mpi_api_wrapper Documentation

Darcy Beurle

Feb 23, 2020

Contents

This project aims to provide a C++ interface to the MPI C library. The official C++ bindings did little to make the MPI library natural to a C++ programmer.

A more natural interface is achieved through the use of advanced C++ features that perform the correct type checks, enforce `const` correctness and reduces the number of errors associated with using the MPI C bindings.

To maintain consistency with the standard C++ library, the project mimics the naming conventions and aims to provide a zero-cost abstraction on top of the C bindings. It is important to note that some function names will change to be clearer and their MPI C equivalents will be noted or obvious from the name.

Please note that due to limited developer resources, not all functions will be wrapped but will happily be included on a user request. If you feel that there is a missing feature please open an issue on the issue tracker.

CHAPTER 1

Contents

1.1 Introduction

The easiest way to use this library is to read the documentation and checkout the test examples. However, a quick demonstration of the code for an all reduce operation of a single scalar is:

```
double sum, local_sum = 1.0;
MPI_Allreduce(&local_sum,
              &sum,
              1,
              MPI_DOUBLE,
              MPI_SUM,
              MPI_COMM_WORLD);
```

where the count and type are specified when they could be deduced. If we pass a scalar value, then we know the count is 1 and the type is already given by the `sum` and `local_sum` variable. If the `MPI_SUM` variable or the `MPI_DOUBLE` is specified incorrectly, there will be a silent error. Instead, we write:

```
mpi::instance instance(argc, argv);

auto const total_sum = mpi::all_reduce(my_local_sum, mpi::sum{});

// When instance is out of scope MPI_Finalize() is automatically called
```

where the sum is a return value from the function can then be made `const` in contrast to the C interface. Here the function defaults the `communicator` to the world type based on the implementation.

1.2 Creating an MPI instance

In order to use the MPI library, we need to create an MPI instance. To accomplish this, we simply create an instantiation of an `mpi::instance` object

```
int main(int argc, char* argv[])
{
    mpi::instance instance(argc, argv);

    // Do parallel work

    return 0;
}
```

Notice that we do not need to call any functions for finalising an MPI program, since this is handled by the destructor of the instance object. This helps when exiting a program from multiple locations to ensure the parallel environment is correctly finalised.

We can take this one step further an allow a creation of an MPI environment where the multiple threads can call the MPI functions. For this we add an additional construction parameter `mpi::thread` which takes the values `single`, `funnelled`, `serialised` and `multiple` which becomes

```
int main(int argc, char* argv[])
{
    mpi::instance instance(argc, argv, mpi::thread::single);

    // Do parallel work

    return 0;
}
```

1.3 Sending and receiving messages

In order to do any meaningful parallel computation, the program needs to communicate data with other processors in the network. One of the most basic tools in the MPI toolbox is the `send` and `receive` functions. There are two types; blocking and non-blocking.

The function names are constant and type tags are used to disambiguate the function calls. For example, a simple blocking send of `value` to process 1 is

```
mpi::send(mpi::blocking{}, 1.0, 1);
```

and the corresponding receive from process 0 on process 1 is

```
auto const recv_value = mpi::recieve<double>(0);
```

To receive a value from a process we must specify the type as a template parameter, otherwise the compiler cannot know what to allocate to receive the value. We can make this extra safe by using `decltype(the_variable_name)` in the template parameter if needed.

In the above example, we saw the use of a default constructed empty `struct` type `mpi::blocking`. Likewise, we can perform a non-blocking send and receive. By doing a non-blocking send, we must promise not to use the data until after the data buffer is safe to use. In order to know when it is safe, the non-blocking send will return a `request` object that we can query the MPI runtime with.

For a non-blocking send of a value of 1.0 to process 1

```
double value{1.0};

auto const request = mpi::send(mpi::async{}, 1.0, 1);
```

(continues on next page)

(continued from previous page)

```
// Do some computation while we wait but don't touch value!

auto const status = mpi::wait(request);

// Phew now it's safe to overwrite value
```

These methods are generic enough to extend naturally to a `std::vector<T>` class, or any contiguous storage type that has a `.resize(entries)` method and a `T::value_type` type alias. Compile-time errors will be produced if the type is not a primitive type such as `int`, `double` etc.

1.4 Reduction operations

Some parallel algorithms require a reduction operation of a particular value. This is so that each process can agree on a particular value for a variable. For this we need to use some *reduction* operator. For example, an all reduce operation of a scalar by performing a sum operation is

```
auto const count = mpi::all_reduce(1, mpi::sum{});
```

which should be equal to the number of processes in the communicator `mpi::size()`. There are other instances of a reduction operation. This could be that a given process needs the minimum value of all the other process's rank

```
auto const process_one_min = mpi::reduce(mpi::rank(), mpi::min{}, 1);

// process_one_min is only meaningful for process 1 (which should be 0)
```

Here we have also introduced the wrapped function `mpi::rank()` which returns an *int* holding the rank of the current process.

No other reduction operations are currently supported.

1.5 Broadcasting

A broadcast operation takes the root process and sends the data out to all of the processors.

For example, if you are the root process you can send a value of *10* to all the other processes in the communicator

```
auto const broad_casted = mpi::broadcast(10);

// All processes now have the value of 10
```

1.6 License

mpi_api_wrapper is covered by the MIT open-source license and uses several other libraries, refer to each dependency for their license.

1.7 Contact

Questions? Bugs? Comments? Open up an issue on GitHub or submit a pull request.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search