GridPACK™ Framework: A Brief Tutorial
Developers

Framework:
Bruce Palmer
Bill Perkins

Applications:
Yousu Chen
Shuangshuang Jin
Xinya Li
Renke Huang
Ruisheng Diao
Introduction
Why Parallel?

► Usual reasons
  ■ More flops
  ■ More memory
  ■ Run bigger problems faster

► For the power grid…
  ■ A lot of interest in running some calculations very fast (e.g. look-ahead dynamic simulation) even if calculations run on single processor
  ■ Contingency calculations can spawn thousands or even millions of individual simulations. These can all be run concurrently
  ■ Bigger calculations are on the horizon (optimization, transmission plus distribution, etc.)
GridPACK™ Overview

- A software framework written in C++ consisting of libraries and templated classes
- Designed from the outset to support parallel execution for all applications
- Communication and data decomposition is mostly hidden from developers using high level abstractions
- Provides access to high performance solver and communication libraries
- Runs on Linux-based platforms from workstations to clusters to DOE’s leadership computing facilities
GridPACK Distribution

- GridPACK source code is hosted on an externally accessible Github repository
- Source code is open source, is freely modifiable by users and can be incorporated into their code
- Webpage is available at [http://gridpack.org](http://gridpack.org)
  - Documentation on how to build GridPACK
  - Documentation on GridPACK modules and how to use them
  - Anonymously download software for current releases
  - Registered users can pull development software directly from Github
GridPACK Framework

GridPACK™ Applications
- Application Driver
- Application Factory
- Base Factory
  - Network-wide Operations
- Base Network Components
  - Neighbor Lists
  - Matrix Elements

Core Data Objects
- Matrices and Vectors
- Power Grid Network

GridPACK™ Framework
- Import Module
  - PTI Formats
  - Dictionary
- Task Manager
- Network Module
  - Exchanges
  - Partitioning
- Math and Solver Module
  - PETSc
- Configure Module
  - XML
- Mapper
- Export Module
  - Serial IO
- Utilities
  - Errors
  - Profiling

Base Factory Components
- Neighbor Lists
- Matrix Elements
GridPACK Software Stack

- Contingency Analysis
- Real Time Path Rating
- Power Flow
- State Estimation
- Dynamic Simulation
- Kalman Filter
- GridPACK
- PETSc
- Parmetis
- Boost
- Sundials
- Global Arrays
- MPI
Software Reuse

- Rename and modify existing GridPACK component or module to create a new application
- Use GridPACK components and/or modules as is
- Inherit from GridPACK component or module to create new application

User Component or Module

GridPACK Component or Module

User Component or Module
Usage Models
Current Usage Model

Currently build and run on Linux-based platforms

- Workstations, clusters and Leadership Class Facilities supported
- Applications written in C++ (Fortran interface available)

Today’s practice:

- Almost all commercial power grid simulation tools are designed to run on Windows platforms
- Most power grid engineers are familiar with Windows. Linux and Linux-like operating systems are much less familiar

How do we bridge the gap?
Usage Scenario: Repackage Vendor’s API

- Repackage vendor applications as C/C++ libraries/modules
- Example: Call TSAT API to initialize dynamic simulation
Usage Scenario: Integrated Demo

- Launch GridPACK jobs on Linux back end from Windows, display results on a web-browser
  - Minimize the learning curve for HPC and Linux
Usage Scenarios – In Progress

- Develop Windows build so that applications can run natively on PCs
- Using GridPACK algorithms as prototypes for insertion in commercial tools to leverage legacy code
  - Example: setting up a task manager in commercial tools
- Distribute libraries needed by GridPACK as Linux packages to simplify building the framework
- Make GridPACK available via the Cloud
  - Create Virtual Machine images of complete development environment that can be copied by anyone with a cloud account
  - Current working with Amazon Web Services to create machine images
Software Overview
GridPACK Objectives

- Simplify development of HPC codes for simulating power grid
- Create high level abstractions for common programming motifs in power grid applications
- Encapsulate high performance math libraries and make these available for power grid simulations
- Promote reuse of power grid software components in multiple applications to reduce development and maintenance costs
- Incorporate as much communication and indexing calculations as possible into high level abstractions to reduce application development complexity
- Compartmentalize functionality to reduce maintenance and development costs
GridPACK Software

- Framework is written in C++. A Fortran interface is available.
- Makes extensive use of object oriented programming and software templates.
- Relies on third party libraries for communication (MPI and GA), partitioning (Parmetis), distributed matrices and vectors, linear and non-linear solvers (PETSc), and advanced programming constructs (Boost).
- CMake build system that can be used to build GridPACK on workstations, clusters, and leadership class facilities (NERSC, ALCF, NCSA). Currently working on Windows build.
Building an Application in GridPACK

- Create classes that describe the behavior of buses and branches. These are derived from base bus and branch classes that define interfaces to the rest of the framework. Most of the physics is defined here.
- Create factory class to control properties of network as a whole and set network state
- Create main application module. Most of the solution algorithm and high level workflow is implemented in this module
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 and controls properties of network as a whole.

- **Mapper**: provides a general mechanism for creating distributed matrices and vectors from network components.
Major GridPACK Modules

- **Math**: Generic wrapper on top of parallel math libraries that provides functionality for creating distributed matrices and vectors. Also provides access to linear and non-linear solvers.

- **IO**: reads external files to create network and set internal control parameters and writes output to files or standard out.

- **Miscellaneous Parallel Support**
  - Task Manager
  - Collective Hash Distribution
  - Global Storage
Framework Module Interactions

- Network
- Components
- Mappers
- Math
Customizing Networks using the BaseNetwork Class

- Template class that can be created with arbitrary user-defined types for the buses and branches
  - `BaseNetwork<MyBus, MyBranch>(const Communicator &comm)`

- 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
Network Topology

Bus

Branch
Partitioning the Network
Process 0 Partition

Ghost Buses and Branches
Process 1 Partition

Ghost Buses and Branches
WECC (Western Electricity Coordinating Council) network partitioned between 16 processors
Network Commands

```cpp
// Declare network type
typedef BaseNetwork<PFBus, PFBranch> PFNetwork;

// Instantiate new network
Communicator world;
shared_ptr<PFNetwork> network(new PFNetwork(world));

// Read in external file
PTI23_parser<PFNetwork> parser(network);
pARSER.parse("network.raw");

// Partition network
network->partition();
```
Factories

- Used to manage interactions between the network and individual network components
- Perform some basic initialization functions
- 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
- Can be used to change the state of network components
- A primary motif in factory methods is that they loop over all bus and branch objects and invoke methods on them
Bus Data

p[0] Printing data for bus 0

(Integer) key: BUS_AREA value: 1
(Integer) key: BUS_NUMBER value: 1
(Integer) key: BUS_OWNER value: 1
(Integer) key: BUS_TYPE value: 3
(Integer) key: BUS_ZONE value: 2
(Integer) key: CASE_ID value: 0
(Integer) key: GENERATOR_BUSNUMBER:0 value: 1
(Integer) key: GENERATOR_IREG:0 value: 0
(Integer) key: GENERATOR_NUMBER value: 1
(Integer) key: LOAD_BUSNUMBER value: 1
(Integer) key: LOAD_NUMBER value: 1
(Integer) key: SHUNT_BUSNUMBER value: 1
(Integer) key: SHUNT_NUMBER value: 1
(String) key: BUS_NAME value: 'BUS-1'
(String) key: GENERATOR_ID:0 value: 1
(Double) key: BUS_BASEKV value: 100
(Double) key: BUS_SHUNT_BL value: 0
(Double) key: BUS_SHUNT_BL:0 value: 0
(Double) key: BUS_SHUNT_GL value: 0
(Double) key: BUS_SHUNT_GL:0 value: 0
(Double) key: BUS_VOLTAGE_ANG value: 0
(Double) key: BUS_VOLTAGE_MAG value: 1.06
(Double) key: CASE_SBASE value: 100
(Double) key: GENERATOR_GTAP:0 value: 1
(Double) key: GENERATOR_MBASE:0 value: 100
(Double) key: GENERATOR_PG:0 value: 232.4
(Double) key: GENERATOR_PMAX:0 value: 0
(Double) key: GENERATOR_PMIN:0 value: 0
(Double) key: GENERATOR_QG:0 value: -16.9
(Double) key: GENERATOR_QMAX:0 value: 99990
(Double) key: GENERATOR_QMIN:0 value: -99990
(Double) key: GENERATOR_RMPCT:0 value: 100
(Double) key: GENERATOR_RT:0 value: 0
(Double) key: GENERATOR_VS:0 value: 1.06
(Double) key: GENERATOR_XT:0 value: 0
(Double) key: LOAD_PL value: 0
(Double) key: LOAD_PL:0 value: 0
(Double) key: LOAD_QL value: 0
(Double) key: LOAD_QL:0 value: 0
(Complex) key: GENERATOR_XTRAN:0 value: (0,0)
(Complex) key: GENERATOR_ZSOURCE:0 value: (0,1)
Branch Data

p[0] Printing data for branch 0
(INTEGER) key: BRANCH_FROMBUS value: 1
(INTEGER) key: BRANCH_INDEX value: 0
(INTEGER) key: BRANCH_NUM_ELEMENTS value: 1
(INTEGER) key: BRANCH_STATUS:0 value: 1
(INTEGER) key: BRANCH_TOBUS value: 2
(INTEGER) key: CASE_ID value: 0
(BOOL) key: BRANCH_SWITCHED:0 value: 0
(STRING) key: BRANCH_CKT:0 value: BL
(DOUBLE) key: BRANCH_B:0 value: 0.0528
(DOUBLE) key: BRANCH_R:0 value: 0.01938
(DOUBLE) key: BRANCH_RATING_A:0 value: 0
(DOUBLE) key: BRANCH_RATING_B:0 value: 0
(DOUBLE) key: BRANCH_RATING_C:0 value: 0
(DOUBLE) key: BRANCH_SHIFT:0 value: 0
(DOUBLE) key: BRANCH_SHUNT_ADMTTNC_B1:0 value: 0
(DOUBLE) key: BRANCH_SHUNT_ADMTTNC_B2:0 value: 0
(DOUBLE) key: BRANCH_SHUNT_ADMTTNC_G1:0 value: 0
(DOUBLE) key: BRANCH_SHUNT_ADMTTNC_G2:0 value: 0
(DOUBLE) key: BRANCH_TAP:0 value: 0
(DOUBLE) key: BRANCH_X:0 value: 0.05917
(DOUBLE) key: CASE_SBASE value: 100
Initializing Network Components

DataCollection

load

MyBus

DataCollection

load

MyBranch
int i;
// Invoke load method on all bus objects
for (i=0; i<p_numBuses; i++) {
    p_network->getBus(i)->load(p_network->getBusData(i));
    if (p_network->getBus(i)->getReferenceBus())
        p_network->setReferenceBus(i);
}

// Invoke load method on all branch objects
for (i=0; i<p_numBranches; i++) {
    p_network->getBranch(i)->load(p_network->getBranchData(i));
}
Components

- All components are derived from the MatVecInterface, GenMatVecInterface and the BaseComponent classes.
  - The MatVecInterface classes are used to generate matrices and vectors from the network.
- Bus components are derived from the BaseBusComponent class.
- Branch components are derived from the BaseBranchComponent class.
Component Reuse

- **BaseBusComponent**
  - **Y-Matrix Bus Component**
  - **Powerflow Bus Component**

- **BaseBranchComponent**
  - **Y-Matrix Branch Component**
  - **Powerflow Branch Component**
Mapper

- Provides 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
Distribute Component Contributions and Eliminate Gaps
MatVecInterface

// Implemented on buses
virtual bool matrixDiagSize(int * isize, int * jsize) const
virtual bool matrixDiagValues(ComplexType *values)

// Implemented on branches
virtual bool matrixForwardSize(int * isize, int * jsize) const
virtual bool matrixReverseSize(int * isize, int * jsize) const
virtual bool matrixForwardValues(ComplexType *values)
virtual bool matrixReverseValues(ComplexType *values)
Powerflow Jacobian from Mapper
(1 Processor)

16351 bus
WECC system
Powerflow Jacobian from Mapper (4 Processors)

16351 bus WECC system
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
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
Input File example

```xml
<?xml version="1.0" encoding="utf-8"?>
<Configuration>
  <Powerflow>
    <networkConfiguration>IEEE14.raw</networkConfiguration>
    <maxIteration>50</maxIteration>
    <tolerance>1.0e-6</tolerance>
    <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>
```
Serial IO

Create reproducible output from head node from all buses and/or branches using functions defined in base component classes and implemented by user.

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

These lines are produced from the serialWrite method in BaseComponent class.
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
Powerflow Components

- Create two new classes to represent buses and branches, PFBus and PFBranch
  - These classes inherit from bus and branch components that are used to form the Y-matrix. These components inherit, in turn, from the BaseBusComponent and BaseBranchComponent.

- Create load methods in to initialize components from network configuration file parameters

- 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
Diagonal Y-matrix contribution

\[ Y_{ii} = -\sum_j Y_{ij} \]
Example: Evaluate Y-matrix parameters on buses

```c++
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

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

\[ Y_{ii} = - \sum_j Y_{ij} \]
Powerflow Application

- Define powerflow network using powerflow bus and branch classes
- Create powerflow factories and mappers using the powerflow networks
  - Implement application-specific methods in the powerflow factory, as needed
- Set up algebraic equations and create Newton-Raphson solver algorithm using linear solvers from math library
typedef BaseNetwork<PFBus, PFBranch> PFNetwork;

Communicator world;

shared_ptr<PFNetwork> network(new PFNetwork(world));

PTI23_parser<PFNetwork> parser(network);
parser.parse("network.raw");

network->partition();

typedef BaseFactory<PFNetwork> PFFactory;
PFFactory factory(network);
factory.load();
factory.setComponents();
factory.setExchange();

network->initBusUpdate();
factory.setYBus();

factory.setSBus();
factory.setMode(RHS);
BusVectorMap<PFNetwork> vMap(network);
shared_ptr<Vector> PQ = vMap.mapToVector();
factory.setMode(Jacobian);
FullMatrixMap<PFNetwork> jMap(network);
shared_ptr<Matrix> J = jMap.mapToMatrix();

shared_ptr<Vector> X(PQ->clone());

double tolerance = 1.0e-6;
int max_iteration = 100;
ComplexType tol = 2.0*tolerance;
LinearSolver solver(*J);

int iter = 0;

// Solve matrix equation J*X = PQ
solver.solve(*PQ, *X);
tol = X->normInfinity();

while (real(tol) > tolerance &&
     iter < max_iteration) {
    factory.setMode(RHS); vMap.mapToBus(X);
    network->updateBuses(); vMap.mapToVector(PQ);
    factory.setMode(Jacobian); jMap.mapToMatrix(J);
    solver.solve(*PQ, *X);
tol = X->normInfinity();
    iter++;
}
Applications built with GridPACK

- Powerflow
- Dynamic Simulation using Reduced Y-matrix
- Dynamic Simulation using Full Y-matrix
- State Estimation
- Kalman Filter
- Contingency Analysis (static and dynamic)
- Real-Time Path Rating
Application Modules

- Full applications that can be imbedded in larger workflows
  - Powerflow
  - State Estimation
  - Dynamic Simulation (Reduced Y-matrix)
  - Dynamic Simulation (Full Y-matrix)

- Other applications can be converted to modules as they are developed
Dynamic Simulation Initialized with State Estimation using Modules

```cpp
// setup and run state estimation calculation
boost::shared_ptr<gridpack::state_estimation::SENetwork> se_network(new gridpack::state_estimation::SENetwork(world));
gridpack::state_estimation::SEAppModule se_app;
se_app.readNetwork(se_network, config);
se_app.initialize();
se_app.readMeasurements();
se_app.solve();
se_app.write();
se_app.saveData();

// setup and run dynamic simulation calculation
boost::shared_ptr<gridpack::dynamic_simulation::DSNetwork> ds_network(new gridpack::dynamic_simulation::DSNetwork(world));
gridpack::dynamic_simulation::DSAppModule ds_app;
se_network->clone<gridpack::dynamic_simulation::DSBus, gridpack::dynamic_simulation::DSBranch>(ds_network);

// transfer results from SE calculation to DS calculation
transferSEtoDS(se_network, ds_network);

// read in faults from input file
gridpack::utility::Configuration::CursorPtr cursor = config->getCursor("Configuration.Dynamic_simulation");
std::vector<gridpack::dynamic_simulation::DSBranch::Event> faults = ds_app.getFaults(cursor);

// run dynamic simulation
ds_app.readGenerators();
ds_app.initialize();
ds_app.setGeneratorWatch();
ds_app.solve(faults[0]);
ds_app.write();
```
Task Manager

// Constructors
TaskManager()
TaskManager(Communicator &comm)

// Set number of tasks
set(int ntasks)

// Get index of next task. Return false if all tasks
// been completed
bool nextTask(int *next_id)
bool nextTask(Communicator &task_comm, int *next_id)

// Cancel tasks (all subsequent calls return false)
void cancel()
Subtasks on Processor Groups

World Group

Parallel tasks running on subgroups
Multiple Levels of Parallelism

- 8 tasks, 4 processors
- 8 processors
- 16 processors (2 levels of parallelism)
Hash Distribution

Distribution of buses on processors:
- P0: Bus 4, Bus 7, Bus 8, Bus 13, Bus 19, Bus 20
- P1: Bus 1, Bus 2, Bus 11, Bus 12, Bus 17, Bus 18
- P2: Bus 3, Bus 9, Bus 10, Bus 14, Bus 15, Bus 16
- P3: Bus 5, Bus 6, Bus 21, Bus 22, Bus 23, Bus 24

Distribution of data on processors:
- P0: 1-V1, 2-V2, 3-V3, 4-V4, 5-V5, 6-V6
- P1: 7-V7, 8-V8, 9-V9, 10-V10, 11-V11, 12-V12
- P2: 13-V13, 14-V14, 15-V15, 16-V16, 17-V17, 18-V18
- P3: 19-V19, 20-V20, 21-V21, 22-V22, 23-V23, 24-V24
Hash Distribution

// Constructor
HashDistribution<_network, _bus_data_type, _branch_data_type> (const boost::shared_ptr<MyNetwork> network)

// Distribute bus values to processors holding the buses
distributeBusValues(std::vector<int> &keys,
                      std::vector<_bus_data_type> &values)

// Distribute branch values to processors holding the branches
// holding the branches
distributeBranchesValues(
    std::vector<std::pair<int, int> > &keys,
    std::vector<int> &branch_ids,
    std::vector<_branch_data_type> &values)
Math Module
GridPACK Math API

- Provides to GridPACK parallel methods to
  - Solve systems of linear equations
  - Solve systems of nonlinear equations
  - Integrate ODE/DAE systems
- Provides parallel data structures for
  - Vectors
  - Matrices (sparse and dense)
- Independent of the rest of GridPACK and can be used on its own

#include <gridpack/math.hpp>
GridPACK Math API Design

- Build API over existing parallel (distributed memory) math libraries
  - PETSc and Trilinos are most used/general
  - Other more specialized: SuperLU, Hypre, SuiteSparse, …
  - Implement wide variety of methods
  - Well tested and maintained
  - No need to reproduce perfectly good code

- Simplified, object-oriented API
  - PETSc and Trilinos have complicated APIs
  - Perhaps sacrificing some functionality
GridPACK Math API Design

- API independent of underlying math library
  - Only one library in a single GridPACK build
- Light weight
- Support complex and real number operations regardless of support by underlying library
  - Using classes templated with element type
    ```cpp
    gridpack::math::Matrix<gridpack::RealType> A;
    gridpack::math::Matrix<gridpack::ComplexType> B;
    ```
  - Only support double precision
GridPACK Math API Design

- Allow run-time selection of available methods
  - Easily change methods without recompilation
  - Supply options to library directly (maybe)
  - Supply library-specific options through GridPACK configuration

- Completely hide underlying library
  - Calling code will not see any PETSc (e.g.) header.
  - Support multiple underlying libraries w/o change to user code
GridPACK Math API
Current PETSc Implementation

- PETSc Versions 3.0-3.7 fully supported
- Only dense and **AIJ** matrix storage schemes
- Linear Solver
  - Interface to PETSc **KSP** library
  - Any solver and pre-conditioner method selected at run-time
    - If the method has PETSc run-time options
    - If the method uses supported matrix storage schemes
  - Multiple instances (w/ different methods) can co-exist
    - Tricks PETSc options database
GridPACK Math API
Current PETSc Implementation

▶ Nonlinear Solver
  ▼ Interface to PETSc \texttt{SNES} library
  ▼ Most solver and line search methods selected at run-time
    ● Jacobian-based (Newton) methods supported
    ● Krylov methods supported (probably)
    ● Gauss-Seidel not supported
  ▼ Linear solver and pre-conditioner methods selected at run-time
▶ Simple Newton-Raphson solver implemented
  ▼ Just uses a GridPACK Linear Solver
GridPACK Math API
Current PETSc Implementation

► ODE/DAE Integrator
  ■ Interface to PETSc TS library
  ■ Time-stepping algorithms available at run-time

► Enhancements
  ■ PETSc dense matrix (Mat) using Global Arrays
  ■ Parallel dense matrix multiplication

► Specifying run-time options
  ■ GridPACK configuration module
  ■ XML format file
  ■ Solver classes each have a PETScOptions field
    ◆ List of PETSc command-line options
    ◆ Knowledge of manipulating PETSc is required
    ◆ Example
<?xml version="1.0" encoding="utf-8"?>
<GridPACK>
  <MathTests>
    <NonlinearSolver>
      <SolutionTolerance>1.0e-10</SolutionTolerance>
      <FunctionTolerance>1.0e-20</FunctionTolerance>
      <MaxIterations>100</MaxIterations>
      <PETScOptions>
        -ksp_atol 1.0e-18
        -ksp_rtol 1.0e-10
        -ksp_max_it 200
        -ksp_monitor
        -ksp_view
        -snes_monitor
        -snes_view
      </PETScOptions>
    </NonlinearSolver>
    <DAESolver>
      <PETScOptions>
        -ts_monitor
        -ts_type rosw
        -ts_max_reject 10
        -ts_max_snes_failures -1
      </PETScOptions>
    </DAESolver>
  </MathTests>
</GridPACK>
Matrices

// Matrix construction
Matrix *clone()

// Setup and initialization functions
void equate(const Matrix &a)
void identity()

// Matrix operations
void scale(const ComplexType &scale)
void multiplyDiagonal(const Vector &x)
void addDiagonal(const Vector &x)
void add(const Matrix &A)
void zero()
void conjugate()
Matrix* multiply(const Matrix &A, const Matrix &B)
Vector* multiply(const Matrix &A, const Vector &x)
Vector* transposeMultiply(const Matrix &A, const Vector &x)
Matrix* transpose(const Matrix &A)
Vectors

// Vector construction
Vector* clone()

// Vector operations
void zero()
void equate(const Vector &x)
void fill(const ComplexType &v)
ComplexType norm1() const
ComplexType norm2() const
ComplexType normInfinity() const
void scale(const ComplexType &v)
void add(const Vector &x)
void add(const Vector &x, const ComplexType &s=1.0)
void elementMultiply(const Vector &x)
void elementDivide(const Vector &x)
Linear Solvers

// Constructor for linear solver
LinearSolver(const Matrix &A)

// Configure solver
void configure(Configuration::Cursor *props)

// Solve A.x=b
void solve(const Vector &b, Vector &x)

<LinearSolver>
   <PETScOptions>
      -ksp_view
      -ksp_type richardson
      -pc_type lu
      -pc_factor_mat_solver_package superlu_dist
      -ksp_max_it 1
   </PETScOptions>
</LinearSolver>
GridPACK Math API
Future Work

- Interface to Trilinos (started)
  - Complete alternative to PETSc
- Interface to SunDIALS (started)
  - Parallel ODE/DAE integrator
  - Independent of underlying math library
  - Credit: Slaven Peles @ LLNL
Optimization
Optimization Component Class Hierarchy

- OptimizationInterface
  - MatVecInterface
  - GenMatVecInterface
  - BaseComponent
    - BaseBusComponent
    - BaseBranchComponent
  - AppBusComponent
  - AppBranchComponent
Equation-Based Interface

// Equation components
Variable
Expression
Constraint

// Example
// Declare variable
double lo=0.0; hi=5.0; x0=3.0;
shared_ptr<Variable> x(new RealVariable(lo,hi,x0));

// Constants
// Declare constants
shared_ptr<Expression> a(new RealConstant(1.0));
shared_ptr<Expression> b(new RealConstant(2.0));

// Define function f (this is not assignment)
shared_ptr<Expression> f = a*x+b;

// Define constraint
shared_ptr<Constraint> C = f <= 6.0;
Optimization Interface

// Return a list of optimization variables associated with this component
std::vector<shared_ptr<Variable> > getVariables()

// Return list of local constraints from component
std::vector<shared_ptr<Constraint> > getLocalConstraints()

// Return contribution to the objective function
shared_ptr<Expression> getObjectiveFunction()

// Return contribution from component to a global constraint
Shared_ptr<Expression> getGlobalConstraint(const char *tag)
Optimizer (Proposed)

// Create objective function
int nbus = network->getNumBuses();
int nbranch = network->getNumBranches();

int i;
shared_ptr<Expression> obj;
for (i=0; i<nbus; i++) {
    if (network->getActiveBus(i)) {
        OptimizationInterface *bus =
            dynamic_cast<OptimizationInterface*>(
            (network->getBus(i).get()));
        obj = obj + bus->getObjectiveFunction();
    }
}

// Similar loop over branches

// Gather objective function contributions from
// other processors
Optimizer Execution Model

1. Loop over network and define variables, objective function and constraints
2. Redistribute variables and equations to match input of high performance optimization software
3. Convert equations to optimizer-specific format and run optimizer

Proudly Operated by Battelle Since 1965
```cpp
std::vector<shared_ptr<Variable> > ACOPFBus::getVariables()
{
    std::vector<shared_ptr<Variable> > ret;
    ret.push_back(p_Vm_ptr);
    if (!getReferenceBus()) ret.push_back(p_Va_ptr);
    int i, ngen;
    ngen = p_Pg_ptr.size();
    for (i=0; i<ngen; i++) {
        if (p_gstatus[i] == 1) {
            ret.push_back(p_Pg_ptr[i]);
            ret.push_back(p_Qg_ptr[i]);
        }
    }
    return ret;
}
```
typedef shared_ptr<Expression> ExpPtr;
ExpPtr ACOPFBus::getObjectiveFunction(){
    ExpPtr ret;
    int ngen = p_Pg_ptr.size();
    int i;
    for (i=0; i<ngen; i++) {
        if (p_gstatus[i]==1) {
            ExpPtr c0(new RealConstant(p_cost0[i]));
            ExpPtr c1(new RealConstant(p_cost1[i]));
            ExpPtr c2(new RealConstant(p_cost2[i]));
            ExpPtr g(c0+c1*p_Pg_ptr[i]+c2*(p_Pg_ptr[i])^2);
            if (!ret) {
                ret = g;
            } else {
                ret = ret + g;
            }
        }
    }
    return ret;
}
GridPACK Application: Dynamic Simulation with Detailed Models
Dynamic Simulation Mini-Framework

Dynamic Simulation Application

DS Bus Component

Base Generator
- Base Exciter
- Base Governor

Base Generator
- Base Exciter
- Base Governor

DS Branch Component

Base Relay

Base Relay

Base Relay

GENCLS

GENROU

GENSAL

EXDC1

ESST1A

ESST4B

WSIEG1

WSHYGP

GGOV1

LVSHBL

FRQTPAT

DISTR1
Dynamic Simulation Framework

Generator Base Class
- Calculate current injection for each generator
- Calculate $\frac{dx}{dt}$ for each generator
- Update state variables for each generator

Top Level Driver
- Direct Solver: KLU, SuperLU, KSP and others
- Iterative Solver: GmRes+ILU, BiCGSTAB+ILU
- Fast Serial Solver: SuiteSparse
Generator Base Class

// load parameters
void load(shared_ptr<DataCollection> data, int idx)

// initialize generator model
void init(double Vmag, double Vang, double tstep)

// return contribution to Norton current
ComplexType I_Norton()

// calculate Norton impedance
ComplexType NortonImpedence()

// predict new state variables for time step
void predictor_currentInjection(bool flag)
void corrector(double t_inc, bool flag)

// correct state variable for time step
void corrector_currentInjection(bool flag)
void corrector(double t_inc, bool flag)
30-second dynamic simulation on BPA WECC system with classical model
Performance: Detailed Model

- 30-second dynamic simulation on BPA WECC system with detailed model
Accuracy and Speed

![Graph showing accuracy and speed comparison between TSAT and Parallel Implementation with API. The x-axis represents the number of cores, while the y-axis shows the total simulation time for t≥0. The graph compares the bus voltage near the fault location and the number of bus voltages saved.]

Legend:
- Blue line: TSAT
- Red line: Parallel Implementation with API
- Green asterisks: no bus voltages saved
- Red circles: 16K bus voltages saved