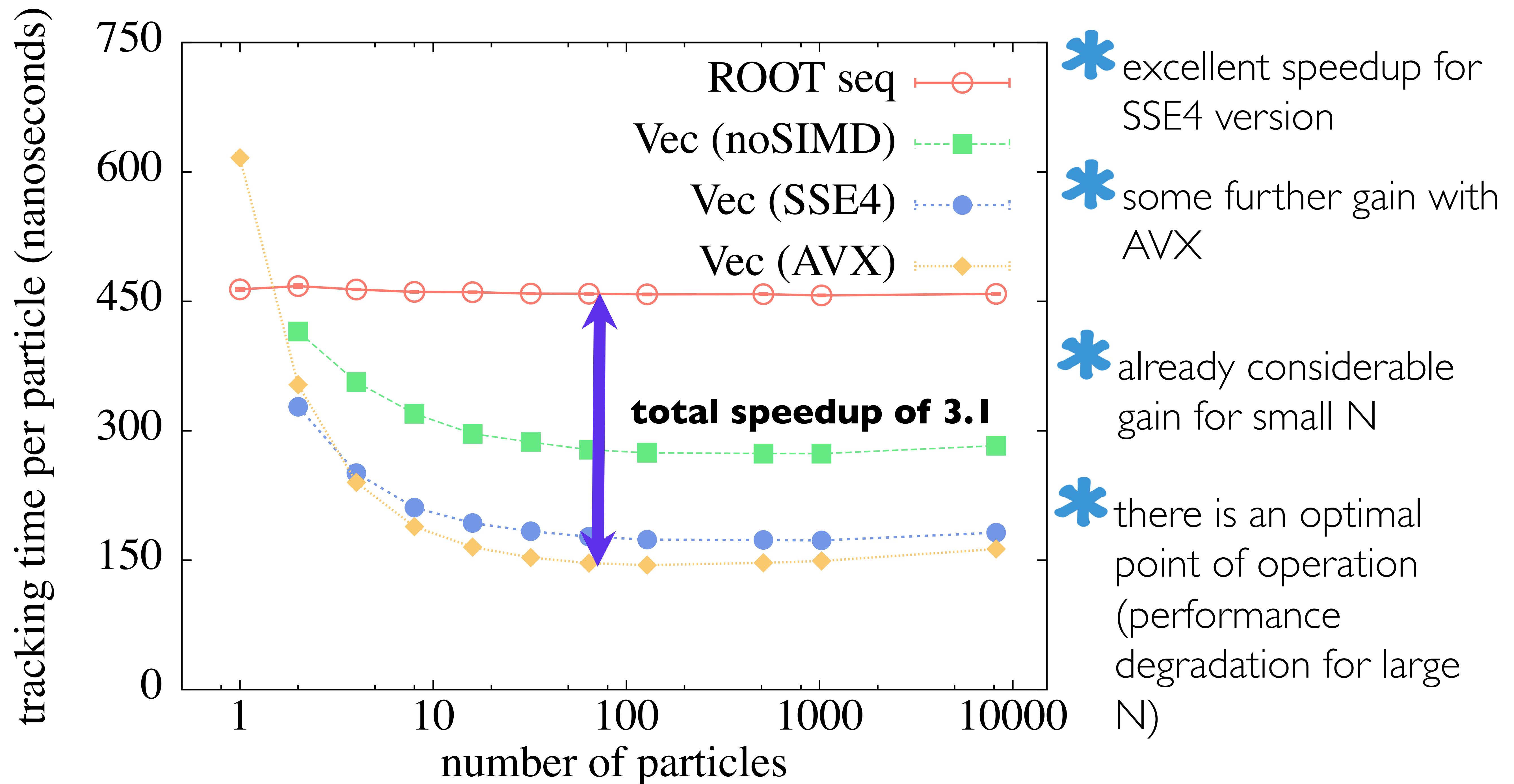
Results from Benchmark: Overall Runtime

- time of processing/navigating N particles (P repetitions) using scalar algorithm (ROOT) versus vector version



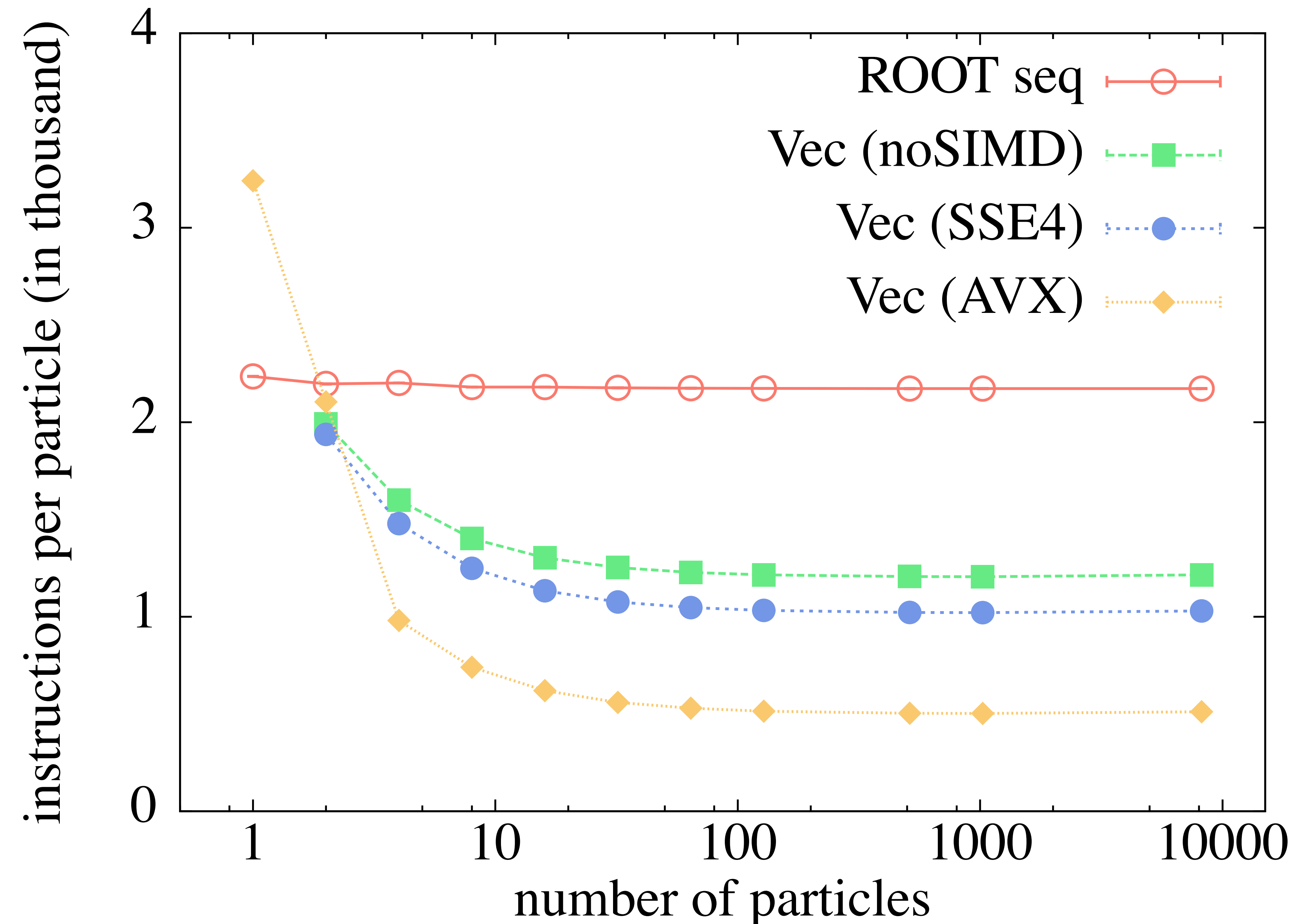
Results from Benchmark: Overall Runtime

- time of processing/navigating N particles (P repetitions) using scalar algorithm (ROOT) versus vector version



Further Metrics: Executed Instructions

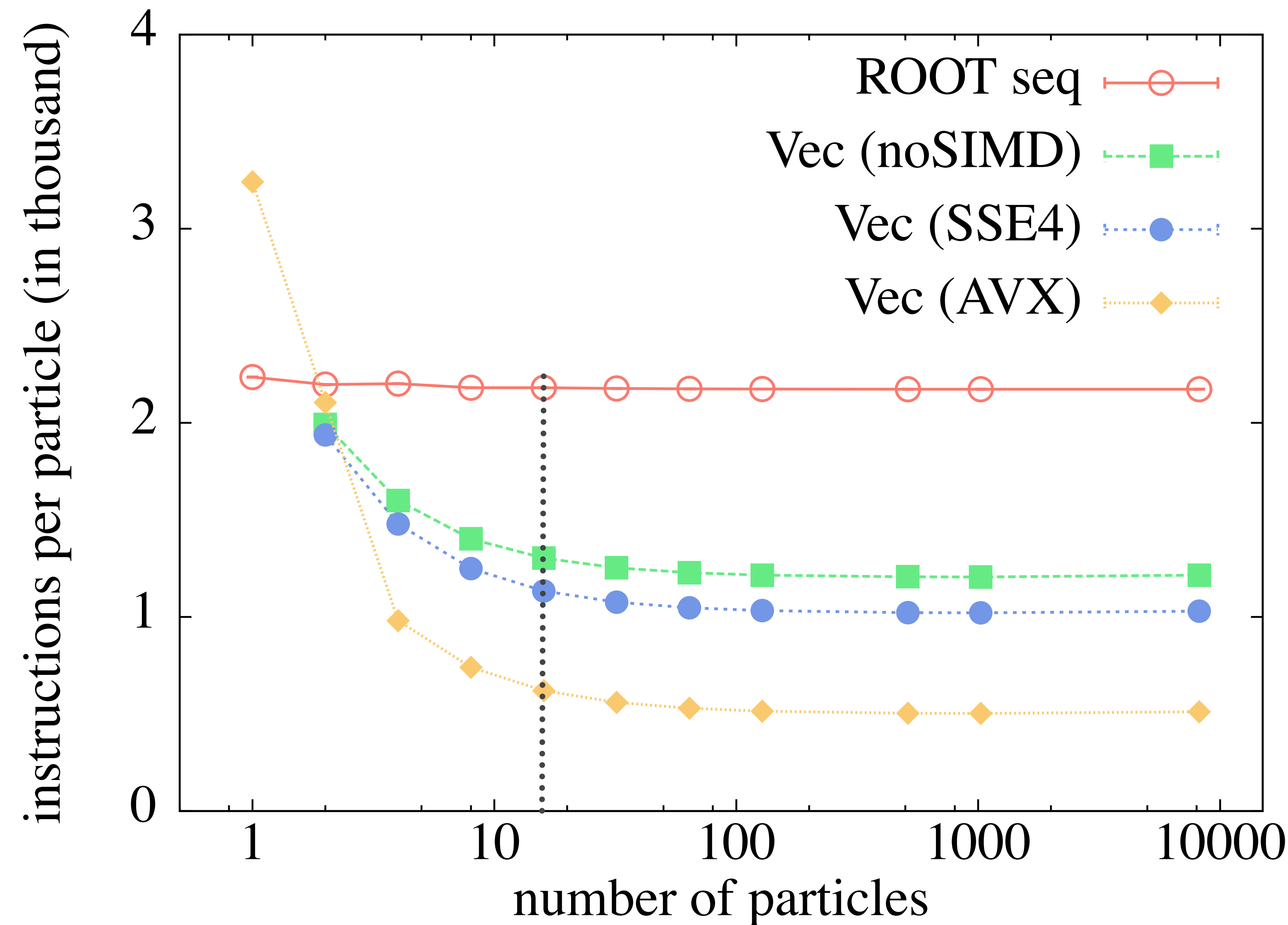
- investigate origin of speedup: study **hardware performance counters**
- developed a "timer" based approach where we read out counter before and after an arbitrary code section (using libpfm)



* gain mainly **due to fewer instructions** (for the same work)

Further Metrics: Executed Instructions

- investigate origin of speedup: study **hardware performance counters**
- developed a "timer" based approach where we read out counter before and after an arbitrary code section (using libpfm)



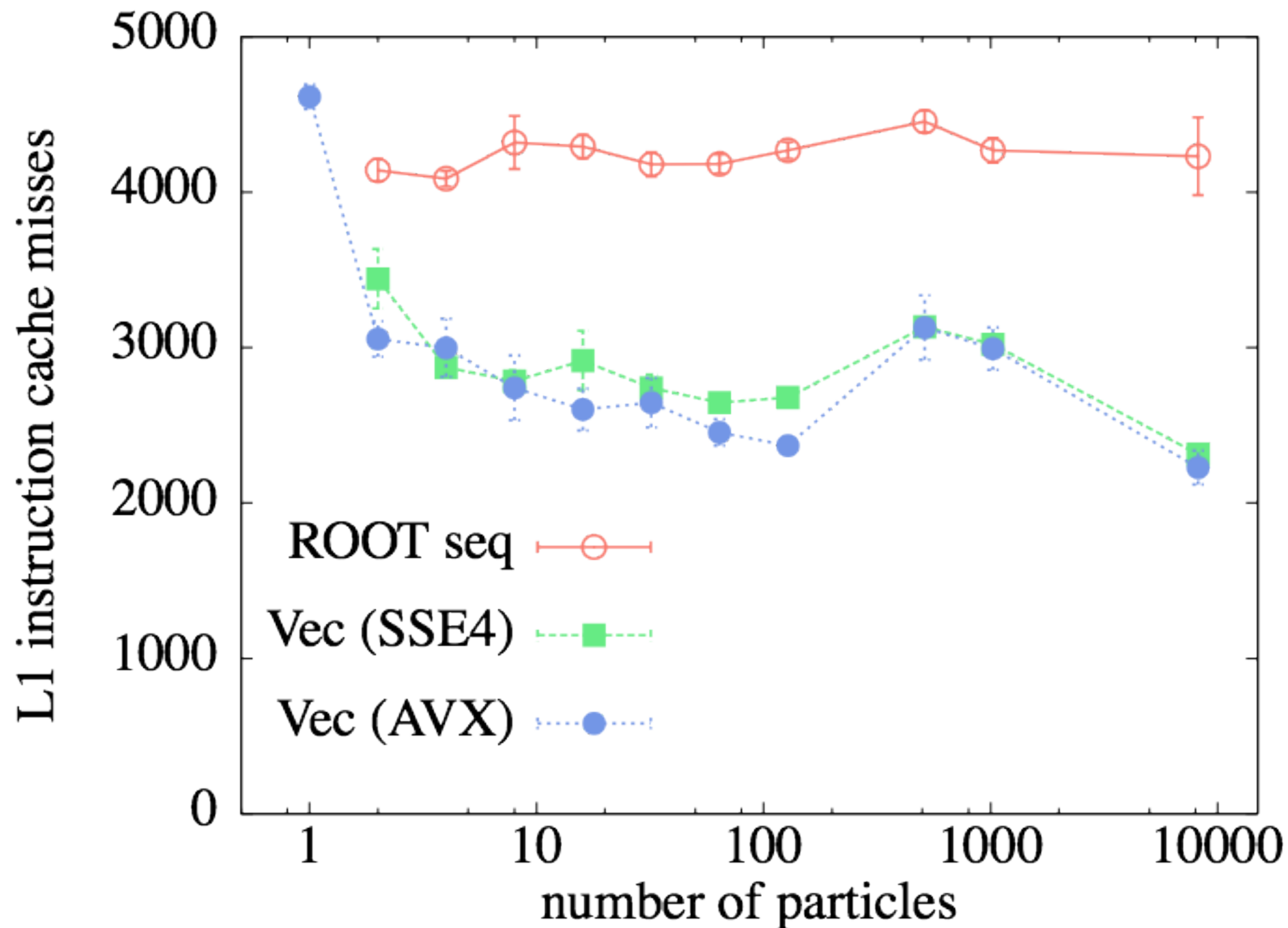
* gain mainly **due to fewer instructions** (for the same work)
* detailed analysis (binary instrumentation) can give statistics, e.g.:

	ROOT	Vec
ALL	17.5 mil	5 mil
MOV	30%	16%
CALL	4%	0.4%
V..PD (SIMD)	4%	**57%**

comparison for N=16 particles (AVX versus ROOT seq)

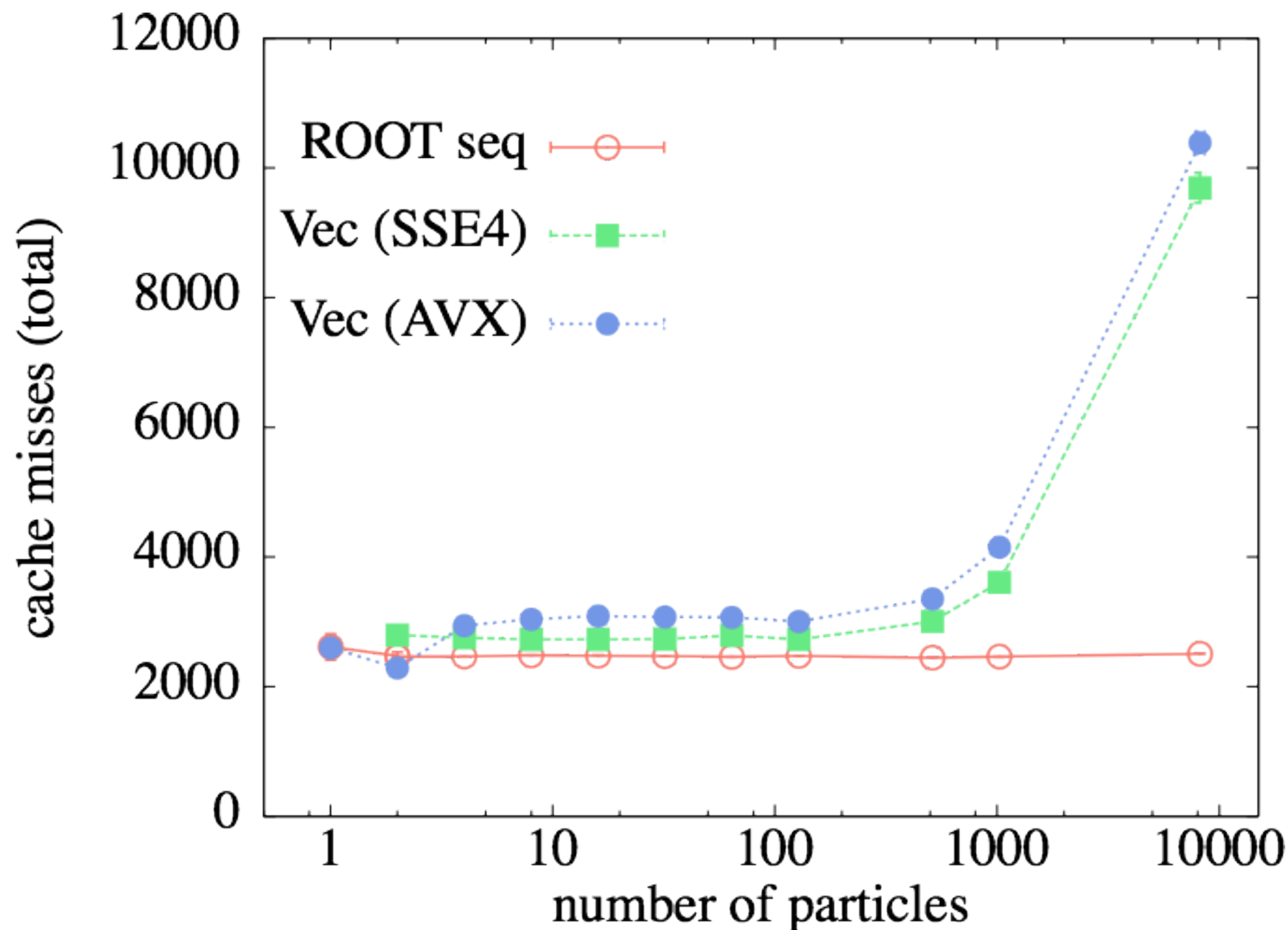
Further Metrics: L1 instruction cache misses

- Better code locality, expected to have more impact when navigation itself is embedded in more complex environment.



Further Metrics: total cache misses

- More data cache misses for large number of particles, can degrade performance
  - likely due to structure-of-array usage in vector
- Realistic N that can be scheduled is an important parameter



Summary / Outlook

- Vectorization is not threading and needs special efforts
- A vector/basket architecture allows to make use of SIMD but also increases locality (less functions calls, more instruction cache friendly)
- Provided a first refactored vector API in ROOT geometry/navigation library and showed good performance gains for individual as well as combined algorithms on commodity hardware
- Very good experience with explicit vector oriented programming model (Vc, Intel Cilk Plus Arrays)

Summary / Outlook

- Vectorization is not threading and needs special efforts
- A vector/basket architecture allows to make use of SIMD but also increases locality (less functions calls, more instruction cache friendly)
- Provided a first refactored vector API in ROOT geometry/navigation library and showed good performance gains for individual as well as combined algorithms on commodity hardware
- Very good experience with explicit vector oriented programming model (Vc, Intel Cilk Plus Arrays)

Outlook

- more complex shapes and algorithms (voxelization), USolids …
- **full flow of vectors in Geant-V prototype** (vectorisation gains vs. scatter-gather overhead)
- Gains on accelerators (GPUs, Intel Phi, ...) using vectorized code

Acknowledgements

- contributors to basic Vc coding:
  - Juan Valles (CERN summer student)
  - Marilena Bandieramonte (University of Catania, Italy)
  - Raman Sehgal (BARC, India)

- help performance analysis / investigation of Intel Cilk Plus Array Notation:
  - Laurent Duhem (Intel)
  - CERN Openlab

Backup slides

Notes on benchmark conditions

- System: Ivybridge iCore7 (4 core, not hyperthreaded (can read out 8 hardware performance counters))
- Compiler: gcc4.7.2 (compile flags -O2 -unroll-loops -ffast-math -mavx)
- OS: slc6
- Vc version: 0.73
- benchmarks usually run on empty system with cpu pinning (taskset -c)

* benchmarks use preallocated pool of test data, in which we take out N particles for processing. Repeat this P times. For repetitions distinguish between random access of N particles (higher cache impact) or sequential access in data pool (as shown here)

* benchmarks shown use NxP=const to time an overall similar amount of work

Backup: A need for tools...

- converting code for data parallelism can be a pain ... (see challenges)
- would be nice to have better tool support for this task, helping at least with often recurring work

A possible direction:

- **source-to-source transformations** (preprocessing)
  - provide trivial vectorized code version of a function
  - unroll inner loops, rewrite early returns, ...
  - Clang/LLVM API very promising for this ... currently investigating

Some tools go into this direction:

- Scout (TU Dresden): Can take code within a loop and **emit instrinsics** code for all kinds of architectures
  - could be used in situations where the compiler does not autovectorize

Example of Vc programming

```
void foo(double const *a,
         double const *b,
         double * out, int np){
    for(int i=0;i<np;i++)
    {
        out[i]=b[i]*(a[i]+b[i]);
    }
}
```

- example in plain C
  - although simple and data parallel (probably) does not vectorize without further hints to the compiler ("restrict")

- example in Vc
  - restructuring the loop stride
  - explicit inner vector declaration
  - always vectorizes (no other hints necessary)
  - architecture independent because Vc::double_v::Size is template constant determined at compile time
  - portable
  - branches/masks supported

```
void foo(double const *a,
         double const *b,
         double * out, int np){
    for(int i=0;i<np;i+=Vc::double_v::Size)
    {
        // fetch chunk of data into Vc vector
        Vc::double_v a_v(&a[i]);
        Vc::double_v b_v(&b[i]);

        // computation just as before
        b_v = b_v*(a_v + b_v);

        // store back result into output array
        b_v.store( &out[i] );
    }
    // tail part of loop has to follow
}
```

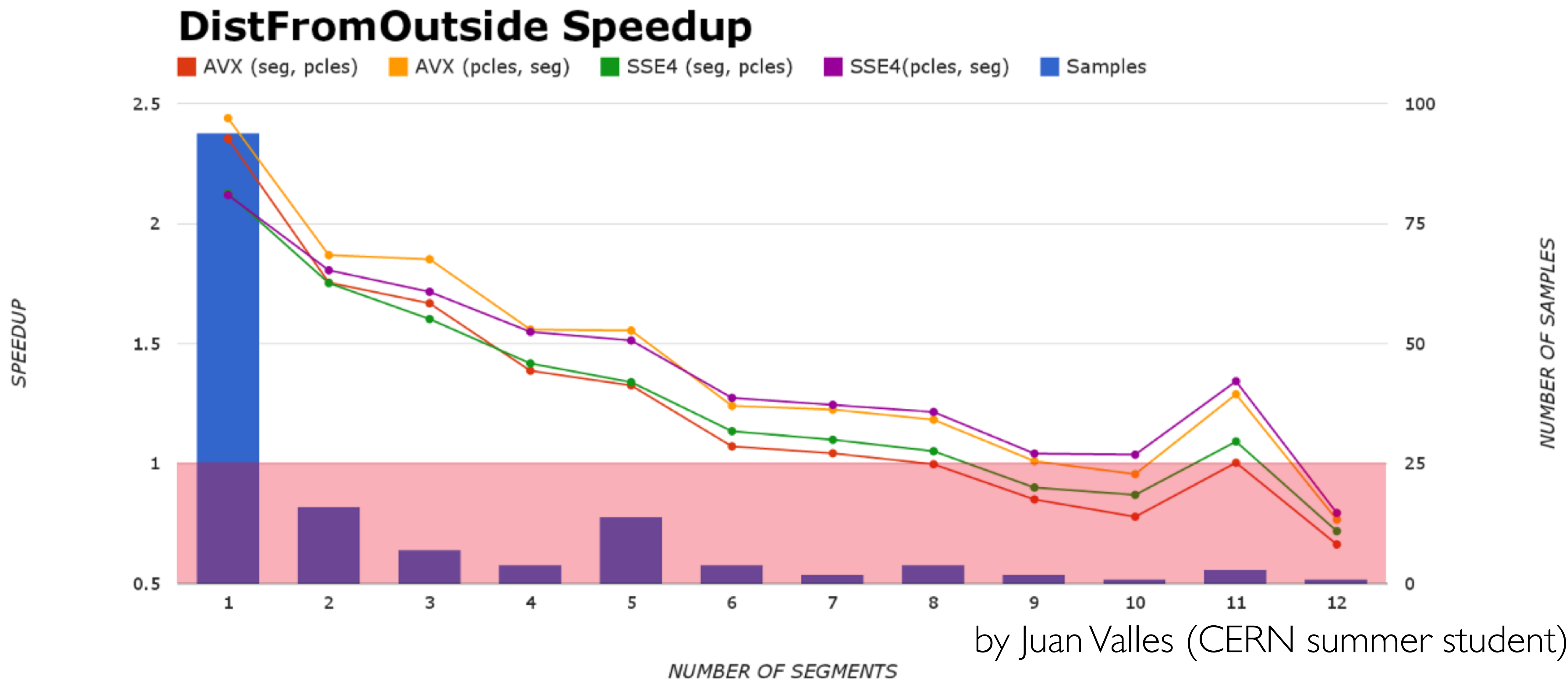
Example with Intel Cilk Plus Array Notation

- Intel Cilk Plus Array Notation indicates to the compiler operations on parallel data and leads to better autovectorization
- Programming can be similar to Vc (but with seemingly more code bloat at the moment) -- somewhat constructed example (is possible in easier manner as well)
- working with small vectors of VecSize wanted because allows for "early returns", finer control

```
// CEAN example
void foo(double const * a,
         double const *b,
         double * out, int np)
{
    int const VecSize=4;
    for(int i=0;i<np;i+=VecSize)
    {
        // cast input as fixed size vector
        double const (*av)[VecSize] = (double const (*)[VecSize]) &a[i];
        double const (*bv)[VecSize] = (double const (*)[VecSize]) &b[i];

        // give compiler hints
        __assume_aligned(av,32);
        __assume_aligned(bv,32);

        // cast output as fixed size vector
        double (*outv)[VecSize] = (double (*)[VecSize]) &out[i];
        __assume_aligned(outv,32);

        // computation and storage in CILK PLUS ARRAY NOTATION
        // will vectorize
     outv[0][:] = bv[0][:]*(av[0][:] + bv[0][:]);
    }
}
```

Status for more complex shapes

- Polycone is one of the more important complex shape used in detector descriptions
- algorithm used in ROOT uses recursive function calls which are not directly translatable to a Vc-type programming; similar for modern approach in USolids which uses voxelization techniques
- Using a simple brute force approach (for all particles just test all segments) has shown to give performance improvements for smaller polycones

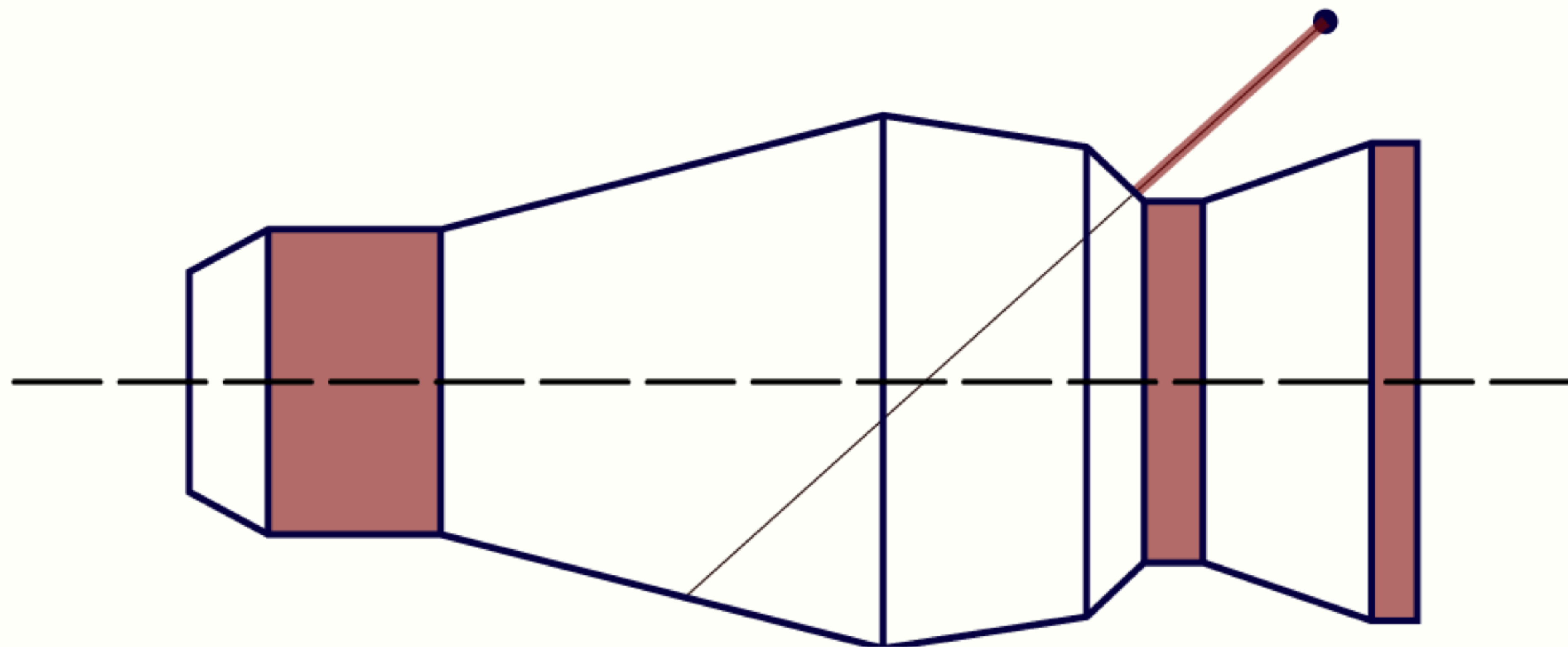


by Juan Valles (CERN summer student)

A differend kind of data parallelism

"From particle data parallelism to segment (data) parallelism"

- for large polycons could use try to vectorize over segments instead of particles (currently developing)
- similar idea could work even for voxelized / tesselated solids



"evaluate distance to shaded segments in a vectorized fashion"