SpECTRE: A Next-Generation Relativistic Astrophysics Code

Nils Deppe
Simulating eXtreme Spacetimes Collaboration

Charm++ Workshop

April 18, 2018
1 SpECTRE

2 Template Metaprogramming

3 TMP & SpECTRE
SpECTRE Goals

Physics:
- Multi-scale, multi-physics in astrophysics
- Binary black holes, neutron stars
- Core-collapse supernovae with micro-physics
- Multi-disciplinary

HPC:
- Open-source
- Efficient
- Exascale
SpECTRE: Previous Generation

- Charm++ for parallelization
- Solve PDEs using discontinuous Galerkin + method of lines
- Uniform, Cartesian grid
- Scalar waves, Newtonian + relativistic Euler, relativistic MHD
- Successful study of task-based parallelism + dG
- Closed-source toy code
Kidder et. al, arXiv: 1609.00098
Desired Features for Next Generation

- Abstract method for communication
- Code modularity
- Type-safe data retrieval
- Almost no run-time errors
Container requirements:

- Container to hold disparate types
- Know types at compile-time
- Efficient lookup, preferably at compile-time

Solutions:

- `unordered_map<string, boost::any>`
- `std::tuple<...>`

Serious drawbacks
1. SpECTRE
2. Template Metaprogramming
3. TMP & SpECTRE
Problem: Iterate sequences and associative container
Choose right function:

```cpp
template <class T,
         typename enable_if_t<is_associative<T>::value>*>
double function(const T& v) {...}
```

```cpp
template <class T,
         typename enable_if_t<is_sequence<T>::value>*>
double function(const T& v) {...}
```
Why You Should Care

Pros:

- Generic code
- Computations at compile time
- Catch errors early
- Domain specific languages
- Growing field

Cons:

- “Difficult” to learn (being improved)
- Difficult to debug (templight on GitHub)
- Difficult to optimize (templight on GitHub)
• “string” in map
• Can be template

```cpp
struct MyTag {
    using type = unordered_map<std::string, double>;
};
```
Sequence:

```cpp
template <typename...>
struct typelist {};  
using my_list = typelist<Tag0, Tag1, Tag2, Tag3>;  
```

Retrieve Tag location in tuple:

```cpp
template <typename Tag>
using tag_index = index_of<my_list, Tag>;  
```
Answer: Compile-time “map”

```cpp
template <typename... Tags>
struct TaggedTuple :
    std::tuple<typename Tags::type...>
{
    using tag_ls = typelist<Tags...>;

    template <typename Tag>
    typename Tag::type& get() {
        return std::get<index_of<tag_ls,
                           Tag>::value>(*this);
    }
};
```
1 SpECTRE

2 Template Metaprogramming

3 TMP & SpECTRE
Requirements:

- Lazy evaluation
- Dependency analysis in evaluation
- Lazily recompute on dependency change
DataBox Dependency Example

\[ \sqrt{g/N} \]

\[ g \]

\[ g_{ij} \]

\[ N \]

\[ \psi_{ab} \]

\[ N^i \]

\[ \partial_i N^j \]

\[ \Phi_{iab} \]
DataBox Solution

- std::shared_future for lazy evaluation
- Compile-time digraph for dependency analysis
- Add-remove creates new DataBox, lightweight pointer copy
Tentacle Setup

- Multi-phase initialization
- Complex communication during initialization
- Start simulation after initialization

Use type traits and TMP
Tentacle Tag with initialization functions
Startup function in Tentacle

```cpp
using TentaclesList = typelist<Observers,
                               Elements,
                               Main>;
```
Retrieve observer on this node:

```cpp
auto* local_observer =
    const_global_cache.template get<Observers>()
    .ckLocalBranch();
```

Use `auto` keyword
No `dynamic_cast`
SpECTRE: Current Generation

- Charm++ for parallelization
- Non-uniform grids
- Very modular
- TMP for better errors, easier development and maintenance
- Open sourcing in-progress on GitHub (sxs-collaboration/spectre)
Charm++ Wish List

- Implement our C++11/14 patches into Charm++
- Better TMP support in Charm++
- Replace interface files with TMP