GridPACK™ Example: Hello World

Bruce Palmer

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by Battelle *Since 1965*

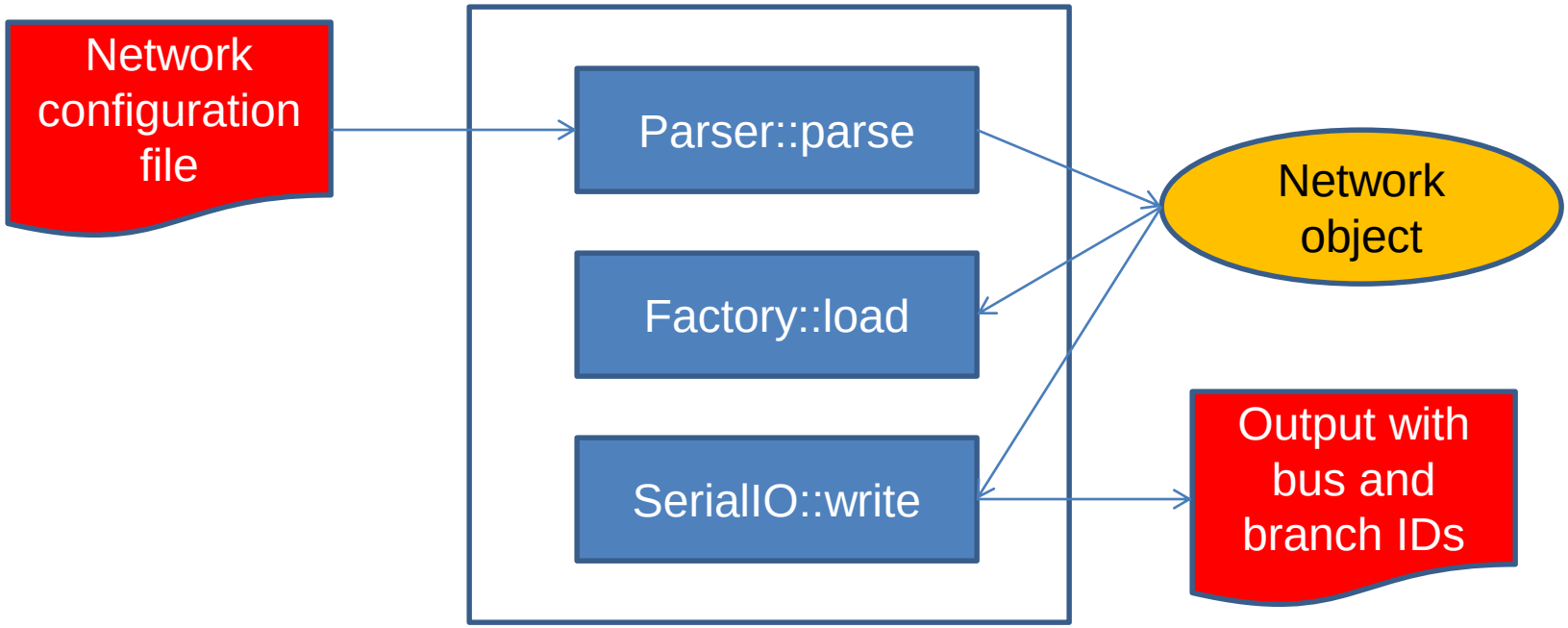
Objective

This example is designed to illustrate some of the basic properties of the GridPACK™ framework by creating a simple network and extracting some information from it. The example will show users how to

- Create a network by parsing an external configuration file
- Create simple bus and branch objects and store information from the input file in them
- Write the information stored in the buses and branches to standard output

The remainder of the tutorial will review this example in exhaustive detail to illustrate some of the features of GridPACK™.

“Hello world”


Network configuration file
Parser::parse
Factory::load
SerialIO::write
Network object
Output with bus and branch IDs

Step 1: Creating the "main" program

- The “main” program is a standard C++ construct that is used to drive the application
- GridPACK™ uses “main” to initialize the communication libraries and initialize the math libraries
- The actual application is treated as a separate class. This helps the application exit cleanly and prevents crashes at program closeout caused by library functions in the destructor being invoked after the libraries have called their terminate functions

“Hello world” main program

```
#include “mpi.h”
#include <ga.h>
#include <macdecls.h>
#include “gridpack/include/gridpack.hpp”
#include “hw_app.hpp”

// Calling program for the hello_world applications
main(int argc, char **argv){
  // Initialize communication and math libraries
  int ierr = MPI_Init(&argc, &argv);
  gridpack::math::Initialize();
  GA_Initialize();
  int stack = 200000, heap = 200000;
  MA_init(C_DBL, stack, heap);

  // Initialize and execute GridPACK application
  gridpack::hello_world::HWApp app;
  app.execute(argc, argv);

  // Terminate communication and math libraries
  GA_Terminate();
  gridpack::math::Finalize();
  ierr = MPI_Finalize();
}
```

Header files

Initialize libraries used in the remainder of the application

Run application

Clean up libraries

More on “main”

- The header files include all files needed to initialize the communication files (mpi.h, ga.h, macdecls.h), the files needed by the gridpack modules (gridpack/include/gridpack.hpp) and the header file for the application (hw_app.hpp)
- The communication libraries and math libraries need to be initialized by calling their initialize functions. The MA_Init function allocates internal memory used by the GA library
- The application is created by instantiating the object “app” and the application is run by calling the “execute” function
- After execution, the libraries are cleaned up by calling their terminate functions. Note that the ordering of the initialize and terminate calls is important

Step 2: Creating the application class

- The application class is designed to manage the application at a high level
- It controls overall program flow and directs information to other parts of the program

The application header file

```
#ifndef _hw_app_h_
#define _hw_app_h_

namespace gridpack {
namespace hello_world {

// Calling program for hello world application
class HWApp
{
 public:

    // Basic constructor
    HWApp(void);

    // Basic destructor
    ~HWApp(void);

    // execute application
    void execute(int argc, char** argv);

  private:
};
} // hello_world
} // gridpack
#endif
```

Preprocessor declarations

Name space declarations

Basic constructor and destructor

Execute application method

More on application header

- All GridPACK™ modules belong to the “gridpack” namespace as do all applications included in the GridPACK™ source tree
- The “hello_world” namespace is used only for this specific application. Users may wish to create their own namespaces for applications built outside the GridPACK™ source tree
- The only important method in this class is “execute”. The constructors and destructors are basically the default methods
- Arguments included in the runtime invocation can be passed to the execute method via the variables argc and argv

The application source code file

- Include all header files required by the method implementations
- Implement methods defined in the header file
- Use default constructor and destructor
- Many GridPACK™ modules make use of Boost library functionality, particularly the boost::shared_ptr<T> template class. The header file for this class needs to be included in most applications.

Basic application functionality

```
#include <iostream>
#include "boost/smart_ptr/shared_ptr.hpp"
#include "hw_app.hpp"
#include "hw_factory.hpp"
#include "gridpack/parser/PTI23_parser.hpp"
#include "gridpack/serial_io/serial_io.hpp"

// Basic constructor
gridpack::hello_world::HWApp::HWApp(void)
{
}

// Basic destructor
gridpack::hello_world::HWApp::~HWApp(void)
{
}
```

The “execute” method implementation

```
1  // Execute application
2  void gridpack::hello_world::HWApp::execute(int argc, char** argv)
3  {
4    // load input file
5    gridpack::parallel::Communicator world;
6    boost::shared_ptr<HWNetwork> network(new HWNetwork(world));
7    // read configuration file
8    std::string filename = "10x10.raw";
9    // Read in external PTI file with network configuration
10   gridpack::parser::PTI23_parser<HWNetwork> parser(network);
11   parser.parse(filename.c_str());
12   // partition network
13   network->partition();
14   // create factory
15   gridpack::hello_world::HWFactory factory(network);
16   factory.load();
17   // Create serial IO objects to export data from buses and branches
18   gridpack::serial_io::SerialBusIO<HWNetwork> busIO(128,network);
19   busIO.header("\nMessage from buses\n");
20   busIO.write();
21   gridpack::serial_io::SerialBranchIO<HWNetwork> branchIO(128,network);
22   branchIO.header("\nMessage from branches\n");
23   branchIO.write();
24 }
```

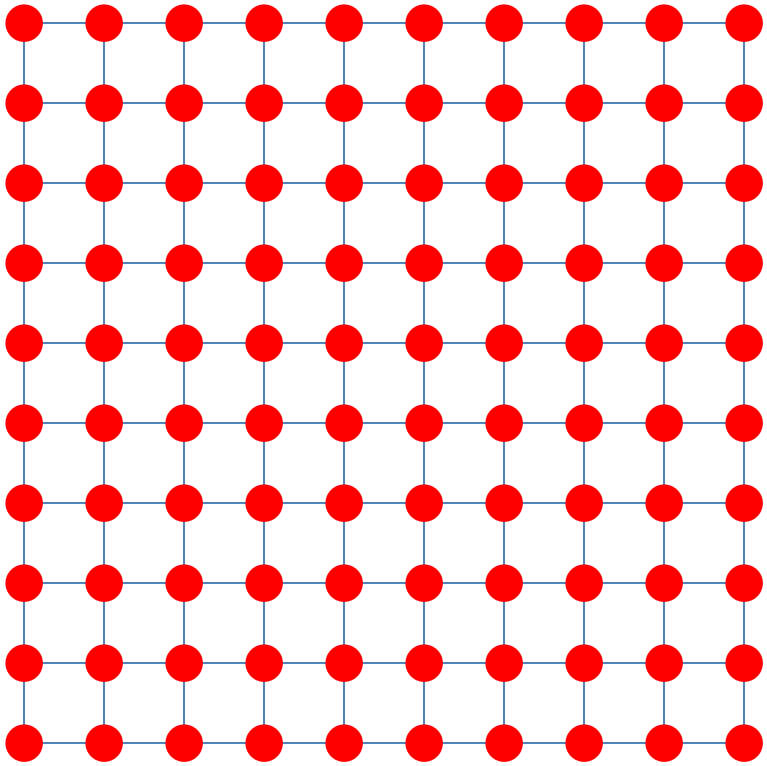
The “execute” method

- (line 5) the Communicator object represents a group of processors within the application. The default communicator used here includes all processors in the application. Processes within a communicator can communicate with each other and collective operations on the communicator must include all processes associated with the communicator
- (line 6) a new network object is instantiated on the “world” communicator. This means that the network will be distributed over all processors in the system. The network contains no buses or branches at this time. The HWNetwork is defined elsewhere

The “execute” method (cont.)

- (line 10) create a PTI23_parser object. This object can be used to import a PTI version 23 network configuration file and create the buses and branches described in the file. These objects will be associated with the network created in line 6
- (line 11) import the file “10x10.raw” to create a network corresponding to a 10x10 square grid of buses with branches connecting nearest neighbors on the grid. At this point all buses and branches exist but they are not optimally distributed on the network

10x10.raw



The “execute” method (cont.)

- (line 13) call the network partition function so that buses and branches are optimally distributed among processors. This means that buses are distributed so that most branches represent connections between buses on the same processor and branches that connect buses on different processors are minimized. Ghost buses and branches are also added to the network at this time
- (lines 15 and 16) create a HWFactory object and use it to initialize the buses and branches in the system using parameters in the 10x10.raw file

The “execute” method (cont.)

- (lines 18-23) create SerialIO objects to export data in buses and branches to standard output. These objects rely on methods defined in the bus and branch classes. The SerialBusIO object exports data from buses and the SerialBranchIO object exports data from branches. The maximum length of a line of text that can exported by either a bus or branch is 128 characters and is specified when the SerialIO object is created. The “header” method is a convenience function that can be used to export text from only process 0. The “write” method exports data from buses and branches

Step 3: Defining the network component classes

- The “hello world” bus and branch classes inherit from the BaseBusComponent and BaseBranchComponent classes respectively.
- The load function from the base classes is overwritten by the application classes. This uses parameters in the DataCollection objects associated with each bus and branch to initialize that application components
- The serialWrite method in the base classes is overwritten so that the write methods of the SerialIO objects will actually produce some output if called

Hello World Component Header File

```
#include "boost/smart_ptr/shared_ptr.hpp"
#include "gridpack/component/base_component.hpp"
#include "gridpack/component/data_collection.hpp"
#include "gridpack/network/base_network.hpp"

namespace gridpack {
namespace hello_world {
       :
typedef network::BaseNetwork<HWBus, HWBranch > HWNetwork;
```

- Include header files for base classes and DataCollection class
- Declare namespaces
- Define HWNetwork

Hello World Bus Component Class

```
class HWBus
  : public gridpack::component::BaseBusComponent {
 public:
   // Simple constructor
   HWBus(void);
   // Simple destructor
   ~HWBus(void);
   // Load values stored in DataCollection object into HWBus object
   void load(const boost::shared_ptr<gridpack::component::DataCollection> &data);
   // Write output from buses to standard out
   bool serialWrite(char *string, const int bufsize, const char *signal = NULL);
 private:
   int p_original_idx;
   friend class boost::serialization::access;
   template<class Archive>
   void serialize(Archive & ar, const unsigned int version)
   {
     ar & boost::serialization::base_object<gridpack::component::BaseBusComponent>(*this)
        & p_original_idx;
   }
};
```

Serialization method

- The serialization method is used within boost classes to take a class and convert it into a string of bits and then back into a class instance in a specific state. This is very useful for moving classes from one processor to another
- Users only need to worry about creating a serialization method that includes all internal variables for the class. Note that pointers are not recursively serialized and any data associated with a class instance via a pointer will not be moved with the rest of the class.
- The serialization method for other component implementations can be created following the template in the previous slide

Serialization method for Hello World Bus Component Class

```
1  void serialize(Archive & ar, const unsigned int version)
2  {
3       ar & boost::serialization::base_object
4              <gridpack::component::BaseBusComponent>(*this)
5          & p_original_idx;
6  }
```

- (line 1) **`serialize`** function declaration that passes the Archive variable **`ar`** to the implementation
- (lines 3-4) the **`ar`** variable is augmented by the string of data coming from the parent of the component class
- (line 5) the **`ar`** variable is augmented by the local data for this class (**`p_original_index`**)
- Additional local variables would require additional declarations like line 5

Bus load method

- The only thing the “hello world” bus class does is cache the original bus index from the input file and then print it out in the serialWrite method
- The bus index is stored as a local variable using the load method, which gets the original bus index by extracting it from the DataCollection object associated with the bus

```
void gridpack::hello_world::HWBus::load(const
        boost::shared_ptr<gridpack::component::DataCollection> &data)
{
   data->getValue(BUS_NUMBER,&p_original_idx);
}
```

BUS_NUMBER is a predefined macro in dictionary.hpp corresponding to the original bus index, which is stored in the internal variable **`p_original_idx`**. dictionary.hpp is located in the $GRIDPACK/src/parser directory

Bus serialWrite method

- Enable each bus to write a string to output describing some current property of the bus

```
bool gridpack::hello_world::HWBus::serialWrite(char *string,
        const int bufsize, const char *signal)
{
  sprintf(string,"Hello world from bus %d\n",p_original_idx);
  return true;
}
```

- Information in **`bufsize`** and **`signal`** is not used in this example
- Print out the original bus ID stored in the internal variable **`p_original_idx`**

Hello World Branch Component Class

- Similar to bus component class

```
class HWBranch
: public gridpack::component::BaseBranchComponent {
 public:
    //  Simple constructor
    HWBranch(void);
    //  Simple destructor
    ~HWBranch(void);
    //  Load values stored in DataCollection object into HWBranch object
    void load(const boost::shared_ptr<gridpack::component::DataCollection> &data);
    // Write output from branches to standard out
    bool serialWrite(char *string, const int bufsize, const char *signal = NULL);
  private:
    int p_original_idx1;
    int p_original_idx2;
    friend class boost::serialization::access;
    template<class Archive>
      void serialize(Archive & ar, const unsigned int version)
      {
        ar & boost::serialization::base_object<
            gridpack::component::BaseBranchComponent>(*this)
          & p_original_idx1 & p_original_idx2;
      }
};
```

Branch load method

- Store IDs of buses at either end of the branch

```
void gridpack::hello_world::HWBranch::load(
   const boost::shared_ptr<gridpack::component::DataCollection> &data)
{
  data->getValue(BRANCH_FROMBUS,&p_original_idx1);
  data->getValue(BRANCH_TOBUS,&p_original_idx2);
}
```

- BRANCH_FROMBUS and BRANCH_TOBUS are macros defined in the dictionary.hpp file located in the $GRIDPACK/src/parser directory

Branch serialWrite method

- Enable each branch to write a string to output describing some current property of the branch

```
bool gridpack::hello_world::HWBranch::serialWrite(char *string,
        const int bufsize, const char *signal)
{
  sprintf(string,"Hello world from the branch connecting bus %d to bus %d\n",
      p_original_idx1, p_original_idx2);
  return true;
}
```

- Again, information in **`bufsize`** and **`signal`** is not used
- Print out the original bus IDs for the buses at either end of the string. These are stored in the internal variables **`p_original_idx1`**, **`p_original_idx2`**

Hello World Factory Class Header File

```
#ifndef _hw_factory_h_
#define _hw_factory_h_
#include "boost/smart_ptr/shared_ptr.hpp"
#include "gridpack/factory/base_factory.hpp"
#include "hw_components.hpp"
namespace gridpack {
namespace hello_world {
class HWFactory
  : public gridpack::factory::BaseFactory<HWNetwork> {
  public:
    // Constructor
    HWFactory(boost::shared_ptr<HWNetwork> network)
      : gridpack::factory::BaseFactory<HWNetwork>(network){
      p_network = network;
    }
    // Destructor
    ~HWFactory() {}
  private:
    NetworkPtr p_network;
};
} // hello_world
} // gridpack
#endif
```

Hello World Factory

- Only functionality from the BaseFactory class is needed
- HWNetwork defined in hw_components.hpp
- NetworkPtr is defined in base_factory.hpp. The network type is determined when HWFactory is defined as subclassing BaseFactory<HWNetwork>
- Only the factory “load” method is used in the application. This transfers data from the DataCollection objects, which are initialized by the parser, to the bus and branch classes