GridPACK™ Framework for Developing Power Grid Applications for HPC Platforms

Bruce Palmer
More extensive tutorials and documentation are available at [https://gridpack.org](https://gridpack.org)
GridPACK™ Development Team

- Bruce Palmer (PI): Parallel code development
- William Perkins: Parallel code development
- Yousu Chen: Power grid application development
- Shuangshuang Jin: Power grid application development
- David Callahan: Data integration
- Kevin Glass: Data integration and optimization
- Ruisheng Diao: Power grid engineering and model validation
- Stephen Elbert: Optimization and economic modeling
- Mallikarjuna Vallem: Synthetic data and model validation
- Nathan Tenney: Automatic builds and testing
- Kevin Lai: Webpage development
- Zhenyu (Henry) Huang: Program management
Advanced Grid Modeling Research

**Scope**
This activity will develop the computational and mathematical scientific advancements (for suitable application in a large-scale, dynamic, stochastic environment) needed to transform the tools and algorithms that underpin electric system planning and operations. In achieving this goal, it will also foster strategic, university-based power systems research capabilities.

**Objectives**
- **Accelerate performance** – enabling faster dynamic state estimation and analysis capabilities at a timescale consistent with data availability (e.g. sub-second level for synchrophasors)
- **Enable predictive capability** – proactively informing operator decision-making to benefit reliability through real-time measurements and improved simulations
- **Integrate model platforms** – capturing the interactions and interdependencies that allow development (and validation) of new control techniques, build strong understanding of the delicate balance between generation and load, and enable dynamic reconfiguration (of previously static assets) driven by technical and economic objectives
Open Source Library – Making Advancements Accessible

Office of Science / ASCR
(PETSc, Hypre, SUNDIALS, Minotaur, Math Center, ...)

Advanced Modeling Grid Research

Other OE Programs
(e.g. GridLAB-D)

GridPACK Library

Other mod/sim efforts

Advanced Operation
Large-scale Planning
Advanced Optimization
Dynamic Paradigm
Stochastic Analysis

Interconnection Model

EERE
(e.g. renewable int)

ARPA-E, NSF, ....

Applications

Commercialization/Users
Why GridPACK™?
Why GridPACK™?

- The power grid, despite its size and complexity, is still being modeled primarily using workstations.
- Serial codes are limited by memory and processor speed, and this limits the size and complexity of existing models.
- Modeling large systems using small computers involves substantial aggregation and approximations.
- Parallel computing can potentially increase memory and computing power by orders of magnitude, thereby increasing the size and complexity of power grid models that can be simulated using computing.
- Parallel computing is more complex than writing serial code, and the investment costs are relatively high.
- Parallel software is a rapidly changing field, and keeping up with new developments can be both expensive and time-consuming.
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
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.
- **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.
Application Flow Diagram

Configuration File
  └── Configure Module
      └── Import Network
          └── Topology File

Network Object
  ├── Partition
  │    └── Network Module
  │        └── Network Component
  │            └── Mapper
  │                └── Math Module
  └── Factory
      └── Network Object
          └── Application Solver
              └── Export Network
                  └── Output File

Network topology, simple fields
Network topology, network components
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
#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

```cpp
#include "gridpack/parser/ParserPTI.hpp"

gridpack::parser::PTI23_parser<MyNetwork> parser(network);
parser.parse("location_of_PTI_file");
```
Network Topology
Network Data

Data Collection
- Objects on Buses
- Objects on Branches
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 0

Ghost Buses and Branches
Process 1 Partition

Ghost Buses and Branches

Process 1
Partitioning of Network

WECC (Western Electricity Coordinating Council) network partitioned between 16 processors
Network Exchanges

Process 0

Bus Exchange

Branch Exchange

Process 1
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.
Initialize Components

Use data in Data Collections to initialize bus and branch components via the load method.
#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
  - The MatVecInterface class is used to generate matrices and vectors from the network
  - A new GenMatVecInterface class is being developed to handle problems that are not covered by MatVecInterface

- Bus components are derived from the BaseBusComponent class

- Branch components are derived from the BaseBranchComponent class
Component Class Hierarchy

- MatVecInterface
- BaseComponent
  - BaseBusComponent
    - AppBusComponent
  - BaseBranchComponent
    - AppBranchComponent
Component Reuse

MatVecInterface

BaseComponent

BaseBusComponent
  Y-Matrix Bus Component
  Powerflow Bus Component

BaseBranchComponent
  Y-Matrix Branch Component
  Powerflow Branch Component
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
- Defines functions used in I/O
Setting Component Mode (Load Method)

- Build Y-Matrix
- Build Powerflow Jacobian
- Build Powerflow Right Hand Side
BaseBusComponent

- Provides methods that are needed by all bus component implementations
- Keeps track of branches that are attached to the bus and buses that are attached via a single branch
- Keeps track of the reference bus
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
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 of contribution in matrix or vector and are NOT responsible for distributing matrices or vectors
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)
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)
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
Distribute Component Contributions and Eliminate Gaps
Powerflow Jacobian from Mapper
(1 Processor)

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

16351 bus
WECC system
Powerflow Jacobian from Mapper (16 Processor)

16351 bus WECC system
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.
Distributed Vector Storage

Vectors are distributed in contiguous segments between processes

Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
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);
Distributed Matrix Storage

Matrices are laid out in row blocks

- Process 0
- Process 1
- Process 2
- Process 3
- Process 4
- Process 5
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
<?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

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
<tr>
<td>43</td>
<td>0.978</td>
<td>-11.688</td>
<td>-</td>
<td>-</td>
<td>5.60</td>
<td>0.60</td>
</tr>
<tr>
<td>51</td>
<td>0.937</td>
<td>-16.934</td>
<td>-</td>
<td>-</td>
<td>63.70</td>
<td>6.40</td>
</tr>
<tr>
<td>52</td>
<td>0.940</td>
<td>-16.426</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>61</td>
<td>0.909</td>
<td>-21.810</td>
<td>-</td>
<td>-</td>
<td>23.20</td>
<td>2.30</td>
</tr>
<tr>
<td>62</td>
<td>0.905</td>
<td>-23.846</td>
<td>-</td>
<td>-</td>
<td>23.40</td>
<td>2.30</td>
</tr>
<tr>
<td>75</td>
<td>0.923</td>
<td>-18.114</td>
<td>-</td>
<td>-</td>
<td>21.30</td>
<td>2.10</td>
</tr>
</tbody>
</table>
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

```java
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(" -------------------------------------------------------------------
");
busIO.write();
```

to produce

<table>
<thead>
<tr>
<th>Bus #</th>
<th>Mag(pu)</th>
<th>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 BaseComponentClass.
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(string)
  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");
  }
  return true;
}
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
Configuration File

Configure Module

File Name

Topology File

Import Network

Topology and parameters from network file

Partition

Components

Network Object

Factory

Network is ready for computation

Solver parameters

Output

Solution

Nonlinear Solver

PQ

Vector Map

Matrix Map

Jacobian

Standard Output
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

```cpp
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();

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++;
}
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      Voltage Magnitude\n");
busIO.write();

Bus Voltages and Phase Angles

<table>
<thead>
<tr>
<th>Bus Number</th>
<th>Phase Angle</th>
<th>Voltage Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000000</td>
<td>1.060000</td>
</tr>
<tr>
<td>2</td>
<td>-4.982589</td>
<td>1.045000</td>
</tr>
<tr>
<td>3</td>
<td>-12.725100</td>
<td>1.010000</td>
</tr>
<tr>
<td>4</td>
<td>-10.312901</td>
<td>1.017671</td>
</tr>
<tr>
<td>5</td>
<td>-8.773854</td>
<td>1.019514</td>
</tr>
<tr>
<td>6</td>
<td>-14.220946</td>
<td>1.070000</td>
</tr>
<tr>
<td>7</td>
<td>-13.359627</td>
<td>1.061520</td>
</tr>
<tr>
<td>8</td>
<td>-13.359627</td>
<td>1.035530</td>
</tr>
<tr>
<td>9</td>
<td>-14.938521</td>
<td>1.055932</td>
</tr>
<tr>
<td>10</td>
<td>-15.097288</td>
<td>1.050985</td>
</tr>
<tr>
<td>11</td>
<td>-14.790622</td>
<td>1.056907</td>
</tr>
<tr>
<td>12</td>
<td>-15.075585</td>
<td>1.055189</td>
</tr>
<tr>
<td>13</td>
<td>-15.156276</td>
<td>1.050382</td>
</tr>
<tr>
<td>14</td>
<td>-16.033645</td>
<td>1.035530</td>
</tr>
</tbody>
</table>
Performance Results

▶ Applications
  - Powerflow
  - Dynamic Simulation
  - Dynamic Contingency Analysis

▶ Strong Scaling Performance
  - Fixed problem size, increasing number of processors
Powerflow Scaling for Artificial 777646 Bus Network

![Graph showing time vs. number of processors for Parsing, Partitioning, Solver, and Total.](image)
Shared Memory Effects

Solver performance for powerflow calculation on artificial 777646 bus network
Processor Configurations and Shared Memory

SMP Node

Processing Core

Shared Memory

16 processors on 2, 4, 8 nodes
Shared Memory Effects

Less memory per processor

More internode communication over network

Solver Time (seconds) vs. Shared Memory Ratio (N/P)
Dynamic Simulation

Simulation of 16351 bus WECC network
Subtasks on Processor Groups

World Group

Parallel tasks running on subgroups
Multiple Levels of Parallelism

- 8 tasks, 4 processors
- 16 processors (2 levels of parallelism)
Dynamic Contingency Analysis

Simulation of 16 contingencies on 16351 bus WECC network

2 levels of parallelism
Acknowledgments

- This work is supported by the U.S. Department of Energy (DOE) through its Advanced Grid Modeling Program.

- Computing resources were provided by Pacific Northwest National Laboratory through its PNNL Institutional Computing program

- GridPACK™ is available for download at https://gridpack.org