WESTGRID

A Brief Introduction to the Boost MPI Library

Patrick Mann, Director of Operations

May 9, 2018

Summary

The C++ Boost library has considerably extended the capabilities of such basic C++ components as the Standard Library. Over the years much of this functionality has been integrated into new versions of the language. Of particular interest for the HPC community is the Boost MPI library. It provides an object-oriented interface to MPI allowing the programmer to concentrate on the parallel architecture rather than the implementation details of packing and sending data.

I'm still learning Boost MPI but I'll review some of the capabilities and give a few simple examples illustrating the basic functionality. It looks very nice to me, and indeed seems to be resulting in significantly cleaner code with essentially no performance penalty.

WEST GRID

Overview

1. C++ - the language, new versions, templates, issues
2. MPI - basics
3. Boost - short introduction
4. Boost/MPI - a few simple examples showing the approach and advantages.

C++ Working group ISO/IEC JTC 1/SC 22

- SC22 is a standardization subcommittee of the ISO/IEC Joint Technical Committee

1998	c++98	Based on Stroustrup's book	Gnu full support
2003	c++03	Fixed problems in c++98 Standard Template Library developed independently.	Gnu full support
2011	c++11	Core: multithreading, generic programming, uniform initialization and performance. Significant updates to the C++ Standard Library based on the Standard Template Library. Included performance requirements. Useful things like "`auto`" and lambda functions.	Gnu full support
2014	c++14	Bug fixes and minor extras.	Gnu full support
2017	c++17	Started to remove deprecated features (`trigraphs`, `auto_ptr`, ..) A lot of additional detail capabilities.	Gnu full support
2020	c++20	In progress	

C++ Philosophy

Bjarne Stroustrup: "*The C++ Programming Language*" is still a great text!

"*It's not what a language allows you to do, but what it encourages you to do.*"

- Flexible allowing programmers to choose their own style.
- Features should be useful in the real world.
- Useful features are more important than protecting against misuse.
- Performance is important
  - nothing between C++ and assembly
  - unused features should not impact performance
  - ...

Templates

Compile-time polymorphism: compiler uses a template to write code.

- Compiler optimizes resultant code.
- Modern compilers are tuned to minimize abstraction penalties.

Standard Template Library (1994) first library of generic algorithms and data structures for C++.

- Mostly superseded by the C++ Standard Library
  - Containers, iterators, localization, general, strings, i/o, threads, some numerics

Issues

Code bloat - lots of code generated, compilation can be slow. Concomitant increase in memory use.

- Less of a problem with fast modern compilers and hardware.

Error messages a pain!

Iterators are great, but as usual in c++ easy to mis-use them.

- ie) sorting can be very slow with poor choice of iterators.

Old C++ MPI Bindings

MPI-1 included C++ bindings: `MPI::Init(argc,argv)`

- Almost no-one used these
- The MPI forum didn't have enough C++ expertise so bugs had crept in.
- C++ bindings were 1:1 so no real advantage.
- Boost uses the old, original MPI C bindings.

And then MPI-3 came along and no-one felt like adding the C++ bindings.

- SO: **MPI-3 does not include the old C++ bindings.**

Boost Compiling & Linking

Just use the MPI scripts: `mpiCC` and `mpirun`

- of course MPI needs to be installed).

Need to link boost libraries both before and after the object file.

- Anyone else having this problem? (Using gnu c++ and OpenMPI on a Mint linux box)

```
CPPFLAGS = -std=c++11
CXX = mpiCC
LDFLAGS = -lboost_mpi -lboost_serialization

%: %.cpp
    $(CXX) $(CPPFLAGS) $(LDFLAGS) $@.cpp $(LDFLAGS) -o $@
```

```
mpirun -np 16 executable_name
```

Receive from all

```
#include <iostream>
#include <boost/mpi.hpp>                        // standard naming
namespace mpi = boost::mpi;                     // useful
using namespace std;

int main()
{
    mpi::environment env {};
    mpi::communicator world;
    int rank = world.rank();
    int tag = 31;                               // always tag the messages
    int master = 0;
    if( rank == master ){
        int i;
        world.recv( mpi::any_source, tag, i ); // receive whichever comes in first
        cout << "received from " << i << '\n';
    } else {
        world.send( master, tag, rank );        // send rank (integer) to master
    }
}
```

Classic MPI send/receive

```
if( rank == sender ){
    char* msg = (char *) "Message from sender";
    int n_msg = strlen(msg) + 1;

    MPI_Send( &n_msg, 1, MPI_INT, receiver, tag, MPI_COMM_WORLD );
    MPI_Send( msg, n_msg, MPI_CHAR, receiver, tag, MPI_COMM_WORLD );
} else if( rank == receiver ){
    int n_msg;

    MPI_Recv( &n_msg, 1, MPI_INT, sender, tag, MPI_COMM_WORLD, &status );
    char* msg = new char[n_msg+1];
    MPI_Recv( msg, n_msg, MPI_CHAR, sender, tag, MPI_COMM_WORLD, &status );
    msg[n] = '\0';

    cerr << "send1: " << rank << "(receiver): received message \""
         << msg << "\"\n";
    Delete[] msg;              // remember to delete temp variables
}
```

As usual need to package up the data, send a byte stream, and then un-package.

Careful with memory management, null terminator, …

Very buggy approach, and hard to debug.

Boost Send String

Boost MPI handles classes/objects: does it's own packing and unpacking.

```
#include <boost/mpi.hpp>
namespace mpi = boost::mpi;

int main( int argc, char *argv[] )
{
    mpi::environment env{argc, argv};
    mpi::communicator world;
    int tag = 10;                              // Usual MPI - tag any message
    if (world.rank() == 0)                     // receiver
    {
        std::string s;
        world.recv(mpi::any_source, tag, s); // receive string from anywhere
        std::cout << s << '\n';
    } else {
        std::string s = "Hello, world!";       // sender
        world.send(0, tag, s);                 // send string s to process 0
    }
}
```

Classic MPI: Class send/recv I

```
class TestClass
{
private:
  int i;
  double a;
  string desc;

public:
  TestClass();
  TestClass( string serial_string ){ …. }  // constructor from string
  ~TestClass(){}

  string Serialize(){
    const char DELIMITER = ' ';
    stringstream s;
    s << i << DELIMITER << a << DELIMITER << desc;
    return s.str();
  }

  friend ostream& operator << ( ostream&, TestClass& );
};
```

Need some sort of serialize for every class.

And a constructor from the serialized data.

Class send/recv II

```
if( rank == sender ){
    TestClass tclass;

    string serialized = tclass.Serialize();            // Serialize the class
    const char* tclass_serial = serialized.data();     // Switch to char stream

    int n = strlen( tclass_serial );                   // As usual send length and then data
    MPI_Send( &n, 1, MPI_INT, receiver, tag, MPI_COMM_WORLD );
    MPI_Send( tclass_serial, n, MPI_CHAR, receiver, tag, MPI_COMM_WORLD );

} else if( rank == receiver ){
    int n;
    MPI_Recv( &n, 1, MPI_INT, sender, tag, MPI_COMM_WORLD, &status );
    char* msg = new char[n+1];
    MPI_Recv( msg, n, MPI_CHAR, sender, tag, MPI_COMM_WORLD, &status );
    msg[n] = '\0';                                     // careful with tmp variables
    TestClass tclass_received( msg );                  // load back into class instance
    delete[] msg;
    ...
}
```

Boost Class Send/Receive

```
#include <boost/serialization.hpp>

class TestClass
{
private:
  int i; double a; string desc;

public:
  TestClass();
  ...

  friend class boost::serialization::access; // serialization needs access to private data

  template<class Archive>                    // use boost serialization templates
    void serialize( Archive &ar, const unsigned int version ){
      ar & i;
      ar & a;
      ar & desc;
    }
  ...
};
```

Still need to serialize, but use the built-in Boost serialization templates.
No need to explicitly write formatters or constructors.

Class Send/Receive II

This one sends/receives a vector of the “TestClass” objects.

```
if( rank == sender ){
    std::vector<TestClass> tvector;
    tvector.push_back( TestClass(1,2.0,"hi") );
    tvector.push_back( TestClass(10,10.0, "hi again") );
    world.send( receiver, tag, tvector );

} else if( rank == receiver ){
    std::vector<TestClass> treceive;
    world.recv( sender, tag, treceive );    // pack/unpack, mem allocation, ..
    for( auto &tvalue: treceive ){          // I really like “auto”
        s << tvalue << '\n';
    }
}
```

- Really nice - pack/unpack handled by serialization library.
  - Memory management by constructor/destructors.
- And a little C++-11 (or -14) with “`auto`” and “`vector<..>`” container.

Gather/Scatter I

Elementary distributed sum of vector:

- first chunk the vector
- Then scatter chunks to processors for summing.

```
vector<vector<double>> chunk_vec;
if( rank == 0 ){
     cout << "vec_norm " << rank << ": master scattering chunks with stride="
          << stride << " n_processors=" << n_processors << endl;

     for( int process=0; process<n_processors; ++process ){
          int istart = process*stride;
          int iend = istart + stride;
          vector<double> chunk;
          for( int i=istart; i<iend; ++i ){
               chunk.push_back(vec[i]);          // fill the chunk
          }
          chunk_vec.push_back(chunk);            // put the chunk in the chunk vector
     }
}
```

Gather/Scatter Chunks II

```
int scatter_process = 0;
vector<double> received_vec;
// Scatter all the chunk_vec's from scatter_process to all other processes.
mpi::scatter( world, chunk_vec, received_vec, scatter_process );

double sum = abs_sum( received_vec );  // every process sums it's chunk

if( rank == scatter_process ){     // send my sum back, and receive all sums
    vector<double> chunk_sums;
    mpi::gather( world, sum, chunk_sums, scatter_process );
    return vec_sum( chunk_sums );  // sum the chunk sums (no parallelization)

} else {                           // other processes send sums back
    mpi::gather( world, sum, scatter_process );
}
```

Boost MPI Functionality

Pretty complete!

- Non-blocking/asynchronous
  - `irecv(..), isend(..), wait_all(..), test(..)`
- Status
- `Reduce(..)`: takes a function to analyze data
  - All_reduce
- Broadcast
- Multiple communicators and processor groups.

Timing and Performance

I have not run any direct comparisons! TODO.

- Literature suggests minimal performance hits from Boost.
- Compilers are optimized from templates.

Boost Docs

- https://www.boost.org/ (the source!)
- https://theboostcpplibraries.com/ (nice textbook)
- https://theboostcpplibraries.com/boost.mpi
- https://github.com/WestGrid/boostWebinar (codes & Makefile for the examples)

And of course tons of stuff out on the web.

Useful Boost libraries:

- boost.Serialization (and string serialization)
- boost.Timer, boost.DateTime
- boost.Log (file out, standard out, severity, …)
- boost.ProgramOptions (options from command-line and file)

Interesting Boost Libraries
(I haven't tried them):

- ODEint (ODE solvers)
- Python (interface from C++)
- Random
- RegEx
- Sort
- uBLAS (matrix, vector, BLAS, dense and sparse, ..)
- Compute (OpenCL, GPUs)

Any experiences with these or others?