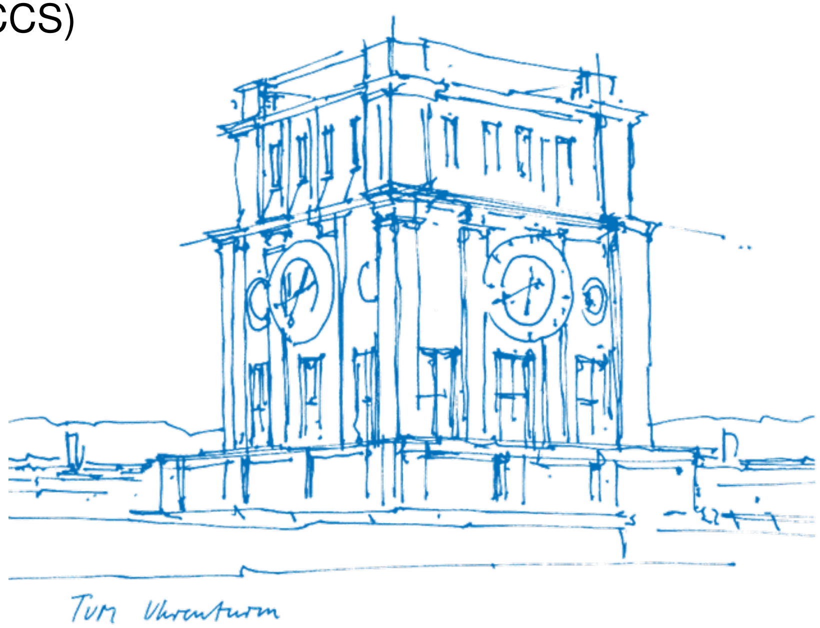
Workshop AutoPas

Fabio Gratl, Philipp Neumann
Technical University of Munich
Department of Informatics
Chair of Scientific Computing in Computer Science (SCCS)
CECAM SWiMM, 18.03.2021



Overview

Requirements

Simulation:

- Linux or WSL
- AutoPas (https://github.com/AutoPas/AutoPas/) $\rightarrow$ branch “swimm“
- CMake ($\geq$ 3.14)
- C++17 compiler: Clang ($\geq$7) / GCC ($\geq$ 7)

Visualization:

- Paraview

Plotting:

- Python3 with pandas, plotly

Introduction

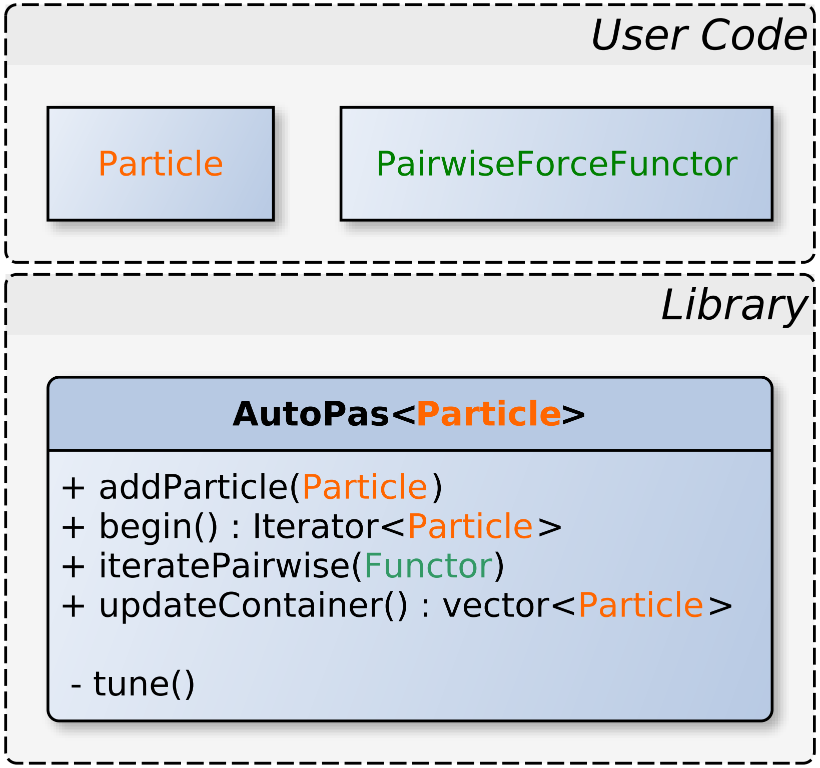
AutoPas

What is AutoPas

- Node-Level `C++17` library
- Black-box particle container
- Facade-like software pattern
- User defines:
  - Properties of particles
  - Force for pairwise interaction
- AutoPas provides
  - Containers, Traversals Data Layouts, ...
  - Dynamic Tuning at run-time

$\Rightarrow$ General base for N-Body simulations

https://autopas.github.io/

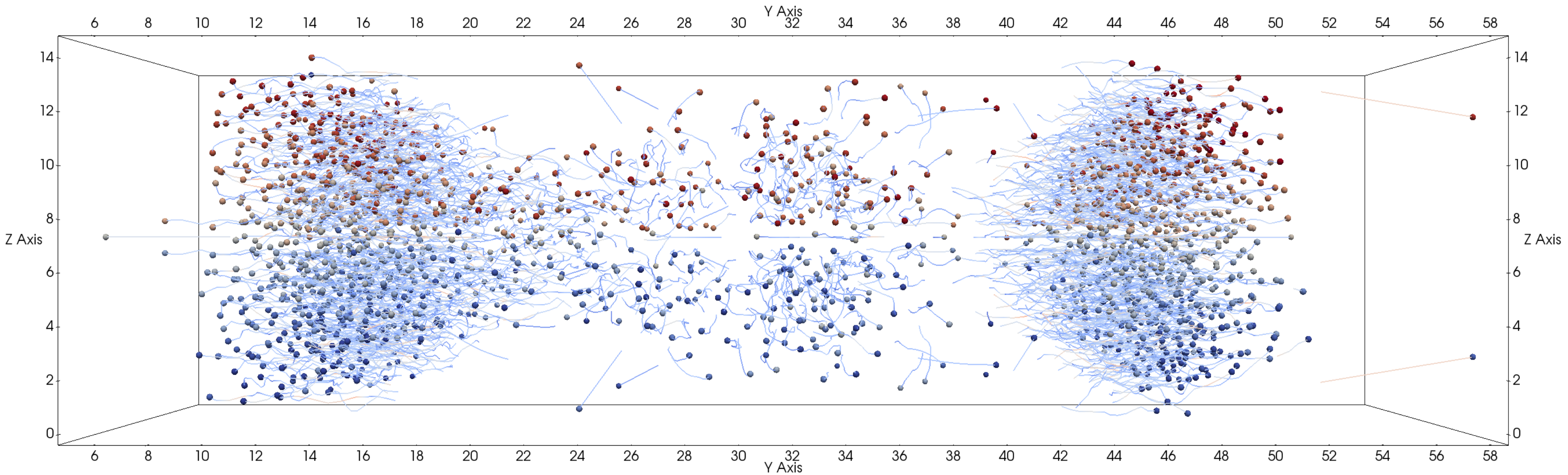


Introduction

MD-Flexible

MD-Flexible

- Example application using AutoPas
- Molecular dynamics simulator
- Demonstrator for all AutoPas features
- Used internally for developing and testing



Hands-On

Working on Individual Particles

Calculate Velocity Updates

```
1  void calculateVelocities(AutoPasTemplate &autopas, const ParticlePropertiesLibraryTemplate &
       particlePropertiesLibrary, const double deltaT) {
2      using autopas::utils::ArrayMath::add;
3      using autopas::utils::ArrayMath::mulScalar;
4
5  #pragma omp parallel
6      for (auto iter = autopas.begin(autopas::IteratorBehavior::ownedOnly); iter.isValid(); ++iter) {
7          auto m = particlePropertiesLibrary.getMass(iter->getTypeId());
8          auto newV = mulScalar((add(iter->getF(), iter->getOldf())), deltaT / (2 * m));
9          iter->addV(newV);
10     }
11 }
```

- Use an iterator to go over all owned particles (6).
- Access the particle via the iterator for reading (7,8) and writing (9).
- Here: Access further particle properties that are bound to the type through a lookup object (7).
- Make use of parallel iterators by creating an OpenMP region (5).

Calculate Velocity Updates

```
1 void calculateVelocities(AutoPasTemplate &autopas, const ParticlePropertiesLibraryTemplate &
    particlePropertiesLibrary, const double deltaT) {
2   using autopas::utils::ArrayMath::add;
3   using autopas::utils::ArrayMath::mulScalar;
4
5 #pragma omp parallel
6   for (auto iter = autopas.begin(autopas::IteratorBehavior::ownedOnly); iter.isValid(); ++iter) {
7       auto m = particlePropertiesLibrary.getMass(iter->getTypeId());
8       auto newV = mulScalar((add(iter->getF(), iter->getOldf())), deltaT / (2 * m));
9       iter->addV(newV);
10  }
11 }
```

- Use an iterator to go over all owned particles (6).
- Access the particle via the iterator for reading (7,8) and writing (9).
- Here: Access further particle properties that are bound to the type through a lookup object (7).
- Make use of parallel iterators by creating an OpenMP region (5).

Calculate Velocity Updates

```
1  void calculateVelocities(AutoPasTemplate &autopas, const ParticlePropertiesLibraryTemplate &
       particlePropertiesLibrary, const double deltaT) {
2      using autopas::utils::ArrayMath::add;
3      using autopas::utils::ArrayMath::mulScalar;
4
5  #pragma omp parallel
6      for (auto iter = autopas.begin(autopas::IteratorBehavior::ownedOnly); iter.isValid(); ++iter) {
7          auto m = particlePropertiesLibrary.getMass(iter->getTypeId());
8          auto newV = mulScalar((add(iter->getF(), iter->getOldf())), deltaT / (2 * m));
9          iter->addV(newV);
10     }
11 }
```

- Use an iterator to go over all owned particles (6).
- Access the particle via the iterator for reading (7,8) and writing (9).
- Here: Access further particle properties that are bound to the type through a lookup object (7).
- Make use of parallel iterators by creating an OpenMP region (5).

Calculate Velocity Updates

```
1  void calculateVelocities(AutoPasTemplate &autopas, const ParticlePropertiesLibraryTemplate &
       particlePropertiesLibrary, const double deltaT) {
2      using autopas::utils::ArrayMath::add;
3      using autopas::utils::ArrayMath::mulScalar;
4
5  #pragma omp parallel
6      for (auto iter = autopas.begin(autopas::IteratorBehavior::ownedOnly); iter.isValid(); ++iter) {
7          auto m = particlePropertiesLibrary.getMass(iter->getTypeId());
8          auto newV = mulScalar((add(iter->getF(), iter->getOldf())), deltaT / (2 * m));
9          iter->addV(newV);
10     }
11 }
```

- Use an iterator to go over all owned particles (6).
- Access the particle via the iterator for reading (7,8) and writing (9).
- Here: Access further particle properties that are bound to the type through a lookup object (7).
- Make use of parallel iterators by creating an OpenMP region (5).

Calculate Velocity Updates

```
1 void calculateVelocities(AutoPasTemplate &autopas, const ParticlePropertiesLibraryTemplate &
    particlePropertiesLibrary, const double deltaT) {
2     using autopas::utils::ArrayMath::add;
3     using autopas::utils::ArrayMath::mulScalar;
4 
5 #pragma omp parallel
6     for (auto iter = autopas.begin(autopas::IteratorBehavior::ownedOnly); iter.isValid(); ++iter) {
7         auto m = particlePropertiesLibrary.getMass(iter->getTypeId());
8         auto newV = mulScalar((add(iter->getF(), iter->getOldf())), deltaT / (2 * m));
9         iter->addV(newV);
10     }
11 }
```

- Use an iterator to go over all owned particles (6).
- Access the particle via the iterator for reading (7,8) and writing (9).
- Here: Access further particle properties that are bound to the type through a lookup object (7).
- Make use of parallel iterators by creating an OpenMP region (5).

Hands-On

Pairwise Force Calculation

Functor AoS

```
template <class Particle>
class LJFunctor : public Functor<Particle, LJFunctor<Particle> > {
  public:

    void AoSFunctor(Particle &i, Particle &j, bool newton3) final {
        double dr = distance(i.getR(), j.getR());
        if (dr > _cutoff) {
            return;
        }

        double f = lennardJonesForce(dr, _sigma, _epsilon);
        i.addF(f);
        if (newton3) {
            j.subF(f);
        }
    }
};
```

- Specify how to calculate your force and how to apply it.
- Currently this has to be done for AoS and SoA but quality of life improvements are under development.

Functor AoS

```
template <class Particle>
class LJFunctor : public Functor<Particle, LJFunctor<Particle> {
  public:

    void AoSFunctor(Particle &i, Particle &j, bool newton3) final {
        double dr = distance(i.getR(), j.getR());
        if (dr > _cutoff) {
            return;
        }

        double f = lennardJonesForce(dr, _sigma, _epsilon);
        i.addF(f);
        if (newton3) {
            j.subF(f);
        }
    }
};
```

- Specify how to calculate your force and how to apply it.
- Currently this has to be done for AoS and SoA but quality of life improvements are under development.

Calculate Pairwise Forces

```
1 void Simulation::calculateForces(autopas::AutoPas<ParticleType> &autopas) {
2     autopas::LJFunctor<Particle> functor(_cutoff, particlePropertiesLib);
3     bool tuningIteration = autopas.iteratePairwise(&functor);
4 }
```

- The functor is applied to all particles via `iteratePairwise()` (3).
- Here: Additional particle properties which are not stored in the particles directly are passed to the functor (2).

Calculate Pairwise Forces

```
1 void Simulation::calculateForces(autopas::AutoPas<ParticleType> &autopas) {
2     autopas::LJFunctor<Particle> functor(_cutoff, particlePropertiesLib);
3     bool tuningIteration = autopas.iteratePairwise(&functor);
4 }
```

- The functor is applied to all particles via `iteratePairwise()` (3).
- Here: Additional particle properties which are not stored in the particles directly are passed to the functor (2).

Hands-On

Updating the Container

Periodic Boundary Conditions

```cpp
1 void applyPeriodic(autopas::AutoPas<Particle> &autoPas, bool forceUpdate) {
2     auto [leavingParticles, updated] = autoPas.updateContainer(forceUpdate);
3     if (updated) {
4         wrapPositionsAroundBoundaries(autoPas, leavingParticles);
5         addEnteringParticles(autoPas, leavingParticles);
6     }
7     auto haloParticles = identifyNewHaloParticles(autoPas);
8     addHaloParticles(autoPas, haloParticles);
9 }
```

```cpp
void addEnteringParticles(autopas::AutoPas<Particle> &autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addParticle(p);
    }
}
```

```cpp
void addHaloParticles(autopas::AutoPas<Particle> &autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addOrUpdateHaloParticle(p);
    }
}
```

- `updateContainer()` updates the internal data container in accordance to the Verlet-like approach and returns all particles that left the domain. (1)
- These leaving particles are inserted on the other side. (4-5)
- Halo Particles are identified via region iterators (not shown here) and copies inserted. (7-8)

Periodic Boundary Conditions

```
1 void applyPeriodic(autopas::AutoPas<Particle> &autoPas, bool forceUpdate) {
2     auto [leavingParticles, updated] = autoPas.updateContainer(forceUpdate);
3     if (updated) {
4         wrapPositionsAroundBoundaries(autoPas, leavingParticles);
5         addEnteringParticles(autoPas, leavingParticles);
6     }
7     auto haloParticles = identifyNewHaloParticles(autoPas);
8     addHaloParticles(autoPas, haloParticles);
9 }
```

```
void addEnteringParticles(autopas::AutoPas<Particle> &autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addParticle(p);
    }
}
```

```
void addHaloParticles(autopas::AutoPas<Particle> &autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addOrUpdateHaloParticle(p);
    }
}
```

- `updateContainer()` updates the internal data container in accordance to the Verlet-like approach and returns all particles that left the domain. (1)
- These leaving particles are inserted on the other side. (4-5)
- Halo Particles are identified via region iterators (not shown here) and copies inserted. (7-8)

Periodic Boundary Conditions

```
1 void applyPeriodic(autopas::AutoPas<Particle> &autoPas, bool forceUpdate) {
2     auto [leavingParticles, updated] = autoPas.updateContainer(forceUpdate);
3     if (updated) {
4         wrapPositionsAroundBoundaries(autoPas, leavingParticles);
5         addEnteringParticles(autoPas, leavingParticles);
6     }
7     auto haloParticles = identifyNewHaloParticles(autoPas);
8     addHaloParticles(autoPas, haloParticles);
9 }
```

```
void addEnteringParticles(autopas::AutoPas<Particle> &autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addParticle(p);
    }
}
```

```
void addHaloParticles(autopas::AutoPas<Particle> &autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addOrUpdateHaloParticle(p);
    }
}
```

- `updateContainer()` updates the internal data container in accordance to the Verlet-like approach and returns all particles that left the domain. (1)
- These leaving particles are inserted on the other side. (4-5)
- Halo Particles are identified via region iterators (not shown here) and copies inserted. (7-8)

Periodic Boundary Conditions

```
1 void applyPeriodic(autopas::AutoPas<Particle> &autoPas, bool forceUpdate) {
2     auto [leavingParticles, updated] = autoPas.updateContainer(forceUpdate);
3     if (updated) {
4         wrapPositionsAroundBoundaries(autoPas, leavingParticles);
5         addEnteringParticles(autoPas, leavingParticles);
6     }
7     auto haloParticles = identifyNewHaloParticles(autoPas);
8     addHaloParticles(autoPas, haloParticles);
9 }
```

```
void addEnteringParticles(autopas::AutoPas<Particle
    > &autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addParticle(p);
    }
}
```

```
void addHaloParticles(autopas::AutoPas<Particle> &
    autoPas, std::vector<Particle> &particles) {
    for (auto &p : particles) {
        autoPas.addOrUpdateHaloParticle(p);
    }
}
```

- `updateContainer()` updates the internal data container in accordance to the Verlet-like approach and returns all particles that left the domain. (1)
- These leaving particles are inserted on the other side. (4-5)
- Halo Particles are identified via region iterators (not shown here) and copies inserted. (7-8)

Hands-On

Supporting SoA

Particle

```
class ParticleBase {
  private:
    size_t _id;
    std::array<double, 3> _r, _f, _v;
    autopas::OwnershipState _ownershipState{OwnershipState::owned};

    enum AttributeNames : int { ptr, id, posX, posY, posZ, forceX, forceY, forceZ, ownershipState };
    using SoAArraysType = typename autopas::utils::SoAType<ParticleBase*, size_t, double, double, double,
        double, double, double, OwnershipState>::Type;

    template <AttributeNames attribute>
    constexpr typename std::tuple_element<attribute, SoAArraysType>::type::value_type get() {
        if constexpr (attribute == AttributeNames::ptr) {
            return this;
        } else if constexpr (attribute == AttributeNames::id) {
            return _id;
        } else ... // all other attributes
    }
    // setter analogous
};
```

- Extra declarations for attributes which are needed in the functor.
- Properties that need to be accessible in the functor should be made accessible via automated getter and setter.

Particle

```cpp
class ParticleBase {
  private:
    size_t _id;
    std::array<double, 3> _r, _f, _v;
    autopas::OwnershipState _ownershipState{OwnershipState::owned};

    enum AttributeNames : int { ptr, id, posX, posY, posZ, forceX, forceY, forceZ, ownershipState };
    using SoAArraysType = typename autopas::utils::SoAType<ParticleBase*, size_t, double, double, double,
        double, double, double, OwnershipState>::Type;

    template <AttributeNames attribute>
    constexpr typename std::tuple_element<attribute, SoAArraysType>::type::value_type get() {
        if constexpr (attribute == AttributeNames::ptr) {
            return this;
        } else if constexpr (attribute == AttributeNames::id) {
            return _id;
        } else ... // all other attributes
    }
    // setter analogous
};
```

- Extra declarations for attributes which are needed in the functor.
- Properties that need to be accessible in the functor should be made accessible via automated getter and setter.

Particle

```cpp
class ParticleBase {
  private:
    size_t _id;
    std::array<double, 3> _r, _f, _v;
    autopas::OwnershipState _ownershipState{OwnershipState::owned};

    enum AttributeNames : int { ptr, id, posX, posY, posZ, forceX, forceY, forceZ, ownershipState };
    using SoAArraysType = typename autopas::utils::SoAType<ParticleBase*, size_t, double, double, double,
        double, double, double, OwnershipState>::Type;

    template <AttributeNames attribute>
    constexpr typename std::tuple_element<attribute, SoAArraysType>::type::value_type get() {
        if constexpr (attribute == AttributeNames::ptr) {
            return this;
        } else if constexpr (attribute == AttributeNames::id) {
            return _id;
        } else ... // all other attributes
    }
    // setter analogous
};
```

- Extra declarations for attributes which are needed in the functor.
- Properties that need to be accessible in the functor should be made accessible via automated getter and setter.

Particle

```cpp
class ParticleBase {
  private:
    size_t _id;
    std::array<double, 3> _r, _f, _v;
    autopas::OwnershipState _ownershipState{OwnershipState::owned};

    enum AttributeNames : int { ptr, id, posX, posY, posZ, forceX, forceY, forceZ, ownershipState };
    using SoAArraysType = typename autopas::utils::SoAType<ParticleBase*, size_t, double, double, double,
        double, double, double, OwnershipState>::Type;

    template <AttributeNames attribute>
    constexpr typename std::tuple_element<attribute, SoAArraysType>::type::value_type get() {
        if constexpr (attribute == AttributeNames::ptr) {
            return this;
        } else if constexpr (attribute == AttributeNames::id) {
            return _id;
        } else ... // all other attributes
    }
    // setter analogous
};
```

- Extra declarations for attributes which are needed in the functor.
- Properties that need to be accessible in the functor should be made accessible via automated getter and setter.

Functor SoA

```
1 template <class Particle>
2 class LJFunctor : public Functor<Particle, LJFunctor<Particle> > {
3     constexpr static auto getNeededAttr() {
4         return std::array<typename Particle::AttributeNames, 9>{
5             Particle::AttributeNames::id, Particle::AttributeNames::posX, ...};
6     }
7     constexpr static auto getComputedAttr() {
8         return std::array<typename Particle::AttributeNames, 3>{
9             Particle::AttributeNames::forceX, ... /* =forceY, forceZ */};
10    }
11  public:
12    void SoAFunctorPair(SoAView<SoAArraysType> soa1, SoAView<SoAArraysType> soa2, bool newton3) final {
13        const auto *const __restrict x1ptr = soa1.template begin<Particle::AttributeNames::posX>();
14        // force calculation similar to AoS Functor
15    }
16 };
```

- Specify what attributes are needed by the functor and which are computed (3-10).
- AutoPas automatically moves the data between the particles (=AoS) and SoA buffers as needed (12-15).

Functor SoA

```
1  template <class Particle>
2  class LJFunctor : public Functor<Particle, LJFunctor<Particle> {
3      constexpr static auto getNeededAttr() {
4          return std::array<typename Particle::AttributeNames, 9>{
5              Particle::AttributeNames::id, Particle::AttributeNames::posX, ...};
6      }
7      constexpr static auto getComputedAttr() {
8          return std::array<typename Particle::AttributeNames, 3>{
9              Particle::AttributeNames::forceX, ... /* =forceY, forceZ */};
10     }
11   public:
12     void SoAFunctorPair(SoAView<SoAArraysType> soa1, SoAView<SoAArraysType> soa2, bool newton3) final {
13         const auto *const __restrict x1ptr = soa1.template begin<Particle::AttributeNames::posX>();
14         // force calculation similar to AoS Functor
15     }
16 };
```

- Specify what attributes are needed by the functor and which are computed (3-10).
- AutoPas automatically moves the data between the particles (=AoS) and SoA buffers as needed (12-15).

Functor SoA

```
1  template <class Particle>
2  class LJFunctor : public Functor<Particle, LJFunctor<Particle> {
3      constexpr static auto getNeededAttr() {
4          return std::array<typename Particle::AttributeNames, 9>{
5              Particle::AttributeNames::id, Particle::AttributeNames::posX, ...};
6      }
7      constexpr static auto getComputedAttr() {
8          return std::array<typename Particle::AttributeNames, 3>{
9              Particle::AttributeNames::forceX, ... /* =forceY, forceZ */};
10     }
11   public:
12     void SoAFunctorPair(SoAView<SoAArraysType> soa1, SoAView<SoAArraysType> soa2, bool newton3) final {
13         const auto *const __restrict x1ptr = soa1.template begin<Particle::AttributeNames::posX>();
14         // force calculation similar to AoS Functor
15     }
16 };
```

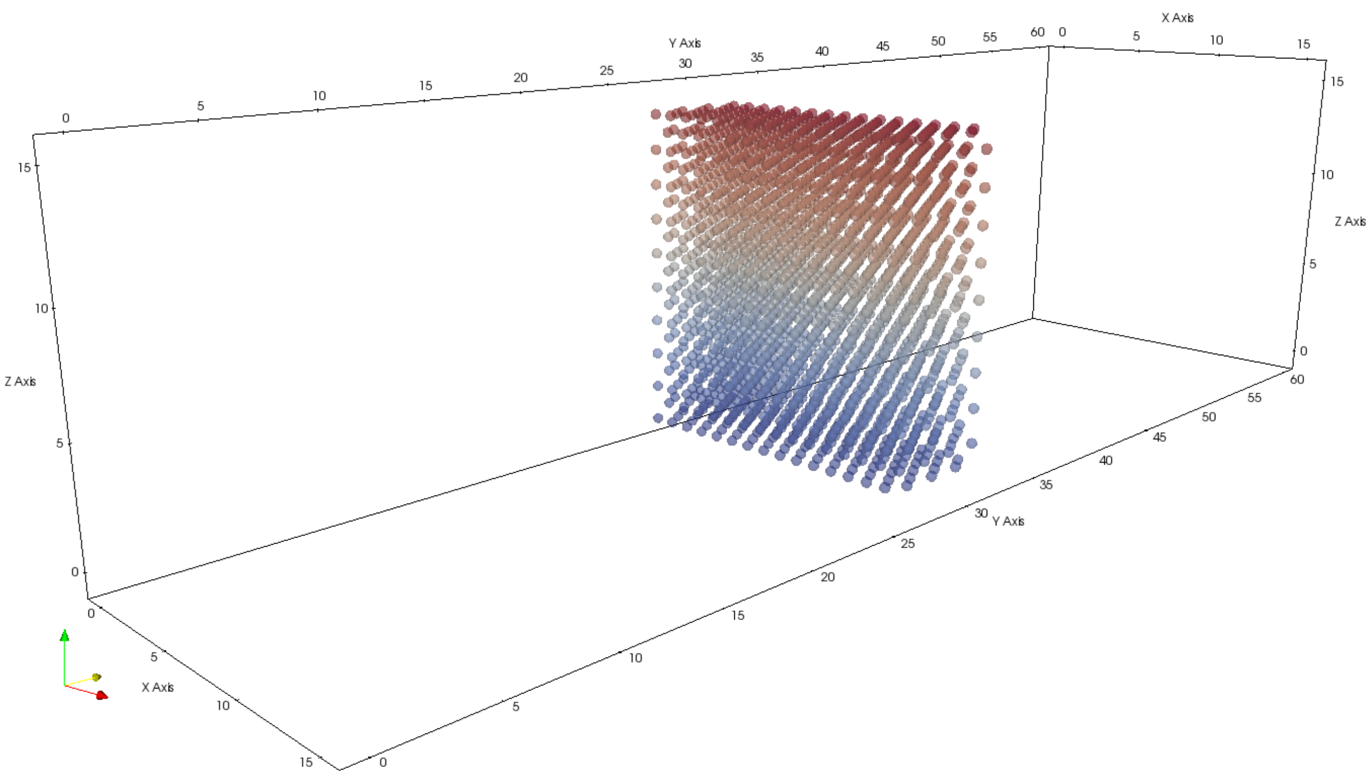
- Specify what attributes are needed by the functor and which are computed (3-10).
- AutoPas automatically moves the data between the particles (=AoS) and SoA buffers as needed (12-15).

Hands-On

Visualization

Visualizing Particles

1. Open Paraview
2. Load vtk Files
3. Apply the Glyph Filter
4. Set Glyph mode to Sphere
5. Disable scaling
6. Show all particles
7. Apply a coloring (id, force, ...)
8. File → Save Animation

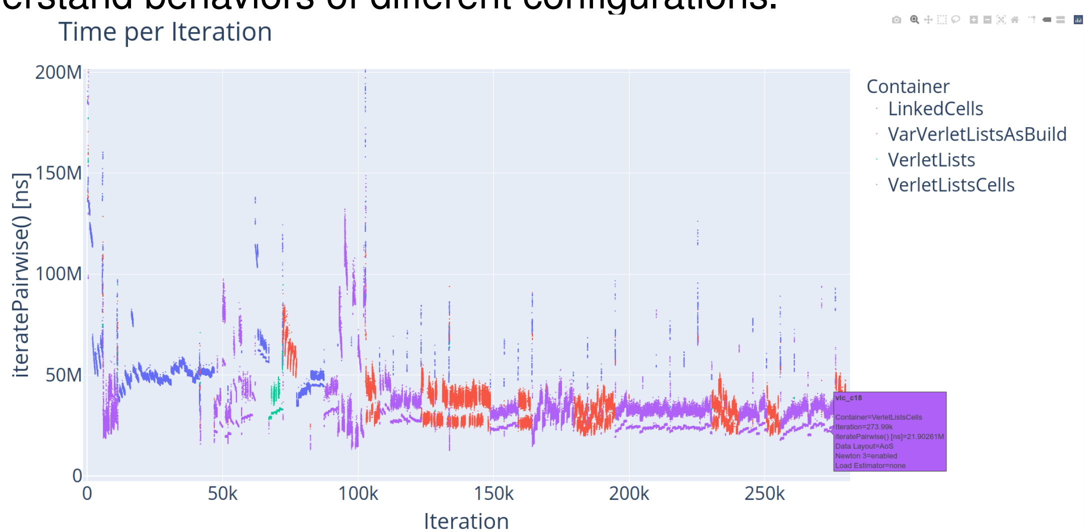


Analyze Performance Data

Activate AutoPas' csv performance data output via `CMake`.

Use `AutoPas/examples/md-flexible/scripts` to analyze csv output:

- plotIterationData.py:
  - Shows time for every iterate pairwise call.
  - Useful to see where most time was spent or spot inefficient tuning decisions.
- plotTuningData.py:
  - Shows smoothed samples that were used by the auto-tuner.
  - Useful to understand behaviors of different configurations.



What was covered?

- Working of individual particles.
  $\Rightarrow$ Calculating velocity updates.
- Pairwise force calculation.
  $\Rightarrow$ Application of a force functor.
- Updating the container object and boundary conditions.
  $\Rightarrow$ Handling leaving and entering particles.
  $\Rightarrow$ Handling halo particles.
- Creating movies of particle trajectories.
- Analyzing AutoPas' performance data.