Concept-based runtime polymorphism with Charm++ chare arrays using value semantics

J. Bakosi, R. Bird, C. Junghans
Los Alamos National Laboratory

A.K. Pandare, H. Luo
North Carolina State University

Apr. 11-12, 2018, LA-UR-18-22990

Introduction / Context

Code project

- Hydrodynamics on 3D unstructured grids for **dynamic*** problems
- Solution adaptation with mesh refinement

Strategy for simulation of real-world problems

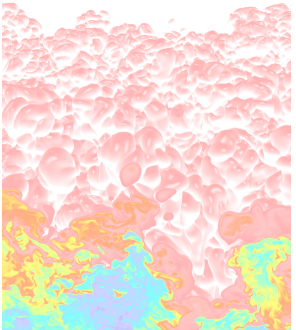
- Build on existing infrastructure (MPI, solvers, libraries)
- Asynchronous, distributed-memory parallel, overdecomposition
- From scratch: *not* based on existing code
- C++11 & Charm++
- Open source: `https://github.com/quinoacomputing/quinoa`

Funding & history

- Started as a hobby project in 2013 (weekends and nights)
- Small funding since 2017

QUINOA

https://github.com/quinoacomputing/quinoa



*A *priori* unknown computational load due to both hardware and software

Near-term plan (2y)

- **Solution-adaptive mesh refinement**
- **Discontinuous Galerkin finite elements** with *NCSU* (see A. Pandare's talk tomorrow)
  - 3rd-order accurate explicit scheme with Runge-Kutta time stepping
  - V&V for smooth and discontinuous problems
  - *p*–refinement
- **Load balancing** for unstructured-mesh PDE solvers with AMR with *Charmworks*
- V&V for discontinuous problems[†] (CG/DG)
- Improve scalability, optimization, cache usage, vectorize, ...
- Compare performance to other LANL codes
- Explore running in containers suitable for the cloud

[†]Kamm et. al, *Enhanced verification test suite for physics simulation codes*, 2008

TPLs: Charm++, Parsing Expression Grammar Template Library, C++ Template Unit Test Framework, Boost, Cartesian product, PStreams, HDF5, NetCDF, Trilinos: SEACAS, Zoltan2, Hypre, RNGSSE2, TestU01, PugiXML, BLAS, LAPACK, Adaptive Entropy Coding library, libc++, libstdc++, MUSL libc, OpenMPI, Intel Math Kernel Library, H5Part, Random123
Compilers: Clang, GCC, Intel
Tools: Git, CMake, Doxygen, Ninja, Gold, Gcov, Lcov, NumDiff

Quinoa: production infrastructure

- 60K lines of *well-commented*[‡] code
- 20+ third-party libraries, 3 compilers
- Unit-, and regression tests (81% coverage)
- Open source: `https://github.com/quinoacomputing/quinoa`
- Code review, github work-flow
- Continuous integration (build & test matrix) with Travis & TeamCity
- Continuous quantified *test code* coverage with Gcov & CodeCov.io
- Continuous quantified *documentation* coverage with CodeCov.io
- Continuous static analysis with CppCheck & SonarQube
- Continuous deployment (of binary releases) to DockerHub

Ported to Linux, Mac, Cray (LANL, NERSC), Blue Gene/Q (ANL)

[‡] Every 3rd line is a comment

Concept-based runtime polymorphism with Charm++ chare arrays using value semantics

Full implementation, more details, and a lot more comments at:

1. https://github.com/quinoacomputing/quinoa/blob/develop/src/Inciter/Scheme.h
2. https://github.com/quinoacomputing/quinoa/blob/develop/src/Inciter/SchemeBase.h
3. https://github.com/quinoacomputing/quinoa/blob/develop/src/Base/Variant.h

Motivation:
In a 30-year-old production code it is practically impossible to add a new hydro scheme

Fact of life:
Different discretization schemes for PDEs can be extremely pervasive on a code

Numerical methods goals:

- Support of multiple discretization schemes
- Easy to add a new scheme
- Scheme selected by user (at runtime)
- Code reuse (in client code)
- Avoid `switch`-mayhem in client code

Software engineering goals:

- Hide, behind a single type, different Charm++ proxy types that model a single concept
- Configured at runtime
- Code reuse (internally)
- Generic
- Extensible
- Maintainable
- Migratable
- Value semantics (internally and client code)
- Avoid `switch`-mayhem in client code
- Concept-based polymorphism (Sean Parent, Adobe)
- Virtual (and overridden) entry methods
- No templates
- Lightweight
- **In other words: runtime polymorphism with chare arrays**

Charm++ supports all this only with reference semantics and `switch`-mayhem

Requirements / Example usage from client code:

```
Scheme s( e );                      // Instantiate a Scheme object
s.coord< tag::bcast >( ... );       // proxy.coord( ... );
s.coord< tag::elem >( 0, ... );     // proxy[0].coord( ... );

// Broadcast to a member function with optional CkEntryOptions
CkEntryOptions opt;
s.coord< tag::bcast >( ..., opt );      // proxy.coord( ..., opt );

// Address array element with optional CkEntryOptions
s.coord< tag::elem >( 0, ..., opt );    // proxy[0].coord( ..., opt );
```

- Ctor configures underlying (child) proxy
- Client code does not know which underlying `Scheme` we dispatch to
- Avoids `switch`-mayhem

Nomenclature

- "Base" proxy and chare array: `discproxy` and `Discretization`
- "Child" proxies and chare arrays:
  - `matcg` and `MatCG` (continuous Galerkin finite elements with a matrix solver)
  - `diagcg` and `DiagCG` (continuous Galerkin with a lumped-mass matrix (diagonal) solver)
  - `dg` and `DG` (discontinuous Galerkin)

Public interface for call to a "base" entry method, `coord()`**:**

```
class Scheme : public SchemeBase {
    using SchemeBase::SchemeBase; // Inherit base constructors
    // discproxy.coord(...)
    template< class Op, typename... Args, typename std::enable_if<
      std::is_same< Op, tag::bcast >::value, int >::type = 0 >
    void coord( Args&&... args ) {
      discproxy.coord( std::forward<Args>(args)... );
    }
    // discproxy[x].coord(...)
    template< typename Op, typename... Args, typename std::enable_if<
      std::is_same< Op, tag::elem >::value, int >::type = 0 >
    void coord( const CkArrayIndex1D& x, Args&&... args ) {
      discproxy[x].coord( std::forward<Args>(args)... );
    }
}
```

Public interface for call to a "child" entry method, `dt()`:

```
class Scheme : public SchemeBase {
    // proxy.dt(...)
    template< class Op, typename... Args, typename std::enable_if<
      std::is_same< Op, tag::bcast >::value, int >::type = 0 >
    void dt( Args&&... args ) {
      boost::apply_visitor( call_dt<Args...>( std::forward<Args>(args)... ),
                            proxy );
    }
    // proxy[x].dt(...)
    template< typename Op, typename... Args, typename std::enable_if<
      std::is_same< Op, tag::elem >::value, int >::type = 0 >
    void dt( const CkArrayIndex1D& x, Args&&... args ) {
      auto e = element< ProxyElem >( proxy, x );
      boost::apply_visitor( call_dt<Args...>( std::forward<Args>(args)... ),
                            e );
    }
}
```

Functor to call the chare entry method, `dt()`:

```
class Scheme : public SchemeBase {
   template< typename... As >
    struct call_dt : Call< call_dt<As...>, As... > {
      using Base = Call< call_dt<As...>, As... >;
      using Base::Base; // inherit base constructors
      template< typename P, typename... Args >
      static void invoke( P& p, Args&&... args ) {
        p.dt( std::forward<Args>(args)... );
      }
    };
}
```

Used with `boost::apply_visitor()`

Dereferencing `operator[]` of a chare proxy

```
template< class ProxyElem >
struct Idx : boost::static_visitor< ProxyElem > {
  Idx( const CkArrayIndex1D& idx ) : x(idx) {}
  template< typename P >
    ProxyElem operator()( const P& p ) const { return p[x]; }
  CkArrayIndex1D x;
};

template< class ProxyElem, class Proxy >
ProxyElem element( const Proxy& proxy, const CkArrayIndex1D& x ) {
  return boost::apply_visitor( Idx<ProxyElem>(x), proxy );
}
```

SchemeBase: types and state

```
class SchemeBase {
  // Variant type listing all chare proxy types modeling the same concept
  using Proxy = boost::variant< CProxy_MatCG, CProxy_DiagCG, CProxy_DG >;
  // Variant type listing all chare element proxy types (behind operator[])
  using ProxyElem =
    boost::variant< CProxy_MatCG::element_t, CProxy_DiagCG::element_t,
                    CProxy_DG::element_t >;

  // Variant storing proxy to which this class is configured for ("child")
  Proxy proxy;
  // Charm++ proxy to data and code common to all discretizations ("base")
  CProxy_Discretization discproxy;
}
```

SchemeBase, ctor: configure underlying scheme

```
class SchemeBase {
  SchemeBase( SchemeType scheme ) :
    discproxy( CProxy_Discretization::ckNew() )
  {
    CkArrayOptions bound;
    bound.bindTo( discproxy );  // Bind child to base when migrated
    if (scheme == SchemeType::MatCG) {
      proxy = static_cast< CProxy_MatCG >( CProxy_MatCG::ckNew(bound) );
    } else if (scheme == SchemeType::DiagCG) {
      proxy = static_cast< CProxy_DiagCG >( CProxy_DiagCG::ckNew(bound) );
    } else if (scheme == SchemeType::DG) {
      proxy = static_cast< CProxy_DG >( CProxy_DG::ckNew(bound) );
    } else Throw( "Unknown discretization scheme" );
  }
}
```

SchemeBase::Call: generic base for all `call_*` classes in `Scheme`

```
class SchemeBase {
  template< class Spec, typename... Args >  // Spec: CRTP to call_*::invoke()
  struct Call : boost::static_visitor<> {
    // Ctor storing called member function arguments in tuple
    Call( Args&&... args ) : arg( std::forward_as_tuple(args...) ) {}

    // Invoke member function with arguments from tuple
    template< typename P, typename Tuple = std::tuple<int> >
    static void invoke( P& p, Tuple&& t = {} )
    { /* See https://stackoverflow.com/a/16868151*/ }

    // Function call operator overloading all types used with variant visitor
    template< typename P > void operator()(P& p) const { invoke(p,arg); }

    std::tuple< Args... > arg;  // Entry method args to be called
  };
}
```

Migration problem

- `boost::variant` (as well as `std::variant` in C++17) when default-constructed is initialized to hold a value of the first alternative of its type list, thus
- calling PUP based on a `boost::visitor` with a templated `operator()` always incorrectly triggers the overload for the first type

Solution: PUP the type!

PUP Scheme/SchemeBase:

```
// Scheme has no state, SchemeBase has two proxies (one is a variant):
class SchemeBase {
  using Proxy = boost::variant< CProxy_MatCG, CProxy_DiagCG, CProxy_DG >;
  // Variant storing proxy to which this class is configured for ("child")
  Proxy proxy;
  // Charm++ proxy to data and code common to all discretizations ("base")
  CProxy_Discretization discproxy;

  void pup( PUP::er &p ) {
    auto v = Variant< CProxy_MatCG, CProxy_DiagCG, CProxy_DG >( proxy );
    p | v;
    proxy = v.get();
    p | discproxy;
  }
}
```

PUP variant: state, pup

```
template< typename... Types >
class Variant {
  Variant( boost::variant< Types... >& v ) : idx( v.which() ), variant(v)
  { boost::apply_visitor( getval(this), v ); }
  boost::variant< Types... > get() { return variant; }  // access
  void pup( PUP::er &p ) {      // pack/unpack
    p | idx;
    p | tuple;
    if (p.isUnpacking())
      boost::mpl::for_each< boost::mpl::vector<Types...> >( setval(this) );
  }
  int idx;                      // Index at which the variant holds a value
  std::tuple< Types... > tuple;       // Can hold any value of the variant
  boost::variant< Types... > variant; // Input/output variant
}
```

PUP variant: get/set

```
// Visitor setting a value of tuple that matches the type of the variant
struct getval : boost::static_visitor<> {
  Variant* const host;
  getval( Variant* const h ) : host(h) {}
  template< typename P > void operator()( const P& p ) const {
    tk::get< P >( host->tuple ) = p;  // C++14: std::get< T >( tuple )
  }
};
// Functor setting the variant based on idx
struct setval {
  Variant* const host;
  int cnt;
  setval( Variant* const h ) : host(h), cnt(0) {}
  template< typename U > void operator()( U ) {
    if (host->idx == cnt++) host->variant = tk::get< U >( host->tuple );
  }           // C++14: std::get< T >( tuple )
};
```

Requirements / Example usage from client code: (once again)

```
Scheme s( e );                          // Instantiate a Scheme object
s.coord< tag::bcast >( ... );           // proxy.coord( ... );
s.coord< tag::elem >( 0, ... );         // proxy[0].coord( ... );

// Broadcast to a member function with optional CkEntryOptions
CkEntryOptions opt;
s.coord< tag::bcast >( ..., opt );      // proxy.coord( ..., opt );

// Address array element with optional CkEntryOptions
s.coord< tag::elem >( 0, ..., opt );    // proxy[0].coord( ..., opt );
```

- Ctor configures underlying (child) proxy
- Client code does not know which underlying `Scheme` we dispatch to
- Avoids `switch`-mayhem

Motivation: (once again)
In a 30-year-old production code it is practically impossible to add a new hydro scheme
(Not in Quinoa!) (Sure, it's not 30 years old, either)

Fact of life:
Different discretization schemes for PDEs can be extremely pervasive on a code
(Not in Quinoa!)

Numerical methods goals:

- Support of multiple discretization schemes **(This works in practice!)**
- Easy to add a new scheme **(See it yourself!)**
  (The implementation is generic. Support for a new scheme is virtually a copy-paste.)
- Scheme selected by user (at runtime)
- Code reuse (in client code)
- Avoid `switch`-mayhem in client code

Conclusion

- C++ allows magic
- Magic is ugly, but
- As long as it is documented and it works, it is usable!