Refactoring and Optimizing Geometry Routines for (SIMD) Vector Particle Processing

-- goals and status report --

R&D!

Sandro Wenzel / CERN-PH-SFT

(for the "Geant-Vector Prototype" team)

Geant4 collaboration meeting, Sevilla, 24.09.2013

Motivation

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

Motivation

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

Dimension 1 (“sharing data”) : multithreading/ multicore

In HEP, mainly to reduce memory footprint

→ Geant4 Release 10!

Motivation

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

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

In HEP, mainly to reduce memory footprint

⇨ Geant4 Release 10!

Dimension II ("troughput increase") : incore micro-parallelism or vectorization

⇨ 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

Dimension I (“sharing data”) : multithreading/ multicore

In HEP, mainly to reduce memory footprint

⇒ Geant4 Release 10!

Dimension II (“troughput increase”) : incore micro-parallelism or vectorization

⇒ Currently not exploited because requires “parallel data” to work on

Research projects (GPU prototype and Geant-Vector Prototype) have started targeting beyond dimension I (see session Thursday):

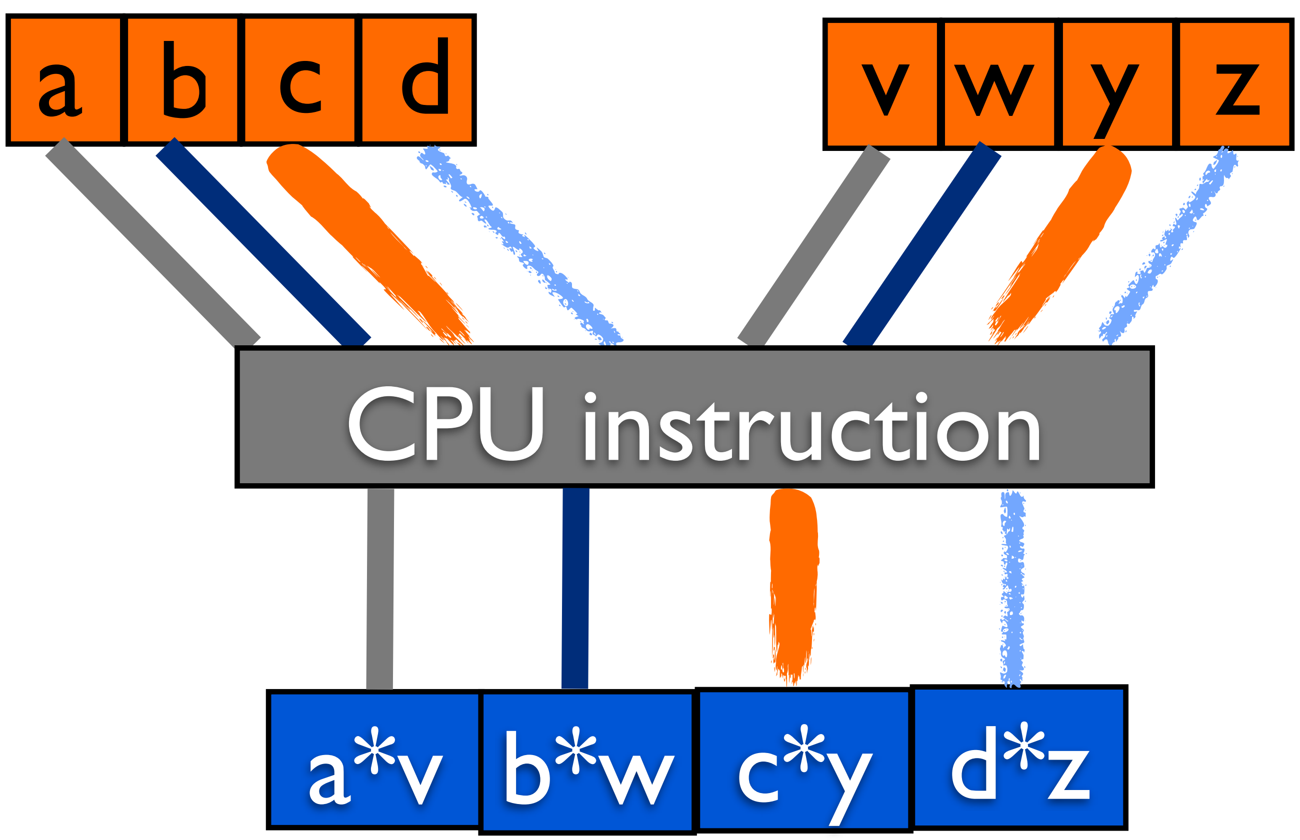
parallel data (“baskets”) = particles from different events grouped by logical volumes

Reminder of vector-microparallelism

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

single instruction multiple data = SIMD

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



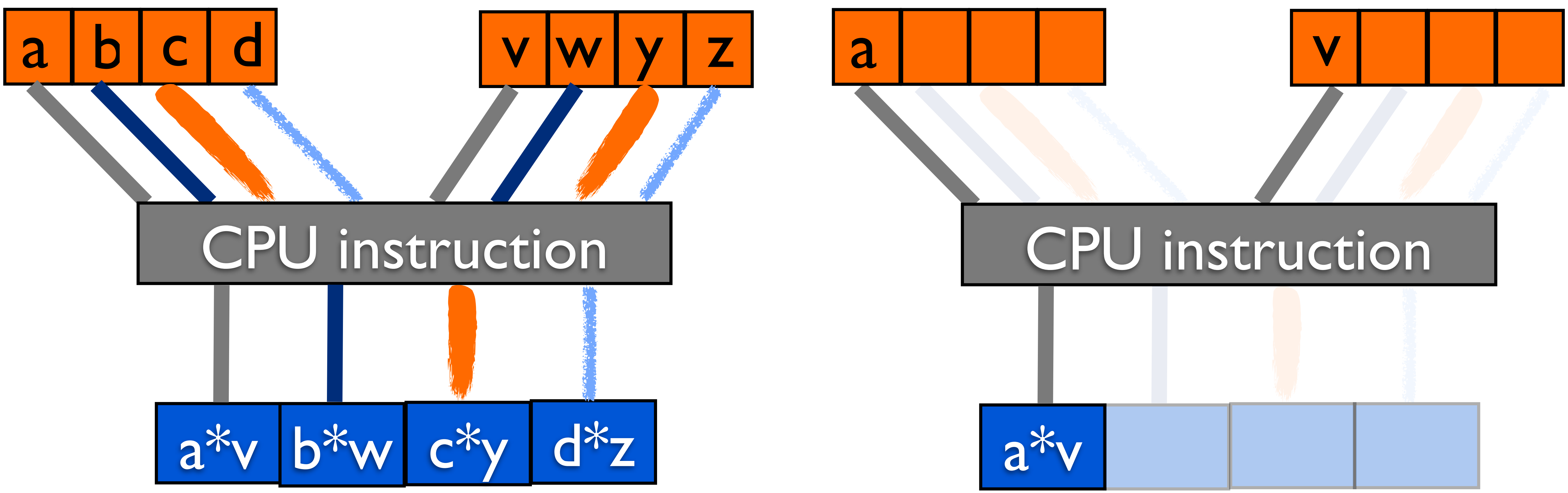
Reminder of vector-microparallelism

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

single instruction multiple data = SIMD

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

optimal usage (vector registers full) **current usage** (3/4 empty for AVX)



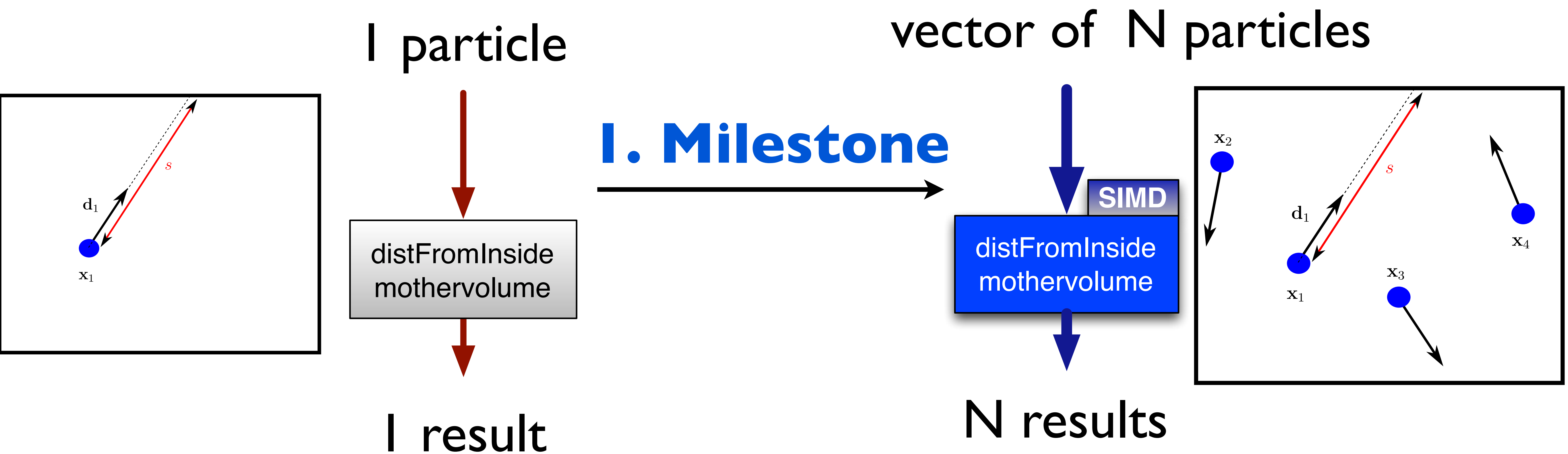
We are loosing factors!

1st Goal: Vector Processing in Simple Geometry Algorithms

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

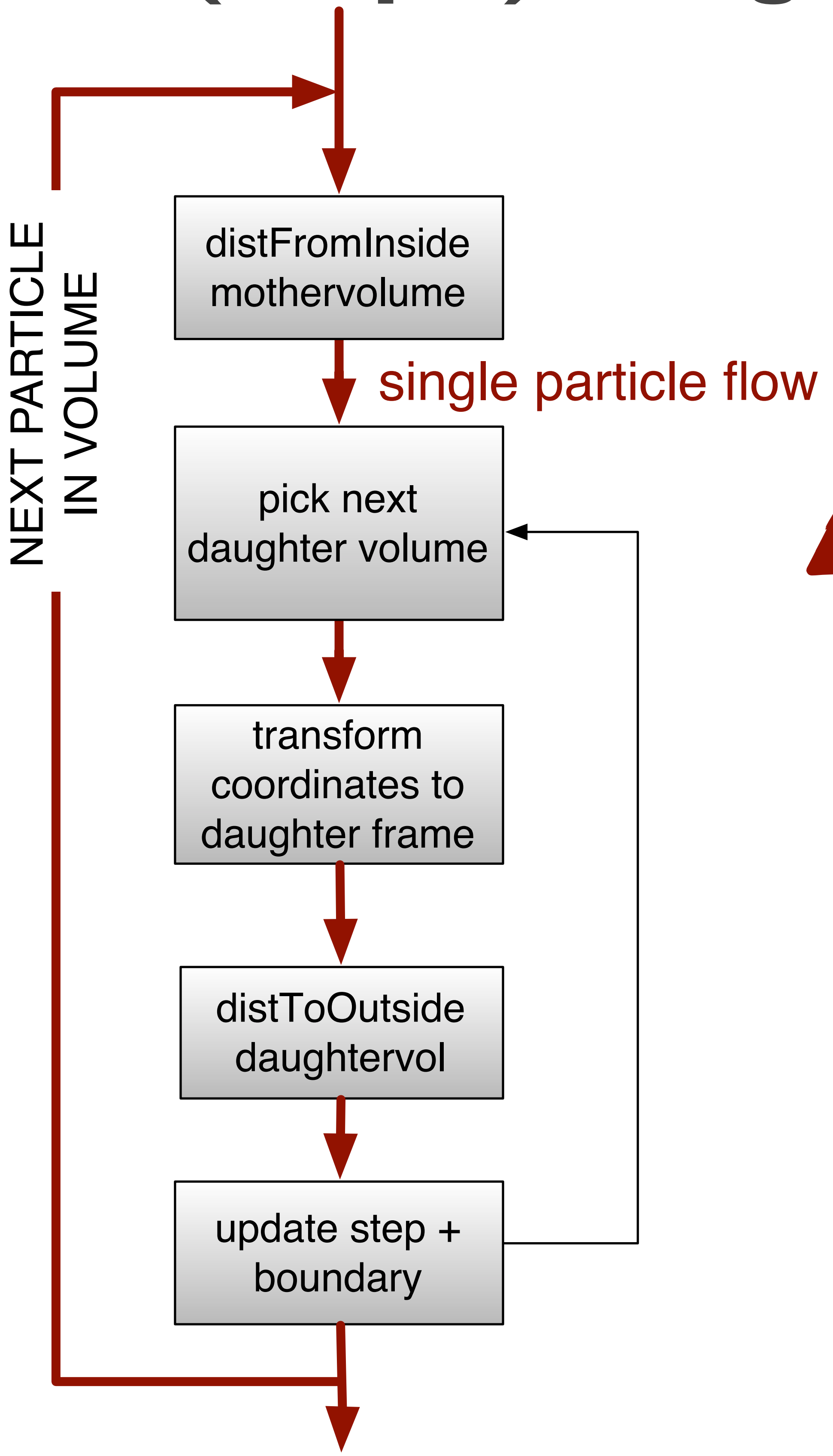
1st Goal: Vector Processing in Simple Geometry Algorithms

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



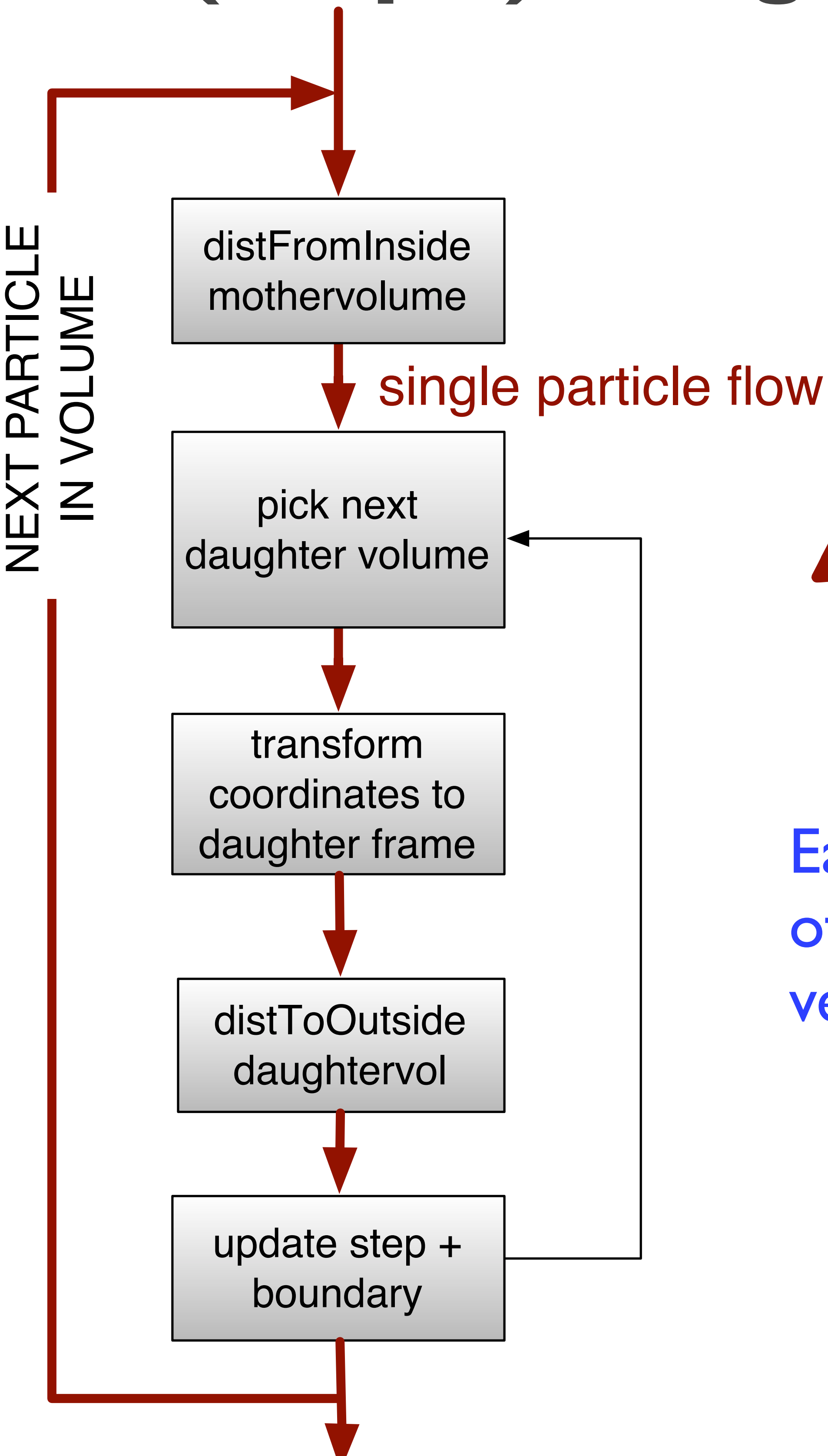
- Provide **new interfaces** to process baskets in basic geometry algorithms
- make efficient use of baskets and try to use SIMD vector instructions wherever possible (**throughput optimization**)

Vector processing in complex algorithm: Scalar (simple) navigation versus vector navigation



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

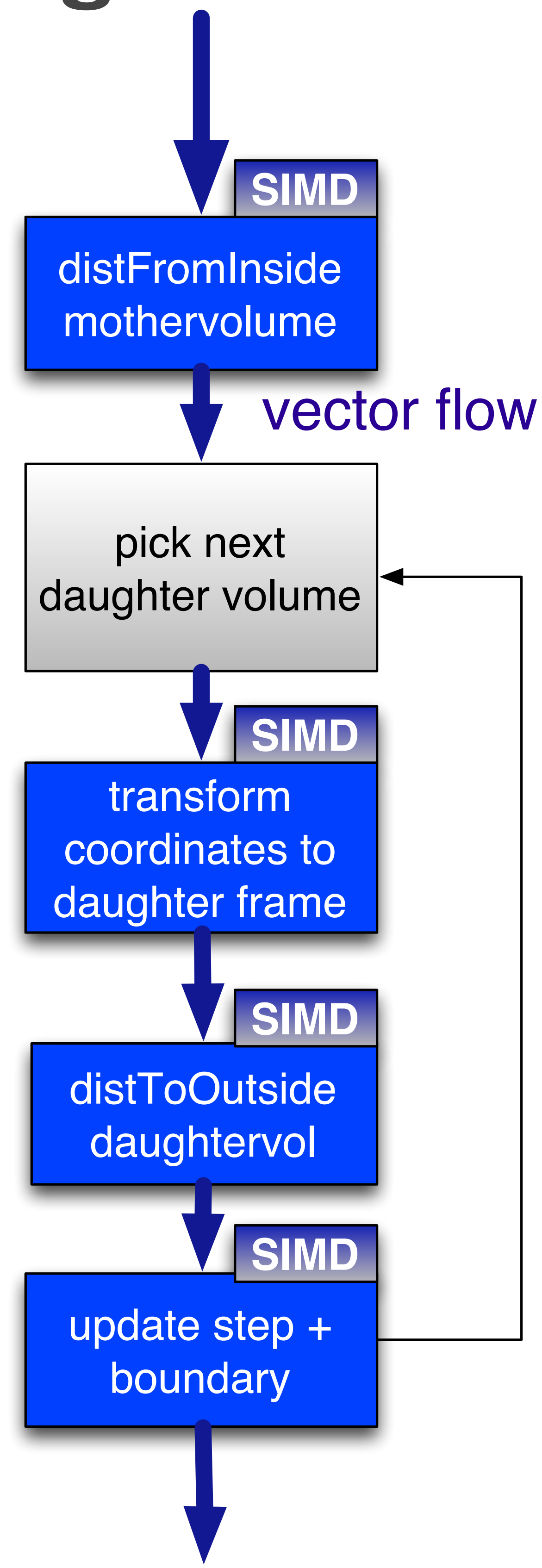
Vector processing in complex algorithm: Scalar (simple) navigation versus vector navigation



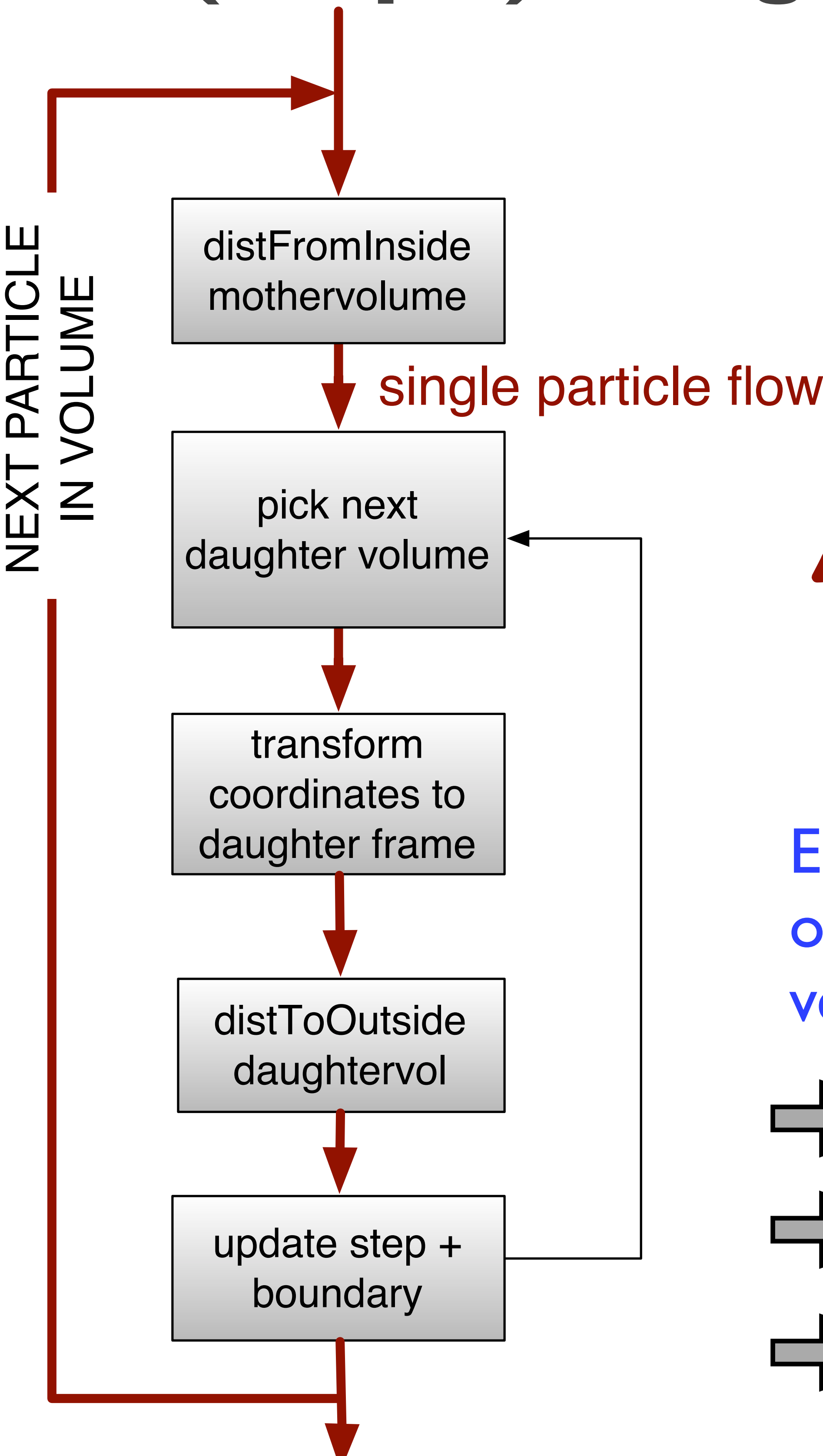
each particles undergoes a series of basic algorithms (with outer loop over particles)

2. Milestone

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



Vector processing in complex algorithm: Scalar (simple) navigation versus vector navigation

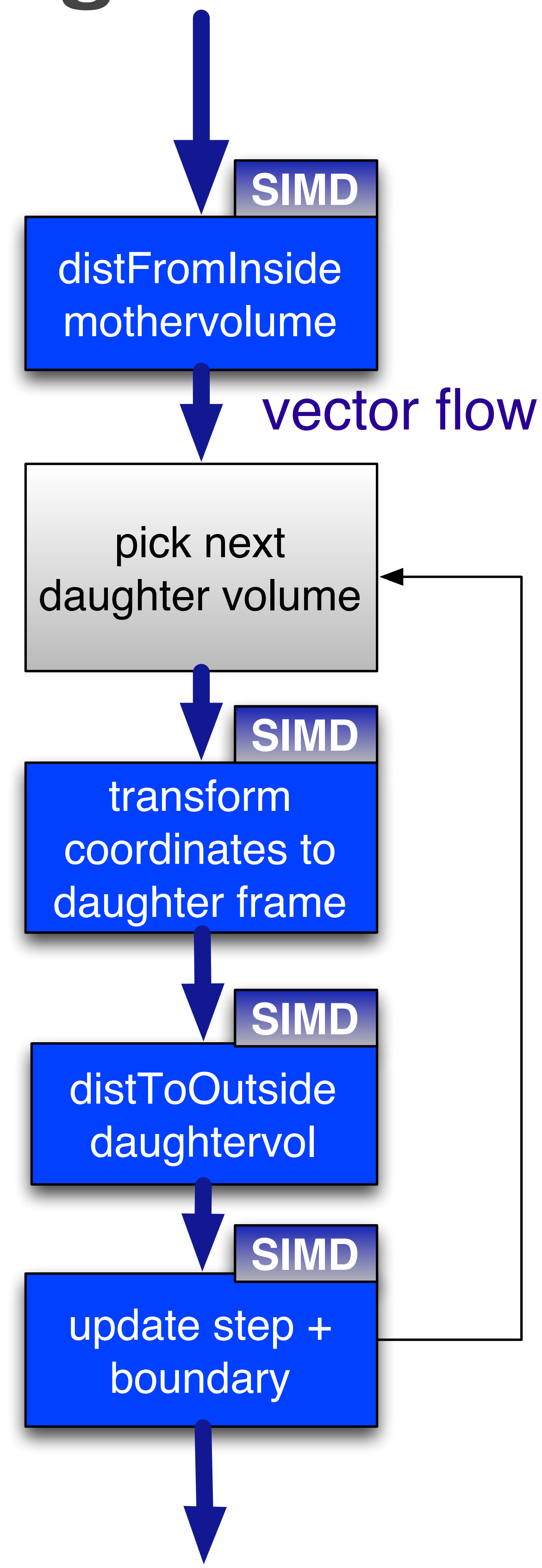


each particles undergoes a series of basic algorithms (with outer loop over particles)

2. Milestone

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

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



The programming model

In order to use SIMD CPU capabilities, need to emit special assembly instructions ("add" versus "vaddp") 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 currently never works (but in a few cases)....

The programming model

In order to use SIMD CPU capabilities, need to emit special assembly instructions ("add" versus "vaddp") 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 currently never 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
  - 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

The programming model

In order to use SIMD CPU capabilities, need to emit special assembly instructions ("add" versus "vaddp") 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 currently never 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
  - 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 emit special assembly instructions ("add" versus "vaddp") 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 currently never 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
  - 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

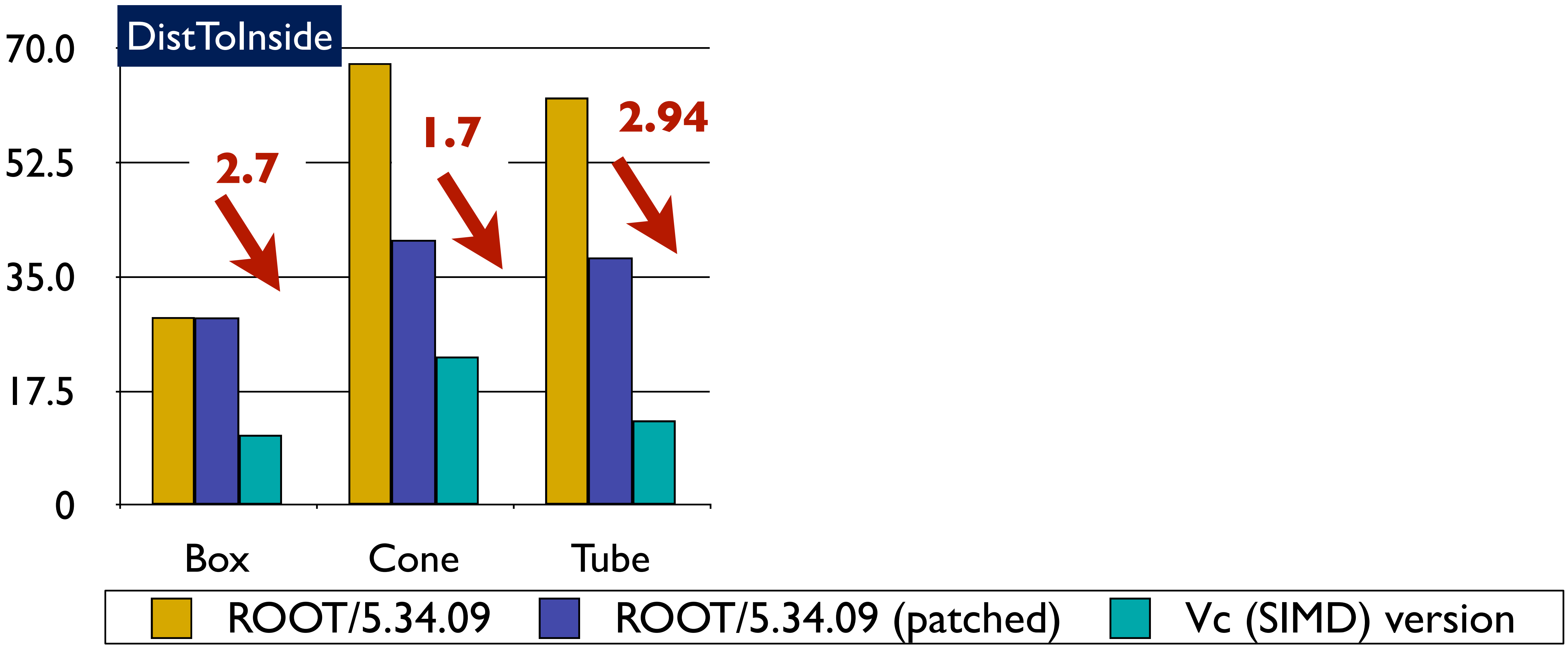
- **language extensions**, such as Intel Cilk Plus Array notation
  - similiar to point 2, investigated but not covered in this talk

Status of simple shape/algorithm investigations

- provided vector interfaces to all shapes and optimized code to simple shapes for functions
  - **“ DistToInside”**, **“DistToOutside”, “Safety”, “IsInside/Contains”**
  - here: using the ROOT shapes (but USolids will come)
  - obtained good experience and results using the Vc programming model
- For simple shapes the **performance gains match our expectations**

Status of simple shape/algorithm investigations

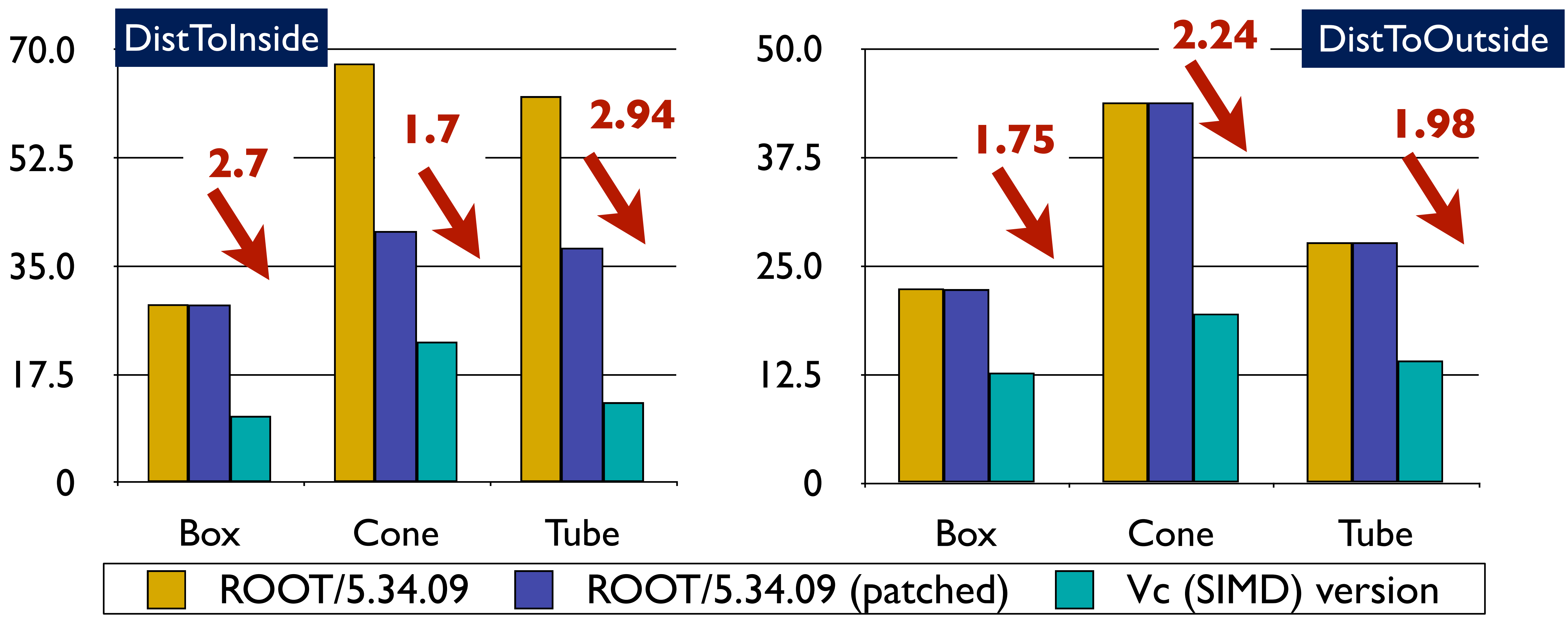
- provided vector interfaces to all shapes and optimized code to simple shapes for functions
  - **" DistToInside", "DistToOutside", "Safety", "IsInside/Contains"**
  - here: using the ROOT shapes (but USolids will come)
  - obtained good experience and results using the Vc programming model
- For simple shapes the **performance gains match our expectations**



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

Status of simple shape/algorithm investigations

- provided vector interfaces to all shapes and optimized code to simple shapes for functions
  - **" DistToInside", "DistToOutside", "Safety", "IsInside/Contains"**
  - here: using the ROOT shapes (but USolids will come)
  - obtained good experience and results using the Vc programming model
- For simple shapes the **performance gains match our expectations**

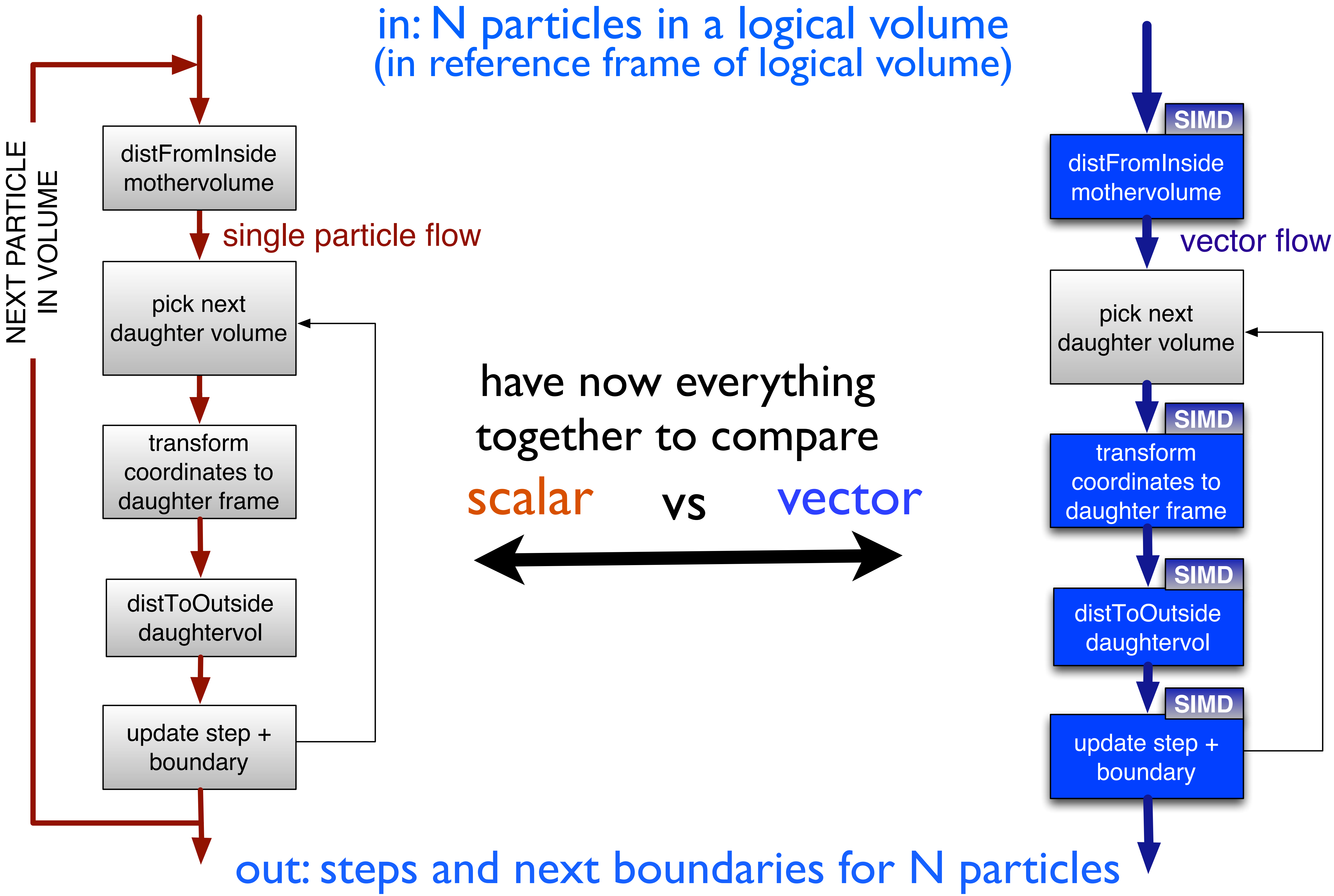


comparison of processing times for 1024 particles (AVX instructions), times in microseconds

Status of refactoring simple algorithms (II)

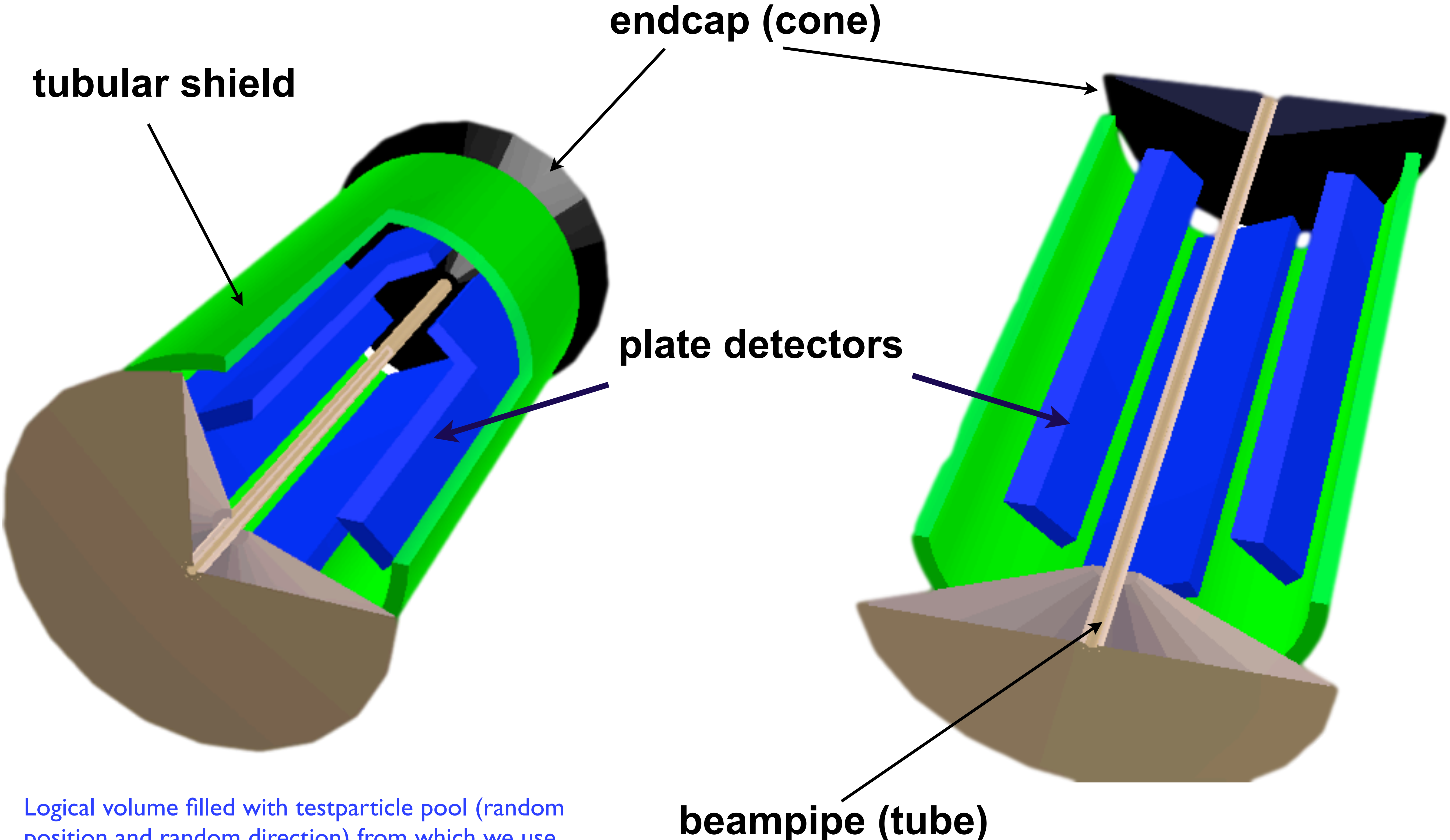
* a lot of work still to do in SIMD-optimizing more complicated shapes; preliminary results available for Polycone (backup)
* outside shapes, vector-optimized other simple algorithmic blocks:
  - coordinate and vector transformations ("master-to-local")
  - min, max algorithms, ...

Benchmarking the Vector Navigation



Need a simple toy detector as logical volume

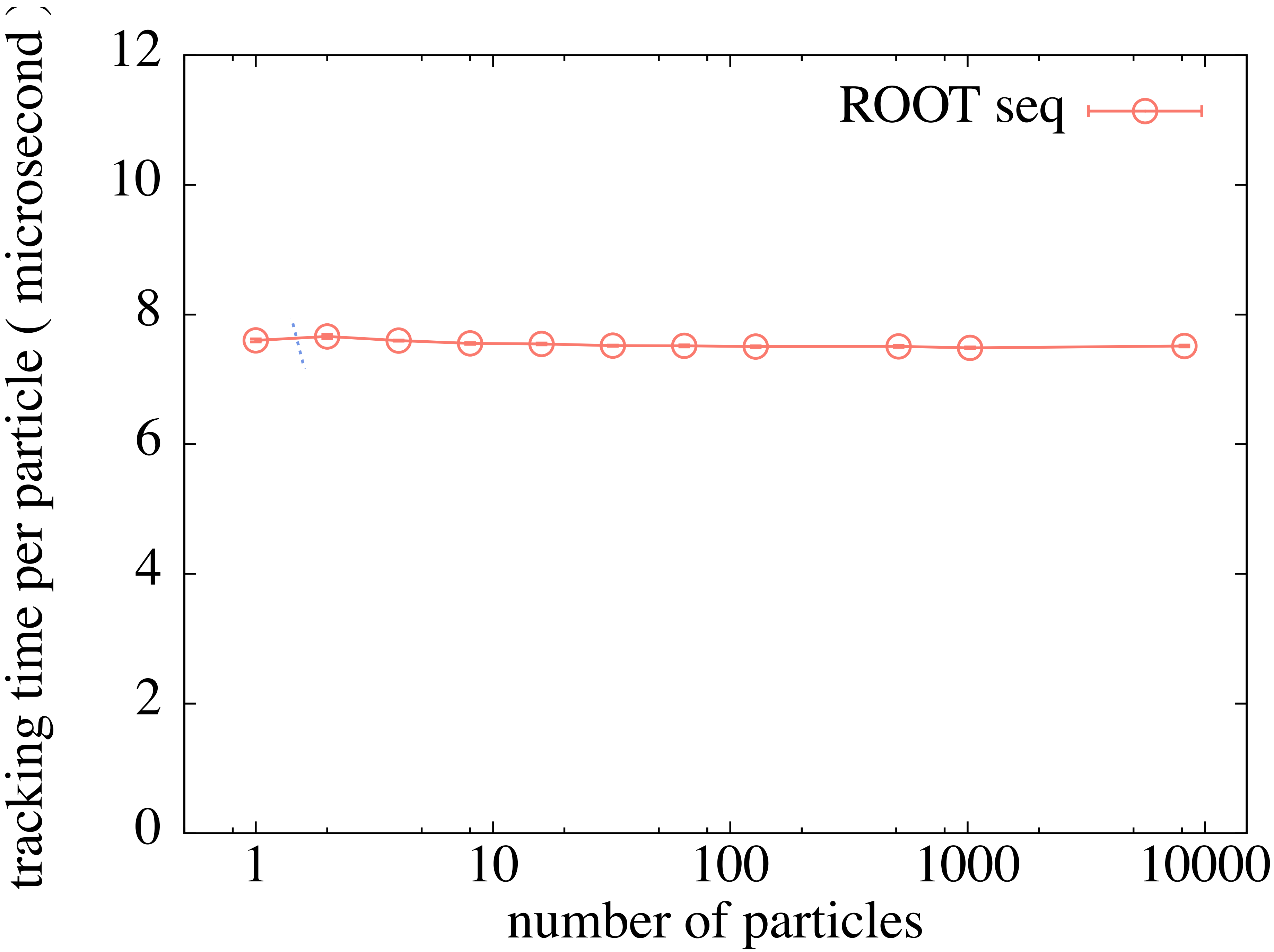
implemented a toy detector for a benchmark ("not to easy; not too complex"): 2 tubes, 4 plate detectors, 2 endcaps (cones), 1 tubular mother volume



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

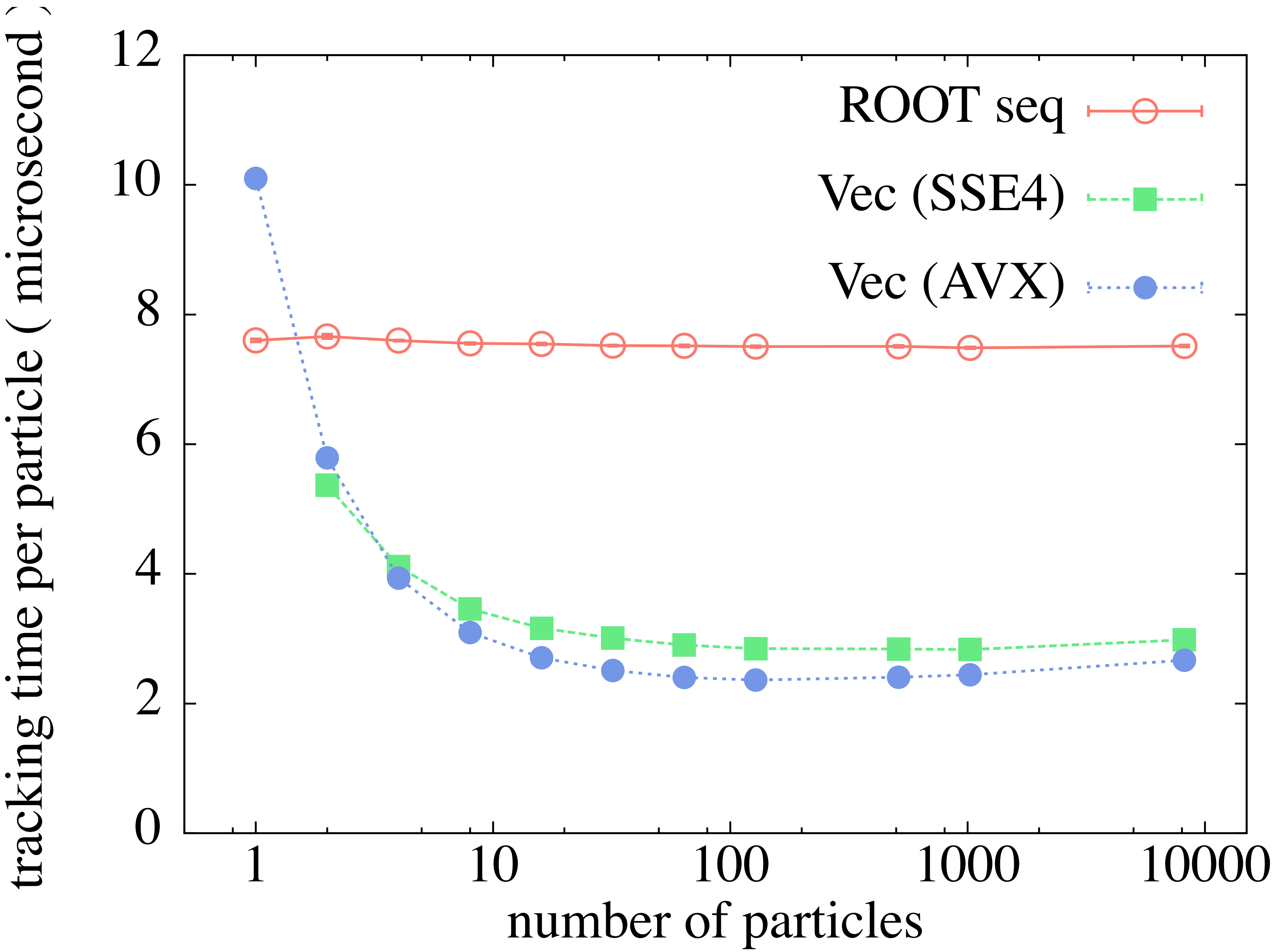
Results from Benchmark: Overall Runtime

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



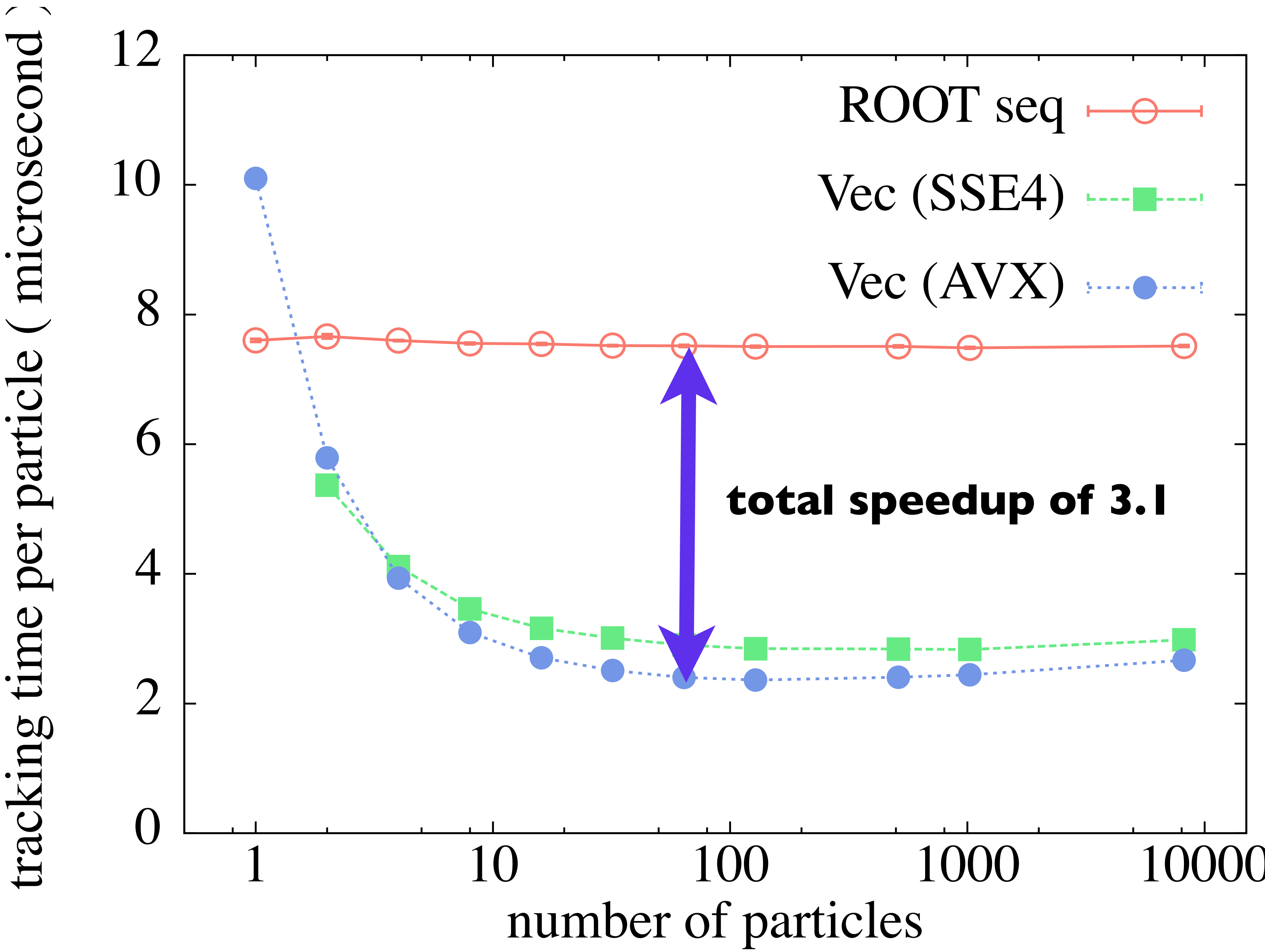
Results from Benchmark: Overall Runtime

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



Results from Benchmark: Overall Runtime

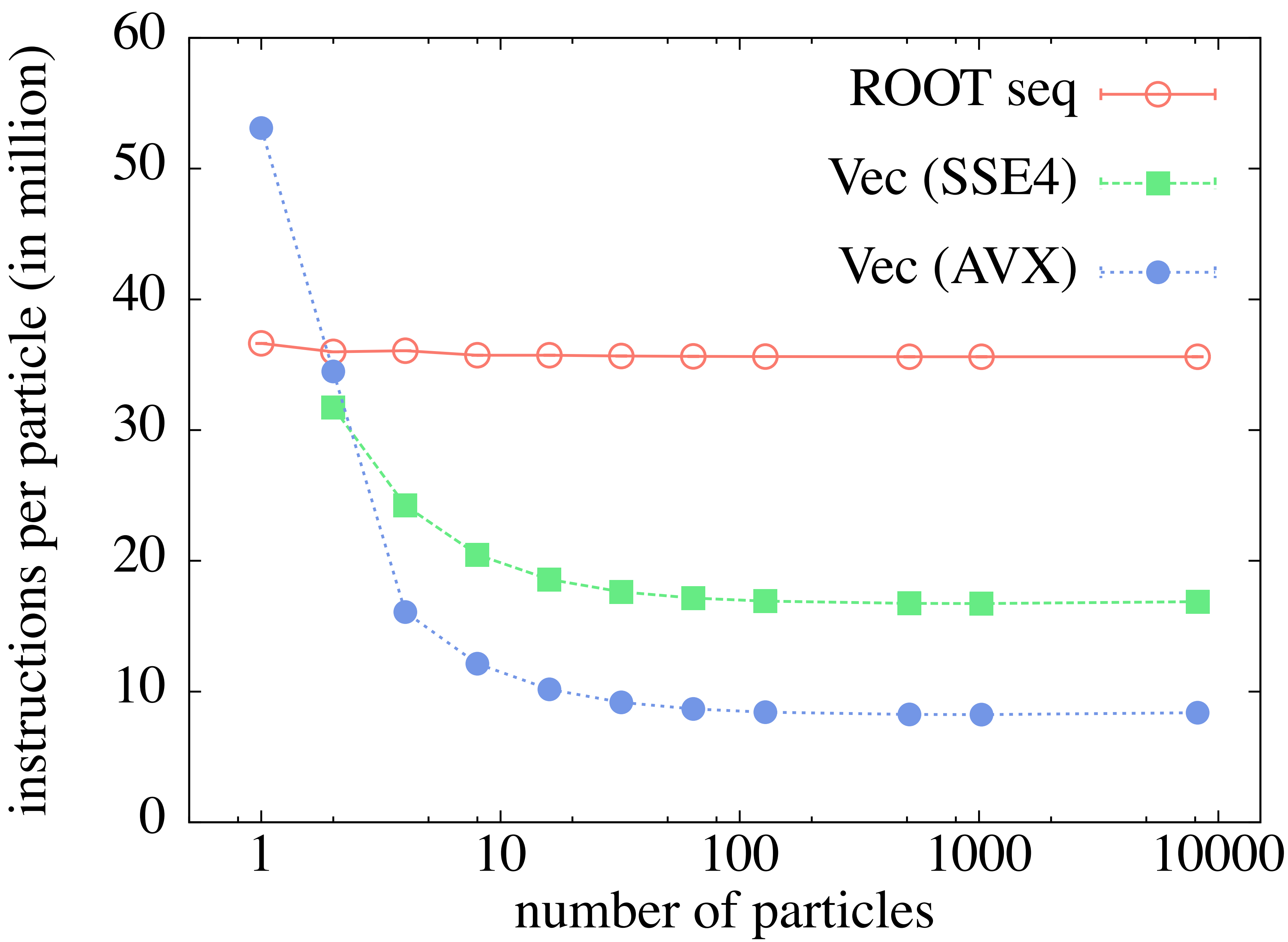
- time of processing/navigating N particles (P repetitions) using scalar algorithm (ROOT) versus vector version



- excellent speedup for SSE4 version
- some further gain with AVX
- already gain considerably for small N
- there is an optimal point of operation (performance degradation for large N)

Further Metrics: Executed Instructions

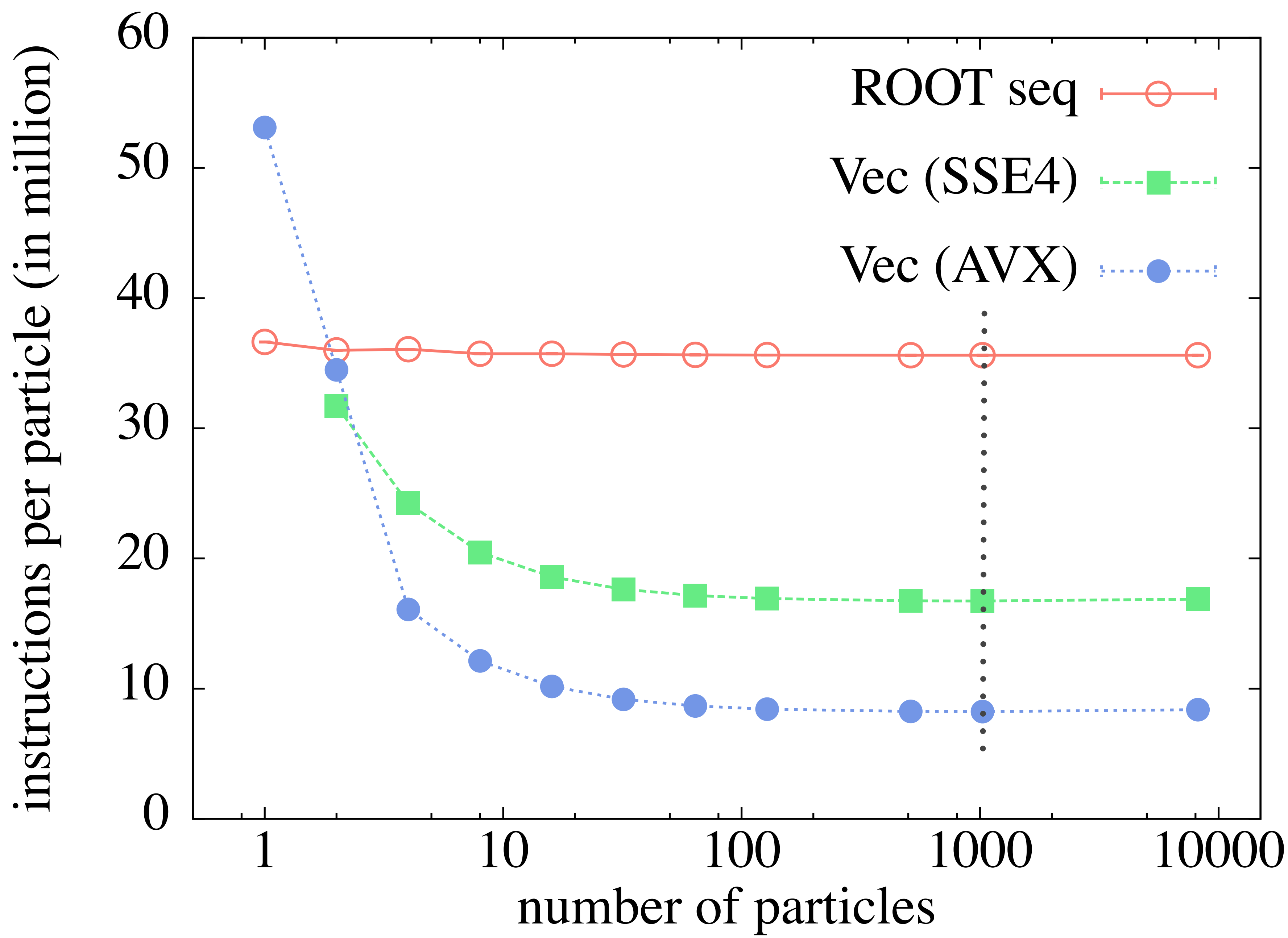
- investigate origin of speedup: study **hardware performance counters**
- developed a “timer” based approach where we read out counter before and after an arbitrary code section (using libpfm)



- gain mainly **due to less 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)



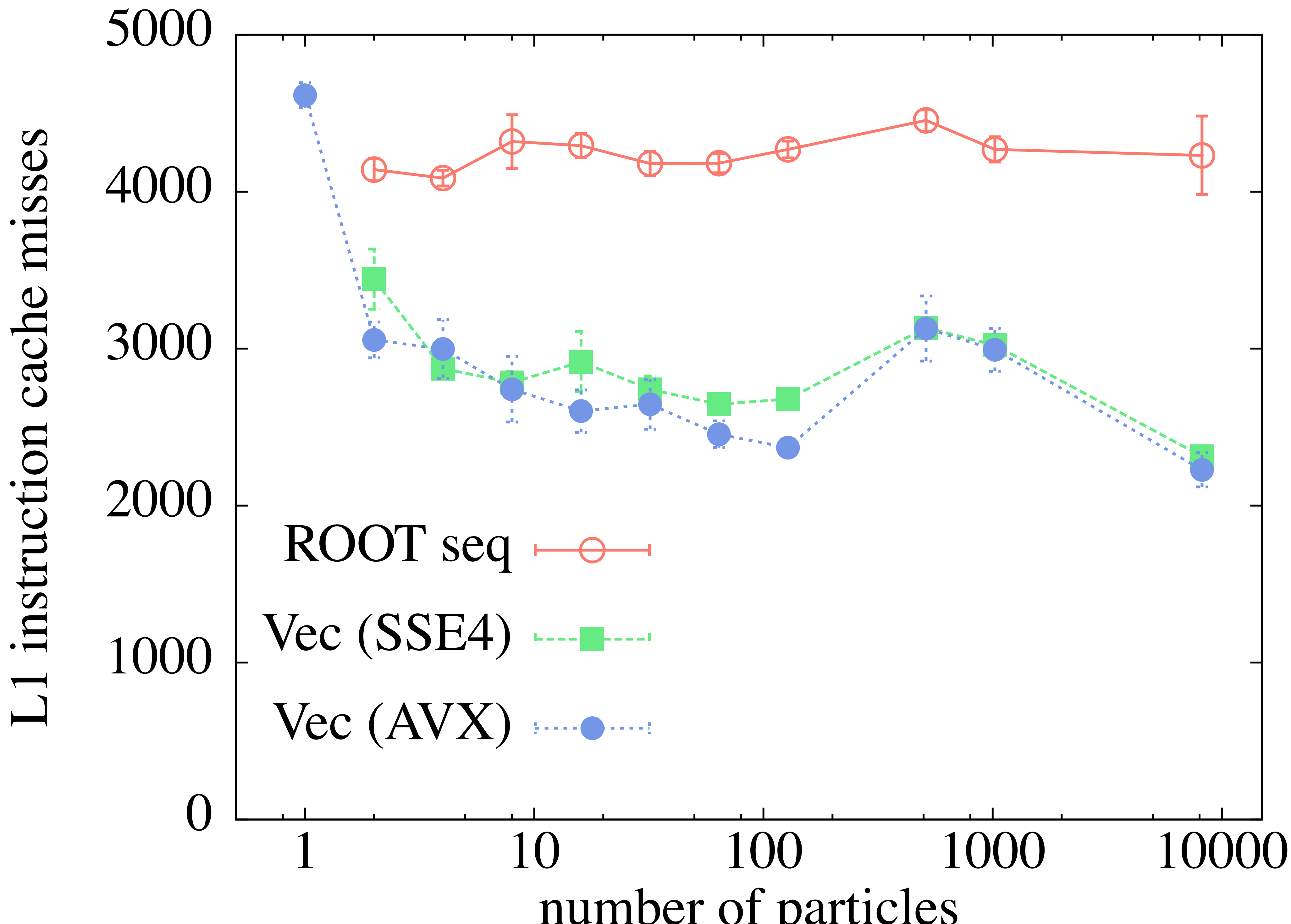
- gain mainly **due to less instructions** (for the same work)
- detailed analysis (binary instrumentation) can give statistics, e.g.:

	ROOT	**Vec**
MOV	30%	15%
CALL	4%	0.4%
V..PD (SIMD instr)	5%	**55%**

comparison for N=1024 particles (AVX versus ROOT seq)

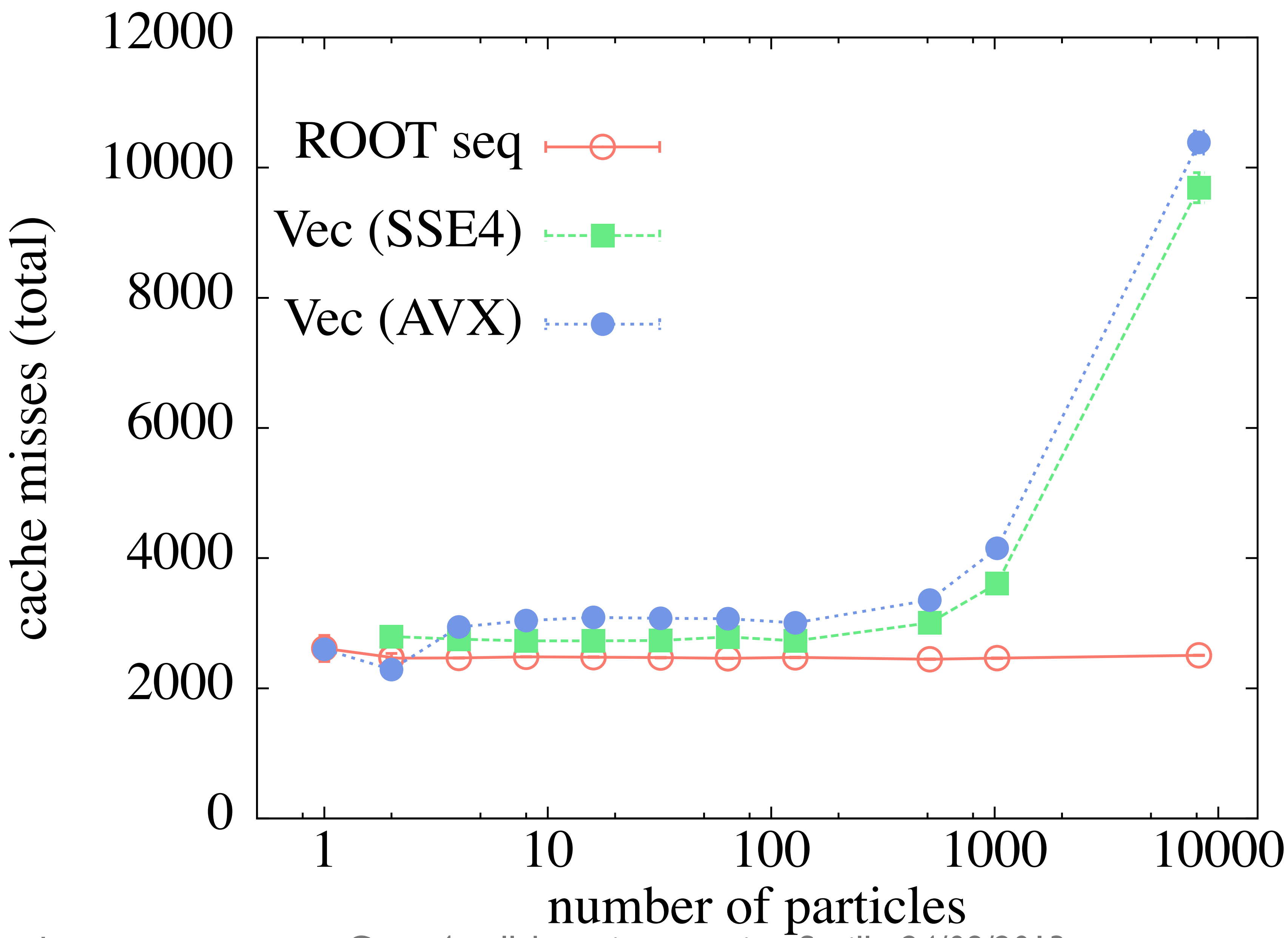
Further Metrics: L1 instruction cache misses

- The number of instruction cache misses is lower in vector treatment, as predicted. Effect will become more important when navigation itself embedded in more complex environment.



Further Metrics: total cache misses

- However, vector version suffers from more data cache misses for large number of particles, responsible for the observed performance degradation
- likely due to structure-of-array usage in vector case (versus array of structures in ROOT case)
- Once we know realistic N, might have to reconsider this option



Summary

- vectorization is not threading and needs to be cared for additionally!
- a vector/basket centric architecture allows to make use of SIMD instruction sets, needs less functions calls, and is 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 complex algorithms on commodity hardware
- Very good experience with explicit vector oriented programming model (Vc, Intel Cilk Plus Arrays)

Summary / Outlook

Summary

- vectorization is not threading and needs to be cared for additionally!
- a vector/basket centric architecture allows to make use of SIMD instruction sets, needs less functions calls, and is 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 complex 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 ...
- Xeon Phi, (GPU)
- **full flow of vectors in Geant-V prototype**

Acknowledgements

Thanks to:

- Geant-V team:
  - J. Apostolakis
  - F. Carminati,
  - A. Gheatta
- 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:
  - Lauren Duhem (Intel)
  - CERN Openlab

AND TO YOU FOR LISTENING!!

Backup slides

Notes on benchmark conditions

- System: Ivybridge iCore7 (4 core, not hyperthreaded (can read out 8hardware 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 testdata, 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 datapool (as shown here)
- benchmarks shown use NxP=const to time an overall similar amount of work

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

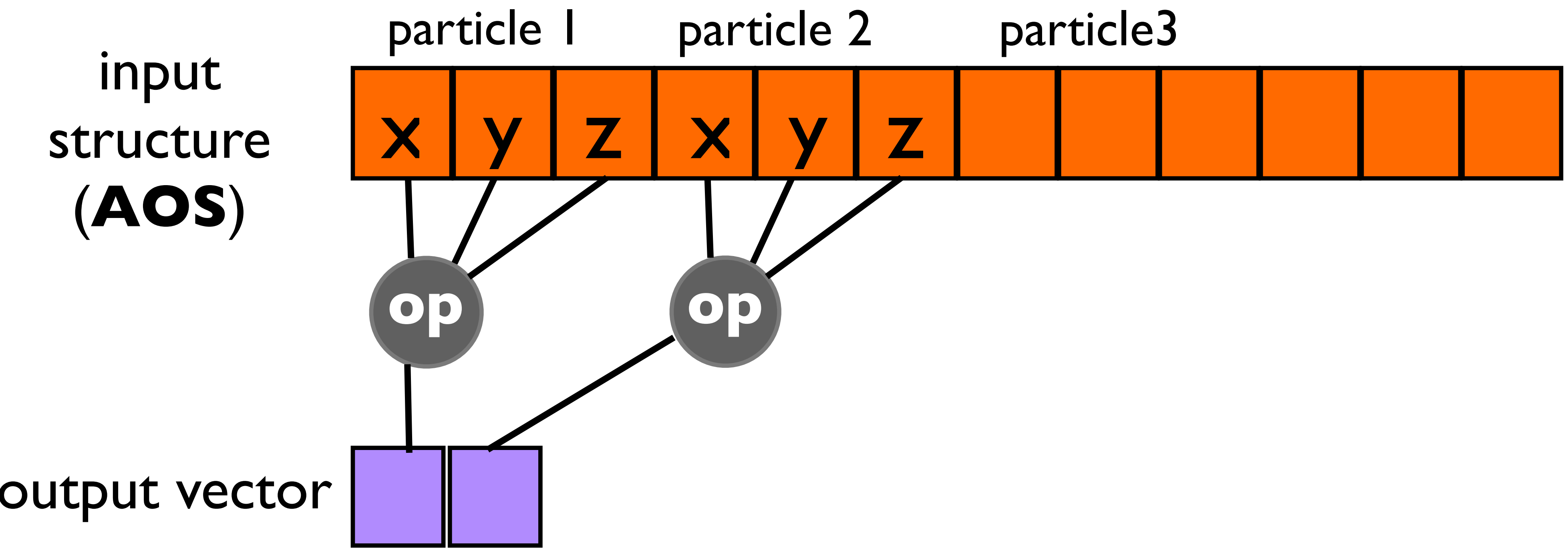
```
void foo(double const *a,
         double const *b,
         double * out, int np){
     for(int i=0;i<np;i+=Vc::double_v::Size)
     {
          // fetch chunk of data into Vc vector
          Vc::double_v a_v(&a[i]);
          Vc::double_v b_v(&b[i]);

          // 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

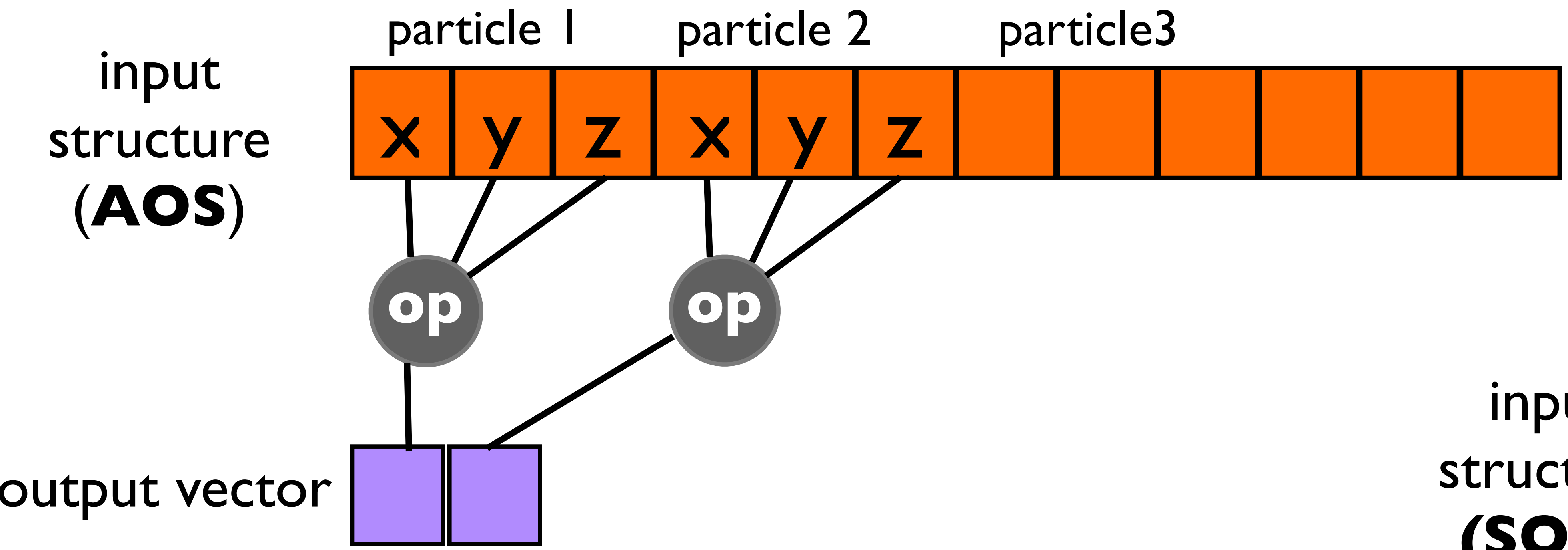
```
// CEAN example
void foo(double const * a,
         double const *b,
         double * out, int np)
{
    int const VecSize=4;
    for(int i=0;i<np;i+=VecSize)
    {
        // cast input as fixed size vector
        double const (*av)[VecSize] = (double const (*)[VecSize]) &a[i];
        double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

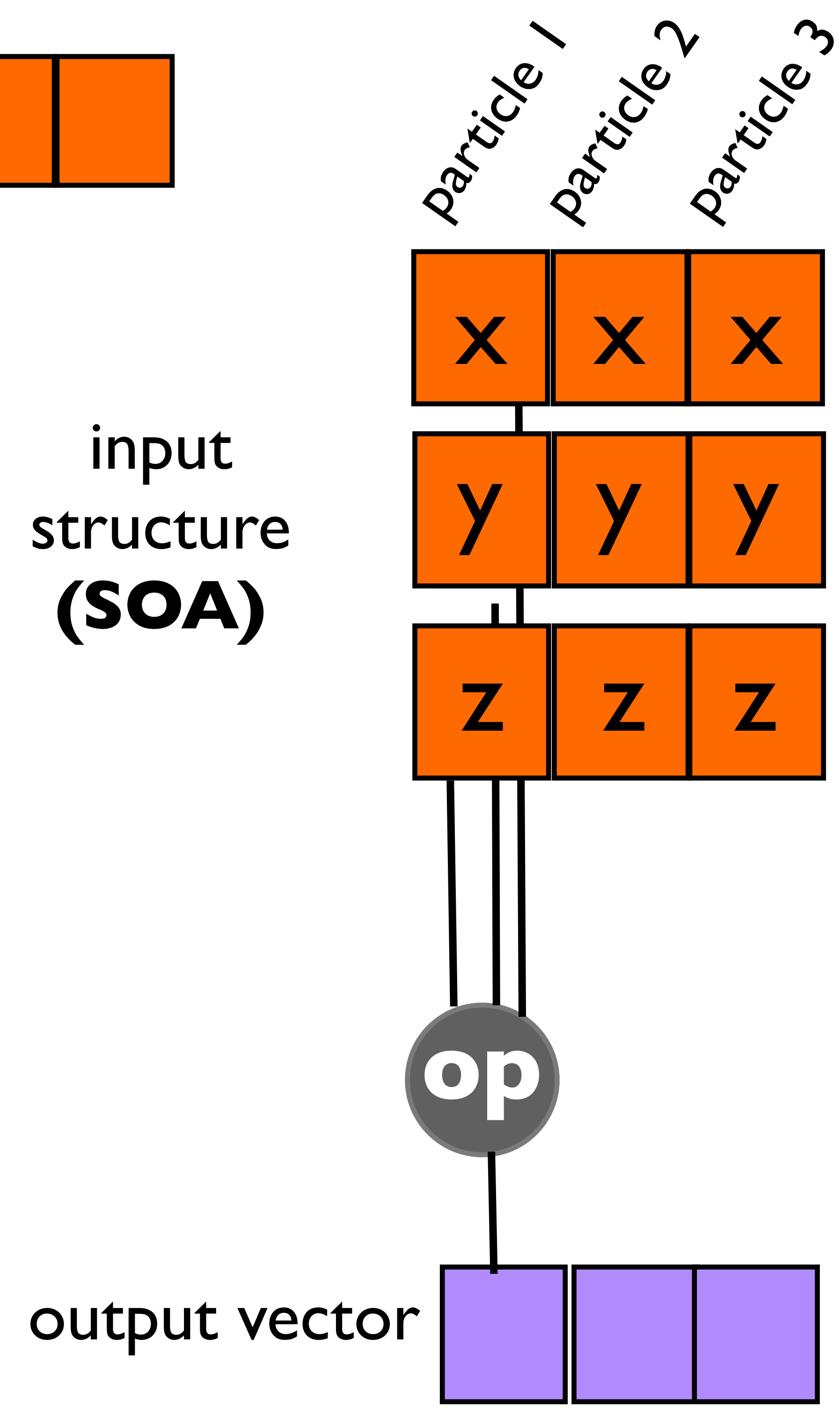
Memory Access Problem / Consideration



- a natural way to do vector processing of particles would be AOS approach
- above memory access pattern typical (3-to-1)

Memory Access Problem / Consideration



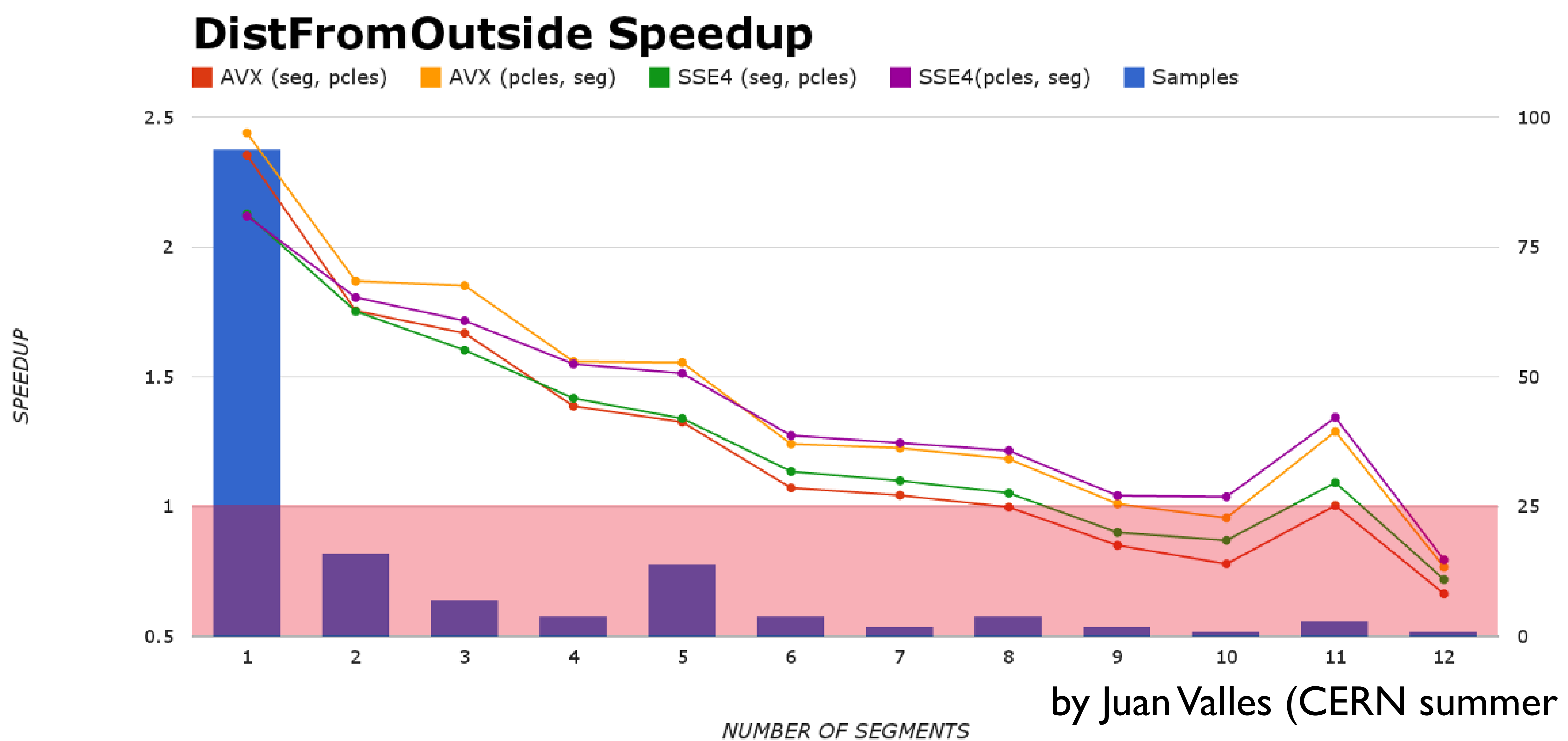


- a natural way to do vector processing of particles would be AOS approach
- above memory access pattern typical (3-to-1)

- SOA approach is better autovectorizable
- memory access in SOA pattern also more efficient

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

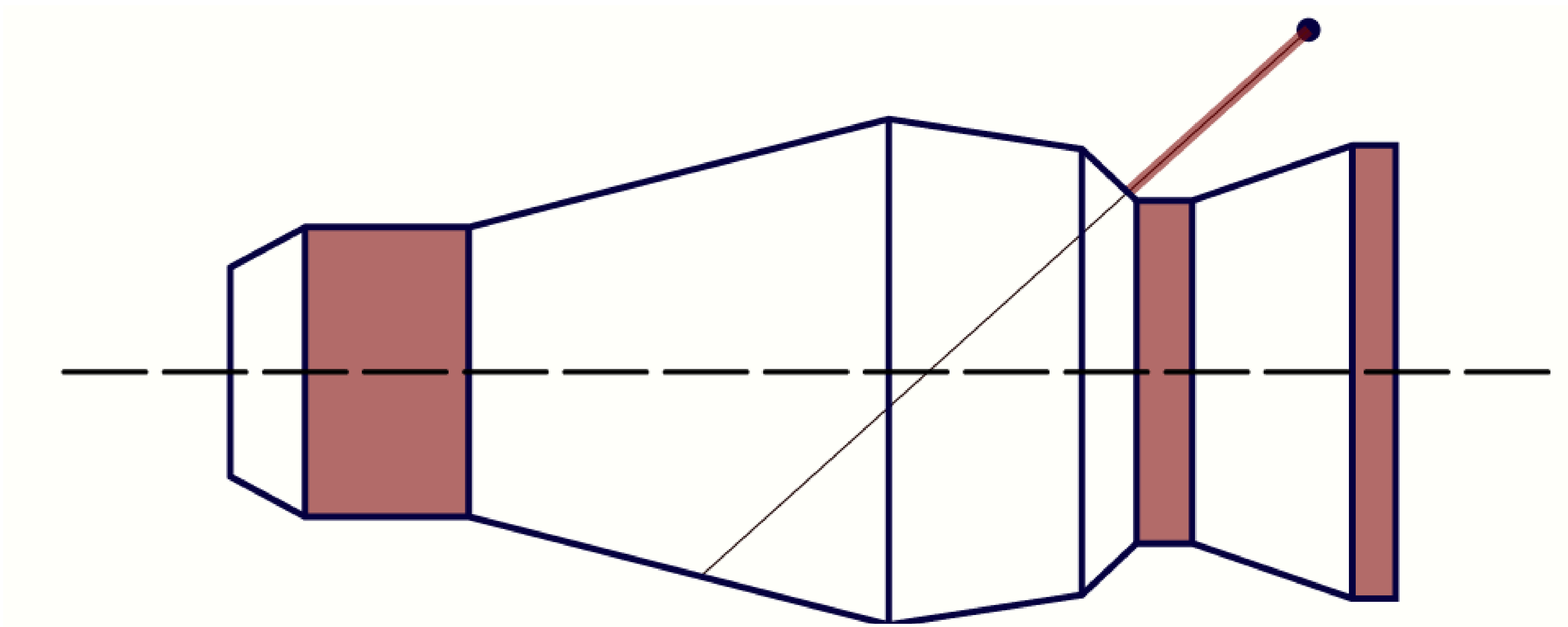


by Juan Valles (CERN summer student)

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"