The first general-coordinate (QED)(GR)PIC code for astrophysical plasmas

Hayk Hakobyan (Columbia/PPPL)

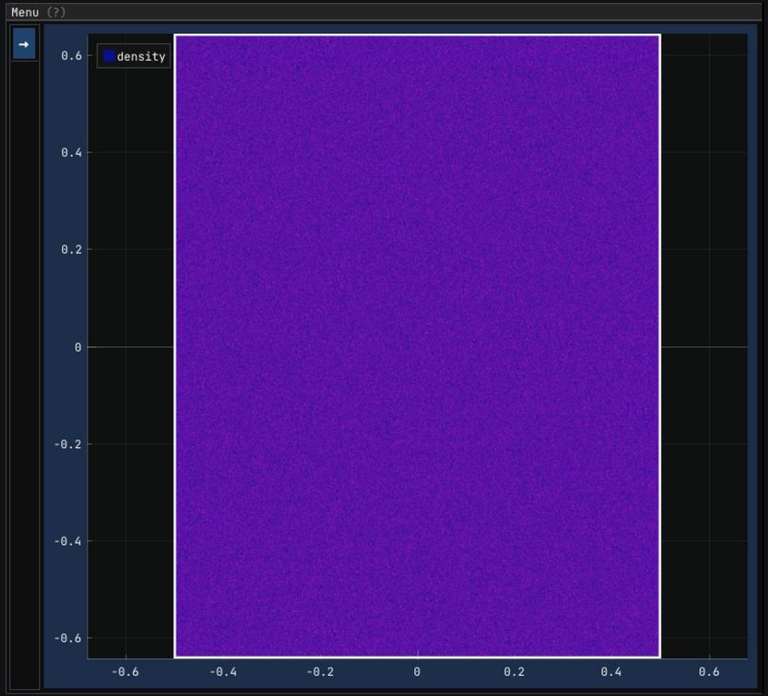
in collaboration with:

A. Chernoglazov (UMD), B. Crinquand (Princeton → U of Toulouse), A. Galishnikova (Princeton), J. Mahlmann (Columbia), A. Philippov (UMD), A. Vanthieghem (Princeton → CNRS), M. Zhou (IAS)

Capabilities of the *Entity*

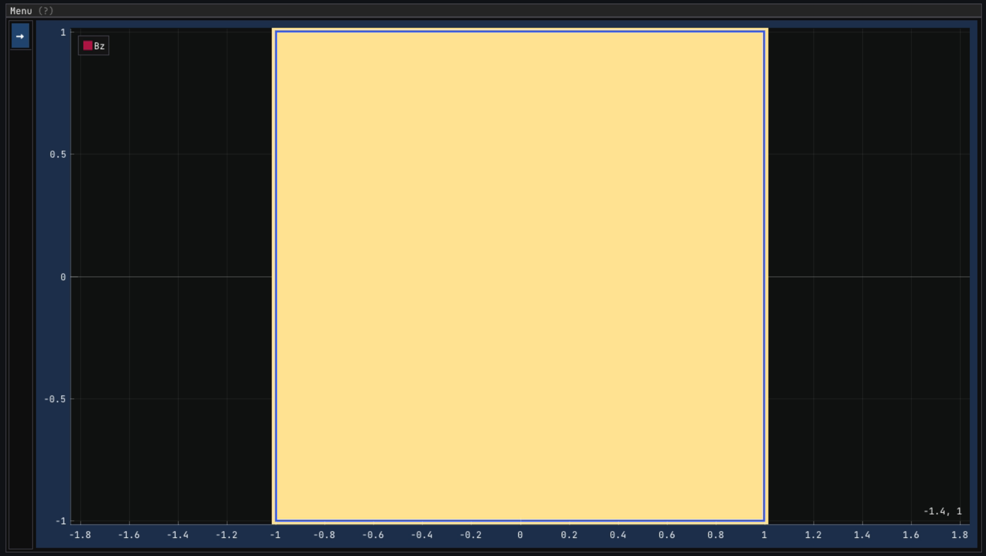
@ haykh.github.io/entity

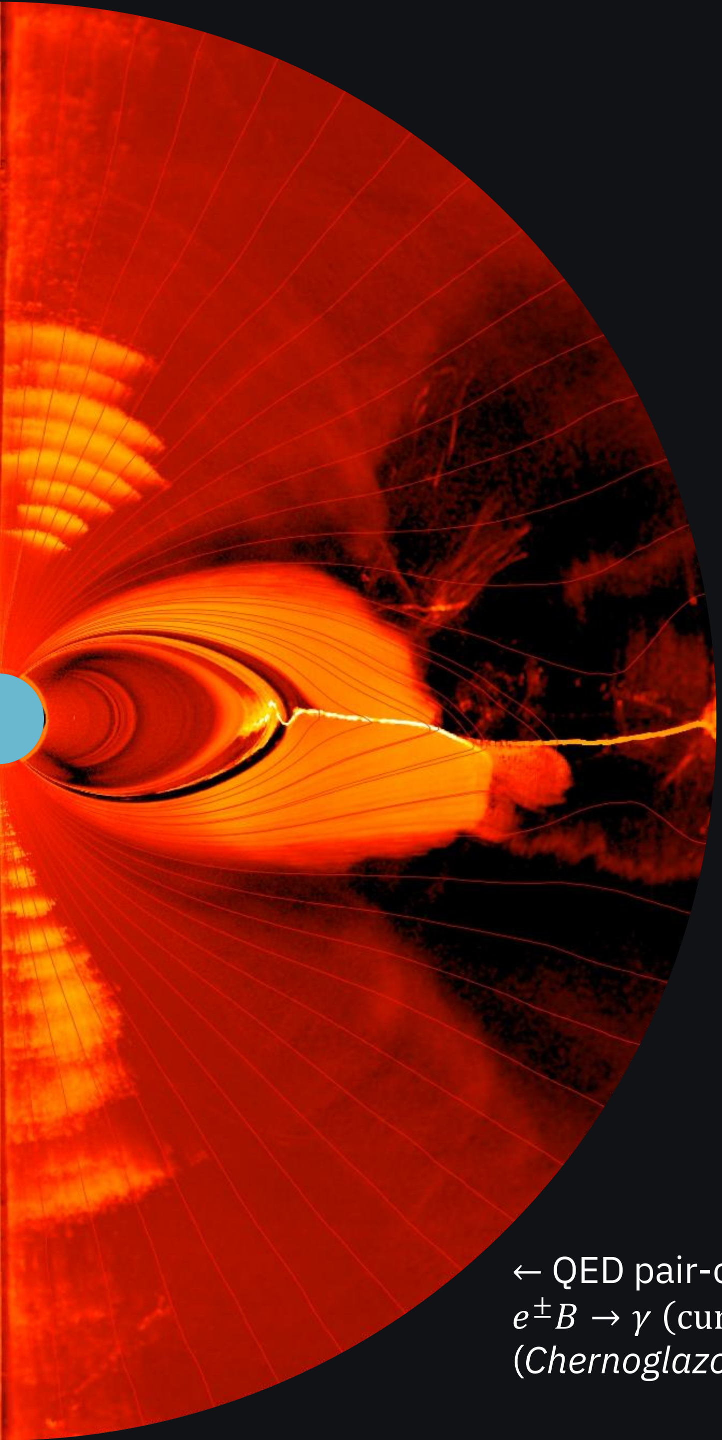
- 1D/2D/3D
- Arbitrary metric/geometry
- General relativity (GR)
- Multi-species
- Radiative processes
- Single-body & two-body QED processes
- Highly modular and flexible



Tearing instability & magnetic reconnection ↑

Weibel instability →



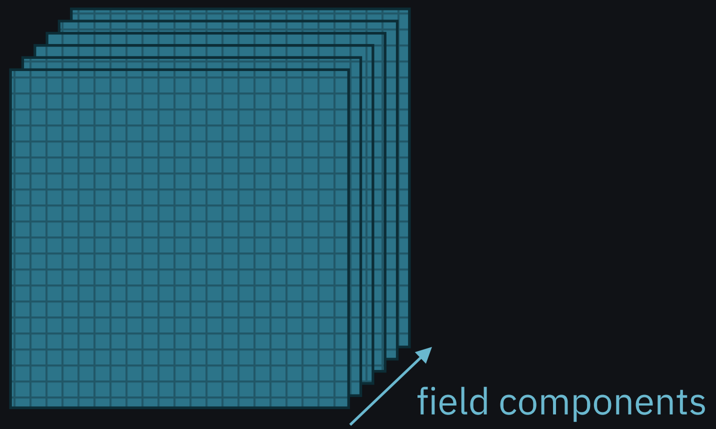


← QED pair-cascade in a neutron star magnetosphere:
$e^{\pm}B \to \gamma\ (\mathrm{curv})$ then $\gamma B \to e^{\pm}$
(*Chernoglazov+, in prep.*)

Architecture of the *Entity*

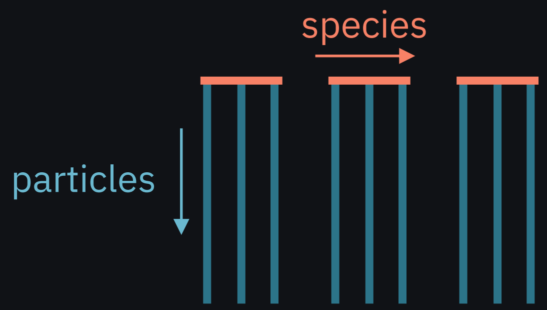
Memory Layout

```
struct Fields {
  Kokkos::View<real_t*** [6]> em;
  Kokkos::View<real_t*** [6]> aux;
  Kokkos::View<real_t*** [3]> cur;
  …
};
```




```
template <Dimension D>
struct Metric {
  KOKKOS_INLINE_FUNCTION
  void v3_Cntrv2Cov(const coord_t<D>& xi,
                    const vec_t<D>& v_Cntrv,
                    vec_t<D>& v_Cov);

  …
};
```

```
struct Particles {
  float mass;
  float charge;

  Kokkos::View<int*>    i1;
  …
  Kokkos::View<float*>  dx1;
  …
  Kokkos::View<real_t*> ux1;
  …
  Kokkos::View<short*>  tag;
};

auto species = std::vector<Particles> {};
```



Architecture of the *Entity*

Kernels

(a) Field solvers

em(:,:,:,:)
aux(:,:,:,:)
cur(:,:,:,:)
metric
→

$$\frac{\partial D^i}{c\partial t} = \frac{1}{\sqrt{h}}\varepsilon^{ijk}\partial_j H_k - \frac{4\pi}{c}\frac{J^i}{\sqrt{h}}$$
$$\frac{\partial B^i}{c\partial t} = -\frac{1}{\sqrt{h}}\varepsilon^{ijk}\partial_j E_k$$

$$H_i = \alpha h_{ij}B^j - \frac{1}{\sqrt{h}}\varepsilon_{ijk}\beta^j D^k$$
$$E_i = \alpha h_{ij}D^j + \frac{1}{\sqrt{h}}\varepsilon_{ijk}\beta^j B^k$$

→ em(:,:,:,:)

```
struct FieldSolver_kernel {
  KOKKOS_INLINE_FUNCTION
  // vectorized over cells
  void operator()(int i1, int i2, int i3) const {}
};
```

(b) Particle pushers

i1(:), …
dx1(:), …
ux1(:), …
em(:,:,:,:)
metric
→

$$\frac{du_i}{cdt} = -\gamma\partial_i\alpha + u_j\partial_i\beta^j - \alpha\frac{1}{2\gamma}u_j u_k\partial_i h^{jk} + \\ + \frac{q}{m\,c^2}\alpha\left(h_{ij}D^j + \frac{1}{\sqrt{h}\gamma}\varepsilon_{ijk}h^{jl}u_l B^k\right)$$
$$\frac{dx^i}{cdt} = \frac{\alpha}{\gamma}h^{ij}u_j - \beta^i$$

→
i1(:), …
dx1(:), …
ux1(:), …

```
struct ParticlePusher_kernel {
  KOKKOS_INLINE_FUNCTION
  // vectorized over particles
  void operator()(std::size_t p) const {}
};
```

Kernels

3D indexing

```
struct FieldSolver_kernel {
  KOKKOS_INLINE_FUNCTION
  // vectorized over cells
  void operator()(int i1, int i2, int i3) const {
    em(i1, i2, i3, ex1) = …;
  }
};
```

1D indexing

```
struct FieldSolver_kernel {
  KOKKOS_INLINE_FUNCTION
  // vectorized over cells
  // from 0 to nx1 * nx2 * nx3
  void operator()(int cell) const {
    auto [i1, i2, i3] = UNRAVEL(cell, nx1, nx2, nx3);
    em(i1, i2, i3, ex1) = …;
  }
};
```

[*] 1D indexing is ~ 20 … 30% faster (on A100 GPU)

Architecture of the *Entity*

Kernels

(c) Current deposition

```
i1(:), …
dx1(:), …
i1_prev(:), …
dx1_prev(:), …
cur(:,:,:,:)
```

→ $\mathcal{J}^i = \sum_p \frac{q}{\Delta t} \left(x^i - x^i_{\text{prev}}\right) S(x^i)$ → `cur(:,:,:,:)`

```
auto scatter_cur = Kokkos::Experimental::create_scatter_view(cur);
for (auto& sp : species) {
  Kokkos::parallel_for("CurrentsDeposit",
                       sp.npart(),
                       CurrentDeposition_kernel(sp, scatter_cur));
}
Kokkos::Experimental::contribute(cur, scatter_cur);

struct CurrentDeposition_kernel {
  // vectorized over particles
  KOKKOS_INLINE_FUNCTION
  void operator()(std::size_t p) {}
};
```

Performance

For PIC, the performance is usually evaluated by the **timestep duration per particle**

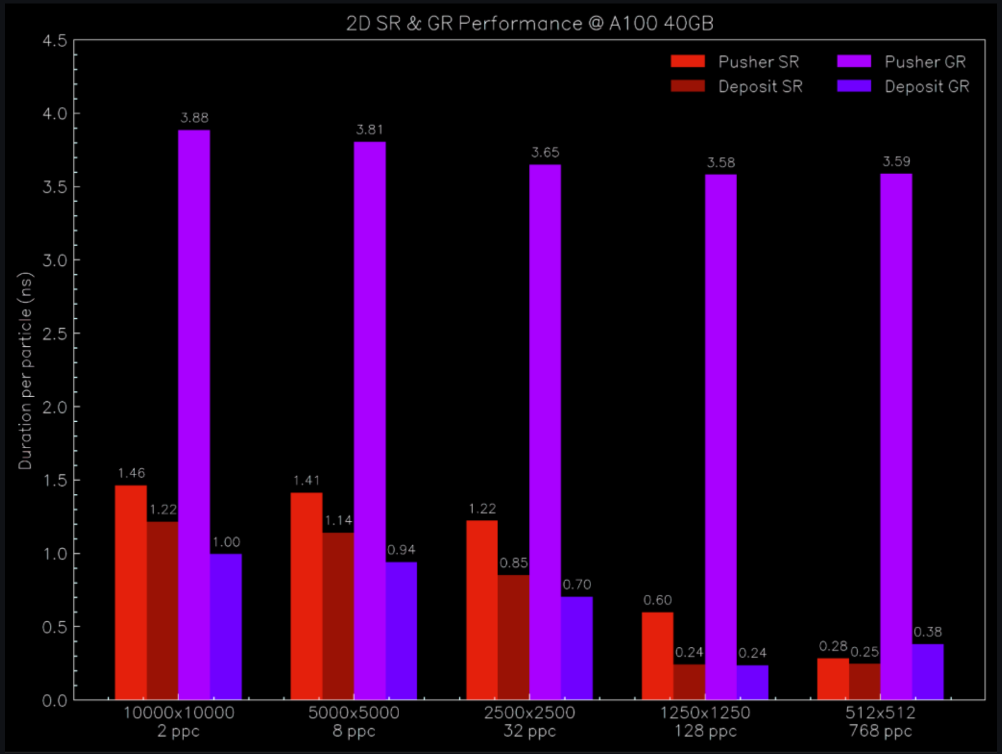
Single core/device performance:

Typical CPU perf.: $\sim 20 \ldots 100$ ns/particle (*Tristan v2* @ Intel Cascade Lake)

- current deposition: $\sim 60\%$
- particle pusher: $\sim 30\%$
- field solver/filtering: $\sim 10\%$

GPU perf.: $\sim 0.5 \ldots 1.5$ ns/particle (*Entity* @ NVIDIA A100)

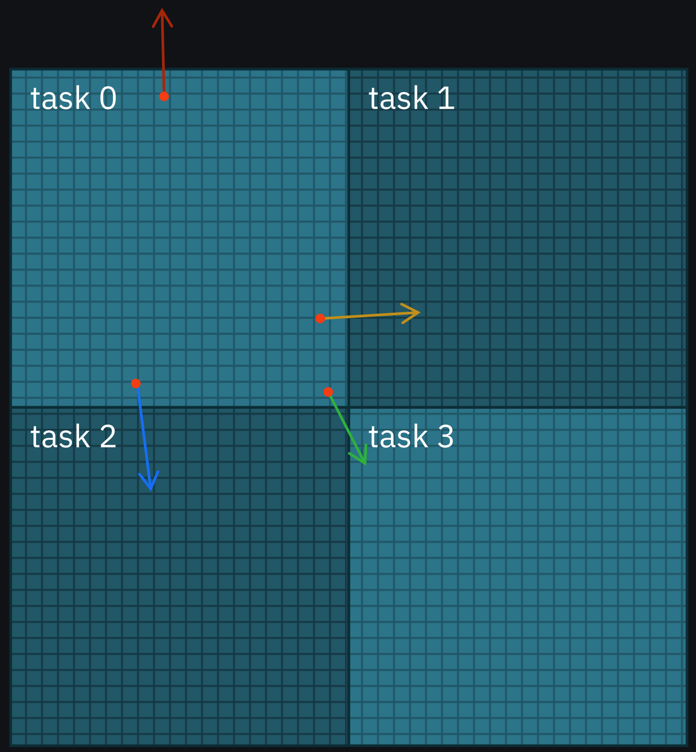
- current deposition: $\sim 35\%$
- particle pusher: $\sim 60\%$
- field solver/filtering: $\sim 5\%$

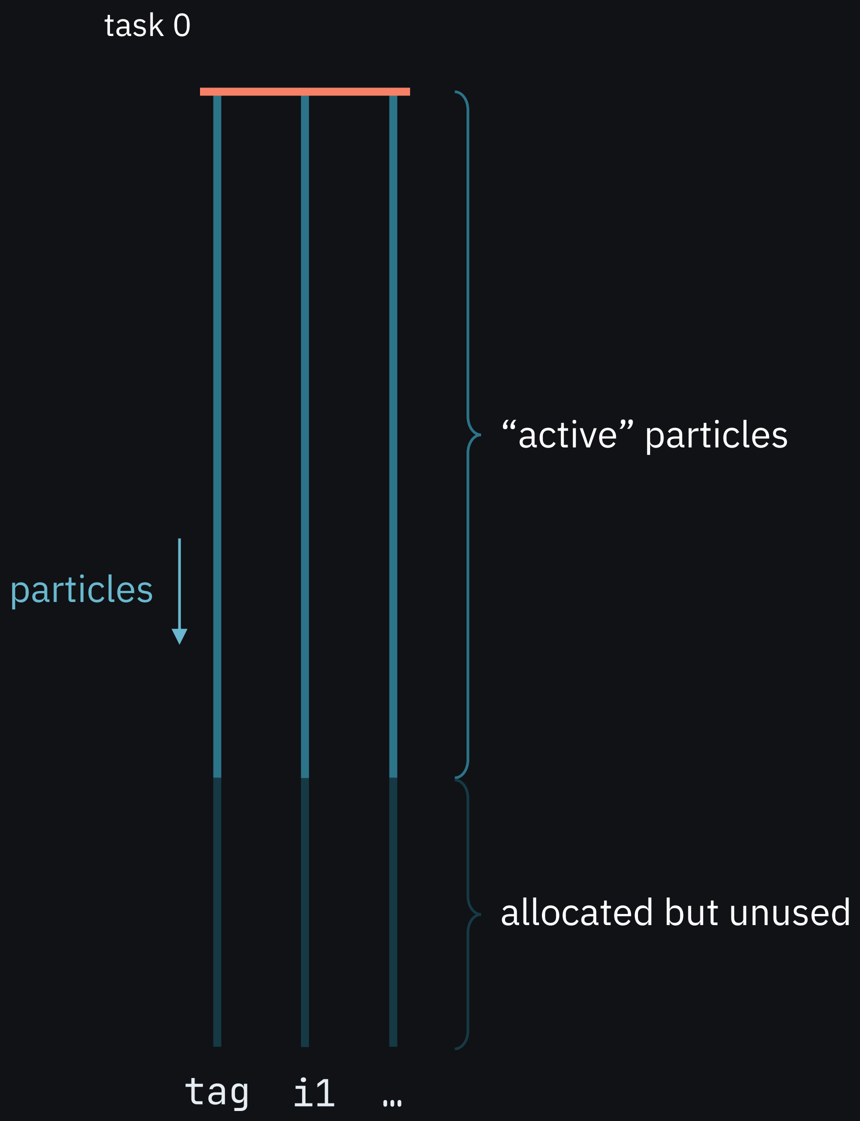


$\sim 2 \cdot 10^8$ particles

Bottlenecks & Culprits

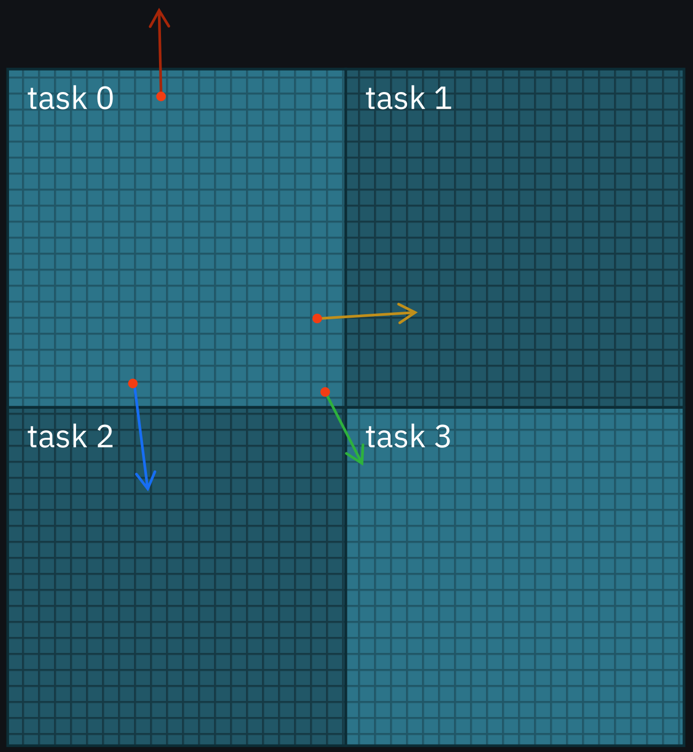
MPI communications of particles

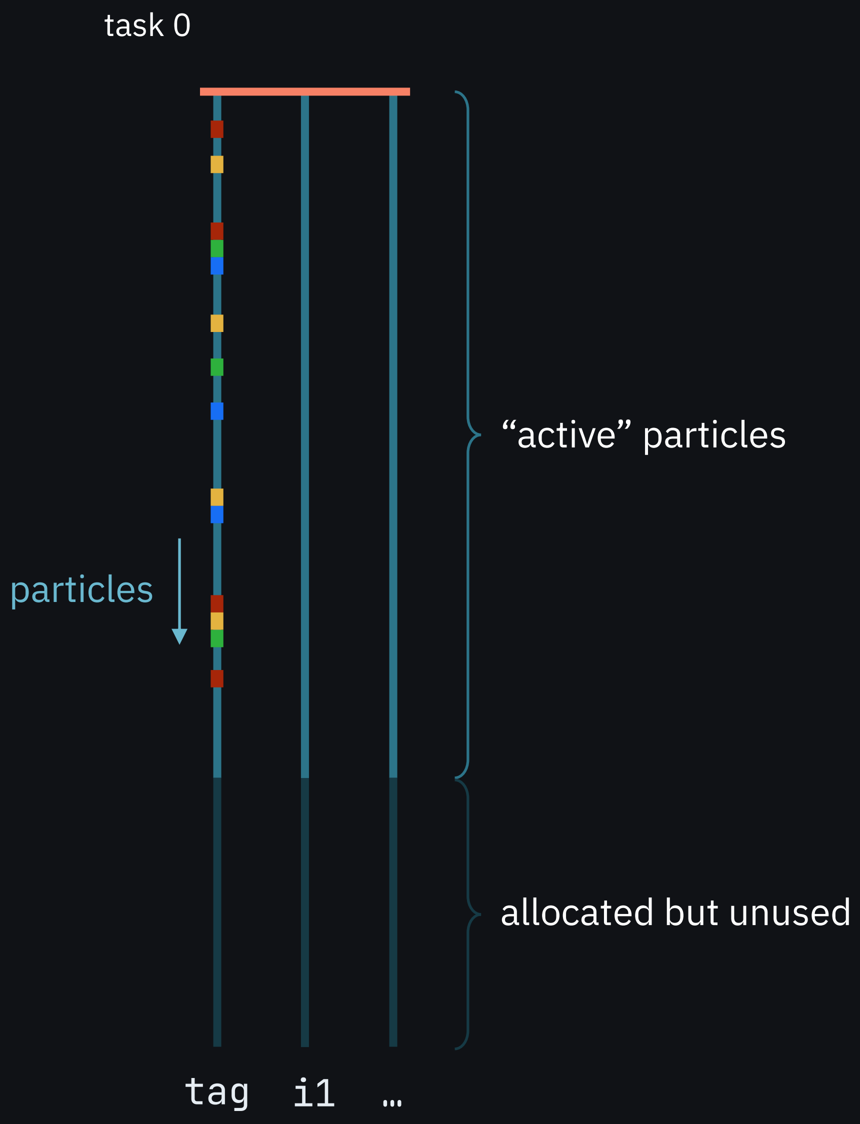




Bottlenecks & Culprits

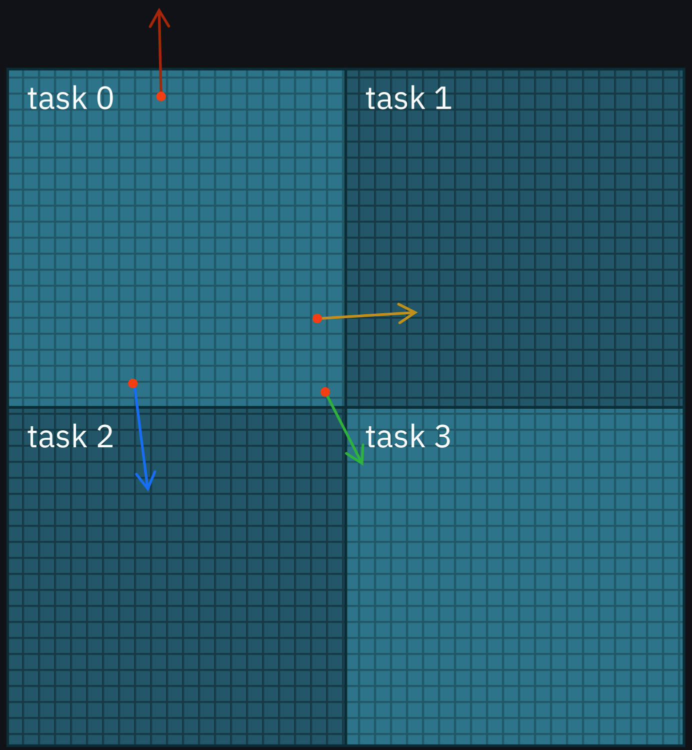
MPI communications of particles

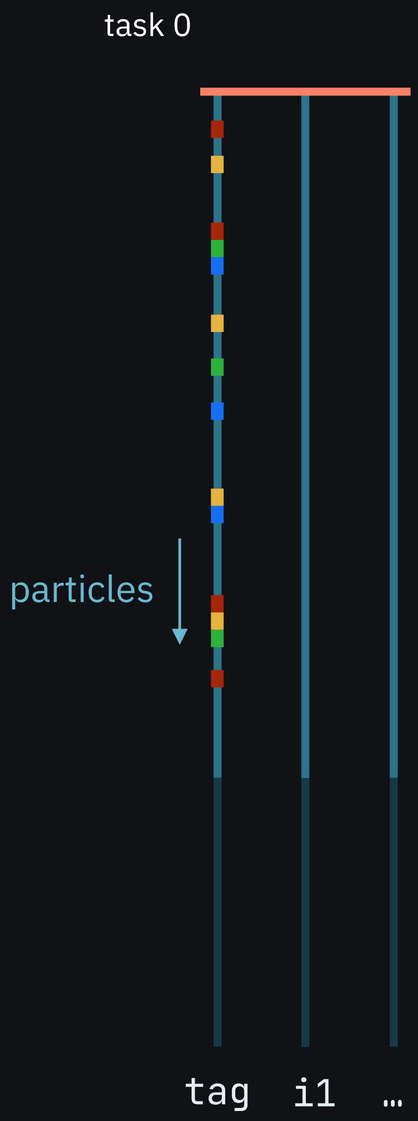




Bottlenecks & Culprits

MPI communications of particles



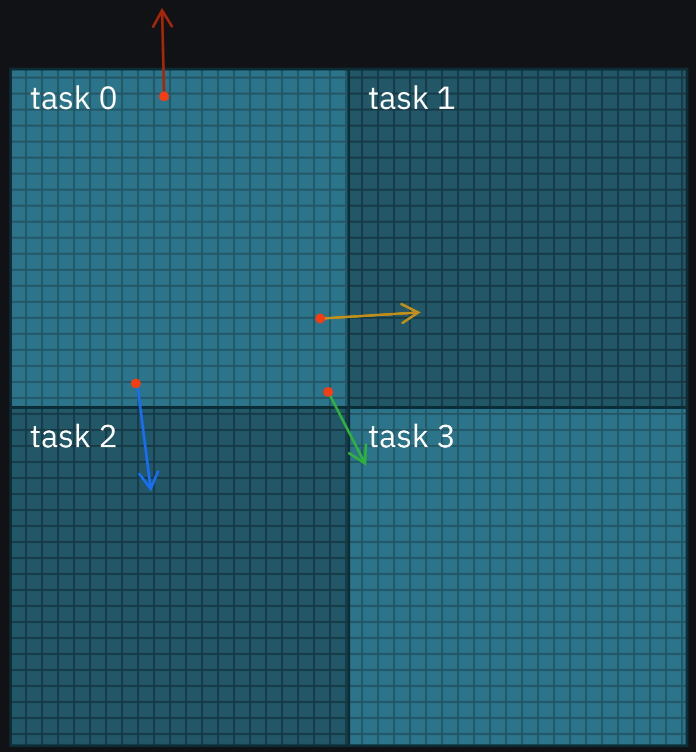


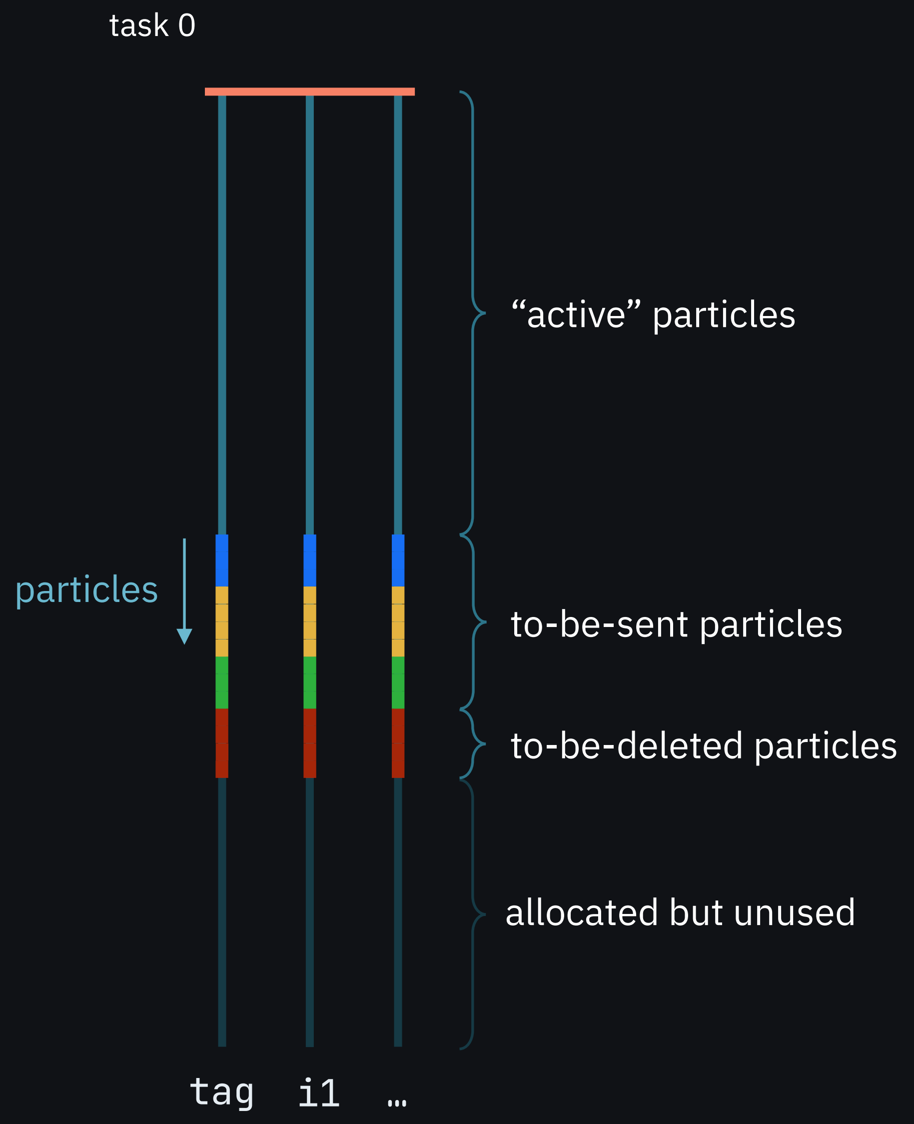

```cpp
using KeyType = Kokkos::View<short*>;
using BinOp   = BinTag<KeyType>;
BinOp bin_op(ntags);
Kokkos::BinSort<KeyType, BinOp>
Sorter(Kokkos::subview(prtl.tag, { 0, npart }),
                                          bin_op,
                                          false);
Sorter.create_permute_vector();
Sorter.sort(Kokkos::subview(prtl.tag, { 0, npart }));
Sorter.sort(Kokkos::subview(prtl.i1, { 0, npart }));
Sorter.sort(Kokkos::subview(prtl.dx1, { 0, npart }));
Sorter.sort(Kokkos::subview(prtl.ux1, { 0, npart }));
…
```

```cpp
template <class KeyViewType>
struct BinTag {
  BinTag(const int& max_bins) : m_max_bins { max_bins } {}

  template <class ViewType>
  Inline auto bin(ViewType& keys, const int& i) const -> int {
    return (keys(i) == 0) ? 1 : ((keys(i) == 1) ? 0 : keys(i));
  }

  Inline auto max_bins() const -> int {
    return m_max_bins;
  }

  template <class ViewType, typename iT1, typename iT2>
  Inline auto operator()(ViewType&, iT1&, iT2&) const -> bool {
    return false;
  }

private:
  const int m_max_bins;
};
```

Bottlenecks & Culprits

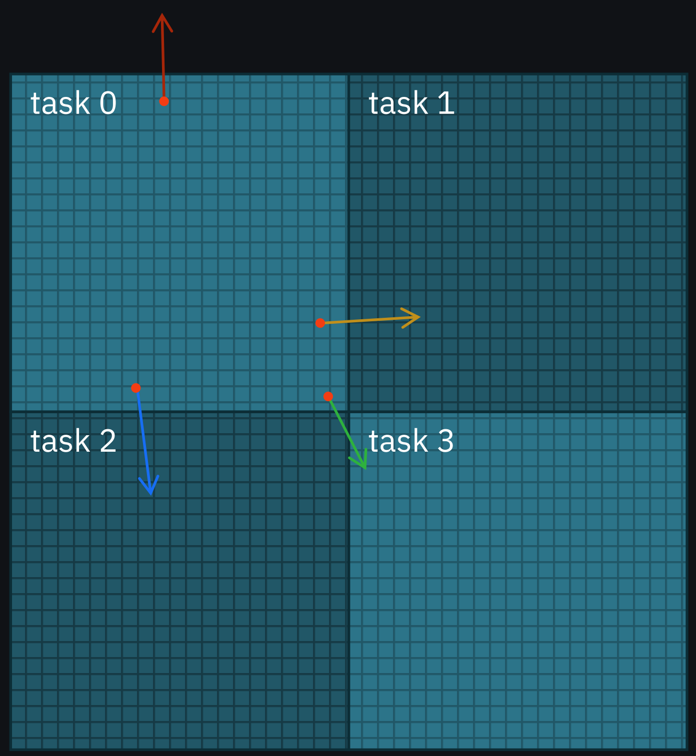
MPI communications of particles

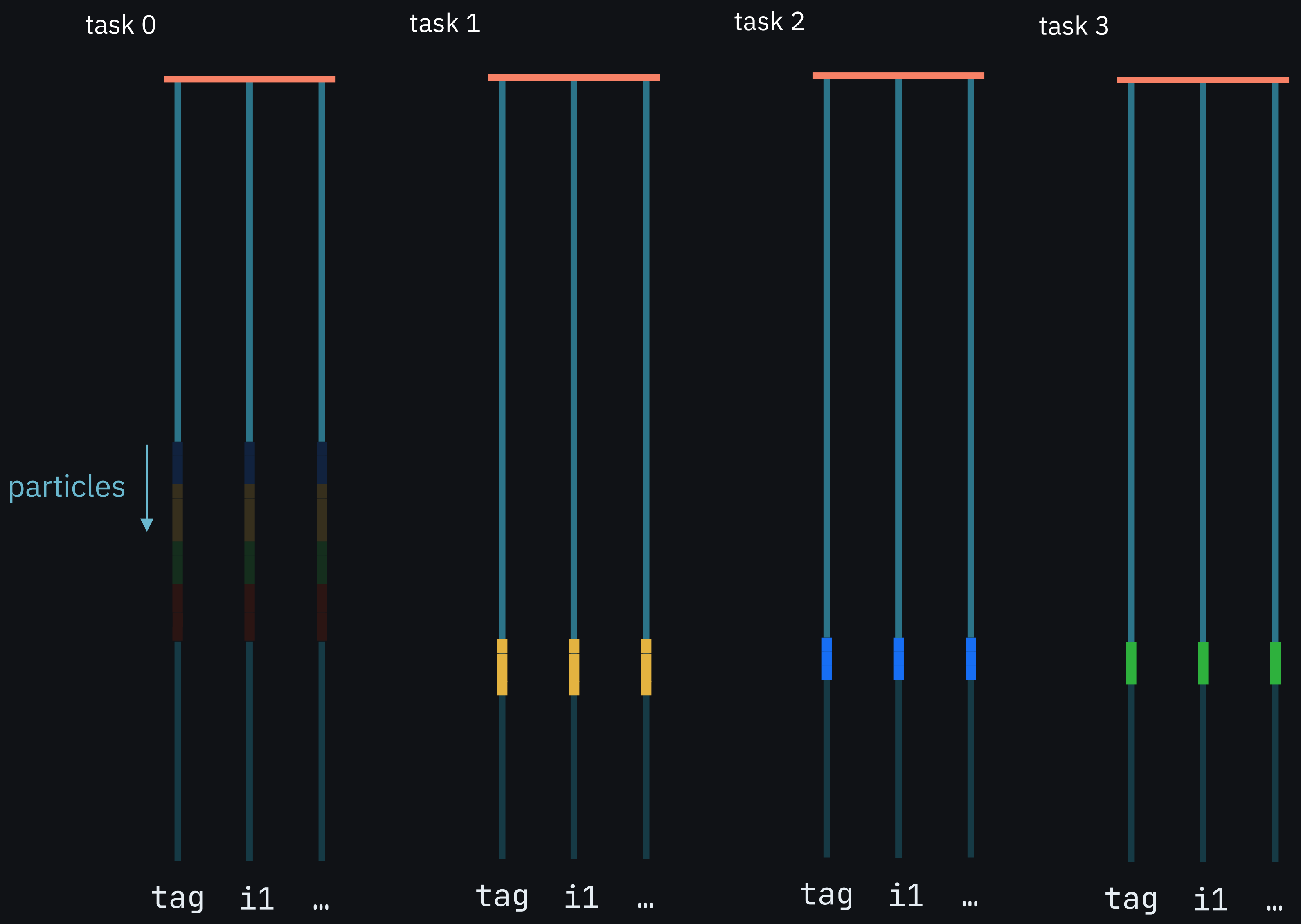




Bottlenecks & Culprits

MPI communications of particles





Bottlenecks & Culprits

MPI communications of particles

Sorting has to be done every timestep
⇒ takes up ~ 80% of the timestep ☹

Possible solutions

1. using *thrust* (similar performance + not portable)

```
thrust::sort_by_key(tag.begin(),
                    tag.begin() + npart,
                    thrust::make_zip_iterator(i1, dx1, ux1, …),
                    comparator);
```

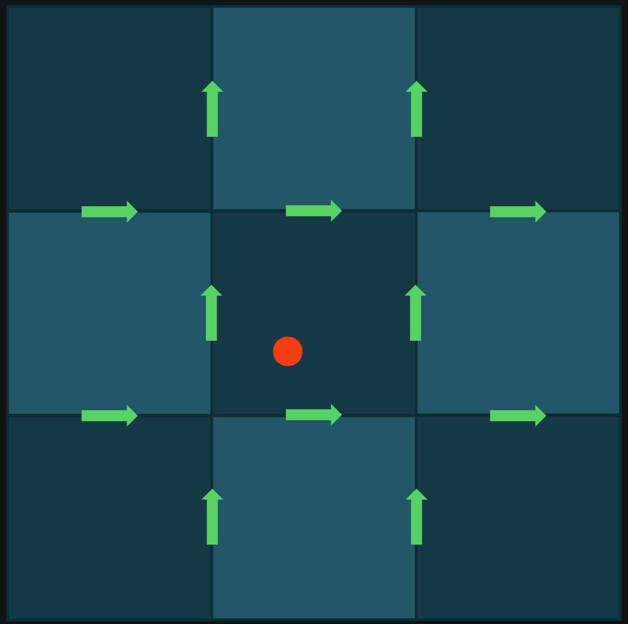
2. pre-extract the data into buffer arrays (not tested)

```
Kokkos::View<std::size_t> send_left_cnt("send_left_cnt");
…
Kokkos::parallel_for(
  "Extract",
  npart,
  KOKKOS_LAMBDA(std::size_t p) {
    if (tag(p) == ParticleTag::sendLeft) {
      Kokkos::atomic_fetch_add(&send_left_cnt(), 1);
      send_left_i1(send_left_cnt) = i1(p);
      tag(p)                      = ParticleTag::dead;
    } else if …
  });
```

```
using KeyType = Kokkos::View<short*>;
using BinOp   = BinTag<KeyType>;
BinOp bin_op(ntags);
Kokkos::BinSort<KeyType, BinOp>
Sorter(Kokkos::subview(prtl.tag, { 0, npart }),
                                       bin_op,
                                       false);
Sorter.create_permute_vector();
Sorter.sort(Kokkos::subview(prtl.tag, { 0, npart }));
Sorter.sort(Kokkos::subview(prtl.i1, { 0, npart }));
Sorter.sort(Kokkos::subview(prtl.dx1, { 0, npart }));
Sorter.sort(Kokkos::subview(prtl.ux1, { 0, npart }));
…
```

Bottlenecks & Culprits

Field interpolation (gathering)



Possible solutions

1. sorting by cells
2. reducing the # of intermediate variables
3. storing fields in smaller chunks (tiles)

Example for just the E^1 component (loads 12 values at different locations)

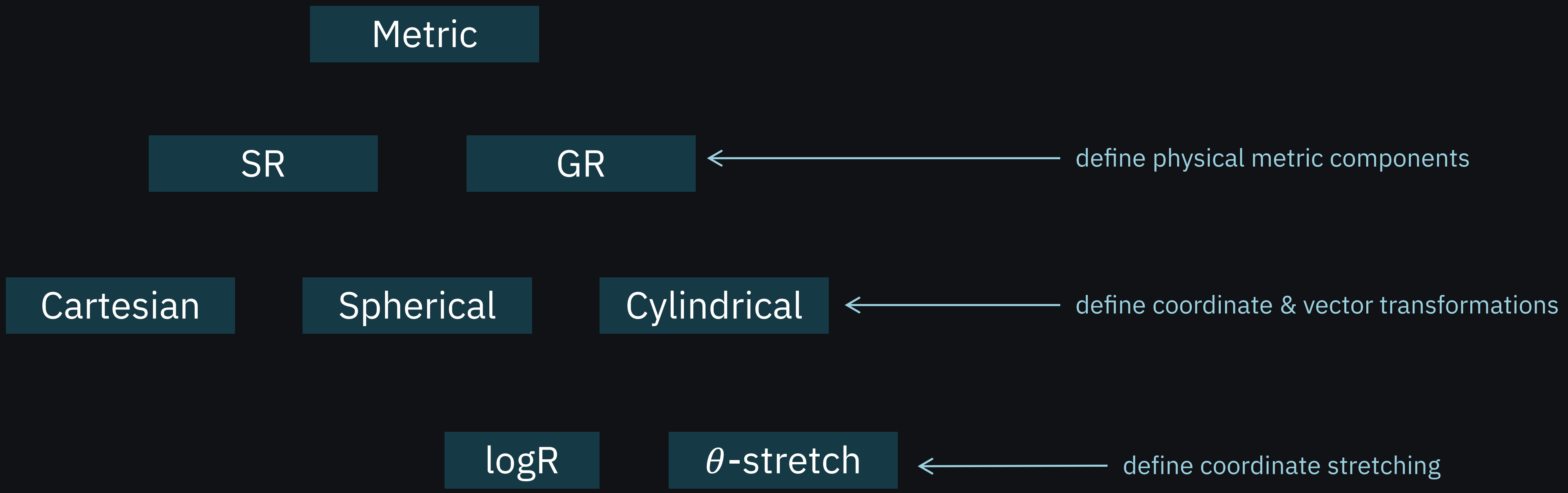
```
struct ParticlePusher_kernel {
  KOKKOS_INLINE_FUNCTION
  // vectorized over particles
  void operator()(std::size_t p) const {
    // interpolate fields …
  }
};
```

```
// Ex1
// interpolate to nodes
c000 = HALF * (EB(i, j, k, em::ex1) + EB(i - 1, j, k, em::ex1));
c100 = HALF * (EB(i, j, k, em::ex1) + EB(i + 1, j, k, em::ex1));
c010 = HALF * (EB(i, j + 1, k, em::ex1) + EB(i - 1, j + 1, k, em::ex1));
c110 = HALF * (EB(i, j + 1, k, em::ex1) + EB(i + 1, j + 1, k, em::ex1));
// interpolate from nodes to the particle position
c00  = c000 * (ONE - dx1_) + c100 * dx1_;
c10  = c010 * (ONE - dx1_) + c110 * dx1_;
c0   = c00 * (ONE - dx2_) + c10 * dx2_;
// interpolate to nodes
c001 = HALF * (EB(i, j, k + 1, em::ex1) + EB(i - 1, j, k + 1, em::ex1));
c101 = HALF * (EB(i, j, k + 1, em::ex1) + EB(i + 1, j, k + 1, em::ex1));
c011 = HALF *
           (EB(i, j + 1, k + 1, em::ex1) + EB(i - 1, j + 1, k + 1, em::ex1));
c111 = HALF *
           (EB(i, j + 1, k + 1, em::ex1) + EB(i + 1, j + 1, k + 1, em::ex1));
// interpolate from nodes to the particle position
c01   = c001 * (ONE - dx1_) + c101 * dx1_;
c11   = c011 * (ONE - dx1_) + c111 * dx1_;
c1    = c01 * (ONE - dx2_) + c11 * dx2_;
ex1_interp = c0 * (ONE - dx3_) + c1 * dx3_;
```

Bottlenecks & Culprits

Virtual metric classes

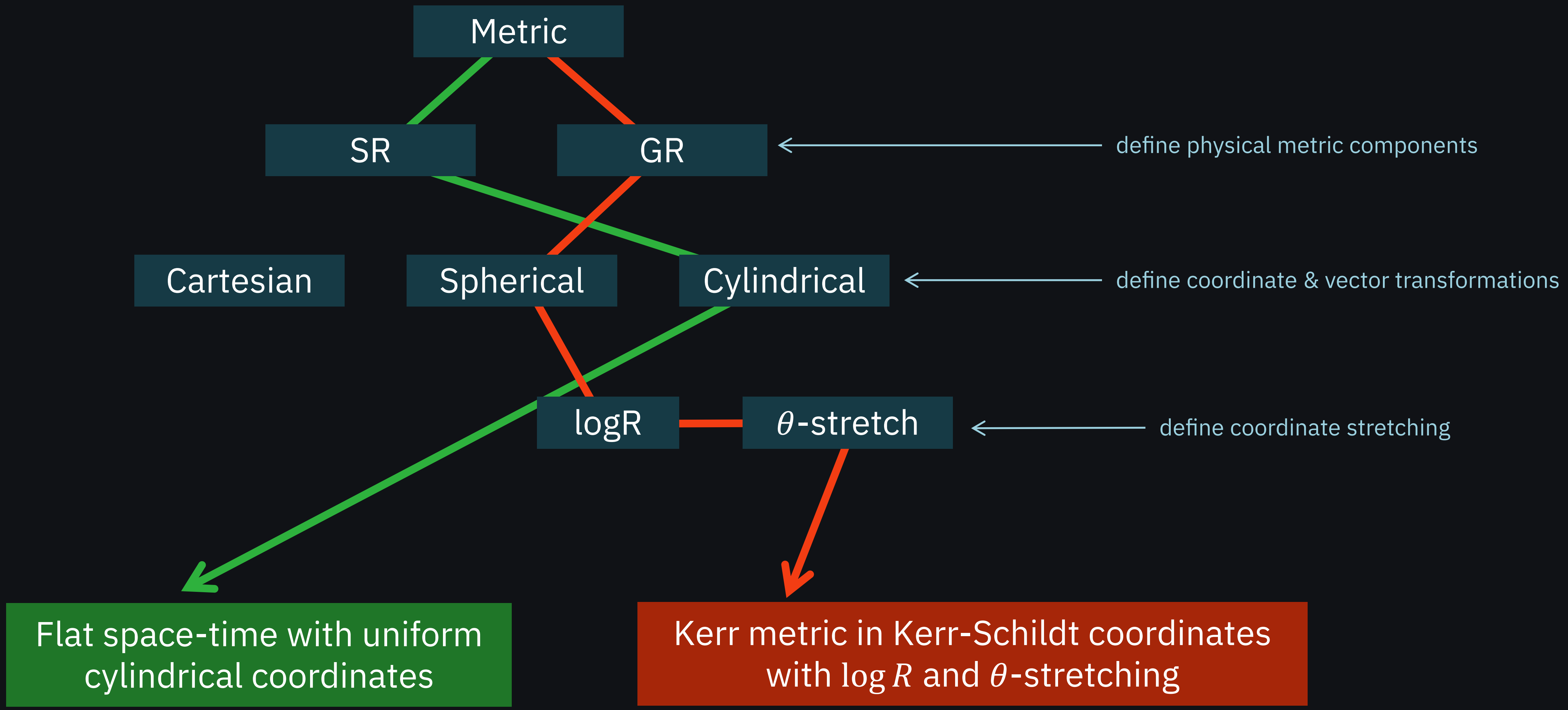
Entity is designed to be coordinate (grid) agnostic, so having flexibility in provided metric structures is critical.



Bottlenecks & Culprits

Virtual metric classes

Entity is designed to be coordinate (grid) agnostic, so having flexibility in provided metric structures is critical.



Takeaways

- *Entity (haykh.github.io/entity)* is the first coordinate-agnostic GR PIC code for astrophysical plasmas written with the Kokkos library.

- preliminary performance is fantastic, and we already use the code for actual science!

challenges by priority:
- performance with MPI
- performance of the field gathering
- reformatting the metric structures
- two-body QED module

dev team

Alexander Chernoglazov {@SChernoglazov: PIC}
Benjamin Crinquand {@bcrinquand: GR, cubed-sphere}
Alisa Galishnikova {@alisagk: GR}
Hayk Hakobyan {@haykh: framework, PIC, GR, cubed-sphere}
Jens Mahlmann {@jmahlmann: cubed-sphere, framework, MPI}
Sasha Philippov {@sashaph: all-around}
Arno Vanthieghem {@vanthieg: PIC, framework}
Muni Zhou {@munizhou: PIC}

We are hiring (ask)!
https://jobregister.aas.org/ad/89c8a064

Supported by the NSF CSSI Grant, and the NVIDIA Corporation Academic Hardware Grant Program.