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

Objective

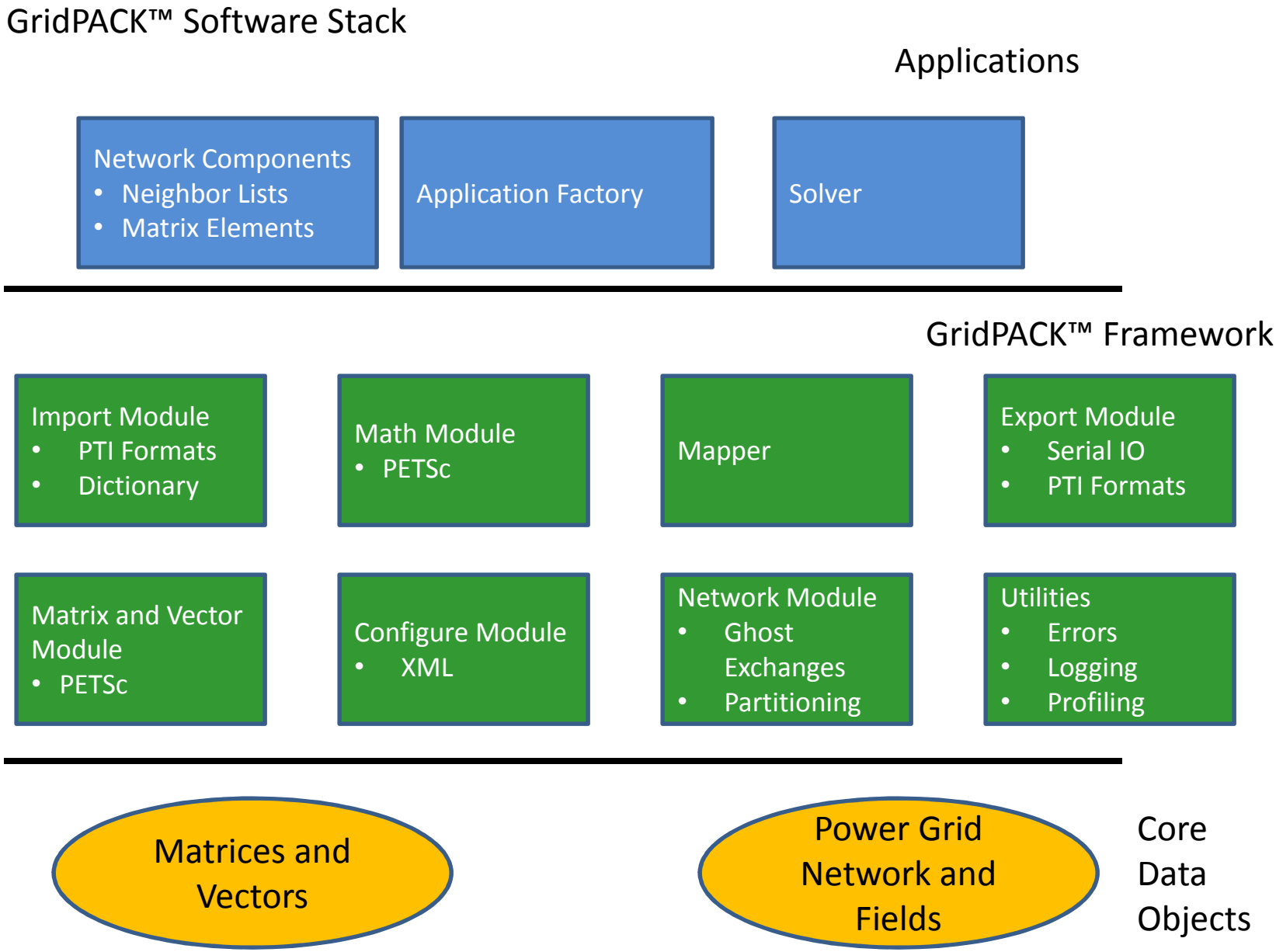
Develop a framework to support the rapid development of power grid software applications on HPC platforms

- 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

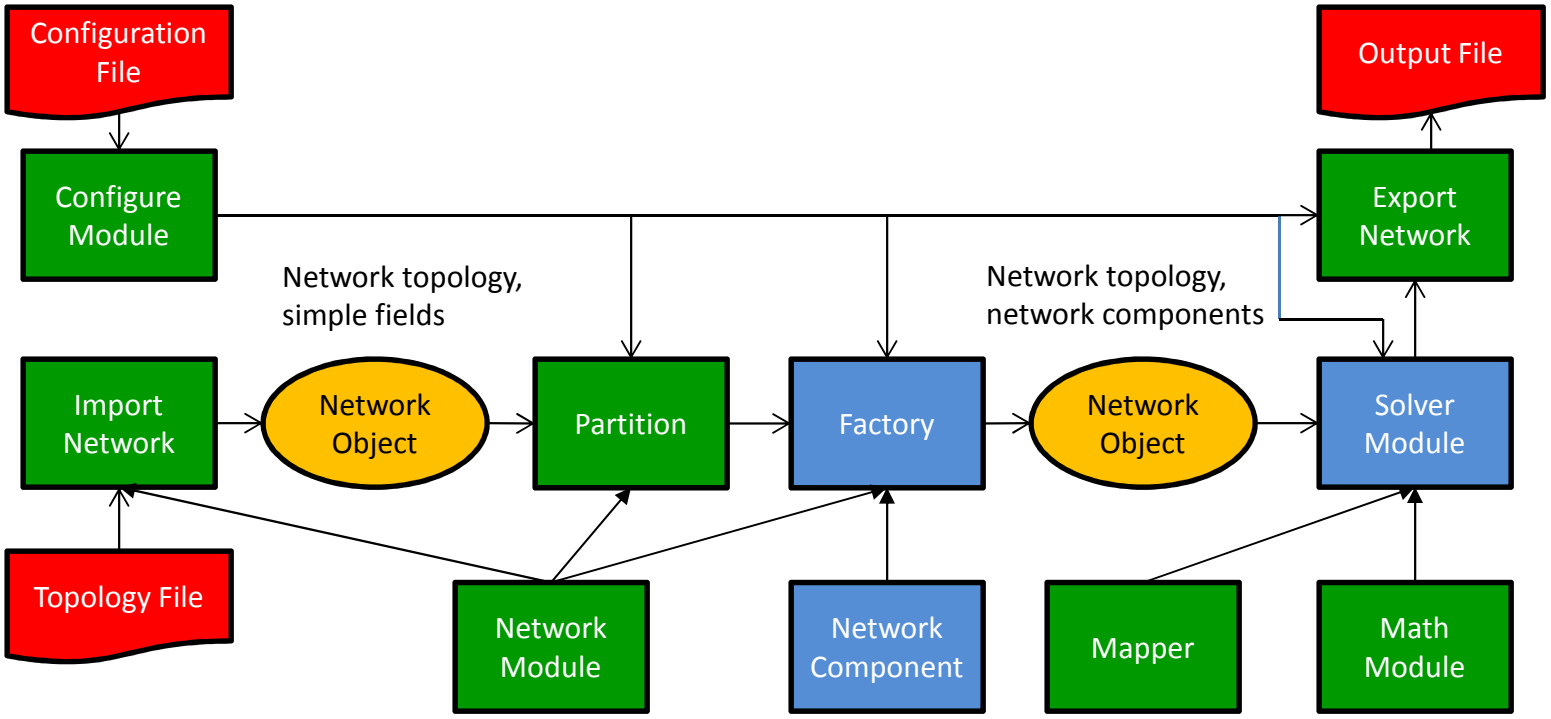
Approach

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

Application Flow Diagram



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

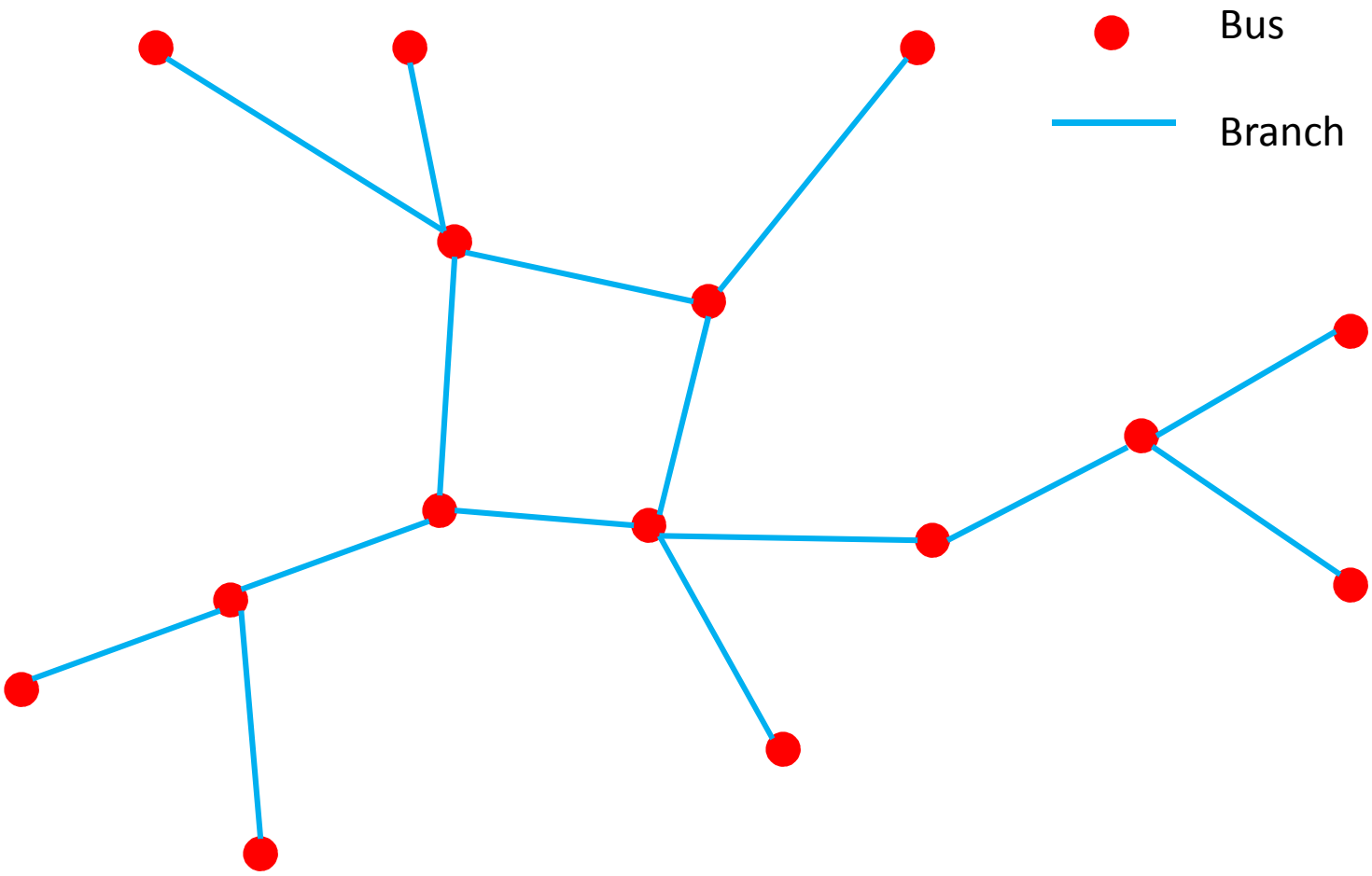
Instantiate a Network

```
#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 datak
```

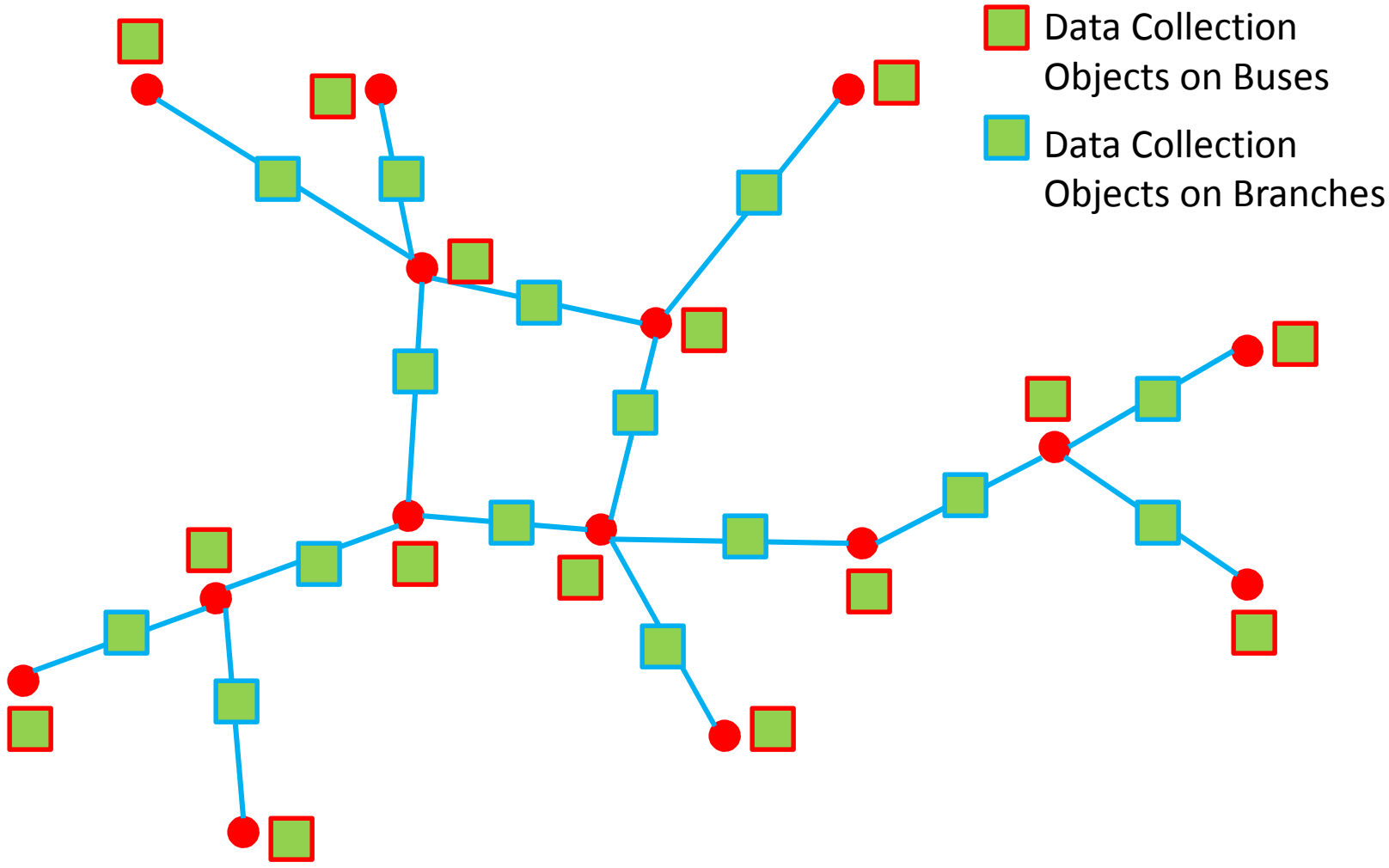
Create Network from External File

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

// 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
```

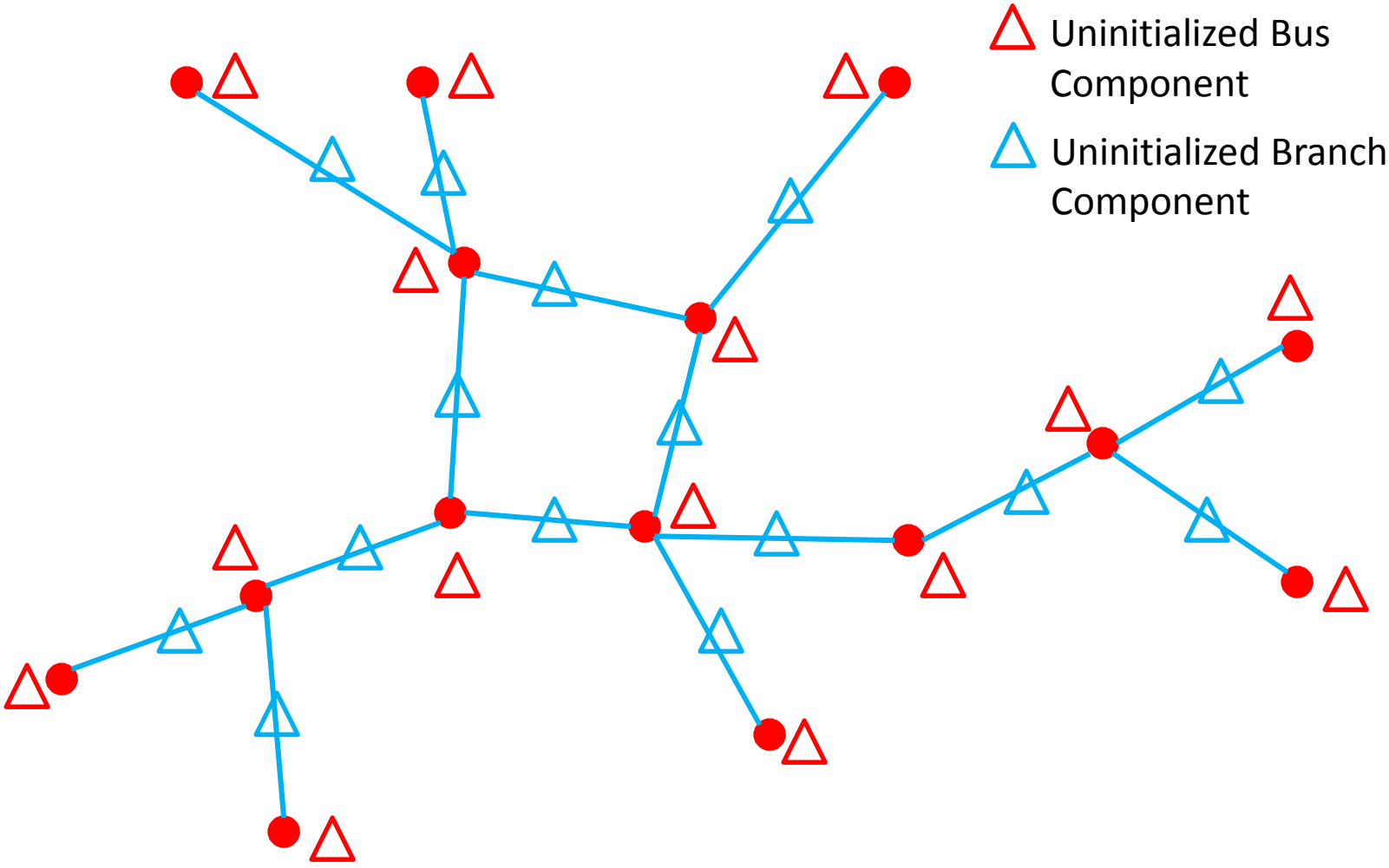
Network Topology



Network Data



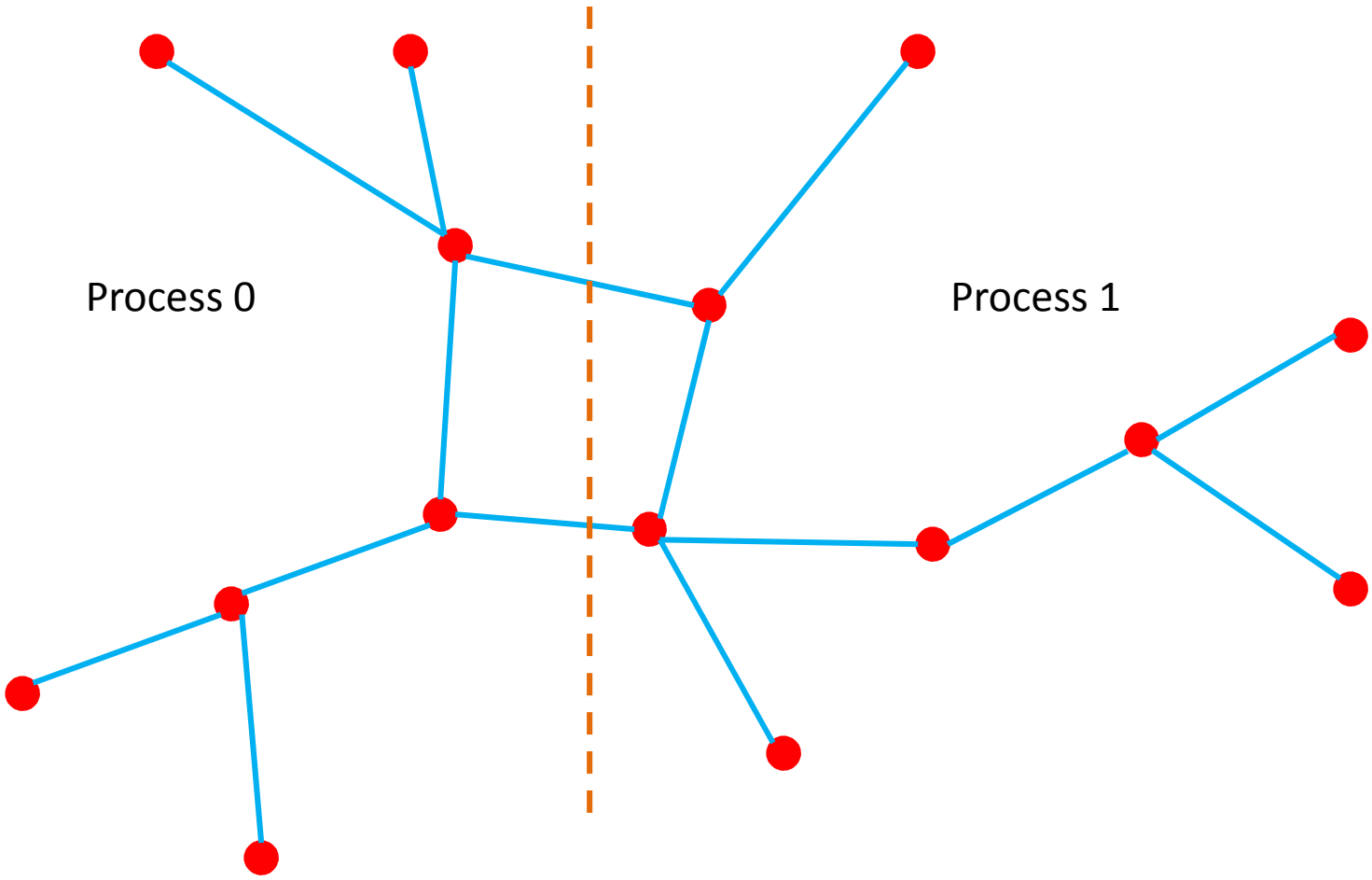
Network Components



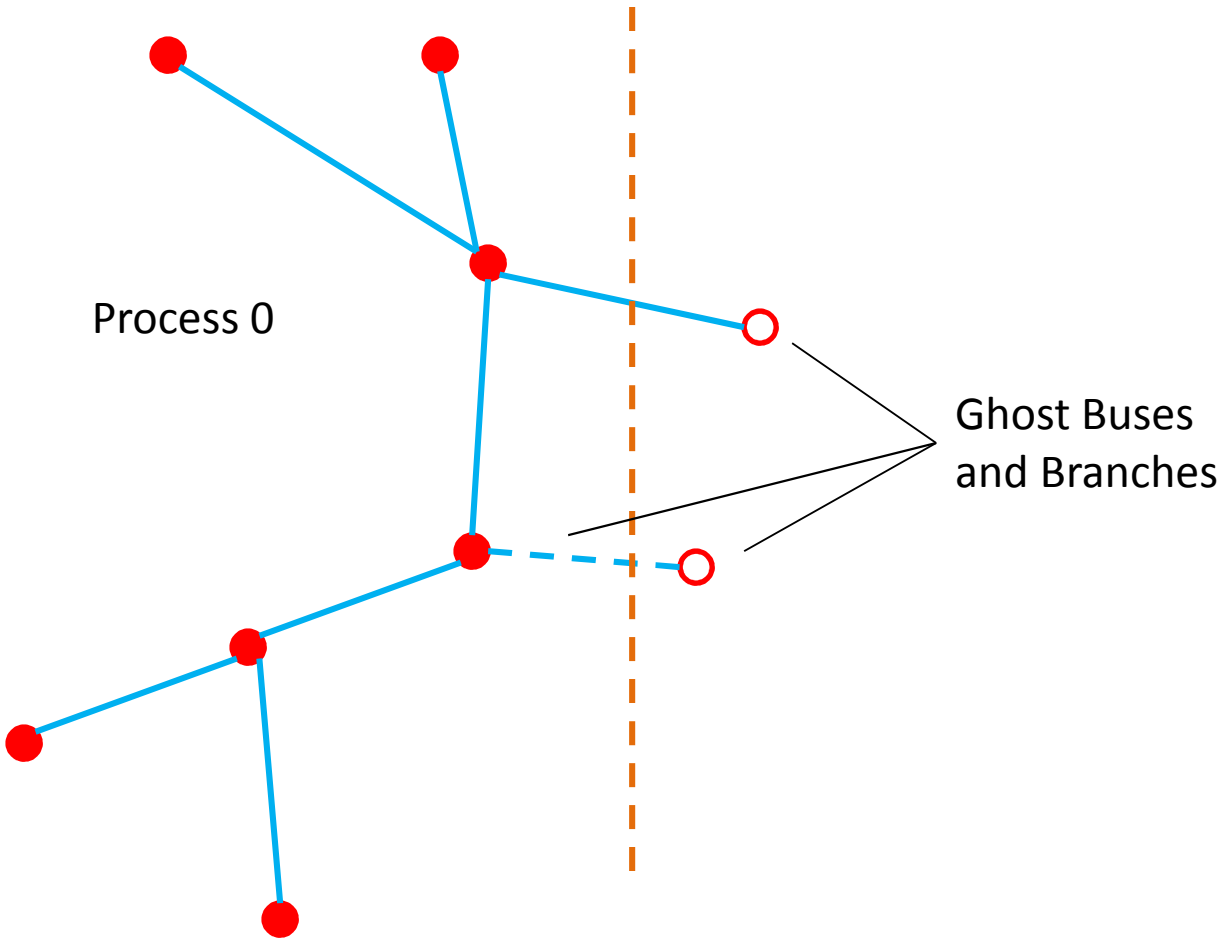
Partition Network

```
// Possibly include some partitioning options before
// invoking 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
```

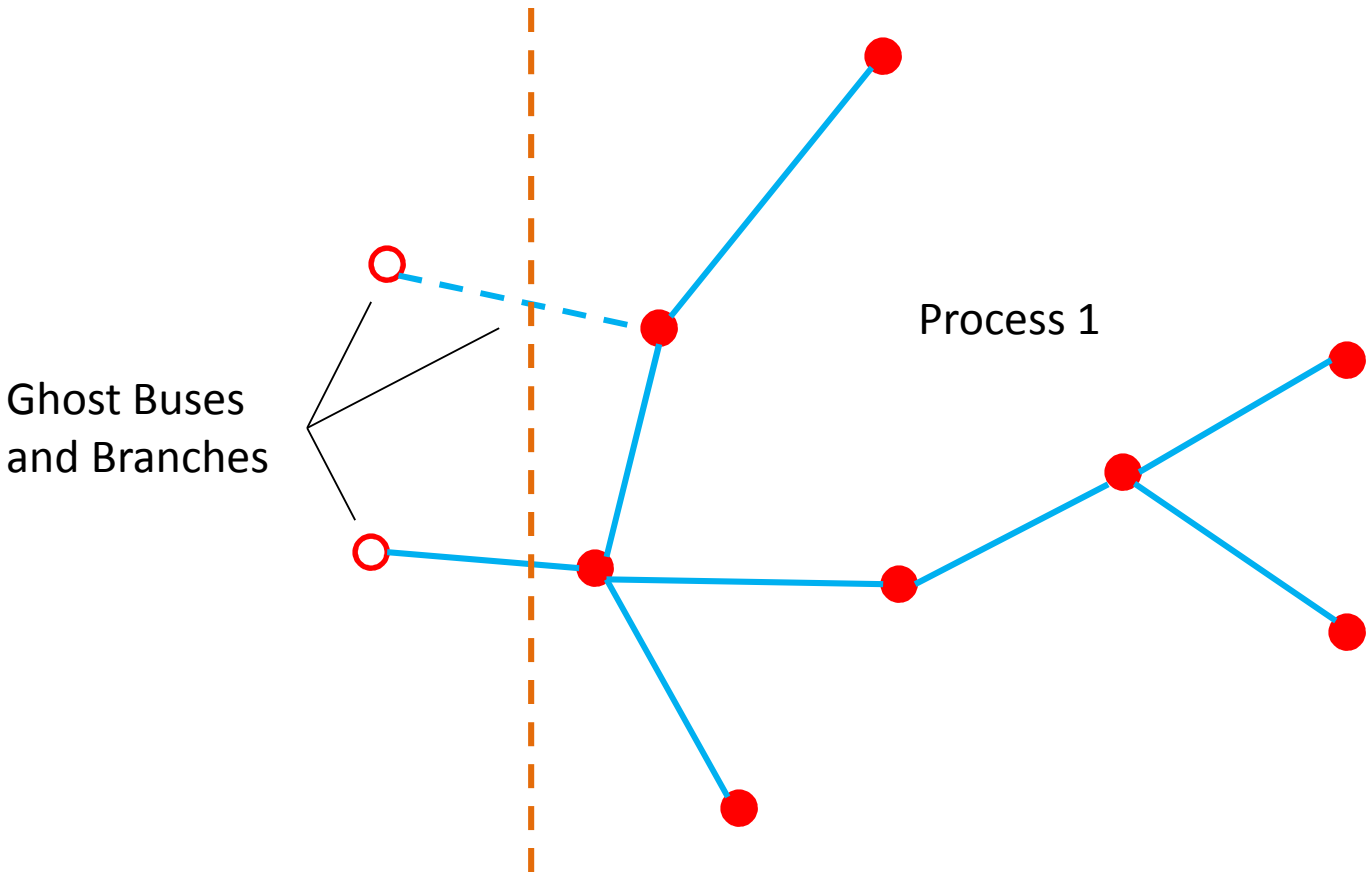
Partitioning the Network



Process 0 Partition



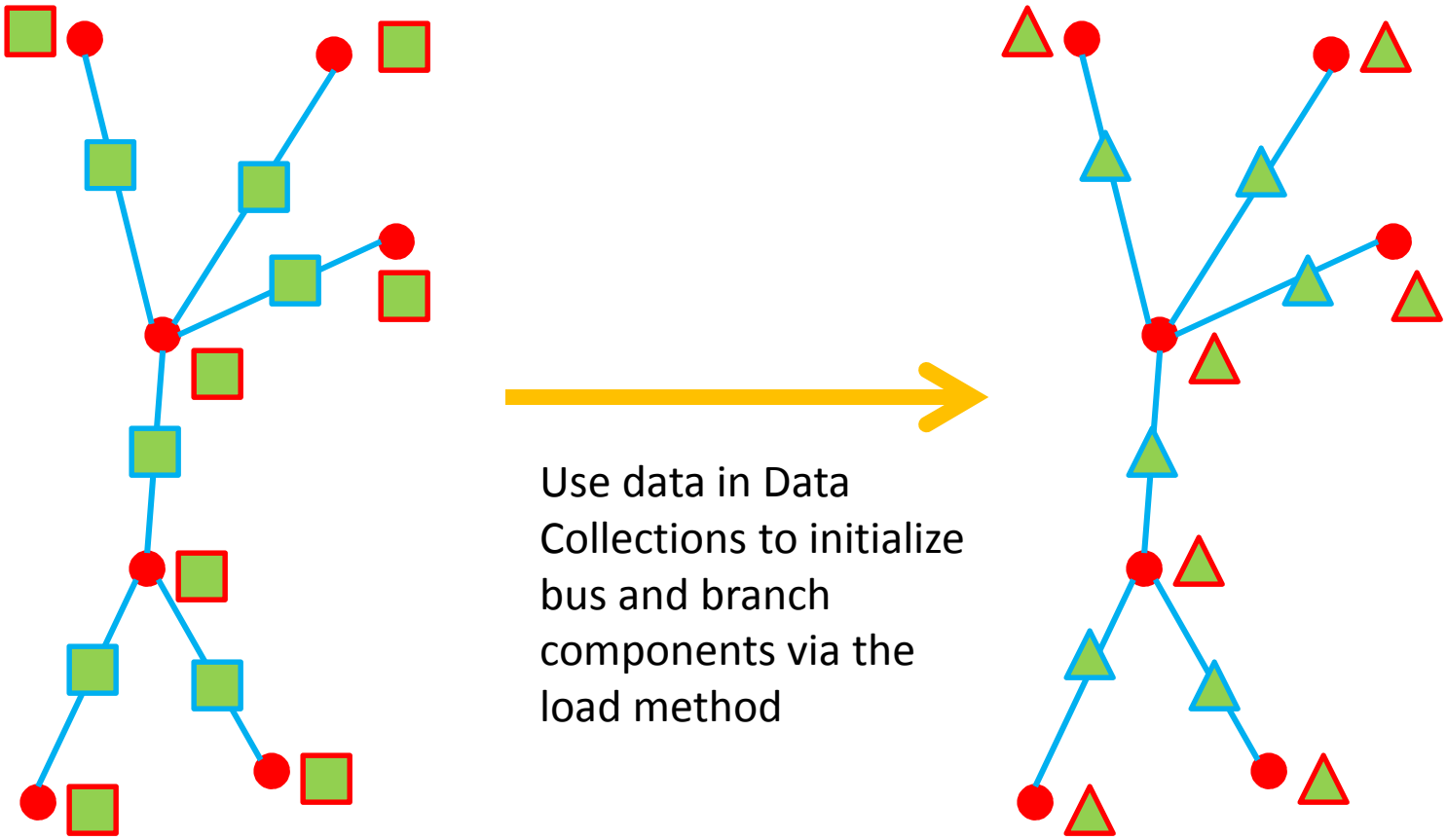
Process 1 Partition



Initialize Components

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

// 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

factory.setComponents();

// This function sets up the neighbor lists in individual
// bus and branch components so that they are pointing to
// the correct neighbors. This should be set AFTER
// partitioning the network. It also guarantees that each
// component knows its global indices
```

Initialize Components



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

Initialize Components (cont.)

```
factory.setExchange();

// This function sets 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
```

Base Network Class

```
// 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 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 correct matrices and vectors needed for solving the problem
- A primary motif in factory methods is that they loop over all bus and branch objects and invoke methods on them that set a particular state

Base Factory Class

```
// Set up lists of pointers in each component to
// neighbors of all buses and branches in the
// network
virtual void setComponents(void);

// Invoke the load method on all bus and branch
// components in the network
virtual void load(void);

// Set up exchange buffers for all bus and branch
// components in the network
virtual void setExchange(void);

// Invoke the setMode method on all bus and branch
// components in the network
virtual void setMode(int mode);
```

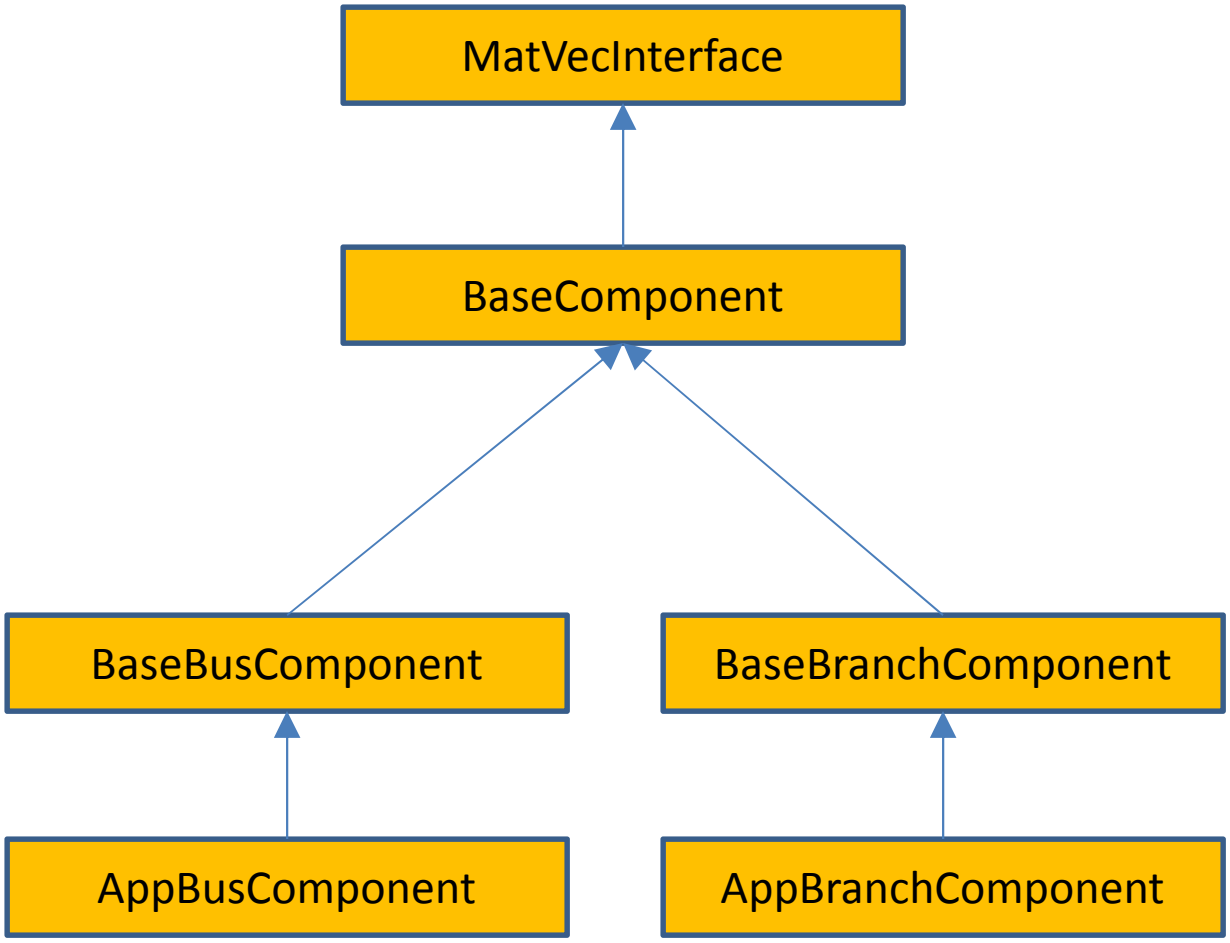
Example of a Factory Method

```
/**
 * 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::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));
  }
}
```

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 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

Diagonal MatVecInterface

```
// Return the size of matrix block on the diagonal and the
// global index of this component. 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 global location of diagonal matrix block plus
// the values of the block in row-major order. Return false
// if component does not contribute to matrix

virtual bool matrixDiagValues(ComplexType *values)
```

Off-diagonal MatVecInterface

```
// Return the size and global indices of an off-diagonal
// matrix block contributed by the component. This
// function returns false if no values are contributed by
// component.

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

// Return the global indices and 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 global index and 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 global index and 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

```
// 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)

// 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. Return false
// if no string is being returned. This functionality is
// used in the serialIO module
virtual bool serialWrite(char *string, char *signal)
```

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

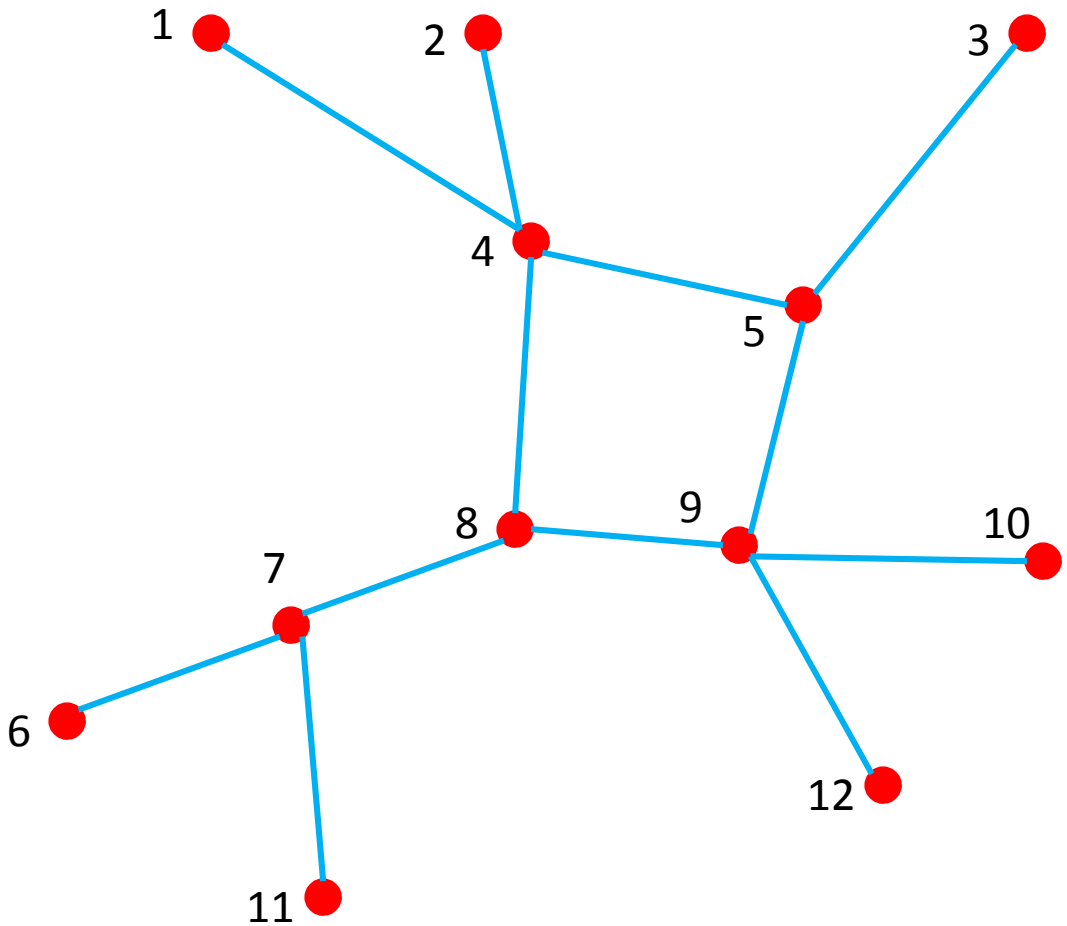
BaseBranchComponent

```
// 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
```

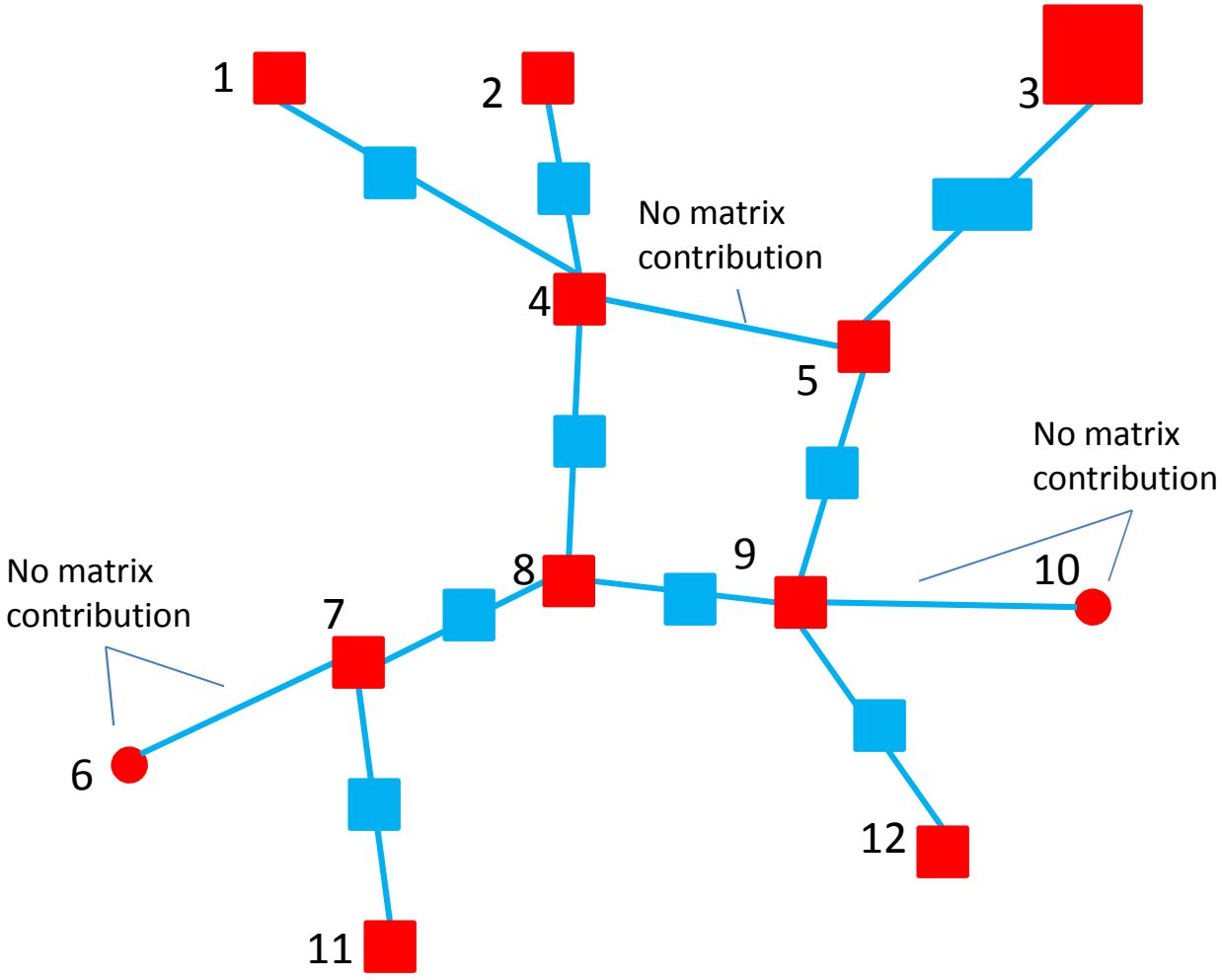
Mapper

- 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

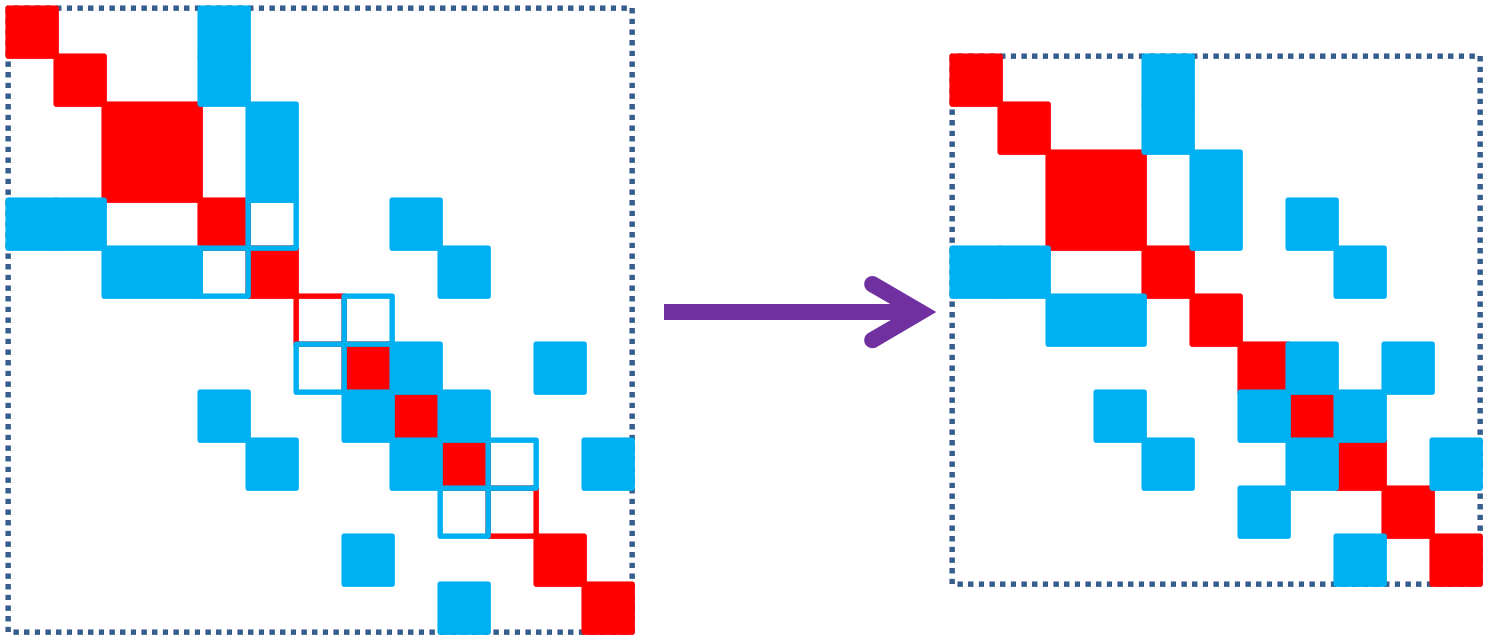
Mapper



Matrix Contributions from Components



Distribute Component Contributions and Eliminate Gaps



Mapper Interface

```
// Instantiate a new mapper that creates a matrix from
// bus and branch components on the network
void 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);

// Other classes include BranchMatrixMap and
// BusVectorMap
```

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 were
// 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 local_size(void) const;
void local_index_range(int &lo, int &hi) const;
```

Vector Class

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

// Access matrix elements
void get_element(const int &i, ComplexType &x) const;
void get_elements(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;
ComplexType norm2(void) const;
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. Also the parallel configuration

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 local_rows(void) const;
int cols(void) const;
```

Matrix Class

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

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

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

Void get_elements(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 multiply_diagonal(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 and instance of a LinearSolver
LinearSolver(const Matrix &A);
void solve(const Vector &b, Vector &x) const;
```

Serial IO

- Works in conjunction with the writeSerial operation in the BaseComponent class
- Designed to send output to standard out of the form

```
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
SerialBusIO(int len,
            boost::shared_ptr<MyNetwork> network)
void write(char *signal)

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

Using Serial IO

Use code fragment

```
if (me == 0) {
  printf("    Bus      Voltage              Generation                Load\n");
  printf("     #   Mag(pu)  Ang(deg)     P (MW)    Q (MVAr)     P (MW)    Q (MVAr)\n");
  printf(" ------------------------------------------------------------------------\n");
}
SerialBusIO busIO(256,network);
busIO.write("standard");
```

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 writeSerial method in BaseComponentClass