Boost.MPI

A C++ API for MPI

Boost.MPI

- A few things that I mentioned in HPCSE are a real hassle
  - Packing complex data structures into a buffer
  - Creating custom MPI datatypes
  - Creating and using custom reduction functions

- The solution is Boost.MPI
  - C++ API for MPI going far beyond the C API
  - Developed as open source library by BoostPro Computing for credit risk simulations at ZKB

Getting started with MPI

- Recall our first program

```
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);

  int rank;
  int size;

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  std::cout << "I am rank " << rank <<
               " of " << size << ".\n";

  MPI_Finalize();

  return 0;
}
```

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

int main(int argc, char** argv)
{
  mpi::environment env(argc, argv);
  mpi::communicator world;
  std::cout << "I am " << world.rank()
  << " of " << world.size()  << ".\n";
  return 0;
}
```

MPI Communicators

- Boost.MPI allows to create MPI communicators more easily than in MPI

```
mpi::communicator comm1(world,mpi::comm_duplicate);
mpi::communicator comm2 = world.split(color,key);
```

A first example of message passing ...

- A parallel “Hello World” program
  - rank 1 sends a string with tag 42 to rank 0
  - rank 0 receives a string with tag 42 from rank 1 and prints it

```
int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);
  int num;

  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  if(num==0) { // "master"
    MPI_Status status;
    char txt[100];
    MPI_Recv(txt, 100, MPI_CHAR,
             1, 42, MPI_COMM_WORLD, &status);
    std::cout << txt << "\n";
  }
  else { // "worker"
    std::string text="Hello world!";
    MPI_Send(const_cast<char*>(text.c_str()), text.size()+1, MPI_CHAR,
             0, 42, MPI_COMM_WORLD);
  }

  MPI_Finalize();

  return 0;
}
```

… simplified using Boost.MPI

- A parallel “Hello World” program
  - rank 1 sends a string with tag 42 to rank 0
  - rank 0 receives a string with tag 42 from rank 1 and prints it

```
int main(int argc, char** argv)
{
  mpi::environment env(argc, argv);
  mpi::communicator world;

  if (world.rank() == 0) {
    std::string msg;
    world.recv(1, 42, msg);
    std::cout << msg << std::endl;
  } else
    world.send(0, 42, std::string("Hello, world!"));
}
```

Custom datatypes

- Boost.MPI allows MPI messages to contain
  - Custom datatypes
  - Variable-sized data structures
  - Pointers, linked lists, trees, ...
  - Pointers to derived classes, ...

- This is made possible by building on Boost.Serialization
  - Include serialization headers
  - Link to serialization library
  - Learn about the serialization library!

Probing for messages

- Using the C API:

```
MPI_Status status;
int count;

MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, &status);
std::cout << "A message is waiting from " << status->MPI_SOURCE
          << "with tag "                  << status->MPI_TAG;
```

- and simplified in Boost.MPI

```
mpi::status stat = world.probe(); // waits for any message from anywhere

std::cout << "A message is waiting from " << stat.source()
          << "with tag "                  << stat.tag();
```

- Looping and waiting is also easy

```
boost::optional<status> stat;
do {
  std::cout << “Still checking\n”;
  stat= world.iprobe(1,42);
} while (!stat);
```

Overlaying communication and computation

- Exchange ghost cells while we compute the interior

```
for (int t=0; t<iterations; ++t) {
  // first start the communications

  if (rank % 2 == 0) {
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[3]);
  }
  else {
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[3]);
  }

  // do calculation  of the interior
  for (int i=2; i<local_N-2;++i)
    newdensity[i] = density[i] + coefficient * (density[i+1]+density[i-1]-2.*density[i]);

  // wait for the ghost cells to arrive
  MPI_Waitall(4,reqs,status);

  // do the boundaries
  newdensity[1] = density[1] + coefficient * (density[2]+density[0]-2.*density[1]);
  newdensity[local_N-2] = density[local_N-2] + coefficient * (
                            density[local_N-1]+density[local_N-3]-2.*density[local_N]);

  // and swap
  density.swap(newdensity);
}
```

… simplified by Boost.MPI

- Exchange ghost cells while we compute the interior

```
for (int t=0; t<iterations; ++t) {
  // first start the communications

  if (rank % 2 == 0) {
    reqs[0] = world.isend(left,0,density[1]);
    reqs[1] = world.irecv(left,0,density[0]);
    reqs[2] = world.isend(right,0,density[local_N-2]);
    reqs[3] = world.irecv(right,0,density[local_N-1]);
  }
  else {
    reqs[0] = world.irecv(right,0,density[local_N-1]);
    reqs[1] = world.isend(right,0,density[local_N-2]);
    reqs[2] = world.irecv(left,0,density[0]);
    reqs[3] = world.isend(left,0,density[1]);
  }

  // do calculation  of the interior
  for (int i=2; i<local_N-2;++i)
    newdensity[i] = density[i] + coefficient * (density[i+1]+density[i-1]-2.*density[i]);

  // wait for the ghost cells to arrive
  mpi::wait_all(reqs,reqs+4);

  // do the boundaries
  newdensity[1] = density[1] + coefficient * (density[2]+density[0]-2.*density[1]);
  newdensity[local_N-2] = density[local_N-2] + coefficient * (
                          density[local_N-1]+density[local_N-3]-2.*density[local_N]);

  // and swap
  density.swap(newdensity);
}
```

Reductions in Boost.MPI

- A nicer syntax also for reductions

```
MPI_Allreduce(rank == 0 ? MPI_IN_PLACE : &sum, &sum, MPI_LONG_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
sum= boost::mpi::all_reduce(world, sum, std::plus<double>());
```

- Boost.MPI allows arbitrary function objects for reductions and automatically optimizes for built-in MPI operations.

Packing and unpacking into a buffer

- Remember how we packed and unpacked

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration
};

int main(int argc, char** argv)
{
  ...

  // now create a buffer and pack the values.
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD,&size_double);
MPI_Pack_size(1, MPI_INT,    MPI_COMM_WORLD,&size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];

// now pack the values into the buffer on the master
if (rank==0) {
  int pos=0;
  MPI_Pack(&p.a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&p.b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&p.nsteps, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
}

// then broadcast the buffer
MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);

// and unpack on the receiving side
int pos=0;
MPI_Unpack(buffer, buffer_size, &pos, &p.a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &p.b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &p.nsteps, 1, MPI_INT, MPI_COMM_WORLD);

// and delete the buffer
delete[] buffer;
```

... vastly simplified by Boost.MPI

- Remember how we packed and unpacked

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration

  template <class Archive>
  void serialize(Archive& ar, unsigned int version)
  {
    ar & a & b & nsteps;
  }
};

int main(int argc, char** argv)
{

  ...

  mpi::broadcast(world,p,0);

  ...
}
```

Using MPI datatypes

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration
};

int main(int argc, char** argv)
{
  ...

  // describe this struct through sizes, offsets and types
  // the safe way getting addresses

  MPI_Aint p_lb, p_a, p_nsteps, p_ub;
  MPI_Get_address(&p,       &p_lb);      // start of the struct is the lower bound
  MPI_Get_address(&p.a,     &p_a);       // address of the first double
  MPI_Get_address(&p.nsteps, &p_nsteps); // address of the integter
  MPI_Get_address(&p+1,     &p_ub);      // start of the next struct is the upper
bound

  int          blocklens[] =  {0, 2, 1, 0};
  MPI_Datatype types[]     =  {MPI_LB, MPI_DOUBLE, MPI_INT, MPI_UB};
  MPI_Aint     offsets[]   =  {0, p_a-p_lb, p_nsteps-p_lb, p_ub-p_lb};

  MPI_Datatype parms_t;
  MPI_Type_create_struct(4, blocklens, offsets, types,&parms_t);
  MPI_Type_commit(&parms_t);

  // broadcast the parms now using our type
  MPI_Bcast(&p, 1, parms_t, 0, MPI_COMM_WORLD);

  ...
}
```

... vastly simplified by Boost.MPI

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration

  template <class Archive>
  void serialize(Archive& ar, unsigned int version)
  {
    ar & a & b & nsteps;
  }
};

BOOST_IS_MPI_DATATYPE(parms)

int main(int argc, char** argv)
{

  ...

  mpi::broadcast(world,p,0);

  ...
}
```

Optimizing communication

- A common MPI usage pattern is sending and receiving updated contents of existing data structures
  - Exchange boundary layers when simulating differential equations using domain decomposition
  - Exchange updated particle positions / velocities in molecular dynamics simulations
- This needs to be fast and efficient
  - Do not allocate the data structures new
  - Avoid copying any data
  - Avoid any overhead of serialization or other libraries
- Solution: use MPI datatypes
- Elegant Boost.MPI solution
  - First send or broadcast the "**skeleton**" of the data structure, to create them on the receiving side
  - Then send the "**content**", automagically creating an MPI data type

Creating MPI Datatypes: skeleton & content

- The master build a list, sends the skeleton and later updates the contents

```
// Generate the list and broadcast its structure
std::list<int> l(list_len);
broadcast(world, mpi::skeleton(l), 0);

// Generate content several times and broadcast out that content
mpi::content c = mpi::get_content(l);
for (int i = 0; i < iterations; ++i) {
  // Generate new random values
  std::generate(l.begin(), l.end(), &random);

  // Broadcast the new content of l
  broadcast(world, c, 0);
}
```

- The workers receive the skeleton, and then updates to the contents

```
// Receive the skeleton and build up our own list
std::list<int> l;
broadcast(world, mpi::skeleton(l), 0);

// Generate content several times and broadcast out that content
mpi::content c = mpi::get_content(l);
for (int i = 0; i < iterations; ++i) {
  // receive the content
  broadcast(world, c, 0);

  // Compute some property of the data.

  ...
}
```