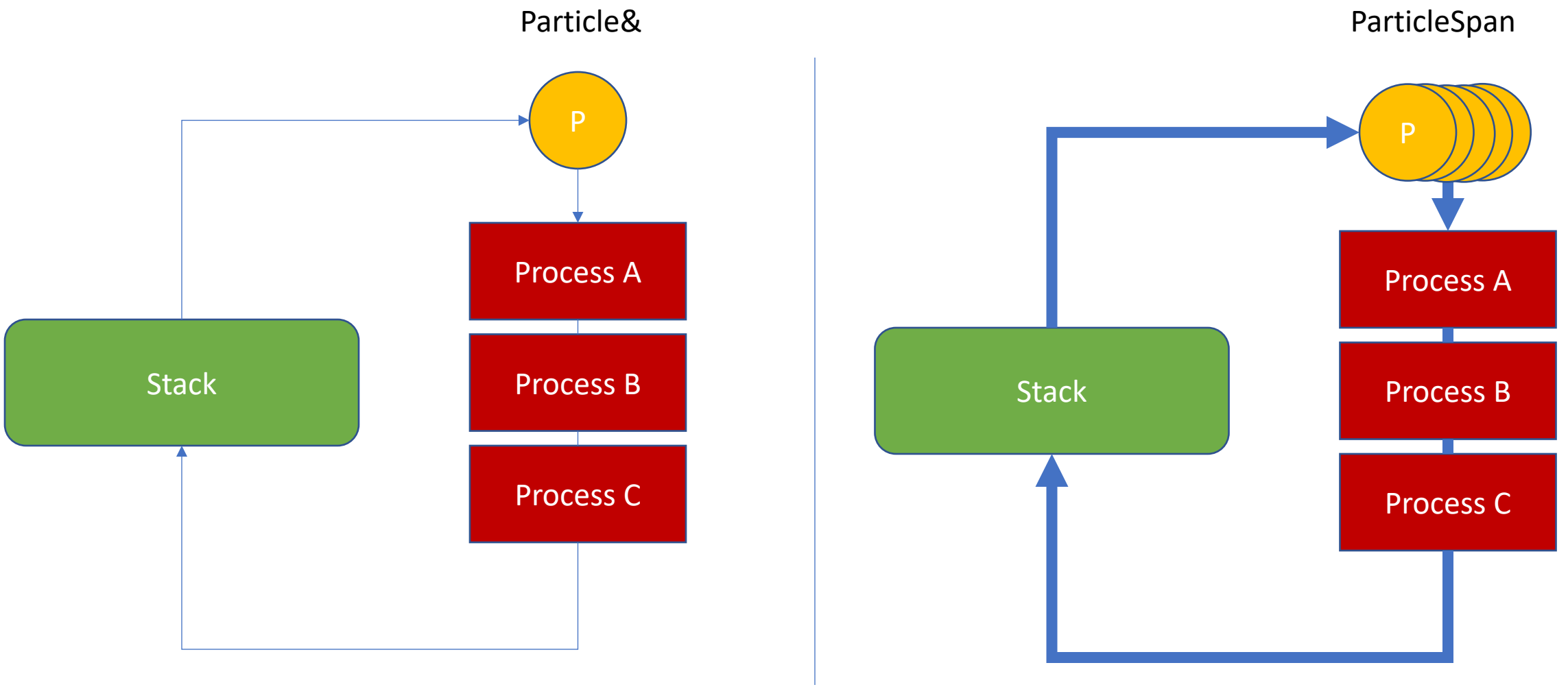
Process N particles in one go

Hans Dembinski, MPIK Heidelberg

Overview

- See https://gitlab.ikp.kit.edu/AirShowerPhysics/corsika/issues/224
- Instead of processing one particle in a Monte-Carlo step, process N particles at once in a Monte-Carlo step
  - Possible because Monte-Carlo steps for all particles independent
  - Apply ProcessA to particle range, apply ProcessB to particle range, ...
- Benefits
  - Faster processing
  - Allows processes to do local optimizations which are otherwise impossible
  - Allows us to create and modify ProcessList at run-time without performance hit

Current main loop vs proposed main loop



Benefits of processing N particles

- CPUs faster when doing same operation on multiple data
  - Reduced CPU cache misses
  - Can use SIMD instructions
  - Pipelining
- Allows encapsulated process optimization
  - Process can internally use optimizations which balance one-time cost against reduced cost per particle
    - Run multiple threads
    - Compute on the GPU
    - Allocate buffers for intermediate results
- Can use ProcessList constructed at run-time instead of static list
  - Configure std::vector<std::variant<ProcessA, ProcessB, ProcessC, ...>> at run-time
  - Runtime Lookup cost is negligible compared to time spend in process body

Syntactic cost is small

- Many calculations with Eigen::Arrays look as if they were numbers (like in Python with Numpy arrays)
- Can always fall back in process to iteration over one particle a time

Particle

- Size of Particle must be multiple of RealType for Eigen
- Particle should be trivial for best copy performance

```
#include <type_traits>

struct Particle {
  float& px() { return px_; }
  float& py() { return py_; }
  float& pz() { return pz_; }
  float& e() { return e_; }


  // "private" variables
  float px_, py_, pz_, e_;
};

static_assert(std::is_trivial<Particle>::value);
```

Trivial struct: no public/private sections, only POD members, no virtual methods

ParticleSpan

- Any Particle field has a fixed memory offset to the next field
- ArrayView can adapt a memory buffer of RealType with regular strides to look like a Eigen::Array
- ArrayView is light-weight, it does not own the memory (copy ok)

```
#include <Eigen/Core>
#include <vector>

class ParticleSpan {
public:
    using iterator = Particle*;
    using ArrayView = Eigen::Map<
        Eigen::Array<float, Eigen::Dynamic, 1>,
        Eigen::Unaligned,
        Eigen::InnerStride<(sizeof(Particle) / sizeof(float))>
    >;

    ParticleSpan(std::vector<Particle>& v)
         : begin_(v.data()), end_(v.data() + v.size()) {};

    iterator begin() { return begin_; }
    iterator end() { return end_; }

    std::size_t size() { return end_ - begin_; }

    ArrayView px() { return ArrayView(&begin_->px_, size()); }
    ArrayView py() { return ArrayView(&begin_->py_, size()); }
    ArrayView pz() { return ArrayView(&begin_->pz_, size()); }
    ArrayView e() { return ArrayView(&begin_->e_, size()); }

private:
    iterator begin_, end_;
};
```

Energy loss

- Templated functions accept Particle and ParticleSpan!
- Important: return decltype(auto) instead of auto
- Use ADL-friendly call to log for Eigen

```
template <class T>
decltype(auto) sqr(const T& x) { return x * x; }


template <class T>
decltype(auto) momentum_squared(T& part) {
  return sqr(part.px()) + sqr(part.py()) + sqr(part.pz());
}

template <class T>
void do_energy_loss(T& part) {
  auto beta_2 = momentum_squared(part) / sqr(part.e());
  // compute energy loss, ignoring all constants
  using std::log; // allow Eigen to find its own log via ADL
  auto energy_loss = log(beta_2 / (1.0 - beta_2)) / beta_2 - 1.0;
  part.e() -= energy_loss;
}
```

Processing

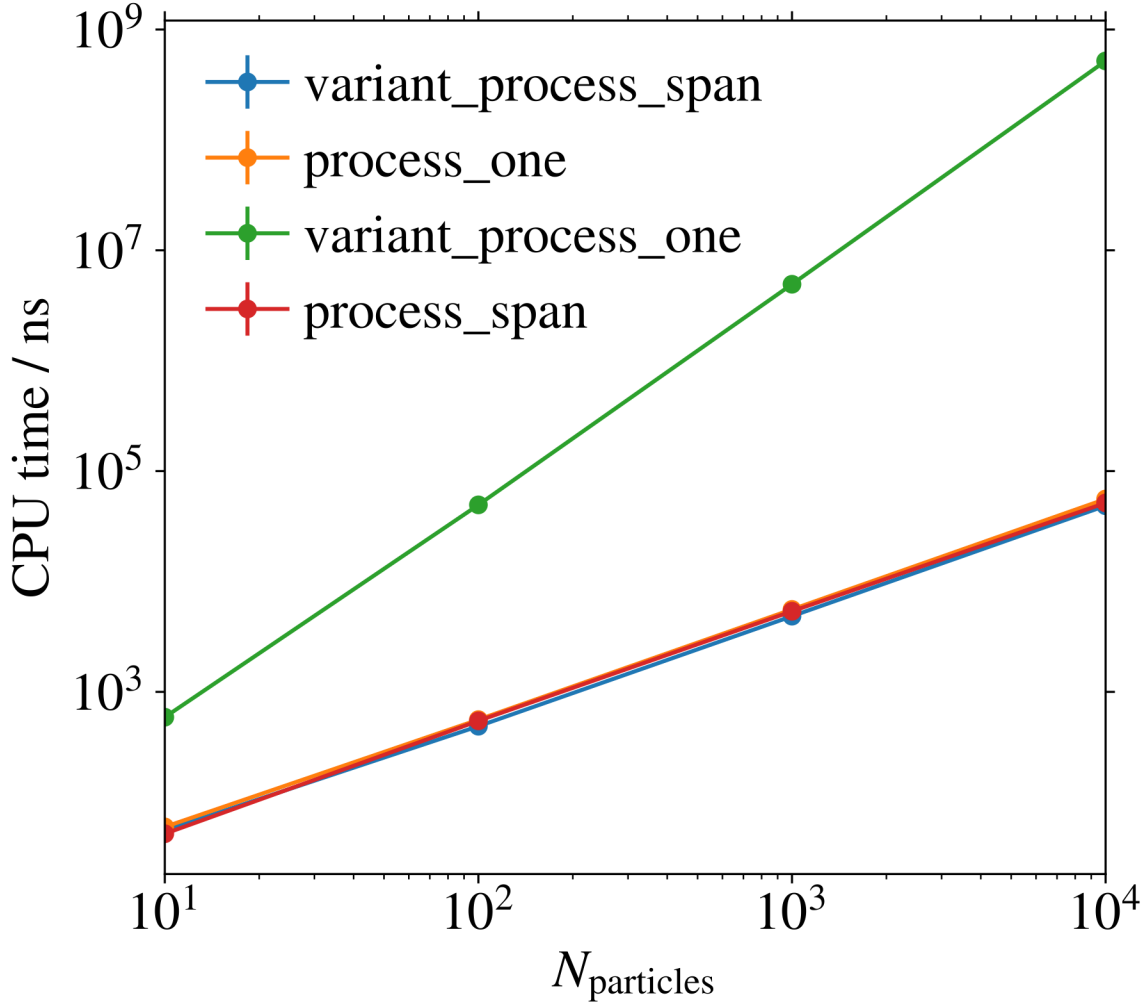
```
struct ContinuousEnergyLoss {
  template <class T>
  void operator()(T& p) const {
    do_energy_loss(p);
  }
};
```

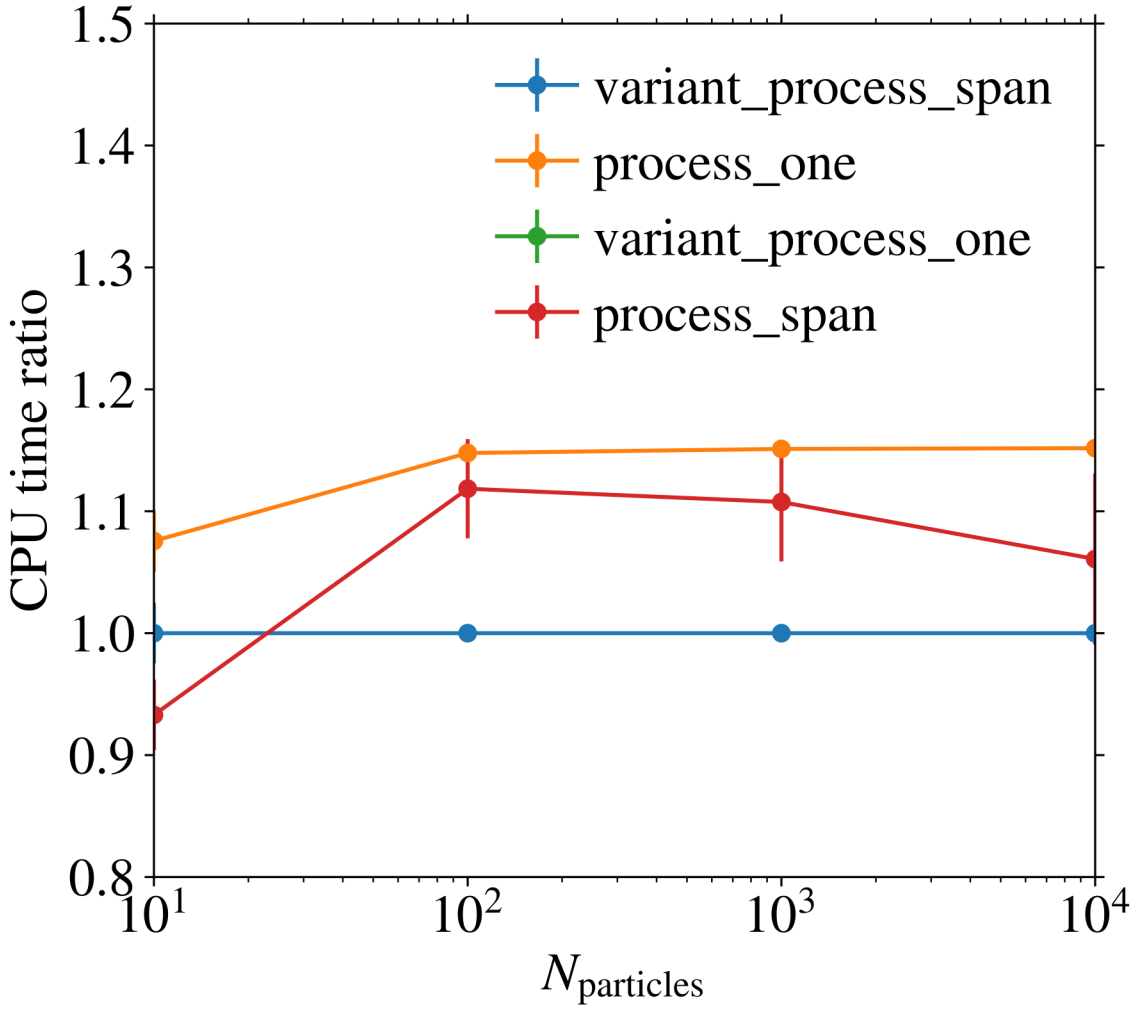
```
struct DummyProcess {
  template <class T>
  void operator()(T& p) const {}
};
```

```
#include <vector>
#include <variant>

std::vector<Particle> stack(1000);
ParticleSpan span(stack);

// Method 1: process one particle at once
for (auto&& p : span)
  do_energy_loss(p);

// Method 2: process block of particles at once
do_energy_loss(span);

std::variant<ContinuousEnergyLoss, DummyProcess> process;
process = ContinuousEnergyLoss(); // set at run-time

// Method 1A: process one particle using std::variant
for (auto&& p : span)
  visit([&span](auto& proc) { proc(span); }, process);

// Method 2A: process block of particles using std::variant
for (auto _ : state)
  visit([&span](auto& proc) { proc(span); }, process);
```

Benchmarks

- Highest performance for variant of processes which acts on a ParticleSpan
- Higher performance gains seem possible with better use of SIMD
- Variant of processes which acts on Particle a time is unacceptably slow

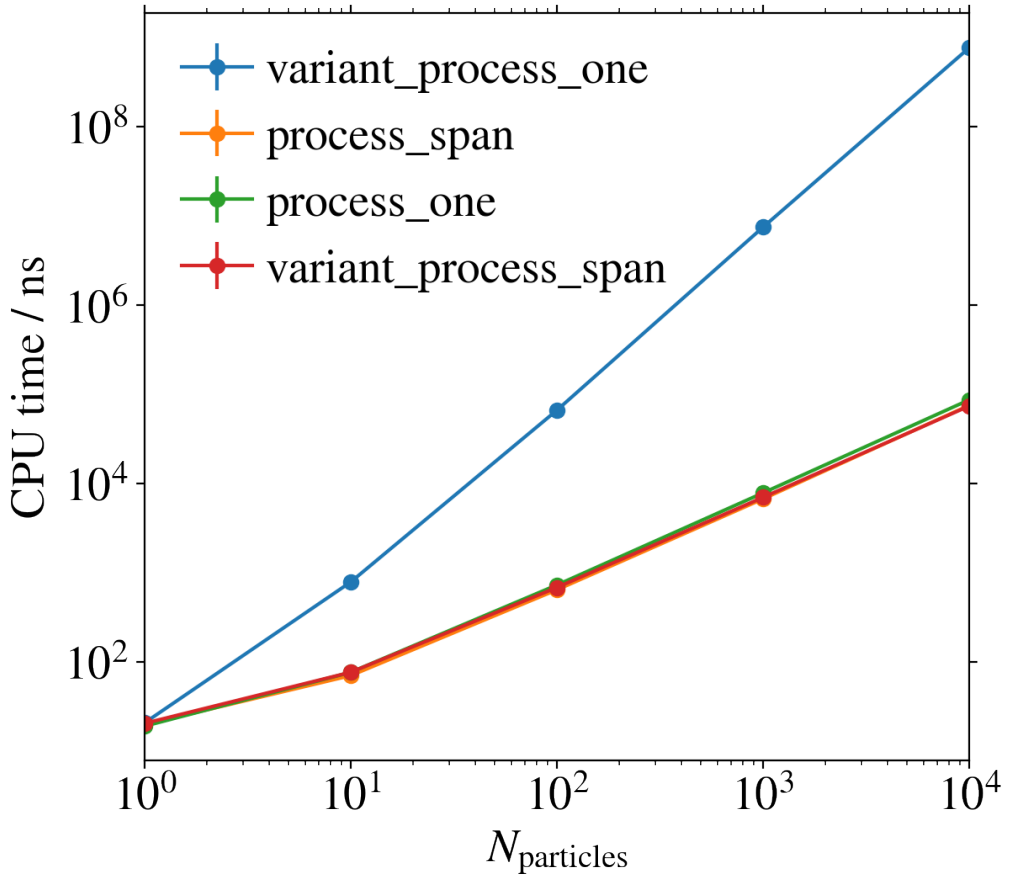


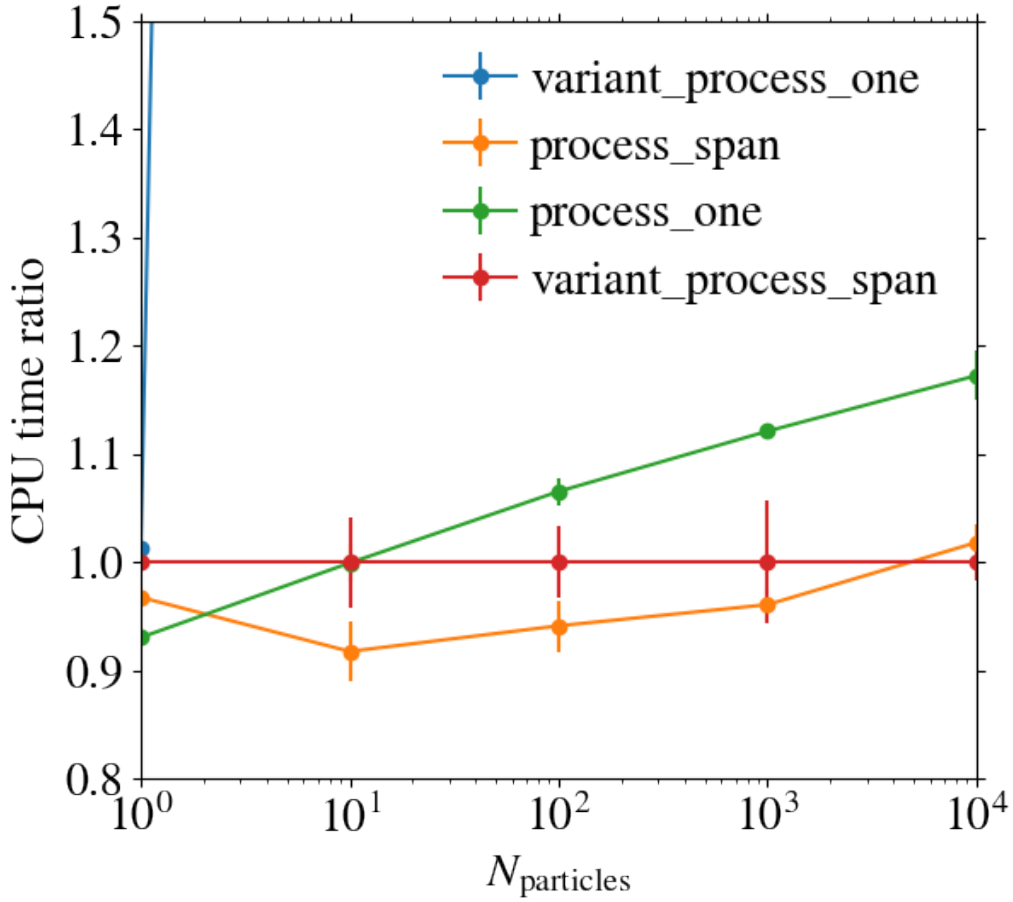


Summary

- We should change to the new main loop proposal
- No show stoppers
- ParticleSpan and Eigen make array computing easy
- Enable individual modular optimisations for each process
- Speed gains not as much as I had expected, but present
- Much simpler ProcessList, configurable at run-time from Python

Benchmarks 2 under load, also with 1 particle





Github repo with this demonstrator

https://github.com/HDembinski/corsika_span_demo