GridPACK™ Framework for Developing Power Grid Applications for HPC Platforms

Bruce Palmer, Bill Perkins, Kevin Glass, Yousu Chen, Shuangshuang Jin, Ruisheng Diao, Mark Rice, David Callahan, Steve Elbert, Henry Huang

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by Battelle *Since 1965*

Objective

Develop a framework to support the rapid development of power grid software applications on HPC platforms

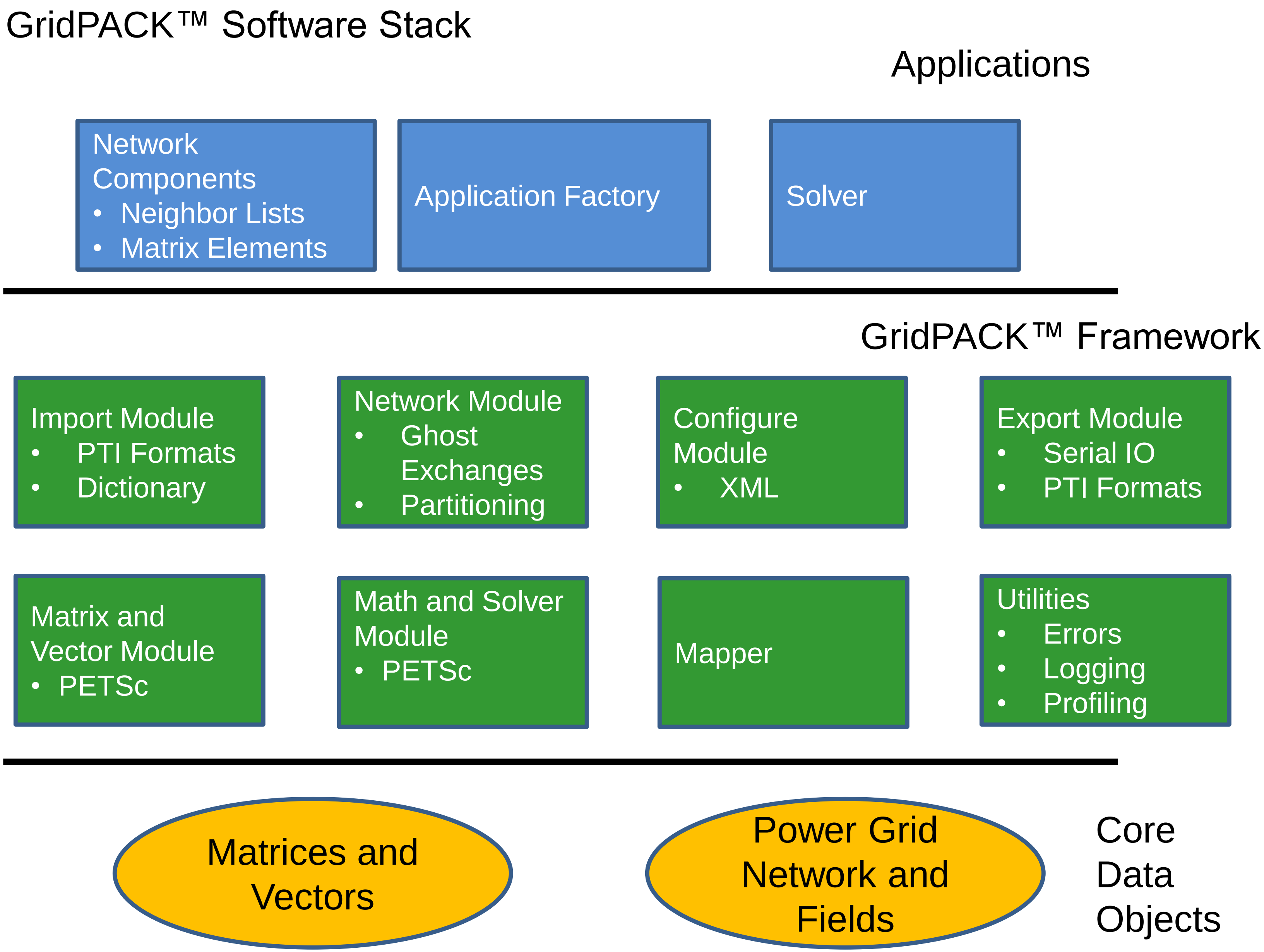
- Extend the penetration of HPC in power grid modeling
- Provide high level abstractions for often used motifs in power grid applications
- Reduce the amount of explicit communication that must be handled by developers
- Allow power grid application developers to focus on physics and algorithms and not on parallel computing

Approach

Develop software modules that encapsulate commonly used functionality in HPC power grid applications

- Setup and distribution of power grid networks
- Input/Output
- Mapping from grid to distributed matrices
- Parallel solvers
- Incorporate advanced parallel libraries whenever possible
  - PETSc, ParMETIS

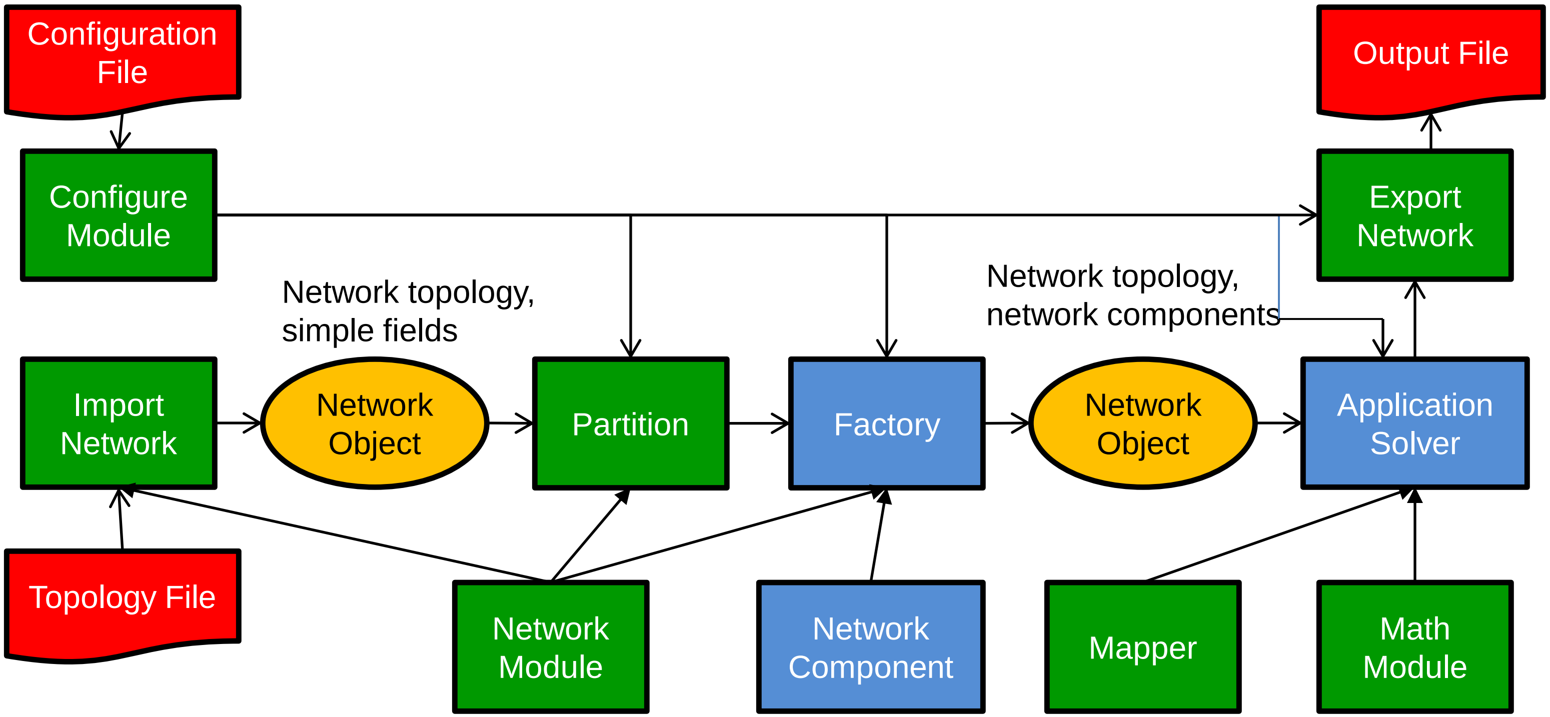
GridPACK™ is currently available as a set of C++ software modules and classes


GridPACK™ Software Stack
Applications
Network Components
• Neighbor Lists
• Matrix Elements
Application Factory
Solver
GridPACK™ Framework
Import Module
• PTI Formats
• Dictionary
Network Module
• Ghost Exchanges
• Partitioning
Configure Module
• XML
Export Module
• Serial IO
• PTI Formats
Matrix and Vector Module
• PETSc
Math and Solver Module
• PETSc
Mapper
Utilities
• Errors
• Logging
• Profiling
Matrices and Vectors
Power Grid Network and Fields
Core Data Objects

Major GridPACK™ Modules

- Network: Manages the topology, neighbor lists, parallel distribution and indexing. Acts as a container for bus and branch components
- Bus and Branch components: define the behavior and properties of buses and branches in network. These components also define the matrices that can be generated as part of the simulation
- Factory: Manages interactions between network and the components

Application Flow Diagram



Building the GridPACK™ Library

- GridPACK™ requires the following libraries
  - MPI (Message Passing Interface): This is the standard parallel communication library and is available on most parallel platforms. It can be downloaded from several sources and built on most Linux workstations
  - Global Arrays: This can be downloaded from http://hpc.pnl.gov/globalarrays/ and built on most Linux platforms.
  - PETSc: This library supports most of the matrix and solver capability in GridPACK™. It can be downloaded and built from http://www.mcs.anl.gov/petsc/.
  - Boost: The Boost libary supplies several extensions to C++ that are used throughout GridPACK™. Boost can be downloaded from http://www.boost.org/.
  - Parmetis: This library supplies the partitioning function in PETSc. It can usually be build within PETSc but if it has not, then it can be downloaded from http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview and built separately

MPI

- MPI is ubiquitous on parallel platforms and is usually already built. Linux workstations probably don't have it installed but it can be downloaded from several sources and built fairly easily
  - MPICH: available from http://www.mpich.org/
  - MVAPICH/MVAPICH2: available from http://mvapich.cse.ohio-state.edu/
  - OPENMPI: available from http://www.open-mpi.org/
- Each of these implementations has its own pluses and minuses, but any of them will be suitable for most GridPACK™ applications
- Follow the instruction for each package to build it

Global Arrays

- GA supplies the internal communication layer for several GridPACK™ modules
- GA can be built on top of MPI or it can use native ports on Infiniband and DMAPP (Cray) networks
- GA uses an autoconfig build to configure the library and then build it using make
  - Building GA on workstation using 2-sided MPI
  - ```
    configure --with-mpi-ts --enable-cxx \
      --disable-f77 --enable-i4 \
      --prefix=$(GA_HOME)
    ```
- Use different option for `--with-mpi-ts` on different platforms
    ```

PETSc

- PETSc is the most complicated library to build
- A typical configure line looks like

```
./configure \
  PETSC_ARCH=arch-linux2-complex-opt --with-prefix=./ \
  --with-mpi=1 --with-cc=mpicc --with-fc=mpif90 \
  --with-cxx=mpicxx --with-c++-support=1 --with-c-support=0 \
  --with-fortran=0 --with-scalar-type=complex \
  --with-fortran-kernels=generic --download-superlu_dist \
  --download-parmetis --download-metis \
  --download-f2cblaslapack=1 --with-clanguage=c++ \
  --with-shared-libraries=0 --with-dynamic-loading=0 \
  --with-x=0 --with-mpirun=mpirun --with-mpiexec=mpiexec \
  --with-debugging=0
```

PETSc (cont)

- PETSC_ARCH is a user-defined name that identifies this particular build
- After configuring, the PETSc build will lead you through the make process. Follow the commands coming from the build
- If Metis and Parmetis are built as part of the PETSc libraries then they do not need to be downloaded and built separately. This guarantees consistency between compilers and other environmental settings across the PETSc and Parmetis libraries.
- This example includes both Parmetis and SuperLU

Boost

- The Boost extensions to C++ are widely available
- The Boost libraries can be configured and built using the commands

```
echo "using mpi ;" > ~/user-config.jam
sh ./bootstrap.sh \
 --prefix="$prefix" \
 --without-icu \
 --with-toolset=gcc \
 --without-libraries=python,log
./b2 -a -d+2 link=static stage
./b2 -a -d+2 link=static install
rm ~/user-config.jam
```

- Some commands may differ depending on platform

ParMetis

- As mentioned above, it is better to build ParMetis (and Metis) as part of PETSc. However, it can be built standalone
- To get ParMetis to build with older GNU compilers, run the following command first
  - ```
    sed -i.org -e \
      's/-Wno-unused-but-set-variable//g' \
      metis/GKlib/GKlibSystem.cmake
    ```
- This removes some newer compiler options
    ```

ParMetis (cont)

- To build Metis, execute the following commands (starting in the top level directory)
  - `cd metis`
  - `make config prefix=/my/install/directory`
  - `make`
  - `make install`

ParMetis (cont)

- After building Metis, execute the following commands
  - `cd ..`
  - `make config cc=mpicc cxx=mpicxx \`
    `prefix=/my/install/directory`
  - `make`
  - `make install`
- Run some tests to check if build is okay
  - `cd Graphs`
  - `mpirun -np 4 ptest rotor.graph rotor.graph.xyz`
  - `mpirun -np 4 ptest rotor.graph`

Configure GridPACK™

- Create a directory that will contain the GridPACK™ build and cd into it
- Configure GridPACK™ using cmake (RHEL5 example)

```
cmake -Wno-dev \
 -D BOOST_ROOT:STRING=‘/top/level/boost/directory’ \
 -D PETSC_DIR:STRING=‘/top/level/petsc/directory' \
 -D PETSC_ARCH:STRING='arch-linux2-cxx-opt' \
 -D PARMETIS_DIR:STRING='/parmetis/lib/directory’ \
 -D GA_DIR:STRING=‘/top/level/ga/directory’ \
 -D MPI_CXX_COMPILER:STRING=‘/mpicxx/location’ \
 -D MPI_C_COMPILER:STRING=‘/mpicc/location’ \
 -D MPIEXEC:STRING=‘/mpiexec/location’ \
 -D CMAKE_BUILD_TYPE:STRING="Debug" \
 -D CMAKE_VERBOSE_MAKEFILE:BOOL=TRUE \
 ..
```

This points to GridPACK™ source directory

Configuring GridPACK™ (cont)

- The string corresponding to "PETSC_ARCH should match the PETSC_ARCH string used in configuring the PETSC library
- The options for CMAKE_BUILD_TYPE are
  - DEBUG (not optimized, built with –g)
  - RELEASE (optimized)
  - RELWITHDEBINFO (optimized but built with –g)
- After configuring, type "make" in the build directory to build all libraries and applications. Individual libraries and applications can be built by cd'ing into those directories and typing "make"

BaseNetwork Class

- Template class that can be created with arbitrary user-defined types for the buses and branches
  - `BaseNetwork<MyBus, MyBranch>`
- Implements partitioning of network between processors
  - Create highly connected sub-networks on each processor with minimal connections between processors
- Implements data exchanges between buses and branches on different processors
- Manages indexing of network components

Instantiate a Network

```
#include "gridpack/network/BaseNetwork.hpp"
#include "gridpack/applications/myapp/mycomponents.hpp"

typedef gridpack::network::BaseNetwork
                  <gridpack::myapp::MyBus,
                   gridpack::myapp::MyBranch> MyNetwork;

boost::shared_ptr<MyNetwork> network(new MyNetwork);

// Create a network object that has the application-specific
// bus and branch models associated with it. The network will
// also have DataCollection objects on each bus and branch.
// At this point, the network is just a container and has no
// topology or data
```

Parser Module

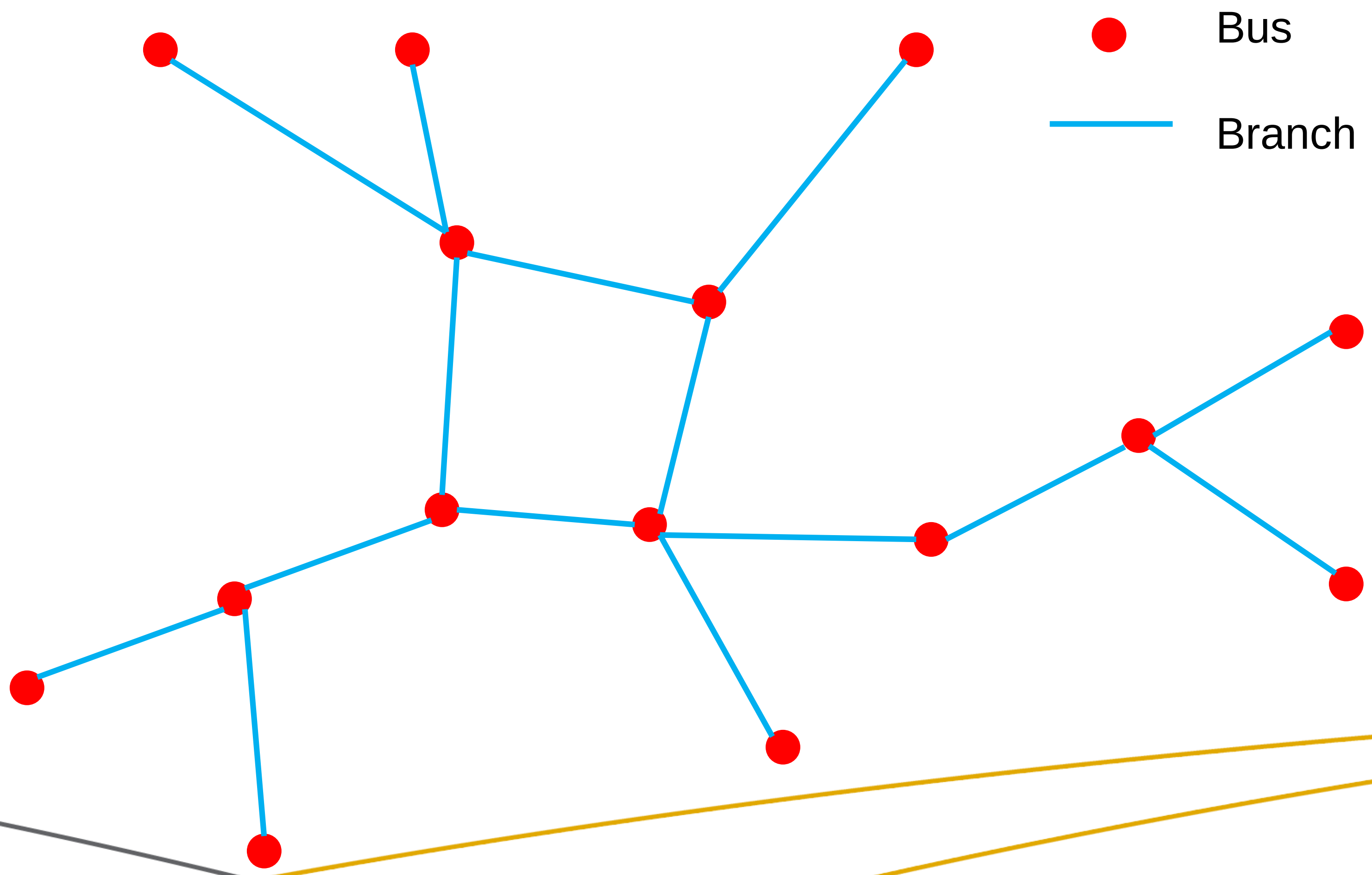
- Currently, only PTI version 23 format is supported.
- Work is under way to develop a parser based on more generic GOSS formats

```
// Constructor
PTI23_parser<MyNetwork>::PTI23_parser(
     boost::shared_ptr<MyNetwork> network);

// Import external network configuration file and generate
// network topology
void parse(const std::string &filename)
```

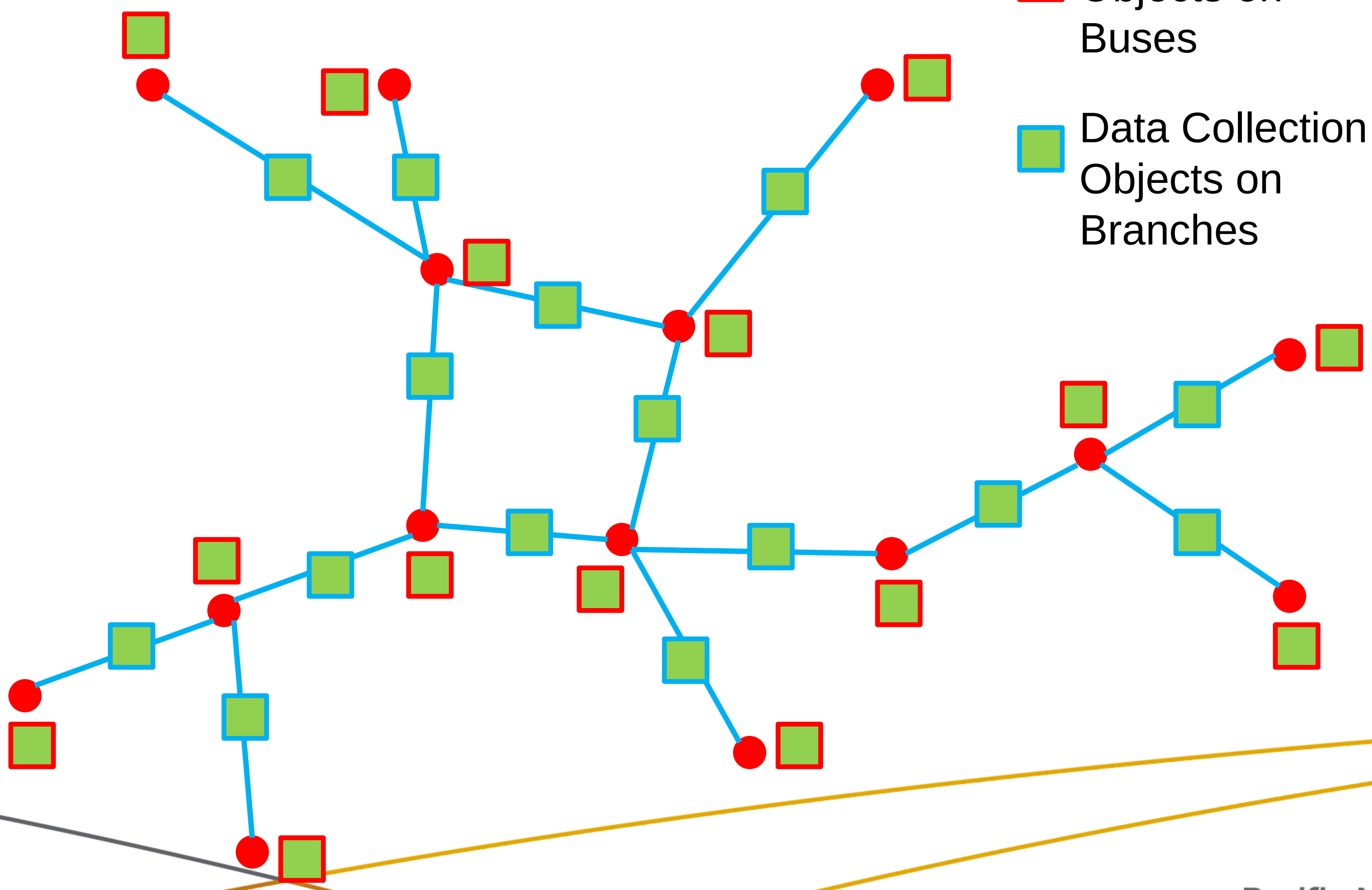
Create Network from External File

```
#include "gridpack/parser/ParserPTI.hpp"
     :
gridpack::parser::PTI23_parser<MyNetwork> parser(network);
parser.parse("location_of_PTI_file");

// The network topology now exists and the data
// collection objects on each bus and branch are filled
// with parameters from the PTI file. The network is
// NOT, however, distributed in an optimal way at this
// point. Also, no ghost buses or ghost branches have
// been added to the network yet, so most calculations
// not possible
```

Network Topology

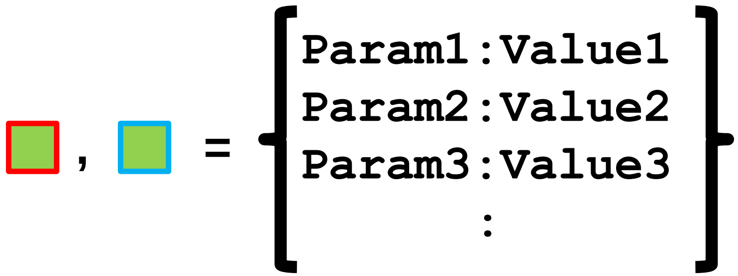


Network Data

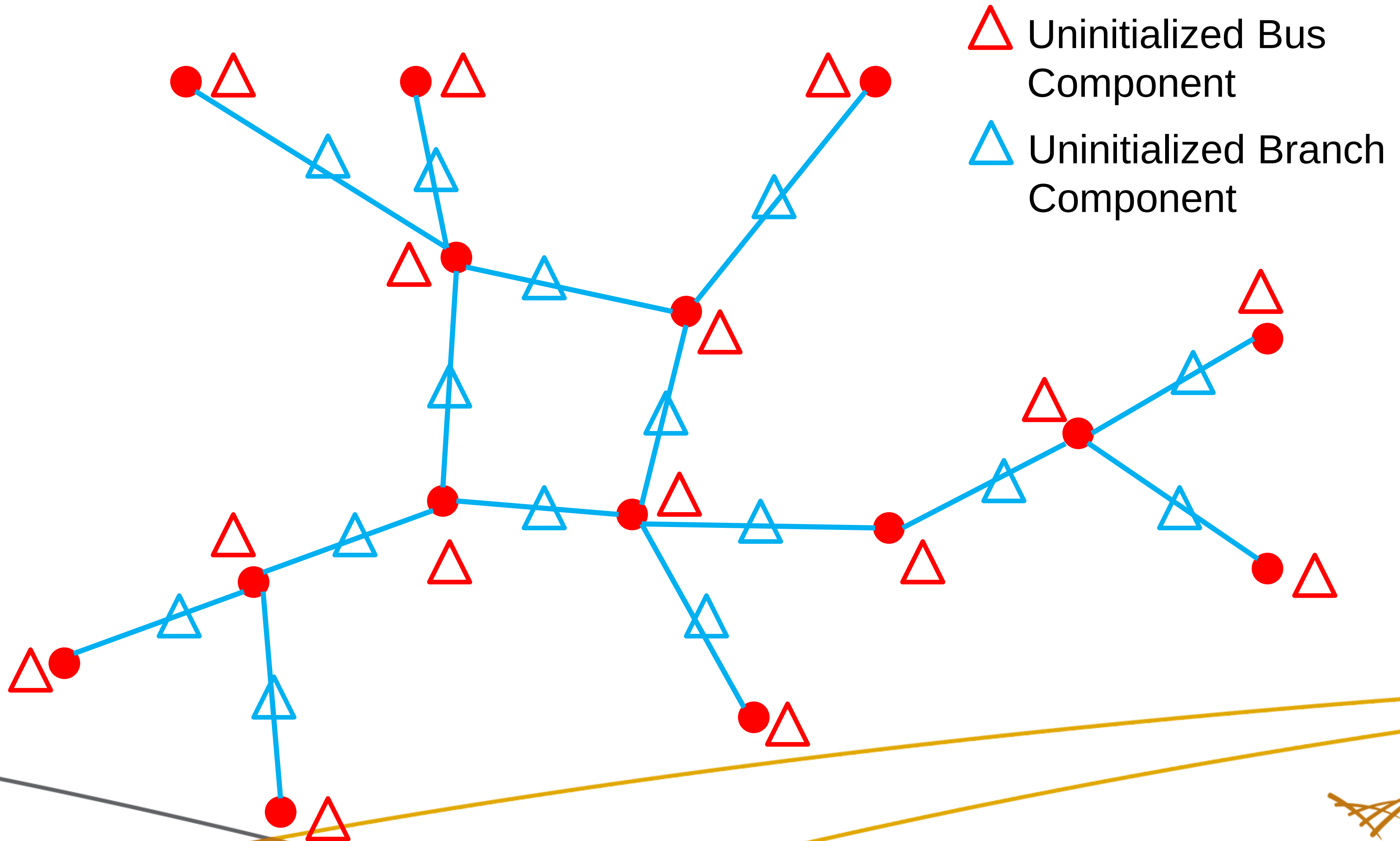


DataCollection Objects

- Each bus and each branch has its own DataCollection object
- DataCollections are lists of key-value pairs representing parameters from network configuration file



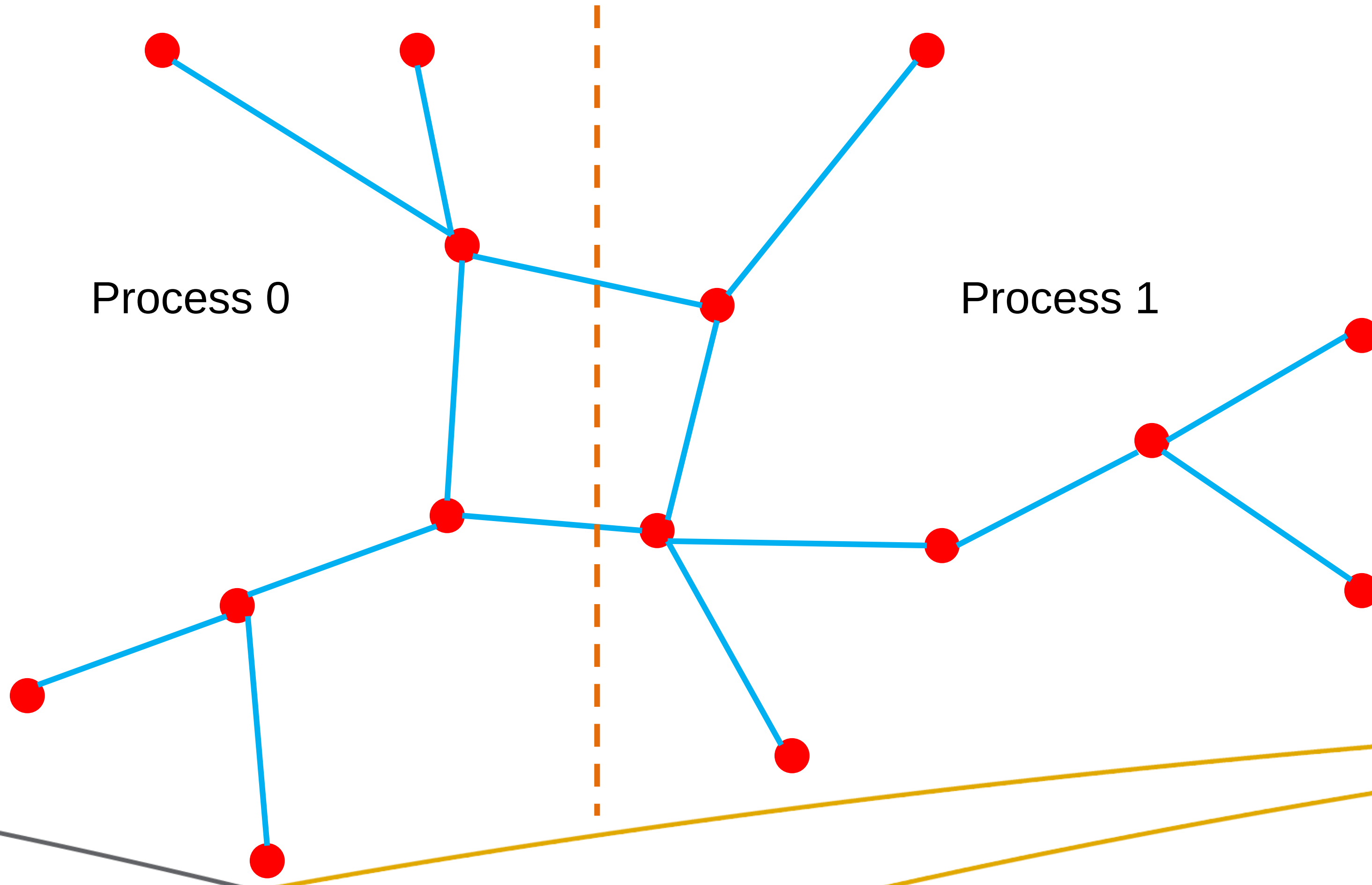
Network Components



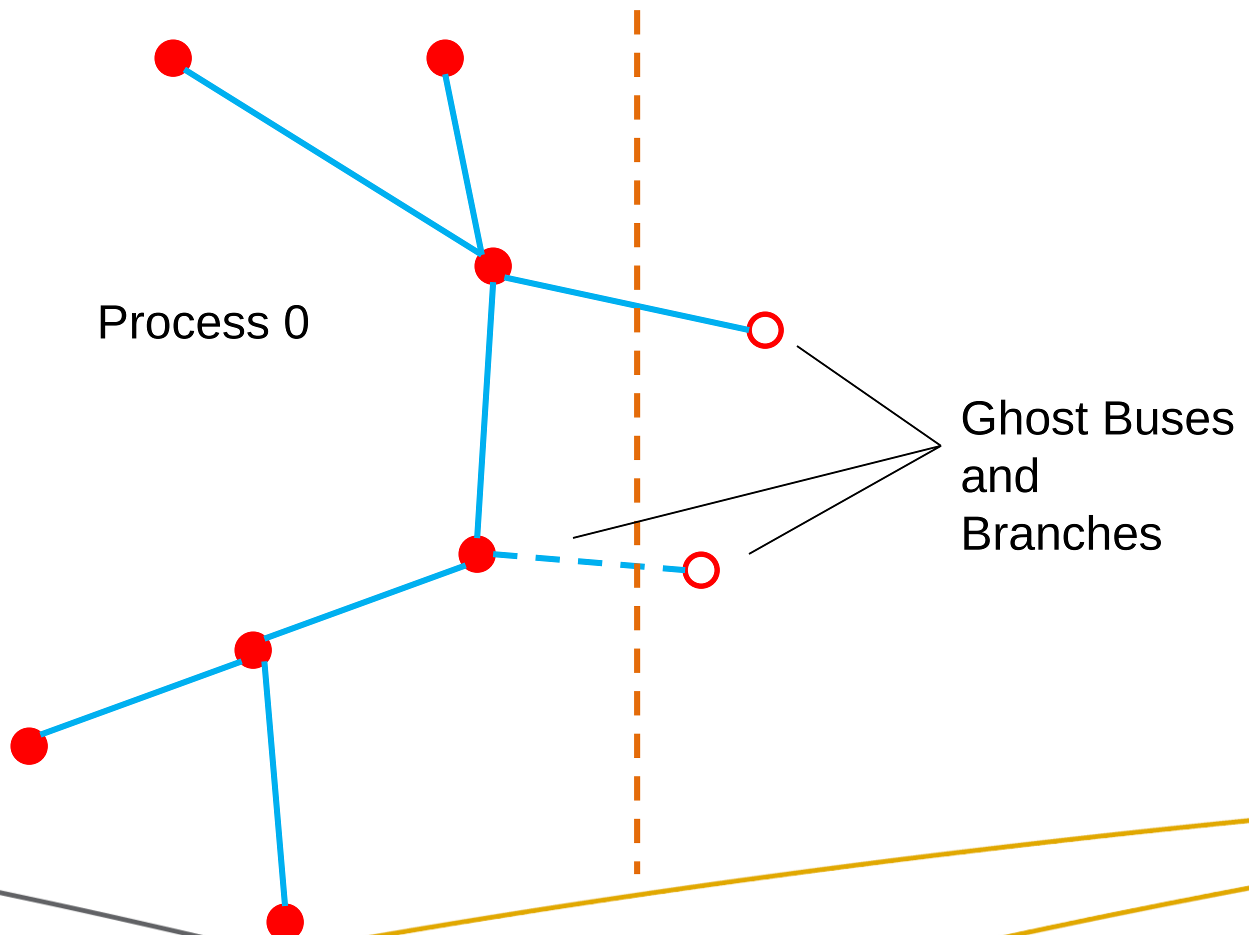
Partition Network

```
// Invoke the partition function

network->partition();

// Network has been properly distributed among
// processors, ghost buses and ghost branches have been
// added to the network, and global indices have been
// set. Local neighbor lists and indices for the ends
// branches have also been set. Network is almost ready
// for calculations
```

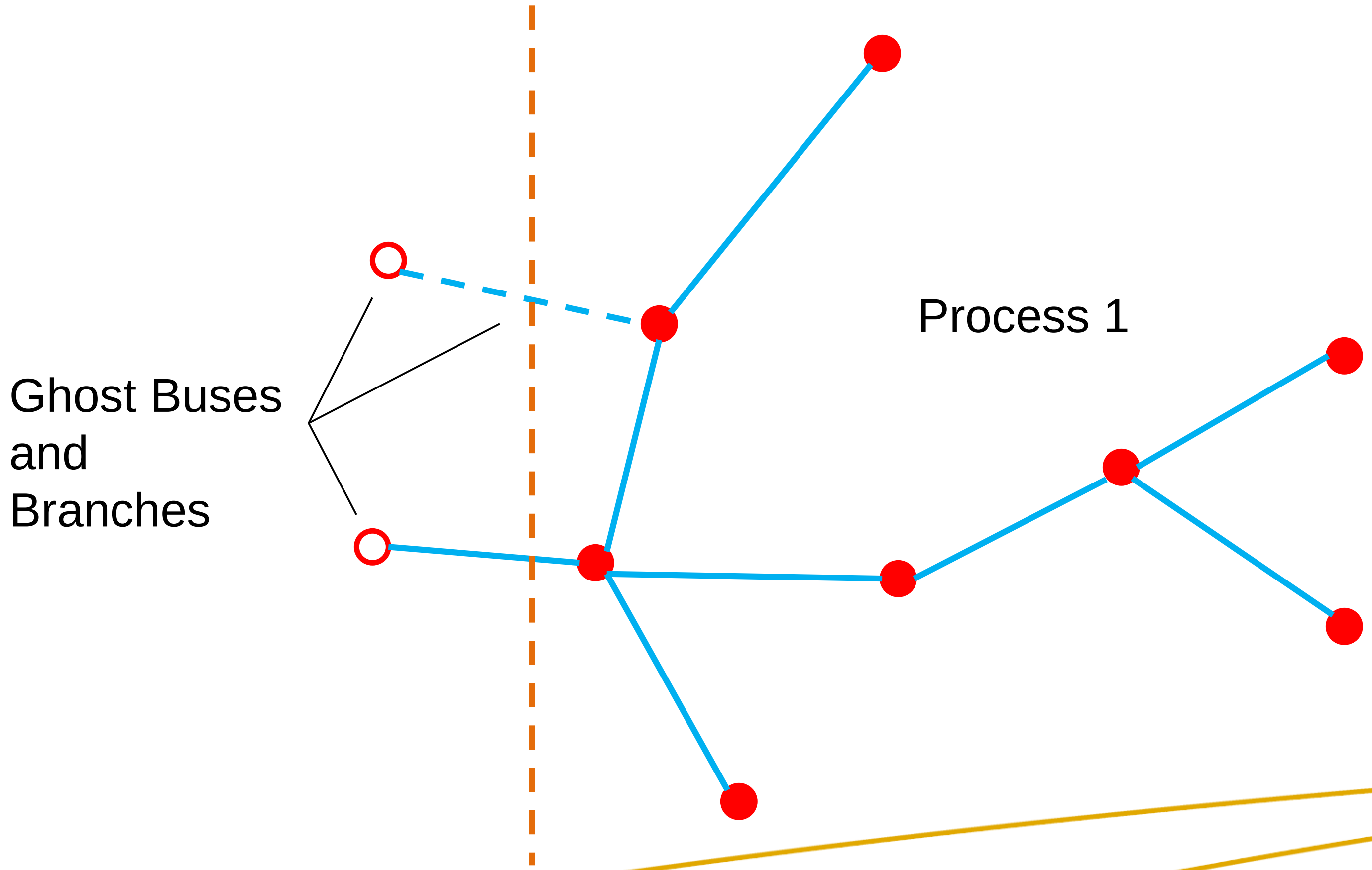
Partitioning the Network



Process 0 Partition



Process 1 Partition



Other Base Network Class Methods

```
// Return number of buses and branches on this process
// (including ghost buses and branches)
int numBuses(void);
int numBranches(void);

// Get local index of reference bus (return -1 if
// reference bus is not on this process)
int getReferenceBus(void) const;

// Return true if bus or branch is local to the process,
// return false for ghost buses and branches
bool getActiveBus(int idx);
bool getActiveBranch(int idx);

// Return pointer to bus or branch object corresponding
// to local index idx
boost::shared_ptr<MyBus> getBus(int idx);
boost::shared_ptr<MyBranch> getBranch(int idx);
```

Base Network Class

```
// Return pointer to DataCollection objects associated
// with bus or branch at local index idx
boost::shared_ptr<gridpack::component::DataCollection>
  getBusData(int idx);
boost::shared_ptr<gridpack::component::DataCollection>
  getBranchData(int idx);

// Remove all ghost buses and branches from the network
void clean(void);

// Set up data structures for exchanges to ghosts buses
// and branches
void initBusUpdate(void);
void initBranchUpdate(void);

// Send data to ghost buses and branches
void updateBuses(void);
void updateBranches(void)
```

Factories

- Factories are used to manage interactions between the network and individual network components
- Factories perform some basic initialization functions
- Factories are designed to set up the system so that it can be used in calculations. They guarantee the all bus and branch objects are in the correct state for generating the matrices and vectors needed for solving the problem
- Factories can be used to change the state network components
- A primary motif in factory methods is that they loop over all bus and branch objects and invoke methods on them

Base Factory Class

```
// Constructor
BaseFactory<MyNetwork>::BaseFactory(
        boost::shared_ptr<MyNetwork> network);

// The "load" operation takes all the data from the
// data collection objects and moves them into the
// corresponding bus and branch components using the
// load functions that have been defined in the
// individual bus and branch classes
virtual void load(void);

// Set up lists of pointers in each component to
// neighbors of all buses and branches in the
// network as well as assigning internal indices used
// by other GridPACK™ components
virtual void setComponents(void);
```

Base Factory Class

```
// Set up internal buffers so that data exchanges to
// fill up ghost branches and ghost buffers will work.
// To exchange data, users need to put the data in the
// component's exchange buffer and then invoke a bus
// or branch ghost exchange in the network
virtual void setExchange(void);

// Invoke the setMode method on all bus and branch
// components in the network. The mode can be used to
// control the behavior of the network components at
// different stages of the calculation
virtual void setMode(int mode);
```

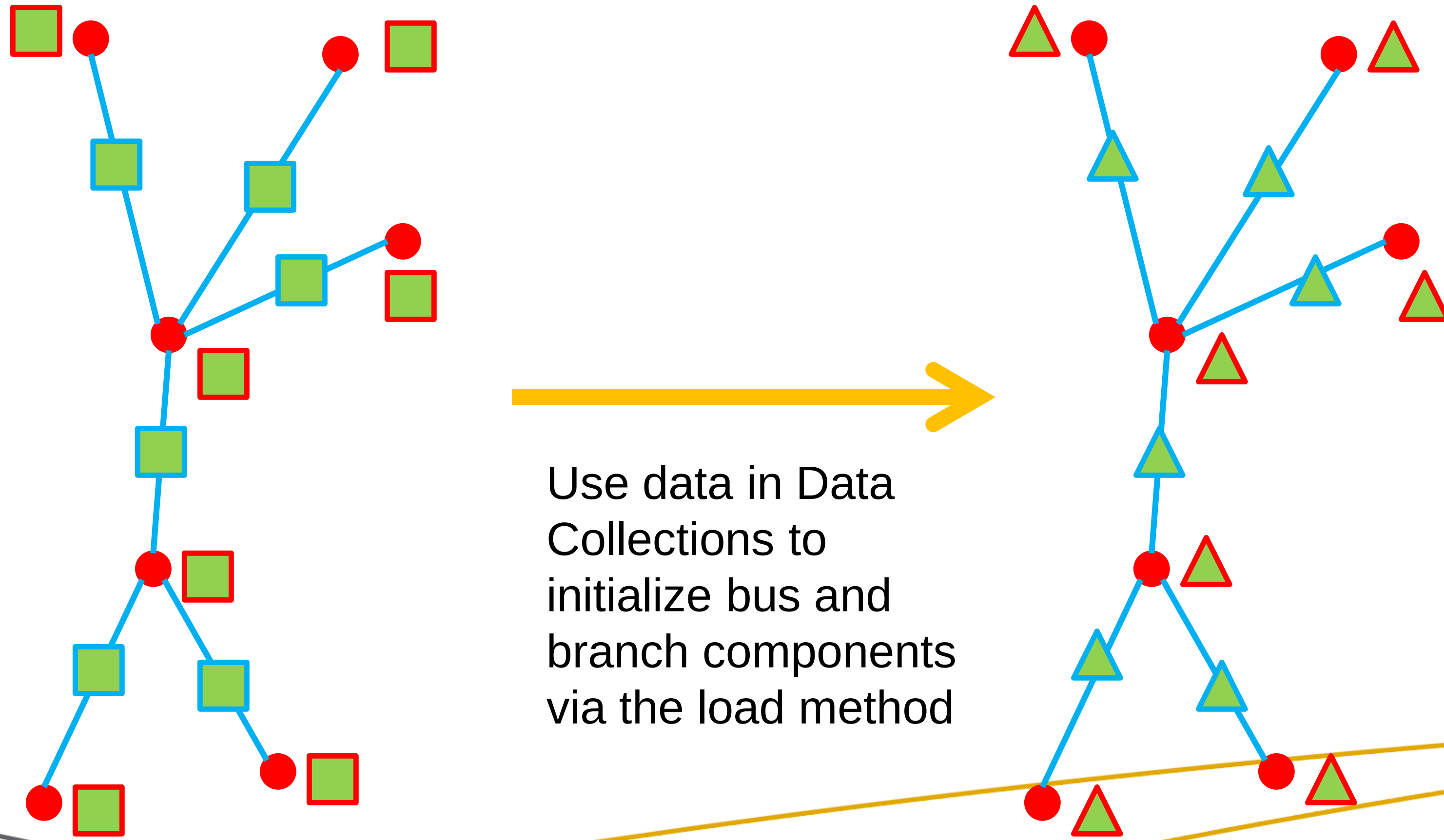
Create a New Factory

```
Class MyFactory
  : public gridpack::factory::BaseFactory<MyNetwork> {

  MyFactory(MyNetworkPtr network);

  ~MyFactory();

  MyFactoryMethod1(…);

  MyFactoryMethod2(…);
      :
      :
}
```

Example of a Factory Method

```
/**
 * Generic method that invokes the "load" method
 * on all branches and buses to move data from
 * the DataCollection objects on the network into the
 * corresponding buses and branches
 */
void gridpack::factory::BaseFactory<MyNetwork>::load(void)
{
  int numBus = p_network->numBuses();
  int numBranch = p_network->numBranches();
  int i;
  // Invoke load method on all bus objects
  for (i=0; i<numBus; i++) {
    p_network->getBus(i)->load(p_network->getBusData(i));
  }
  // Invoke load method on all branch objects
  for (i=0; i<numBranch; i++) {
    p_network->getBranch(i)->load(p_network->getBranchData(i));
  }
}
```

Initialize Components



Use data in Data Collections to initialize bus and branch components via the load method

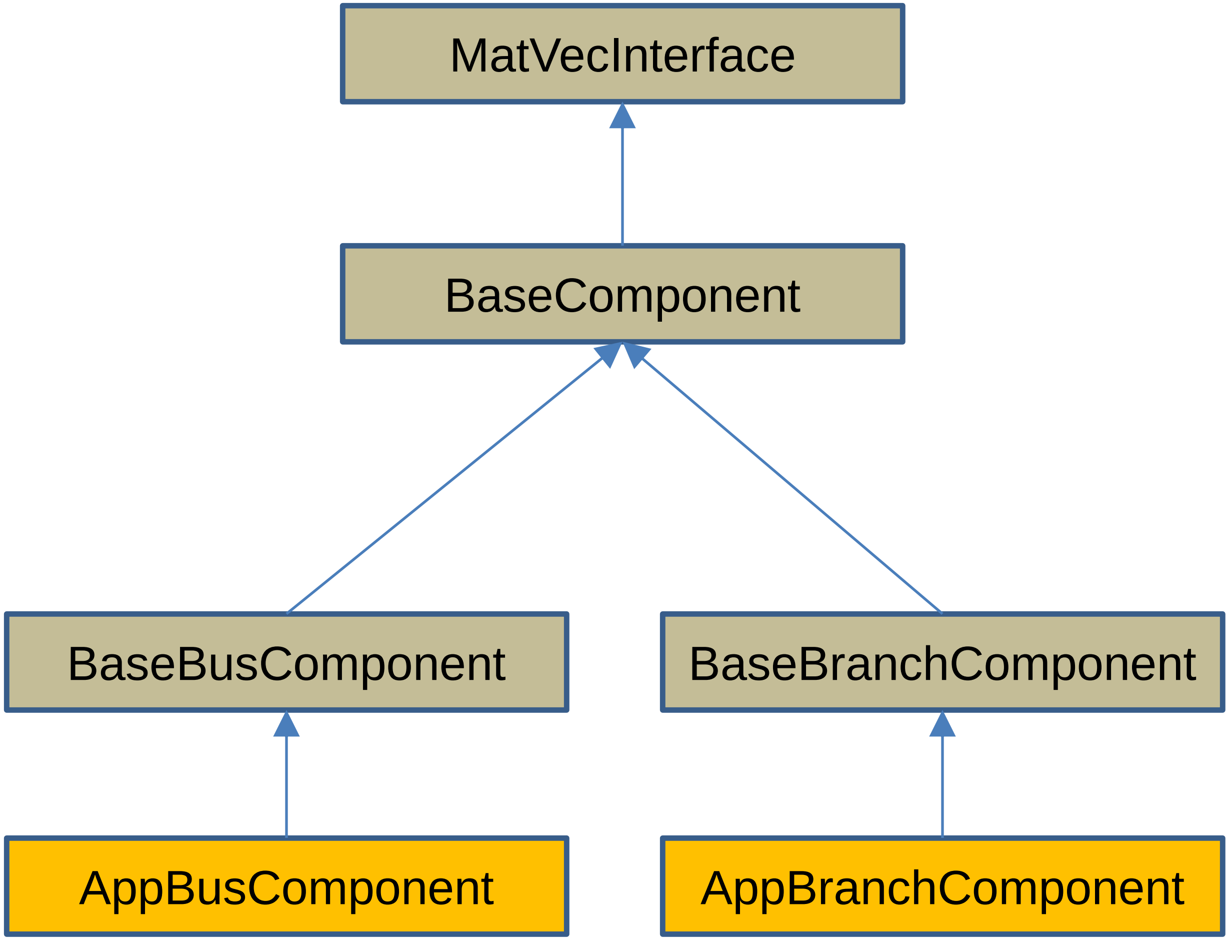
Initialize Network Components

```
#include "gridpack/applications/myapp/MyFactory.hpp
   :
gridpack::myapp::MyFactory factory(network);

// Initialize components with data from DataCollection
// objects
factory.load();

// Set up internal indices used by mappers to create
// matrices and vectors and set pointers for
// neighboring buses and branches
factory.setComponents();

// Set up buffers for ghost exchanges
factory.setExchange();
```

Components

- All components are derived from the MatVecInterface class and the BaseComponent class
- Bus components are derived from the BaseBusComponent class
- Branch components are derived from the BaseBranchComponent class

Component Class Hierarchy



The MatVecInterface

- Designed to allow the GridPACK™ framework to generate distributed matrices and vectors from individual bus and branch components
- Buses and branches are responsible for describing their individual contribution to matrices and vectors
- Buses and branches are NOT responsible for determining location in matrix or vector and are NOT responsible for distributing matrices or vectors

Data Type

- Matrices and vectors are assumed to be complex
- ComplexType used for everything, including real values (a real number is just a complex number with the imaginary part set to zero)

Diagonal MatVecInterface

```
// Return the size of matrix block on the diagonal.
// Usually implemented on bus components. This function
// returns false if the component does not contribute
// anything to the matrix

virtual bool matrixDiagSize(int *isize,
                            int *jsize) const

// Return the values of the block in row-major order.
// Return false if component does not contribute to matrix

virtual bool matrixDiagValues(ComplexType *values)
```

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by Battelle *Since 1965*

Off-diagonal MatVecInterface

```
// Return the size an off-diagonal matrix block
// contributed by the component. This function returns
// false if no values are contributed by component. These
// functions are usually implemented on branches. The
// Forward function is called for an ij pair when i
// corresponds to the "from" bus defining a branch.
// The Reverse function is called when i corresponds
// to the "to" bus

virtual bool matrixForwardSize(int *isize,
                               int *jsize) const
virtual bool matrixReverseSize(int *isize,
                               int *jsize) const

// Return the values of off-diagonal matrix block.
// Values are in row-major order.

virtual bool matrixForwardValues(ComplexType *values)
virtual bool matrixReverseValues(ComplexType *values)
```

Vector MatVecInterface

```
// Return the block size of component contribution to
// a vector. Return false if a component does not
// contribute to vector

virtual bool vectorSize(int *isize) const

// Return the values of the vector block contributed
// by component

virtual bool vectorValues(ComplexType *values)
```

BaseComponent

- This class provides a few methods that are needed by all network components (bus or branch)
- Provides methods for moving data from DataCollection objects to components and sets up buffers used for ghost bus and ghost branch exchanges
- Provides a mechanism for changing component behavior so that different matrices can be extracted from components during different phases of the calculation

BaseComponent

```
// These methods need to be redefined in application
// components. Default implementations are all no-ops

// Load data from DataCollection object into component
virtual void load(const shared_ptr<DataCollection> &data)

// Return the size of the buffer needed for data exchanges
// Note that all bus components must return the same value
// for this function and all branch components must return
// the same value
virtual int getXCBufSize(void)

// Assign the location of the data exchange buffer to an
// internal component pointer. This buffer is allocated
// and deallocated by the network
virtual void setXCBuf(void *buf)
```

BaseComponent

```
// Write out a single string describing current state of
// the component. The character string “signal” can be used
// to control the behavior of the component. The size of
// buffer “string” is contained in the variable bufsize.
// Return false if no string is being returned. This
// functionality is used in the serialIO module
virtual bool serialWrite(char *string, const int bufsize,
                         const char *signal)
```

BaseComponent::setMode

```
// Set internal behavior of component by specifying a "mode"
void setMode(int mode)
```

- Each bus and branch component should have an internal mode variable corresponding to an enumerated type
- Different values of the mode variable can generate different behaviors. For example, the network components in a power flow application must be able to generate both the Y-matrix and the Jacobian for a powerflow calculation
  - If the mode is set to “YBus”, the MatVecInterface functions return values appropriate for creating the Y-matrix
  - If the mode is set to “Jacobian”, the MatVecInterface functions return values appropriate for generating the Jacobian matrix used in the power flow equations

BaseBusComponent

- Provides methods that are needed by all bus component implementations
- Sets up lists of branches that are attached to the bus and buses that are attached via a single branch
- Keeps track of the reference bus

BaseBusComponent

```
// Get pointers to branches that are connected to bus
void getNeighborBranches(vector<shared_ptr
                    <BaseComponent> > &nghbrs) const

// Get pointers to buses that are connected to bus via
// a single branch
void getNeighborBuses(vector<shared_ptr
                    <BaseComponent> > &nghbrs) const

// If bus is reference bus, set status to true
void setReferenceBus(bool status)

// Return true if this bus is reference bus
bool getReferenceBus(void) const
```

BaseBranchComponent

- Provides methods that are needed by all branch component implementations
- Keeps track of the buses at each end of the branch and makes these available to the application

BaseBranchComponent

```
// Get pointers to buses at either end of the branch.
// Bus 1 refers to the “from” bus and bus 2 refers to
// the “to” bus

shared_ptr<BaseComponent> getBus1(void) const
shared_ptr<BaseComponent> getBus2(void) const
```

DataCollection Objects

- Every bus and branch in the network has an associated DataCollection object
- The DataCollection object is a container for key-value pairs of parameters associated with a particular bus or branch
- Different buses and branches can contain different numbers of key-value pairs and different types of key-value pairs
- If a bus or branch has a list of values associated with the same key, then these can be indexed
- Key-value pairs are usually added to the DataCollection objects when the network is read in from a network configuration file

Data Collection Structure

DataCollection instance

```
double key1:value1
int    key2:value2
bool   key3:value3
double key4:value4

double key5[0]:value5[0]
double key5[1]:value5[1]
double key5[2]:value5[2]
```

DataCollection Names

Names assigned by parser are defined in the dictionary.hpp file

```
/**
 * Non-blank alphanumeric branch circuit identifier
 * type: string
 */
#define BRANCH_CKT "BRANCH_CKT"
/**
 * Branch resistance; entered in pu
 * type: real float
 */
#define BRANCH_R "BRANCH_R"
/**
 * Branch reactance; entered in pu
 * type: real float
 */
#define BRANCH_X "BRANCH_X"
```

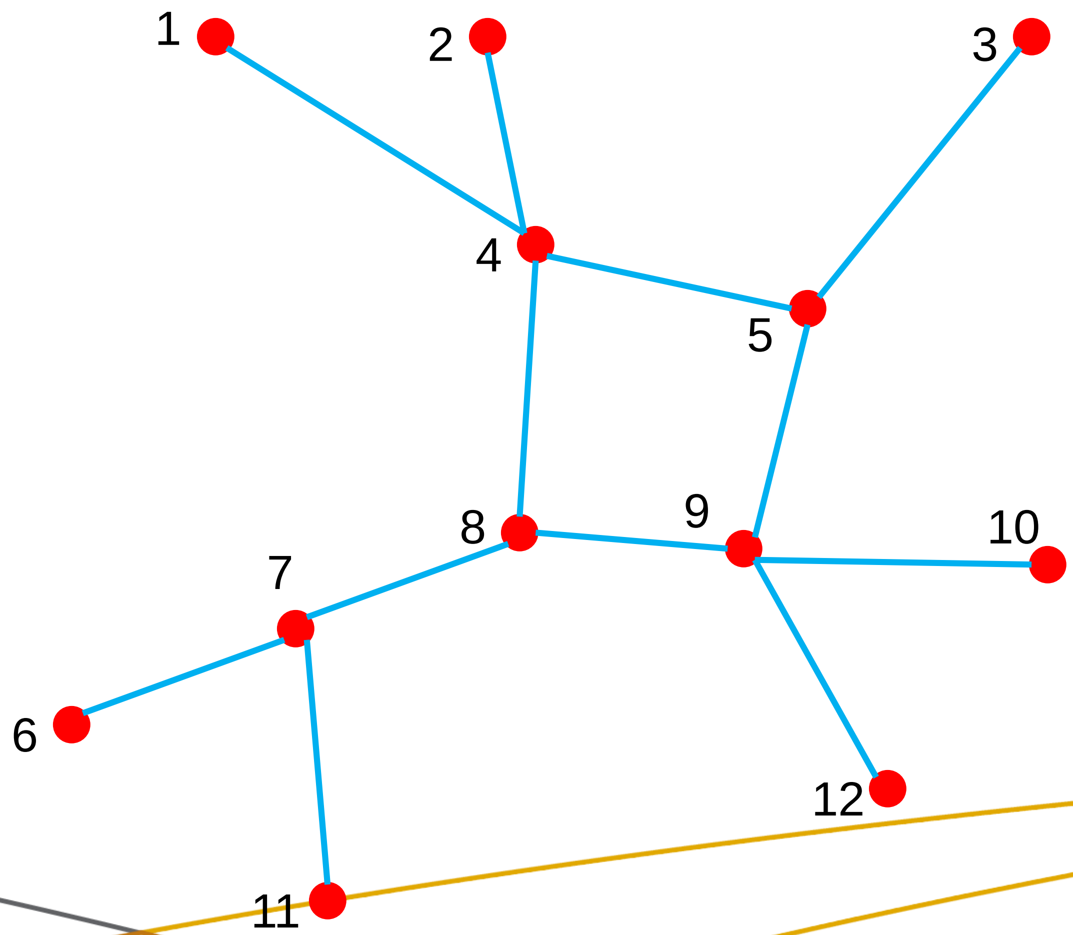
DataCollection Accessors

```
bool getValue(char *name, int *value);
bool getValue(char *name, long *value);
bool getValue(char *name, bool *value);
bool getValue(char *name, std::string *value);
bool getValue(char *name, float *value);
bool getValue(char *name, double *value);
bool getValue(char *name, gridpack::ComplexType *value);

bool getValue(char *name, int *value, int idx);
bool getValue(char *name, long *value, int idx);
bool getValue(char *name, bool *value, int idx);
bool getValue(char *name, std::string *value, int idx);
bool getValue(char *name, float *value, int idx);
bool getValue(char *name, double *value, int idx);
bool getValue(char *name, gridpack::ComplexType *value,
              int idx);
```

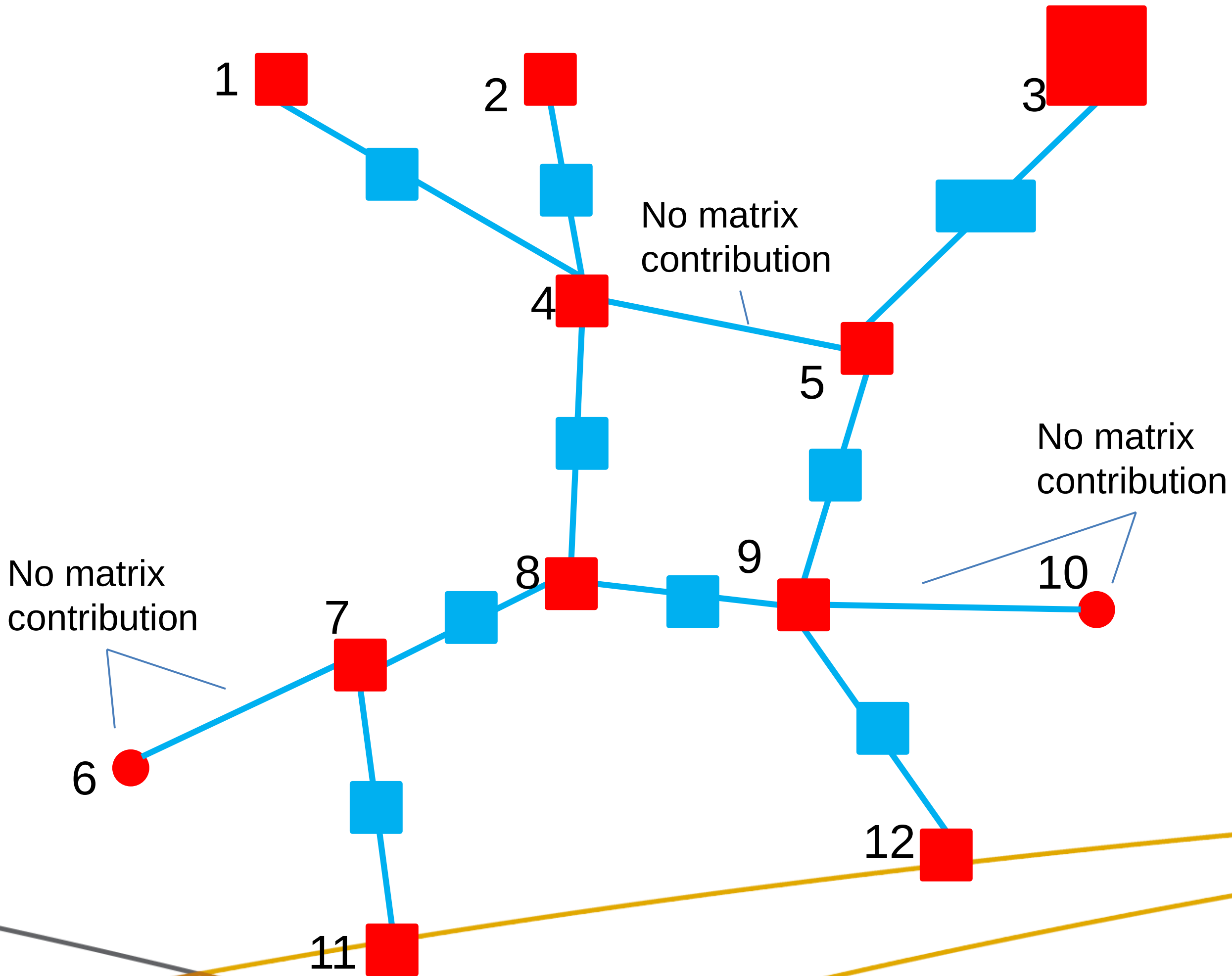
Mapper

- Provide a flexible framework for constructing matrices and vectors representing power grid equations
- Hide the index transformations and partitioning required to create distributed matrices and vectors from application developers
- Developers can focus on the contributions to matrices and vectors coming from individual network elements

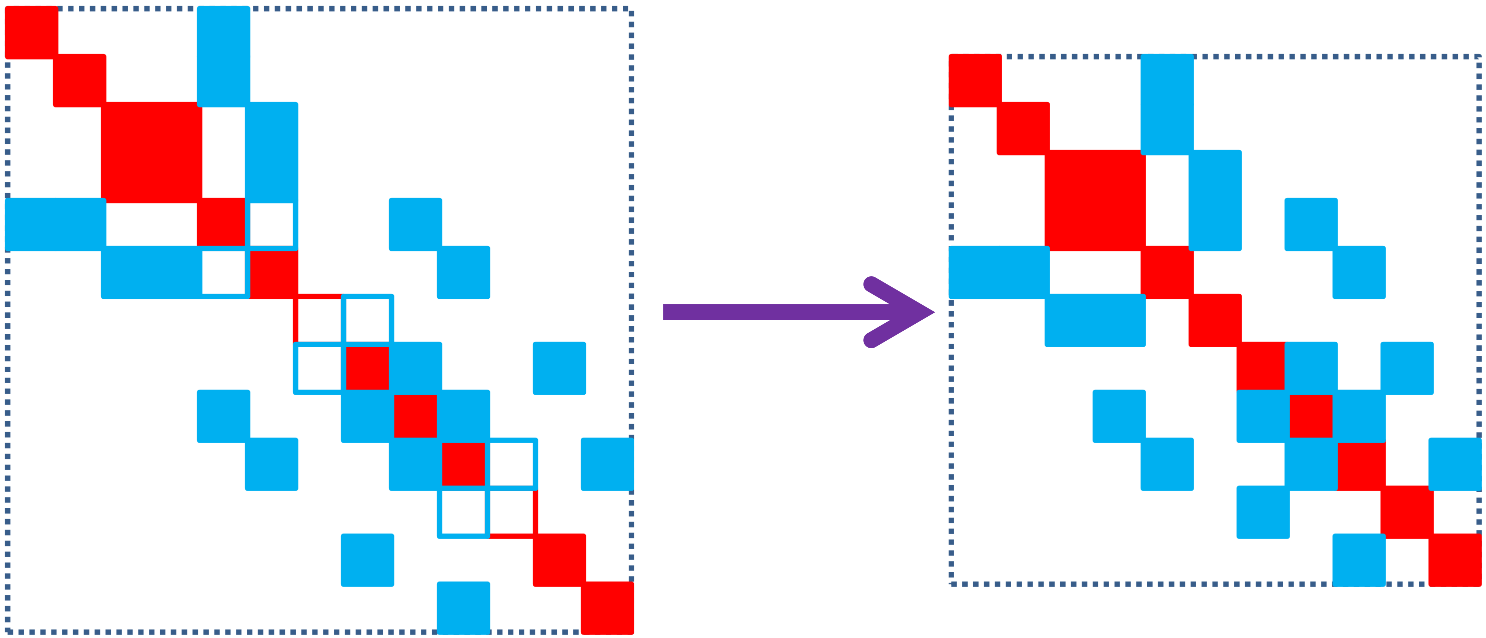
Mapper



Matrix Contributions from Components



Distribute Component Contributions and Eliminate Gaps



Matrix Mapper Interface

```
// Instantiate a new mapper that creates a matrix from
// bus and branch components on the network
FullMatrixMap<MyNetwork>::FullMatrixMap(
   shared_ptr<MyNetwork> network);

// Create a matrix from the network
shared_ptr<Matrix> mapToMatrix(void);

// Reset matrix based on current network values
void mapToMatrix(shared_ptr<Matrix> &M);
```

Vector Mapper Interface

```
// Instantiate a new mapper that creates a vector from
// bus components on the network
BusVectorMap<MyNetwork>::BusVectorMap(
  shared_ptr<MyNetwork> network);

// Create a vector from the network
shared_ptr<Vector> mapToVector(void);

// Reset vector based on current network values
void mapToVector(shared_ptr<Vector> &V);

// Push current values in vector back into buses
void mapToBus(Vector &vector);
```

Mapper Behavior

- The matrix or vector that is produced by a mapper is controlled by
  - The functions that are implemented in the MatVecInterface by the application developer
  - The current value of the mode variable. If the application needs to create different matrices or vectors based on different modes, then separate mappers should be created for each mode
  - When calling any of the mapper functions, the mode should always be set to the same value as the mode that was in place when the mapper was created

Math Module

- The math module is a wrapper on top of a parallel solver library. It supports
  - Distributed sparse and dense matrices and distributed vectors
  - Basic manipulations of matrices and vectors, e.g. matrix additions, matrix-vector multiplication, scaling of matrices, creation of identity matrix, etc.
  - Linear solvers that support different algorithms and preconditioners for solving the matrix equation Ax=b
  - Nonlinear solvers

Math Library

```
// Initialize math library. Call any initialization
// routines that are necessary and read in any
// configuration files. Currently, PETSc options are
// listed in a gridpack.petscrc file. This is where
// preconditioners and other PETSc configuration
// parameters are specified.

extern void Initialize(void);

// Is math library initialized?
extern bool Initialized(void);

// Shut down math library
extern void Finalize();
```

Vector Class

```
// Specify parallel configuration and local contribution
// to vector in constructor

Vector(const parallel::Communicator &comm,
       const int &local_length);

// Accessors for vector properties
int size(void) const;
int localSize(void) const;
void localIndexRange(int &lo, int &hi) const;
```

Distributed Vector Storage

Vectors are distributed in contiguous segments between processes

Vector Class

```
// Set vector elements
void setElement(const int &i, const ComplexType &x);
void setElements(const int &n, const int *i,
                 const ComplexType *x);

// Access vector elements
void getElement(const int &i, ComplexType &x) const;
void getElements(const int &n, const int *i,
                 ComplexType *x) const;

// Indicate vector is ready to use
void ready(void);
```

Basic Vector Operations

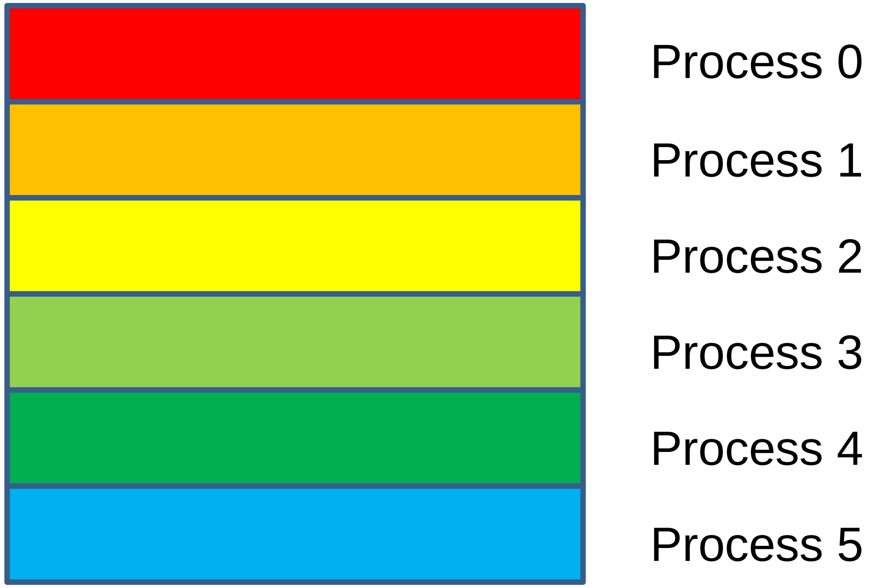
```
// Basic operations that can be performed on vectors
void zero(void);
void fill(const ComplexType &v);
ComplexType norm1(void) const; // L1 norm
ComplexType norm2(void) const; // L2 norm (standard)
void scale(const ComplexType &x);
void add(const Vector &x, const ComplexType &scale = 1.0);
void equate(const Vector &x);
void reciprocal(void);
```

Matrix Class

```
// Specify dimensions and storage format of matrix in
// constructor.

Matrix(const parallel::Communicator &dist,
       const int &local_rows,
       const int &cols,
       const StorageType &storage_type=Sparse);

// Accessors for matrix properties

int rows(void) const;
int localRows(void) const;
void localRowRange(int &lo, int &hi) const;
int cols(void) const;
```

Distributed Matrix Storage

Matrices are laid out in row blocks



Matrix Class

```
// Set matrix elements
void setElement(const int &i, const int &j,
                const ComplexType &x);

void setElements(const int &n, const int *i,
                 const int *j, const ComplexType *x);

// Access matrix elements
void getElement(const int &i, const int &j,
                ComplexType &x) const;

Void getElements(const int &n, const int *i,
                  const int *j, ComplexType *x) const;

// Indicate matrix is ready
void ready(void);
```

Basic Matrix Operations

```
// Basic operations that can be performed on matrices

void equate(const Matrix &A);
void scale(const ComplexType &x);
void multiplyDiagonal(const Vector &x);
void add(const Matrix &A);
void identity(void);
void zero(void);

// Matrix-Vector operations
extern Matrix *add(const &A, const &B);
extern Matrix *transpose(const Matrix &A);
extern Vector *column(const Matrix &A, const int &cidx);
extern Vector *diagonal(const Matrix &A);
extern Matrix *multiply(const Matrix &A, const Matrix &B);
extern Vector *multiply(const Matrix &A, const Vector &x);
```

Linear Solver

```
// Solve equation using an instance of a LinearSolver

LinearSolver(const Matrix &A);
void solve(const Vector &b, Vector &x) const;
void configure(CursorPtr cursor);

// Most of the solver functionality can be accessed by
// requesting it in the input deck

<LinearSolver>
  <PETScOptions>
    -ksp_view
    -ksp_type richardson
    -pc_type lu
    -pc_factor_mat_solver_package superlu_dist
    -ksp_max_it 1
  </PETScOptions>
</LinearSolver>
```

Configure

- Configure is designed to take user input, in the form of an XML-based input file, and transfer that information to any parts of the code that might need it. Configure is designed to handle relatively limited amounts of data, it is not designed for handling large data objects like the network. Examples of user input include
  - Location of network configuration file
  - Type of solvers to use
  - Solution parameters such as convergence tolerance, maximum number of iterations, etc.
  - Control parameters for different types of data output

Configuration Module

```
// Access common instance of configuration module,
// shared across all modules
static Configuration* configuration()

// Open external configuration file
bool open(std::string file, Communicator comm)

typedef Configuration Cursor

// Set the "cursor" in the Configuration object so
// that it only looks at key-value pairs within the
// block delimited by "key"
Cursor* getCursor(std::string key)
```

Configuration Accessors

```
// Return default value if no value for that key is
// set in input file
bool get(std::string, bool default_value)
int get(std::string, int default_value)
double get(std::string, double default_value)
std::string get(std::string,
                std::string default_value)
std::vector<double> get(std::string,
                std::vector<double> default_value)

// Return value for key, value of function is
// false if no value found
bool get(std::string, bool *value)
bool get(std::string, int *value)
bool get(std::string, double *value)
bool get(std::string, std::string *value)
bool get(std::string, std::vector<double> *value)
```

Input File example

```xml
<?xml version="1.0" encoding="utf-8"?>
<Configuration>
  <Powerflow>
    <networkConfiguration>IEEE14.raw</networkConfiguration>
    <LinearSolver>
      <PETScPrefix>nrs</PETScPrefix>
      <PETScOptions>
        -ksp_atol 1.0e-08
        -ksp_rtol 1.0e-12
        -ksp_monitor
        -ksp_max_it 50
        -ksp_view
      </PETScOptions>
    </LinearSolver>
  </Powerflow>
</Configuration>
```

Using the Configuration Module

```
// Using example input file from previous slide. Note that
// “open” is a collective operation, all others are local
Configuration *config = Configuration::configuration();
config->open(“input.xml”,MPI_COMM_WORLD);

// This returns “IEEE14.raw” in variable filename
std::string filename;
config.get(“Configuration.Powerflow.networkConfiguration”,
           &filename);

// This also returns “IEEE14.raw” in variable filename
Cursor *cursor = config->getCursor(“Configuration.Powerflow”);
cursor->get(“networkConfiguration”,&filename);
```

Serial IO

- Works in conjunction with the writeSerial operation in the BaseComponent class
- Designed to send output to standard out from buses and/or branches

11	0.942	-16.250	-	-	-	-
12	0.943	-16.176	-	-	16.70	1.70
13	0.926	-15.878	-	-	16.10	1.60
21	0.964	-12.162	-	-	196.20	19.60
23	0.964	-12.162	-	-	0.10	0.10
31	0.967	-10.454	-	-	79.20	7.90
32	0.967	-10.454	-	-	79.20	7.90
41	0.978	-11.654	-	-	106.70	10.70
43	0.978	-11.688	-	-	5.60	0.60
51	0.937	-16.934	-	-	63.70	6.40
52	0.940	-16.426	-	-	-	-
61	0.909	-21.810	-	-	23.20	2.30
62	0.905	-23.846	-	-	23.40	2.30
75	0.923	-18.114	-	-	21.30	2.10

Serial IO Classes

```
// Write serial IO from buses. "len" is the maximum size
// string that is written. The string "signal" is passed
// to the writeSerial method in the BaseComponent class.
// The "write" method will trigger the writeSerial
// in the base and branch components, the "header" method
// is a convenience method for writing single strings
// from the head node
SerialBusIO(int len,
            boost::shared_ptr<MyNetwork> network)
void write(char *signal)
void header(char *str)

// Write Serial IO from branches
SerialBranchIO(int len,
            boost::shared_ptr<MyNetwork> network)
void write(char *signal)
void header(char *str)
```

Using Serial IO

Use code fragment

```
SerialBusIO busIO(256,network);
busIO.header("    Bus       Voltage                 Generation                 Load\n");
busIO.header("     #    Mag(pu)   Ang(deg)      P (MW)     Q (MVAr)      P (MW)    Q (MVAr)\n");
busIO.header(" -------------------------------------------------------------------\n");
busIO.write();
```

to produce

```
   Bus       Voltage                 Generation                 Load
     #    Mag(pu)   Ang(deg)      P (MW)     Q (MVAr)      P (MW)    Q (MVAr)
-------------------------------------------------------------------
  11    0.942     -16.250        -          -            -           -
  12    0.943     -16.176        -          -          16.70        1.70
  13    0.926     -15.878        -          -          16.10        1.60
  21    0.964     -12.162        -          -         196.20       19.60
  23    0.964     -12.162        -          -           0.10        0.10
  31    0.967     -10.454        -          -          79.20        7.90
  32    0.967     -10.454        -          -          79.20        7.90
  41    0.978     -11.654        -          -         106.70       10.70
```

These lines are produced from the serialWrite method in BaseComponentClass

serialWrite method

```
bool gridpack::myapp::MyBus::serialWrite(char *string,
  const int bufsize, const char* signal){
  sprintf(string,"  %4d%7.3f%12.3f",getOriginalIndex(),
          p_volt, p_angle);
  int len = strlen(string)
  char *ptr = string + strlen
  if (p_generator) {
    sprintf(ptr,"   %f12.3  %f12.3",p_gen_p, p_gen_q);
  } else {
    sprintf(ptr,"           -           -");
  }
  len = strlen(ptr);
  ptr += len;
  if (p_load) {
    sprintf(ptr,"   %f12.3  %f12.3\n",p_load_p, p_load_q);
  } else {
    sprintf(ptr,"           -           -\n");
  }
}
```

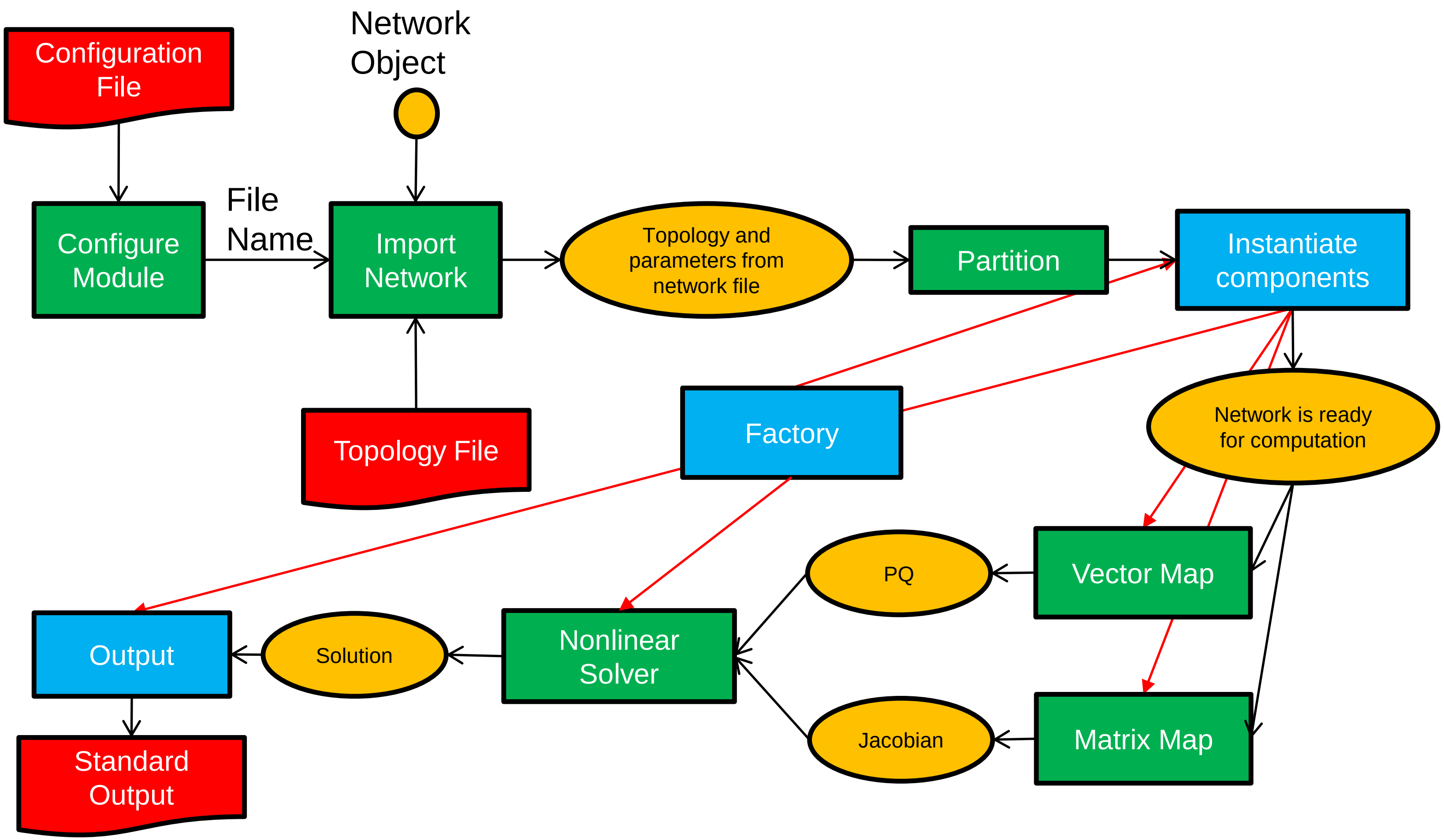
Directing Serial IO to Files

- Use open and close methods to redirect output from standard out to a file
- Same serial IO objects can be used to create multiple files

```
// Open file with name filename and direct all
// subsequent output to file
void open(const char *filename)

// Close file and redirect output back to standard
// out
void close(void)
```

Powerflow Application Example

- Create elements of Y-matrix and solve powerflow equations using a Newton-Raphson procedure.
  - Powerflow components: set network parameters and evaluate matrix and vector elements
  - Powerflow factory: coordinate higher level functions over the whole network
  - Powerflow application: control program flow and implement higher level solver routine


Network Object
Configuration File
Configure Module
File Name
Import Network
Topology and parameters from network file
Partition
Instantiate components
Topology File
Factory
Network is ready for computation
PQ
Vector Map
Output
Solution
Nonlinear Solver
Jacobian
Matrix Map
Standard Output

Powerflow Components

- Create two new classes to represent buses and branches, PFBus and PFBranch
  - These classes inherit from BaseBusComponent and BaseBranchComponent
  - Create load methods in to initialize components from network configuration file parameters
  - Implement routines to evaluate elements of Y-matrix on both bus and branches
  - Implement functions in MatVecInterface to create Y-matrix, Jacobian matrix and right-hand-side (PQ) vector
  - Set up buffers for data exchanges between processors
  - Implement serialWrite method to create output

PFBus::load

```
void gridpack::powerflow::PFBus::load(
    const boost::shared_ptr<gridpack::component
    ::DataCollection> &data)
{
  data->getValue(CASE_SBASE, &p_sbase);
    :
  int itype; data->getValue(BUS_TYPE, &itype);
  if (itype == 3) {
    setReferenceBus(true);
  } elseif (…){…}
  bool lgen;
  int i, ngen, gstatus;
  double pg, qg;
  if (data->getValue(GENERATOR_NUMBER, &ngen)) {
    for (i=0; i<ngen; i++) {
      lgen = true;
      lgen = lgen && data->getValue(GENERATOR_PG, &pg,i);
          :
      if (lgen) {
        p_pg.push_back(pg);
          :
      }
    }
  }
}
```

Standard names defined in dictionary.hpp

"ngen" can be used to determine whether this bus has generators

Evaluate Y-matrix parameters on buses

```
void gridpack::powerflow::PFBus::setYBus(void)
{
  gridpack::ComplexType ret(0.0,0.0);
  std::vector<boost::shared_ptr<BaseComponent> > branches;
  getNeighborBranches(branches);
  int size = branches.size();
  int i;
  for (i=0; i<size; i++) {
    gridpack::powerflow::PFBranch *branch
      = dynamic_cast<gridpack::powerflow::PFBranch*>
        (branches[i].get());
    ret -= branch->getAdmittance();
    ret -= branch->getTransformer(this);
    ret += branch->getShunt(this);
  }
  if (p_shunt) {
    gridpack::ComplexType shunt(p_shunt_gs,p_shunt_bs);
    ret += shunt;
  }
  p_ybusr = real(ret);
  p_ybusi = imag(ret);
}
```

Loop over branches

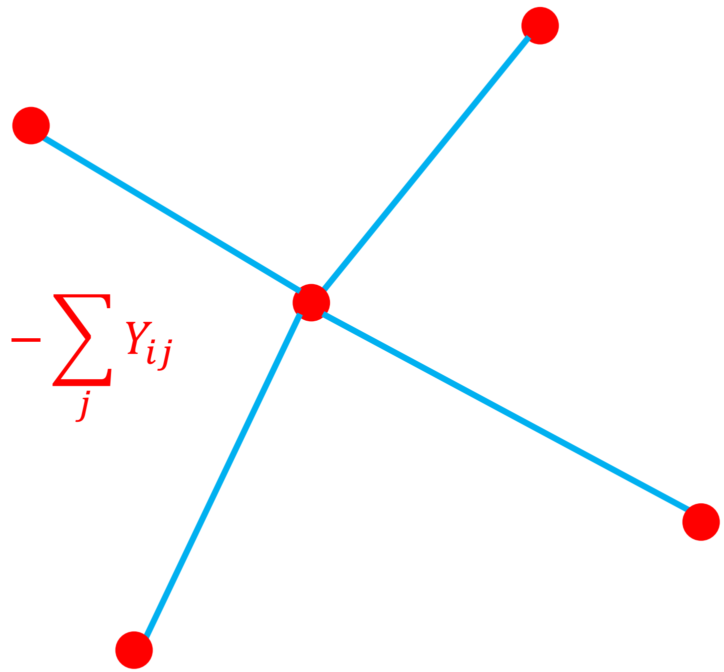
Functions defined on branches

Y-matrix components assigned to internal variables

$$Y_{ii} = -\sum_j Y_{ij}$$

Need to loop over branches attached to bus to evaluate bus contributions to Y-matrix

Diagonal Y-matrix contribution



$$Y_{ii} = -\sum_j Y_{ij}$$

Evaluate Y-matrix parameters on branches

```
void gridpack::powerflow::PFBranch::setYBus(void)
{
  gridpack::ComplexType ret(p_resistance,p_reactance);
  ret = -1.0/ret;
  gridpack::ComplexType a(cos(p_phase_shift),sin(p_phase_shift));
  a = p_tap_ratio*a;
  if (p_xform) {
    p_ybusr_frwd = real(ret/conj(a));
    p_ybusi_frwd = imag(ret/conj(a));
    p_ybusr_rvrs = real(ret/a);
    p_ybusi_rvrs = imag(ret/a);
  } else {
    p_ybusr_frwd = real(ret);
    p_ybusi_frwd = imag(ret);
    p_ybusr_rvrs = real(ret);
    p_ybusi_rvrs = imag(ret);
  }
  gridpack::powerflow::PFBus *bus1 =
    dynamic_cast<gridpack::powerflow::PFBus*>(getBus1().get());
  gridpack::powerflow::PFBus *bus2 =
    dynamic_cast<gridpack::powerflow::PFBus*>(getBus2().get());
  p_theta = (bus1->getPhase() - bus2->getPhase());
}
```

Evaluating contributions for Y_{ij} and Y_{ji}

Functions defined on <u>buses</u>

PFBus::matrixDiagSize

```
bool gridpack::powerflow::PFBus::matrixDiagSize(int *isize,
     int *jsize) const
{
  if (p_mode == Jacobian) {
    *isize = 2;
    *jsize = 2;
    return true;
  } else if (p_mode == YBus) {
    *isize = 1;
    *jsize = 1;
    return true;
  }
}
```

The Y-matrix is built as a complex valued matrix, the Jacobian is written as a real-valued matrix, hence the 2x2 blocks instead of 1x1 blocks

PFBus::matrixDiagValues

```
bool gridpack::powerflow::PFBus::matrixDiagValues(ComplexType *values)
{
  if (p_mode == YBus) {
    gridpack::ComplexType ret(p_ybusr,p_ybusi);
    values[0] = ret;
    return true;
  } else if (p_mode == Jacobian) {
    if (!getReferenceBus()) {
      values[0] = -p_Qinj - p_ybusi * p_v *p_v;
      values[1] = p_Pinj - p_ybusr * p_v *p_v;
      values[2] = p_Pinj / p_v + p_ybusr * p_v;
      values[3] = p_Qinj / p_v - p_ybusi * p_v;
      if (p_isPV) {
        values[1] = 0.0;
        values[2] = 0.0;
        values[3] = 1.0;
      }
      return true;
    } else {
      values[0] = 1.0;
      values[1] = 0.0;
      values[2] = 0.0;
      values[3] = 1.0;
      return true;
    }
  }
}
```

Note different return values depending on properties of bus

PFBranch::matrixForwardValues

```
bool gridpack::powerflow::PFBranch::matrixForwardValues(
   ComplexType *values)
{
  if (p_mode == Jacobian) {
    gridpack::powerflow::PFBus *bus1
      = dynamic_cast<gridpack::powerflow::PFBus*>(getBus1().get());
        :
    bool ok = !bus1->getReferenceBus();
    ok = ok && !bus2->getReferenceBus();
    if (ok) {
      double cs = cos(p_theta);
      double sn = sin(p_theta);
      values[0] = (p_ybusr_frwd*sn - p_ybusi_frwd*cs);
      values[1] = (p_ybusr_frwd*cs + p_ybusi_frwd*sn);
      values[2] = (p_ybusr_frwd*cs + p_ybusi_frwd*sn);
      values[3] = (p_ybusr_frwd*sn - p_ybusi_frwd*cs);
          :
      bool bus1PV = bus1->isPV();
      bool bus2PV = bus2->isPV();
      if (bus1PV & bus2PV) {
          :
      }
      return true;
    } else {
      return false;
    }
  } else if (p_mode == YBus) {
    values[0] = gridpack::ComplexType(p_ybusr_frwd,p_ybusi_frwd);
    return true;
  }
```

For the reverse values function, use –p_theta instead of p_theta

Don’t contribute anything to the matrix if one end of the branch is attached to the reference bus

Mapping Vectors

- The MatVecInterface functions vectorSize and vectorValues work in a similar way to the matrix functions except that all data structures are 1D. The vector functions are only implemented on buses
- The setValues function can be used to move values in a vector back onto buses. This is often done for the solution vector.

```
void gridpack::powerflow::PFBus::setValues(
    gridpack::ComplexType *values)
{
  p_a -= real(values[0]);
  p_v -= real(values[1]);
  *p_vAng_ptr = p_a;
  *p_vMag_ptr = p_v;
}
```

Use values in the solution vector to modify internal bus variables

Set up buffers for ghost bus and/or branch exchanges

```
int gridpack::powerflow::PFBus::getXCBufSize(void)
{
  return 2*sizeof(double);
}
```

Specify maximum amount of data to be exchanged

```
void gridpack::powerflow::PFBus::setXCBuf(void *buf)
{
  p_vAng_ptr = static_cast<double*>(buf);
  p_vMag_ptr = p_vAng_ptr+1;
  *p_vAng_ptr = p_a;
  *p_vMag_ptr = p_v;
}
```

Assign slots in the allocated buffer to internal variables and initialize exchange variables

Note: all buses exchange the same amount of data, all branches exchange the same amount of data. Not all of the exchanged data needs to be used by any one particular bus or branch.

Setting the Mode Variable

- Both bus and branch components should define an enumerated type that defines the mode. Modes in powerflow app include

```
enum PFMode{YBus, Jacobian, RHS}
```

- The bus and branch classes should define a simple modifier that sets the mode variable. Some other actions can also be taken when the mode is set

```
void gridpack::powerflow::PFBus::setMode(int mode)
{
  p_mode = mode;
}
```

Creating an Application Network

- Once the application bus and branch classes are defined, it is possible to declare a typedef for the application network. This is not strictly necessary, but it is usually a tremendous convenience

```
typedef gridpack::network::BaseNetwork<PFBus, PFBranch > PFNetwork;
```

Powerflow Factory

- Create a powerflow factory class by subclassing the BaseFactory class
- Inherit base class methods such as setComponents and setExchange

```
class PFFactory
    : public gridpack::factory::BaseFactory<PFNetwork> {
                    :
                    :
};
```

New Factory Methods

- A typical application-specific factory method would be the setYBus function
- Loop over buses and invoke bus method, then loop over branches and invoke branch method

```
void gridpack::powerflow::PFFactory::setYBus(void)
{
  int numBus = p_network->numBuses();
  int numBranch = p_network->numBranches();
  int i;

  for (i=0; i<numBus; i++) {
    dynamic_cast<PFBus*>(p_network->getBus(i).get())->setYBus();
  }

  for (i=0; i<numBranch; i++) {
    dynamic_cast<PFBranch*>(p_network->getBranch(i).get())->setYBus();
  }
}
```

Powerflow Main Application

- A small "main" program calls the powerflow application class and executes the simulation. This separates some initialization from the rest of the application and guarantees that objects exit cleanly
- An application class controls overall program flow and implements the solution algorithm
  - Read in network from external configuration file
  - Initialize network and set up matrices and vectors
  - Implement solution method
  - Export results to standard out or to files

Powerflow “main” Program

```
main(int argc, char **argv)
{
  int ierr = MPI_Init(&argc, &argv);
  gridpack::math::Initialize();
  GA_Initialize();
  int stack = 200000, heap = 200000;
  MA_init(C_DBL, stack, heap);

  gridpack::powerflow::PFApp app;
  app.execute();

  GA_Terminate();
  gridpack::math::Finalize();
  ierr = MPI_Finalize();
}
```

Initialize communication libraries (MPI and GA) and math component. Initializing math component invokes initialization routines in underlying math libraries.

Execute application

Powerflow Execution: Setting Up Network

```
gridpack::parallel::Communicator world;
boost::shared_ptr<PFNetwork> network(new PFNetwork(world));
```

Create network

```
gridpack::utility::Configuration *config
    = gridpack::utility::Configuration::configuration();
config->open("input.xml",world);
```

Open input file

```
gridpack::utility::Configuration::Cursor *cursor;
cursor = config->getCursor("Configuration.Powerflow");
std::string filename = cursor->get("networkConfiguration",
    "No network configuration specified");
```

Read in name of network configuration file

```
gridpack::parser::PTI23_parser<PFNetwork> parser(network);
parser.parse(filename.c_str());
```

Read in network

```
network->partition();
```

Partition network between processors

Powerflow Application: Using Factory to Complete Network Initialization

```
gridpack::powerflow::PFFactory factory(network);
factory.load();
factory.setComponents();
factory.setExchange();
network->initBusUpdate();
factory.setYBus();
```

Move data from DataCollection objects to bus and branch objects

Set internal indices and buffers

Set up data exchanges between buses

Evaluate Y-matrix components

Powerflow Application: Creating Mappers and Generating Matrices and Vectors

```
factory.setMode(YBus);
gridpack::mapper::FullMatrixMap<PFNetwork> mMap(network);
boost::shared_ptr<gridpack::math::Matrix> Y = mMap.mapToMatrix();
```

Y-matrix

```
factory.setSBus();
factory.setMode(RHS);
gridpack::mapper::BusVectorMap<PFNetwork> vMap(network);
boost::shared_ptr<gridpack::math::Vector> PQ = vMap.mapToVector();
```

Right hand side vector

```
factory.setMode(Jacobian);
gridpack::mapper::FullMatrixMap<PFNetwork> jMap(network);
boost::shared_ptr<gridpack::math::Matrix> J = jMap.mapToMatrix();
```

Jacobian

```
boost::shared_ptr<gridpack::math::Vector> X(PQ->clone());
```

Solution vector

Powerflow Application: Initialize Newton-Raphson Loop

```
double tolerance = 1.0e-6;
int max_iteration = 100;
ComplexType tol;
```

Set parameters

```
gridpack::math::LinearSolver solver(*J);
solver.configure(cursor);
```

Set up linear solver an set solver parameters from input file

```
int iter = 0;

X->zero();
solver.solve(*PQ, *X);
tol = PQ->normInfinity();
```

Solve linear equation and evaluate norm of solution vector

Powerflow Application: execute Newton-Raphson Iterations

```
while (real(tol) > tolerance && iter < max_iteration) {

  factory.setMode(RHS);
  vMap.mapToBus(X);
  network->updateBuses();
```

Move solution vector back to buses and exchange data between buses

```
  vMap.mapToVector(PQ);
  factory.setMode(Jacobian);
  jMap.mapToMatrix(J);
```

Create new Jacobian and right hand side vector from updated bus values

```
  X->zero();
  solver.solve(*PQ, *X);
  tol = PQ->normInfinity();
  iter++;
}
```

Resolve equations

```
factory.setMode(RHS);
vMap.mapToBus(X)
```

Push final result back onto buses

Powerflow Application: Export Results to Standard Output

```
gridpack::serial_io::SerialBusIO<PFNetwork> busIO(128,network);

busIO.header("\n   Bus Voltages and Phase Angles\n");
busIO.header("\n   Bus Number      Phase Angle");
busIO.header("      Voltage Magnitude\n");

busIO.write();
```

Write header

Write data from buses

```
        Bus Voltages and Phase Angles

        Bus Number        Phase Angle        Voltage Magnitude
                 1           0.000000              1.060000
                 2          -4.982589              1.045000
                 3         -12.725100              1.010000
                 4         -10.312901              1.017671
                 5          -8.773854              1.019514
                 6         -14.220946              1.070000
                 7         -13.359627              1.061520
                 8         -13.359627              1.090000
                 9         -14.938521              1.055932
                10         -15.097288              1.050985
                11         -14.790622              1.056907
                12         -15.075585              1.055189
                13         -15.156276              1.050382
                14         -16.033645              1.035530
```