MPI3 Wrapper

Alfredo A. Correa

Lawrence Livermore National Laboratory

October 16, 2017

The problem with the standard C-interface

```
int status = MPI_Send(&number, 1, MPI_INT, 1, 0,
    ↪ MPI_COMM_WORLD);
...// concurrently with
int status = MPI_Recv(&number, 1, MPI_INT, 0, 0,
    ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- Too many arguments
- Force use of raw pointers
- No automatic arguments, redundant, unchecked
- No type-safe, type-erasure by **void***
- Only primitive types
- Only contiguous
- Error codes
- Use of handles with no well defined semantics

MPI3 Wrapper: Design

Principles

- Automatic Resource Management (RAII)
- Value Semantics
- Type-safe
- Generic Programming
- Ranges
- ~~Object Oriented~~ ~~Inheritance RTP~~

Important Details

- Header only
- Simplified syntax
- Handle

Initialization

```
#include "mpi3/environment.hpp"
int main(int argc, char** argv){
    mpi3::environment env(argc, argv); // MPI_Init
    auto& world = env.world(); // communicator is extracted from
        ↪ the environment
    // ... your code here
    // 'env' destroyed automatically here: MPI_Finalize
}
```

- RAII
- No global communicator (explicitly passed or copied)

Communicators

```
#include "mpi3/main.hpp"
#include "mpi3/communicator.hpp"
int mpi3::main(int, char**, mpi3::communicator& world){
   assert(world.size()==8); // run in 8 processes
   mpi3::communicator comm = (world <= 1);
   assert( comm.size()==2 or comm.size()==6);
   return 0;
}
```

- Value semantics by default

```
mpi3::communicator world2 = world;
assert( world2==world );
```

- Operations on communicators

```
mpi3::communicator hemisphere = world/2;
mpi3::communicator interleaved = world%2;
```

MPI is 3 libraries in 1, MPRMAShMem

- Message Passing (MP) (Real Space-QMCPack)
- Remote Memory Access (RMA)
- Shared Memory (ShMem) (AFQMC)

Message Passing (MP): Ranges of values

In STL

```
std::copy(origin.begin(), origin.end(), destination.begin())
```

Message Passing (MP): Ranges of values

In STL

```
std::copy(origin.begin(), origin.end(), destination.begin())
```

In the wrapper

```
int mpi3::main(int, char**, mpi3::communicator& world){
    assert(world.size()==2);
    if(world.rank()==0){
        std::vector<double> v={1.,2.,3.};
        world.send(v.begin(), v.end(), 1); // to rank 1
    }else if(world.rank()==1){
        std::vector<double> v(3);
        world.receive(v.begin(), v.end(), 0); // from rank 0
//          world.receive(v.begin())
    }
}
```

Message Passing (MP): Values

```
    cout << A;
    cin >> A;
```

Message Passing (MP): Values

```
    cout << A;
    cin >> A;
```

```
int mpi3::main(int, char**, mpi3::communicator& world){
    assert(world.size() > 1);
    if(world.rank() == 0){
        int a = 5;
        world[1] << a;
    }else if(world.rank() == 1){
        int a = -1, b = -1;
        world[0] >> a; // specific source (any tag)
        world >> b; // any source (any tag)
        assert(a == 5);
    }
}
```

Remote Memory Access

RMA memory is handled by (non-copyable/movable) memory windows. Communicators can created windows for existing memory.

```
mpi3::window<double> w=world.make_window<double>(ptr, n);
```

Operations on windows, consists in remote operations and synchronization

```
w.fence();
w.put(begin, end, rank);
w.fence();
```

Shared Memory (SM)

SM uses the underlying OS capability to share memory among processes. A special type of communicator can be created by splitting a given communicator.

```
mpi3::shared_communicator node = world.split_shared();
```

Features specific to shared communicators

```
node.all_reduce_in_place(begin, end);
mpi3::shared_window<int> win =
    node.make_shared_window<int>(node.rank()==0?0:100);
```

Serialization

Non-POD types require active code on both sides to transmit information.

```
#include<boost/serialization/string.hpp>
struct MyType{
    std::string name; int n=0; double* data; // ...
    template<class Archive>
    void save(Archive& ar, const unsigned int) const{
        ar << name << n << boost::serialization::make_array
    ↪ (data, n);
    }
    template<class Archive>
    void load(Archive& ar, const unsigned int){
        ar >> name >> n; delete[] data; data=new double[n];
        ar >> boost::serialization::make_array(data, n);
    }
};
```

Serialization can be faster than (non-contiguous) datatypes.

MPI-aware Allocator: at least use/steal the Allocator

`gitlab.com/correaa/boost-mpi3/blob/master/allocator.hpp`
Uses `MPI_Allocate` and `MPI_Free` internally (RAII).
Containers:

```
#include "mpi3/allocator.hpp"
...
std::vector<double>
    → std::vector<double, mpi3::allocator<double>>
```

Buffers:

```
std::vector<char>
    → std::vector<double, mpi3::uallocator<char>>
```

Tips:

- Use **typedefs** (**using** `vector = std::vector<...>`)
- Use generic interfaces (**template**<**class** `Vector> fu(Vector&&...)`)
- Use it for frequently communicated data

Conclusions

- MPI3 standard is very large (MPRMAShMem)
- Calls for simplification
- Concentrate in the important things
- Custom types: Datatypes vs. Serialization

Visit `https://gitlab.com/correaa/boost-mpi3`