MPI‘s C++ Interface

Christian Terboven

terboven@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen

MPI and C++: the Basics

- Let‘s state that clear: most C++ programs use the plain C interface of MPI, and that is perfectly fine!

- In addition to that, C++ programs may use the C++ bindings
- Simple example: MPI_Init function:
  - C: `int MPI_Init(int* argc, char*** argv)`
  - C++: `void MPI::Init(int& argc, char**& argv)`
    `void MPI::Init()`
- Similar: MPI_Finalize function
  - C: `int MPI_Finalize()`
  - C++: `void MPI::Finalize()`
- What is different?
  - The functions are defined within the namespace `MPI`
  - Arguments are declared with *references* instead of *pointers*

MPI and C++: the Basics

- You might have already noticed: the design of MPI was based on the notion of objects

- Most MPI functions are methods of MPI C++ classes
- MPI class names are derived from the language neutral MPI types by dropping the `MPI_` prefix and scoping the type within the `MPI` namespace: `MPI_DATATYPE` becomes `MPI::Datatype`

- The following is an excerpt of the C++ classes of MPI-1:

```
namespace MPI {
   class Comm {...};                          class Errhandler {...};
   class Intracomm : public Comm {...};        class Exception {...};
   class Graphcomm : public Intracomm {...};  class Op {...};
   class Datatype {...};                       class Status {...};
};
```

MPI and C++: the Basics

- An implementation is advised to provide the C++ binding in the `mpi.h` file as well

- Most constants are of type `const int`:
  - `MPI::ANY_SOURCE`, `MPI::ANY_TAG`, …
  - `MPI::MAX_PROCESSOR_NAME`, …

- The elementary datatypes are const types, `const MPI::Datatype`:
  - `MPI::CHAR`, `MPI::INT`, `MPI::DOUBLE`, …
  - `MPI::INTEGER`, `MPI::REAL`, …

- The predefined communicators are of type `MPI::Intracomm`:
  - `MPI::COMM_WORLD`
  - `MPI::COMM_SELF`

MPI and C++: the Basics + Communication

- Collective operators are of type `const MPI::Op`:
  - `MPI::MAX`, `MPI::SUM`, …
- Those functions are typically `virtual const`:
  - MPI_Send: `void Comm::Send(const void* buf, int count, const Datatype& datatype, int dest, int tag) const`
- The same holds for the collective communication functions:
  - MPI_Barrier: `void Intracomm::Barrier() const`
- If a function has just one argument that is intended to be an output and is not a status object, that argument is dropped and the function returns that value:
  - `int MPI::Comm::Get_rank()`

MPI and C++: Inquiry

- The following functions are not bound to any objects (excerpt):
  - `void Get_processor_name(char* name, int& resultlen)`
  - `double Wtime(), double Wtick()`

- As MPI_Status is now a class, it provides member functions:
  - `int Status::Get_source() const`
  - `void Status::Set_source(int source)`
  - Same for *tag* and *error*

- Detailed information: read and follow the links at:
  http://www.mpi-forum.org/docs/mpi-20-html/node21.htm#Node21

MPI and C++: Error handling

- C++ functions do not return error codes
  - If the default error handler is set to `MPI::ERRORS_THROW_EXCEPTIONS`, then the C++ exception mechanism will be used
  - An error handler can be set for the classes `MPI::Comm`, `MPI::File` and `MPI::Win` using the member function `Set_errhandler()`
  - Better don‘t mix this with C code

- The class `MPI::Exception` is basically a wrapper around an `int`, it also provides a way to return an error description string:

```
Exception::Exception(int error_code);
int Exception::Get_error_code() const;
int Exception::Get_error_class() const;
const char* Exception::Get_error_string() const;
```

MPI and C++: example code

```
01 #include "mpi.h"
02 #include <iostream>
03
04 int main(int argc, char* argv[])
05 {
06    MPI::Init(argc, argv);
07    MPI::COMM_WORLD.Set_errhandler(MPI::ERRORS_THROW_EXCEPTIONS);
08
09    try {
10       int rank = MPI::COMM_WORLD.Get_rank();
11       std::cout << "I am " << rank << std::endl;
12    }
13    catch (MPI::Exception e) {
14       std::cout << "MPI ERROR: " << e.Get_error_code() \
                   << " - " << e.Get_error_string() \
                   << std::endl;
16    }
17
18    MPI::Finalize();
19    return 0;
20 }
```

Do you have any questions?