A SpECTRE With a New 'face

Nils Deppe
Simulating eXtreme Spacetimes Collaboration

Charm++ Workshop

April 11, 2018

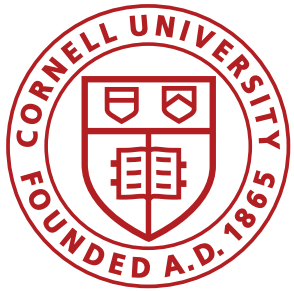


Table of Contents

Physics:

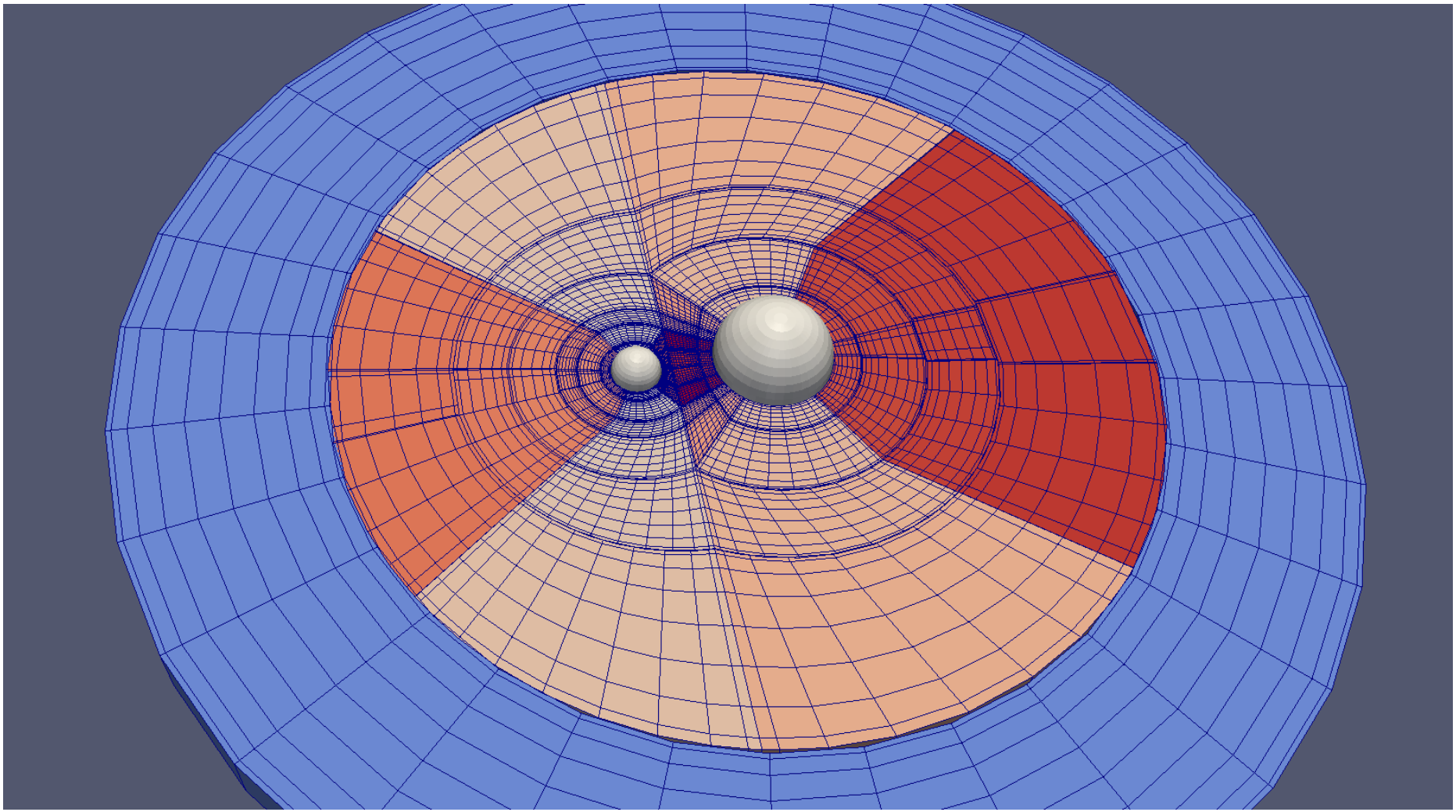
- Multi-scale, multi-physics relativistic astrophysics
- Binary black holes, binary neutron stars
- Core-collapse supernovae with micro-physics
- Multi-disciplinary

HPC:

- Open-source, github.com/sxs-collaboration/spectre
- Efficient
- Exascale

Domain Decomposition and Local Time Stepping

B. Szilagyi, arXiv: 1405.3693



Dual Frames

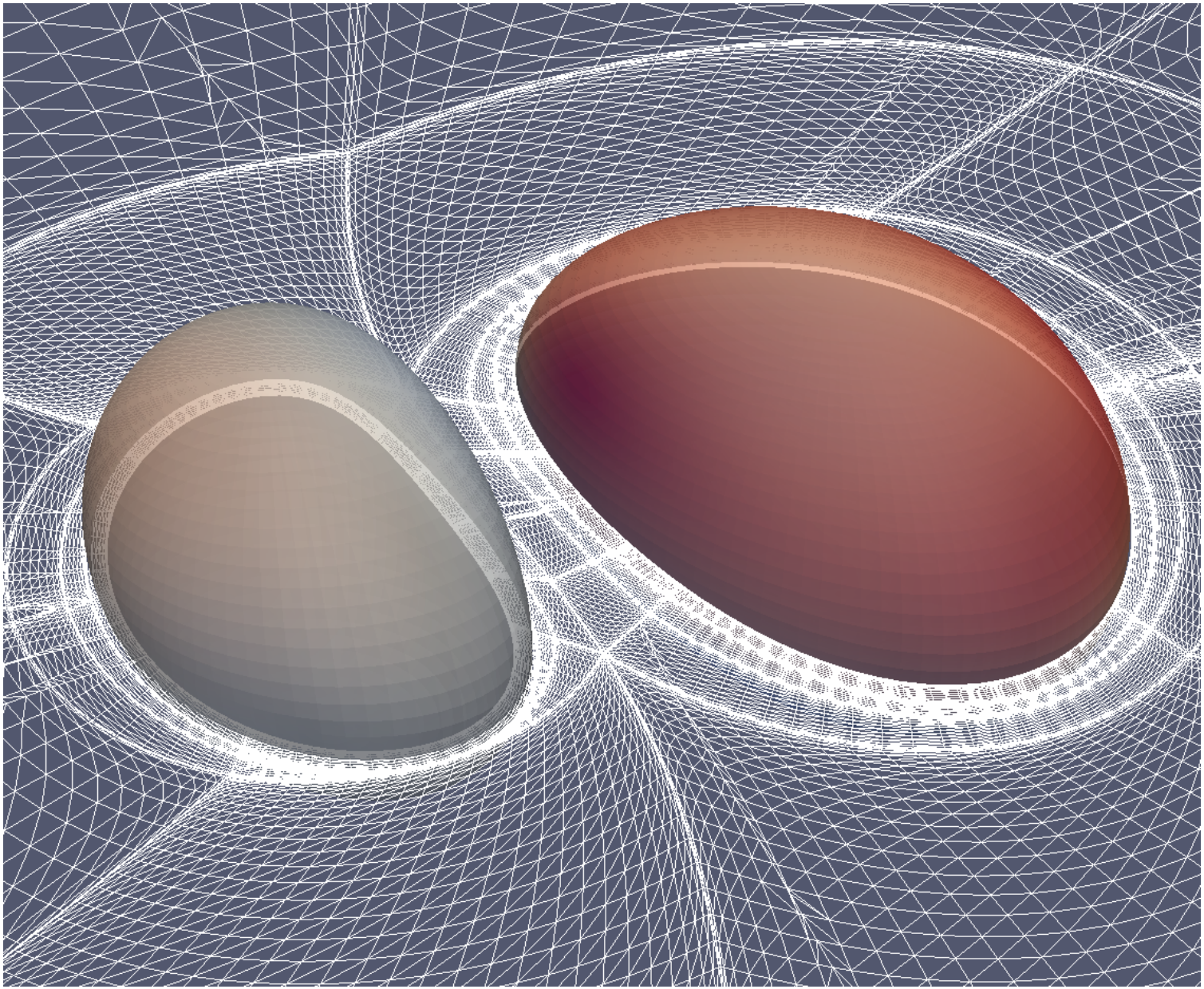


Table of Contents

Eliminating Charm++ Interface Files

Issues with interface files:

- Restrictive
- Error-prone (undefined behavior $\implies$ difficult bugs)
- Maintenance burden (users and Charm++ devs)
- Can't handle modern C++ $\implies$ inefficient generated code

Issues with interface files:

- Restrictive
- Error-prone (undefined behavior $\implies$ difficult bugs)
- Maintenance burden (users and Charm++ devs)
- Can't handle modern C++ $\implies$ inefficient generated code

Metaprogramming!

We think:

- Reduce user code
- Really easy to use
- Similar to current model: familiarity $\Longrightarrow$ faster adoption
- Error-free code generation
- Eliminate runtime errors

Any others??

1. **Invoking entry methods**
2. Creating chares
3. Reductions

Entry methods `MyEntryMethod0`, `MyEntryMethod1`, and Charm++ proxy `my_proxy`.

No arguments:

```
charmxx::invoke<MyEntryMethod0>(my_proxy);
```

Passing arguments:

```
charmxx::invoke<MyEntryMethod1>(my_proxy, arg0,
                                arg1, std::move(arg2));
```

```
struct MyEntryMethod1 {
  static void apply(const Arg0& arg0,
                    const Arg1& arg1,
                    Arg2&& arg2) noexcept {
    /* do work */
  }
};
```

- Entry methods are "member functions"

```
struct MyEntryMethod1 {
  static void apply(const Arg0& arg0,
                    const Arg1& arg1,
                    Arg2&& arg2) noexcept {
    /* do work */
  }
};
```

- Entry methods are "member functions"
- How to handle attributes?

```
struct MyEntryMethod1 {
  static void apply(const Arg0& arg0,
                    const Arg1& arg1,
                    Arg2&& arg2) noexcept {
    /* do work */
  }
};
```

- Entry methods are "member functions"
- How to handle attributes?
- Where is chare member data?

Possible ways of controlling attributes:

- Inside entry method class:

```
struct EntryMethod0 {
  using attributes = charmxx::AttrList<
            charmxx::attrs::Inline>;
  /* apply function*/
```

- At call site:

```
charmxx::invoke<MyEntryMethod1,
          charmxx::AttrList<charmxx::attrs::Inline>>(
          my_proxy, arg0, arg1, std::move(arg2));
```

- Chares hold a `TaggedTuple` i.e. compile-time hash table
- Tags to chare as template parameter, maybe:

```
charmxx::Chare<charmxx::TagList<ParticleCoordinate,
                                ParticleVelocity>>
  my_chare{start_coord, start_velocity};
```

- `TaggedTuple` passed to entry methods

Passing Member Data To Entry Methods

```
struct MyEntryMethod2 {
  template <class... Tags>
  static void apply(charmxx::TaggedTuple<Tags...>&
                      member_data,
                    const double& delta_time) noexcept {

    const auto& vel =
      charmxx::get<ParticleVelocity>(member_data);
    auto& coord =
      charmxx::get<ParticleCoordinate>(member_data);

    coord += vel * delta_time;
  }
};
```

In MyEntryMethod2.hpp:

```
#include "UpdateCoordinate.hpp"

struct MyEntryMethod2 {
  template <class... Tags>
  static void apply(charmxx::TaggedTuple<Tags...>&
                        member_data,
                    const double& delta_time) noexcept {
    update_coordinate(
      charmxx::get<ParticleCoordinate>(member_data),
      charmxx::get<ParticleVelocity>(member_data),
      delta_time);
  }
};
```

In UpdateCoordinate.hpp:

```
void update_coordinate(double& coord, const double& vel,
                       const double& delta_time);
void update_coordinate(Vector& coord, const Vector& vel,
                       const double& delta_time);
```

In UpdateCoordinate.cpp:

```
void update_coordinate(double& coord, const double& vel,
                       const double& delta_time) {
  coord += vel * delta_time;
}

void update_coordinate(Vector& coord, const Vector& vel,
                       const double& delta_time) {
  coord += vel * delta_time;
}
```

1. Invoking entry methods
2. **Creating chares**
3. Reductions

Charm++ will supply class templates, need "names"

```
struct MyChareName {
  // Singleton, Array, Group, or Nodegroup
  using chare_type = charmxx::Array;

  // For arrays must specify index:
  using array_index = MyAwesomeArrayIndex;

  // List of the tags that are member data
  using tags =
    charmxx::TagList<ParticleCoordinate,
                     ParticleVelocity>;

  // To put proxy into a TaggedTuple:
  using type = charmxx::compute_type<chare_type, tags>;
};
```

Create using:

```
auto my_proxy =
      charmxx::MyChareName>(start_coord,
                            start_velocity);
auto my_proxy2 = charmxx::create<MyChareName2>(
      my_proxy, start_coord, start_velocity);
```

- `create` replaces `ckNew`
- Chare name is also tag!

Bonus! Custom Array Indices

Can handle custom array indices more easily, e.g.

```
template <size_t VolumeDim>
class ElementIndex {
 public:
  ElementIndex(const ElementId<VolumeDim>& id) noexcept;

 private:
  std::array<SegmentIndex, VolumeDim> segments_;
};
```

`ElementId` indexes block, x, y, and z in domain

1. Invoking entry methods
2. Creating chares
3. **Reductions**

- Reductions become quite straight forward:

```
charmxx::contribute_to_reduction<
      ProcessReducedProductOfDoublesEntryMethod>(
  my_send_double, array_proxy[my_index],
  target_proxy, charmxx::Reduction::product_double);
```

- Reductions become quite straight forward:

```
charmxx::contribute_to_reduction<
      ProcessReducedProductOfDoublesEntryMethod>(
  my_send_double, array_proxy[my_index],
  target_proxy, charmxx::Reduction::product_double);
```

- Reducing custom data also simpler
- Supply generic data structure for custom reductions

```
charmxx::ReductionData<int,
                       std::unordered_map<std::string,
                                          int>,
                       std::vector<double>>;
```

Function to reduce custom data structure:

```
charmxx::ReductionMsg* reduce_reduction_data(
      const int number_of_messages,
      charmxx::ReductionMsg** const msgs) noexcept {
  /* custom reduction function*/
}
```

Function to reduce custom data structure:

```
charmxx::ReductionMsg* reduce_reduction_data(
      const int number_of_messages,
      charmxx::ReductionMsg** const msgs) noexcept {
  /* custom reduction function*/
}
```

Inside an entry method:

```
charmxx::ReductionData<int,
                       std::unordered_map<std::string,
                                          int>,
                       std::vector<int>> my_send_data{
  10, my_send_map, std::vector<int>{array_index,10,-8}};

charmxx::contribute_to_reduction<
      &reduce_reduction_data,
      ProcessCustomReductionEntryMethod>(
  my_send_data, array_proxy[my_index], target_proxy);
```

Summary

- Charm++ interface files replaced with basic metaprogramming
- Users do not need to metaprogram
- Large number of errors eliminated
- Most remaining errors compile time
- Integrate into Charm++ v7?