Simone Pellegrini, Radu Prodan and Thomas Fahringer

A Lightweight C++ Interface to MPI
MPI is a relatively old standard
- MPI 1.0 was introduced in 1994

Designed upon an abstraction level which is the **lowest common denominator** across C, C++, and Fortran
- i.e. functions calls, pointers, data types, etc...

MPI standard evolved over 17 years by solely adding new primitives
- MPI 3.0 will contain several hundreds routines
- Abstraction level stays the same
MPI did not keep the pace with the evolution of languages in the last 17 years

- Objects Oriented Programming in Fortran 2000
- C++ Templates and template meta-programming
- Lambdas and variadic templates in C++11

HPC community is also slowly adopting C++
MPI C++ Bindings Deprecated

- MPI committee deprecated the C++ binding in MPI 2.3
  - They will be removed in MPI 3.0

- Why?
  - Poor overall integration in the C++ environment (e.g. with the standard template library STL)
  - Level of abstraction far lower than typical C++ libraries
  - C bindings are preferred even in C++ applications
  - Weaken type safety when mapping common C++ constructs to MPI
Outline

- Overview of existing MPI C++ bindings
  - Boost.MPI

- Introduction of MPP (MPI C++) interface
  - MPI endpoints as streams
  - Integration with user and legacy data types

- Performance evaluation
  - Interface overhead
  - QUAD_MPI real case scenario
if ( rank==0 ) {
    MPI_Send ((const int[1]) { 2 }, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );

    std::array<float,2>& val = { 3.14f, 2.95f};
    MPI_Send (&val.front(), val.size(), MPI_FLOAT, 1, 0, MPI_COMM_WORLD );
}
else if (rank == 1) {
    int n;
    MPI_Recv (&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    std::vector<float> values(n);
    MPI_Recv (&values.front(), n, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
60% of MPI routines parameters can be either
- **Inferred** by the compiler (type and size of sent/received data)
- **Defaulted** (tag = 0, comm = MPI_COMM_WORLD, status = MPI_STATUS_IGNORE)

Send/Recv both requires a **void*** which
- **Weaken** type safety
- Transfer **constant values** (a.k.a. R-Values) inefficient
  - Unless c99’s compounds literals are used as in line 1
- Miss an integration with C++ references
Overview of existing MPI C++ bindings
  - Boost.MPI

Introduction of MPP (MPI C++) interface
  - MPI endpoints as streams
  - Integration with user and legacy data types

Performance evaluation
  - Estimate the interface overhead
  - QUAD_MPI real case scenario
```cpp
if (world.rank() == 0) {
    world.send(1, 1, 2);
    world.send(1, 0, {3.14f, 2.95f});
} else if (world.rank() == 1) {
    int n;
    world.recv(0, 1, n);
    std::vector<float>& values(n);
    world.recv(0, 0, values);
}
```
Advantages
- Much easier to read (less syntax)
- Type and size of the transmitted data is inferred
- Includes support for C++ references

Disadvantages
- No default parameters
- MPI endpoints not handled as streams using the insertion `<<` and extraction `>>` C++ stream operations
  - Communication statements cannot be combined, e.g.
  - `cout << "Hello" << "world"`
- Use of software serialization for transmitting user data types with poor performance
  - No integration with `MPI_Datatypes`
Overview of existing MPI C++ bindings
  - Boost.MPI

Introduction of MPP (MPI C++) interface
  - MPI endpoints as streams
  - Integration with user and legacy data types

Performance evaluation
  - Estimate the interface overhead
  - QUAD_MPI real case scenario
Interface to MPI smoothly integrated into the C++ programming language
- Combines concepts of object-oriented programming and templates

**Key features:**
- **Header only** interface
- Reduce the number of parameters MPI routines and infer as much as possible at compile-time
- Improved integration with user and legacy data types with safer type checking
- Use **compiler optimizations** to reduce overhead
  - e.g. function inlining, constant propagation
- Focus on **point-to-point** communications
using namespace mpi;

if ( comm::world.rank() == 0 ) {
    comm::world(1) << 2 << { 3.14f, 2.95f };
}

else if ( mpi::world.rank() == 1 ) {
    int n;
    comm::world(0) >> n;
    std::vector<float>& values(n);
    comm::world(0) >> values;
}
MPP Features

- Concept of **endpoint**
  - `mpi::endpoint comm::operator() (int rank);`
  - Can be used as input/output stream using the insertion `<<` and extraction `>>` operators (lines 3-4)

- Infer **type** and **size** of the transmitted data and perform **type checking**

- Message **tag** is optional
  - By default messages are sent with a `tag 0` (lines 3,9)
  - User can provide a different value, if needed (lines 4,7)
Asynchronous Communications

- **Future pattern** utilized for simplifying the use of asynchronous communications

```cpp
1 float real;
2 mpi::request<float>&& req =
    mpi::comm::world(mpi::any) > real;
3 // ... do something else ...
4 use( req.get() );
```

- **T& request::get()** blocks for the communication to complete and directly returns a reference to received value
User and Legacy Data types

- A good MPI interface must be capable of dealing with non-primitive data types

- BoostMPI
  - Generic but inefficient approach based on software serialization

- MPI_Datatypes: MPI’s mechanism to deal with user-defined types
  - Very efficient, but cumbersome to use, unknown to most programmers
  - Programmers need to manually specify the starting address of the object in memory and its layout
    - The number of elements that compose the type
    - For each element recursively, the displacement from the starting address of the container type and their MPI_Datatype
We support sending user/legacy data types in MPP via the \textit{type traits} design pattern.

For any new data type the programmer specifies via a C++ template specialization class:

- How to determine its \textit{starting memory address}
- The \textit{number of elements} composing the type
- Type of elements
Type Traits to Handle STL Vectors

template <class T>
struct mpi_type_traits<std::vector<T>> {
    static inline const T* get_addr( const std::vector<T>& vec ) {
        return mpi_type_traits<T>::get_addr(vec.front());
    }
    static inline const size_t get_size( const std::vector<T>& vec ) {
        return vec.size();
    }
    static inline MPI_Datatype get_type( const std::vector<T>& ) {
        return mpi_type_traits<T>::get_type(T());
    }
};

vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };
comm::world(1) << v

typedef mpi_type_traits<vector<int>> vect_traits;
MPI_Ssend( vect_traits::get_addr(v), vect_traits::get_size(v),
        vect_traits::get_type(v), ... );
Outline

- Overview of existing MPI C++ bindings
  - Boost.MPI

- Introduction of our MPP (MPI C++) interface
  - Endpoints as streams
  - Integration with user and legacy data types

- Performance evaluation
  - Estimate the interface overhead
  - QUAD_MPI real case scenario
Library Overhead

- 3 interfaces
  - Standard MPI C bindings
  - Boost.MPI
  - MPP

- The MPI library utilized is the same, i.e. Open MPI 1.4.2

- 2 micro-benchmarks to measure
  - Throughput
  - User data type overhead
Simple **ping-pong** application using shared memory communication

- **--mca btl sm** flag
- Minimize communication
- Emphasize library overhead

![Throughput graph](image)
std::list<double> Linked List

- Ping-pong application
- Processes allocated on different nodes
- Infiniband interconnection
QUAD_MPI

- Integral approximation using the quadrature rule
  - Rank 0 assigns a sub-range \([A, B)\) of the problem to other \(P-1\) processes
  - \(P\) processes compute in parallel the partial results for \([A, B)\)
  - Result is aggregated via reduction

- MPP reduces the input code by 30% in terms of number of characters

- MPP has better performance due to elimination of temporary variable assignments
Conclusions and Future Work

- Lightweight C++ interface to MPI

- Higher abstraction level
  - C++ stream operators for send/receive operations
  - Infer data type and size with improved type checking
  - Integrations with MPI_Datatypes instead of serialization with improved performance

- Missing features
  - Integration of collective communications
  - Simplify utilization of other cumbersome features of the MPI library (e.g. dynamic processes, topologies)
Thanks for your Attention

Questions?