GridPACK$^{\text{TM}}$: A Framework for Developing Power Grid Simulations on High Performance Computing Platforms

Bruce Palmer[*]
bruce.palmer@pnnl.gov

William Perkins[*]
william.perkins@pnnl.gov

Yousu Chen[†]
yousu.chen@pnnl.gov

Shuangshuang Jin[†]
shuangshuang.jin@pnnl.gov

David Callahan[‡]
david.callahan@pnnl.gov

Kevin Glass[*]
kevin.glass@pnnl.gov

Ruisheng Diao[*]
ruisheng.diao@pnnl.gov

Mark Rice[*]
mark.rice@pnnl.gov

Stephen Elbert[*]
stephen.elbert@pnnl.gov

Mallikarjuna Vallem[*]
mallikarjun.vallem@pnnl.gov

Zhenyu (Henry) Huang[*]
zhenyu.huang@pnnl.gov

ABSTRACT

This paper describes the GridPACK$^{\text{TM}}$framework, which is designed to help power grid engineers develop modeling software capable of running on high performance computers. The framework makes extensive use of software templates to provide high level functionality while at the same time allowing developers the freedom to express whatever models and algorithms they are using. GridPACK$^{\text{TM}}$contains modules for setting up distributed power grid networks, assigning buses and branches with arbitrary behaviors to the network, creating distributed matrices and vectors and using parallel linear and non-linear solvers to solve algebraic equations. It also provides mappers to create matrices and vectors based on properties of the network and functionality to support IO and to manage errors. The goal of GridPACK$^{\text{TM}}$is to substantially reduce the complexity of writing software for parallel computers while still providing efficient and scalable software solutions. The use of GridPACK$^{\text{TM}}$is illustrated for a simple powerflow example and performance results for powerflow and dynamic simulation are discussed.

Keywords

Electric Power Grid, High Performance Computing, Software Frameworks

1. INTRODUCTION

The electric power grid has been characterized as being the largest machine in the world, but in spite of this it is still being modeled primarily on workstations running serial programs. Many other technologies (e.g. the internal combustion engine[4]) are, on the other hand, being modeled in ways that can fully exhaust the resources of the largest available computing systems. Power grid engineers have spent enormous effort and ingenuity reducing simulations of the grid to manageable sizes, but these reductions have resulted in approximations and loss of detail which may be hiding or obscuring important features and behaviors of the electric power grid. Furthermore, as more energy is derived from renewable sources, the complexity and unpredictability of the grid will increase. The influx of more information from data sources such as smart meters is also making the task of modeling even small networks more challenging. The power grid is clearly an appealing target for high performance computing (HPC) but few tools are available to assist power grid engineers interested in writing code that runs on HPC platforms.

Existing power grid modeling tools that are widely used by today's utilities are built on serial kernels, some of which represents legacy code going back decades. These codes used array-based models of programming, in spite of the heterogeneity of power grid networks. In many cases, codes have not made use of modern, object-oriented constructs, even though these would be a natural fit. Furthermore, most of the code used in power grid modeling is proprietary commercial software, so there is no access to the source code and development is not focused on creating modules that can be used in a general context. Even when access to the source code is available, it still requires significant code redesign and reconstruction to utilize HPC technologies, partly because these serial codes have been highly optimized to run on single processors. Some recent efforts have been to made to construct parallel versions of power grid simulations, but these have been single application development efforts and again, have not resulted in software that is useful across multiple applications. Recent examples of parallel power grid applications include power flow[18], contingency analysis[13, 6], state estimation[8, 6] and dynamic simulation[15].

This paper will describe the GridPACK$^{\text{TM}}$framework for developing parallel power grid simulations that run on HPC

[*]Pacific Northwest National Laboratory, Richland, WA, 99352
[†]Pacific Northwest National Laboratory, Seattle, WA
[‡]Northwest Institute for Advanced Computing, Seattle, WA

platforms with high levels of performance and scalability. Frameworks have appeared in other contexts and been used to reduce the programming burden on domain scientists by making complex but commonly used motifs available through libraries or other mechanisms. Both the Community Climate System Model[7] and Weather Research and Forecasting Model[19] are framework-based approaches to developing climate and weather models. The Cactus framework is designed to support grid-based applications and is widely used in the numerical general relativity community[10]. The Common Component Architecture[2] is a framework designed to support modularization of codes and has been used successfully in some groundwater applications[21]. Other examples of frameworks or modular approaches to code development can be found, particularly among large software projects with a broad developer base.

The GridPACK^{TM}framework is designed to allow power system engineers to focus on developing working applications from their models without getting bogged down in the details of decomposing the computation across multiple processors, managing data transfers between processors, working out index transformations between power grid networks and the matrices generated by different power applications, and managing input and output. The framework relies heavily on software templates to allow users to develop application-specific components that can then be plugged into generic modules that handle many of the more complicated book-keeping tasks. The description of network components and their contributions to algebraic equations are left to the application developer, but many other functions can be handled by the framework itself. These include creating distributed network objects, setting up distributed matrices and vectors, linear and non-linear solvers and IO. This approach provides flexibility where it is needed, in the specification of the models and equations that are to be addressed by the application, while encapsulating much of the tedious book-keeping and index calculations associated with programming in a distributed environment. The following will summarize the overall design of the GridPACK^{TM}framework and then describe the major modules and some applications that have been developed with them.

2. GRIDPACK^{TM}SOFTWARE STACK

GridPACK^{TM}is currently available as a collection of C++ modules and has taken a strongly object-oriented approach to software development. Power grids can be thought of as directed graphs with nodes (referred to as buses in power grid terminology) representing elements such as generators, loads, etc. and edges (branches in power grid terminology) representing transmission lines and transformers. The core software objects are user-developed descriptions of the properties and behavior of buses and branches. Bus and branch objects specify all the properties of the power grid and are also responsible for evaluating the contribution of the network element to whatever equations are going to be used in the power grid analysis.

The network components are derived from base classes that specify a set of virtual functions that can be implemented by the user. Depending on the application, some or all of these functions can be implemented. Those that are not implemented default to simple no-ops. In addition, a few functions are provided that are used either internally by other parts of the framework or are intended for use by developers as is. These include operations such as obtaining a list of objects that are connected to the requesting object and returning the unique integer identifier of a bus.

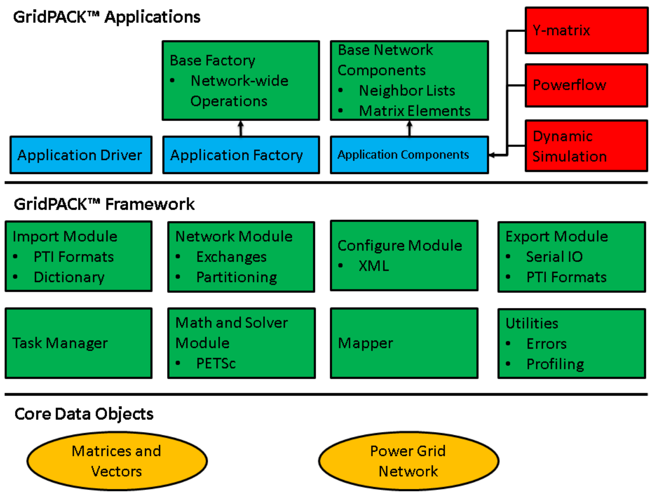

Figure 1: A schematic diagram of the GridPACK^{TM}software stack. Framework modules are colored green, user-supplied modules are colored blue. User modules that can be used across multiple applications are colored red.

The user-developed base and branch components can then be embedded in the rest of the framework and used to create different power grid applications. A schematic picture of the framework software stack is shown in Figure 1. The core data objects in most power grid applications are a representation of the power grid network and matrices and vectors that are generated by the equations describing the system. A key requirement of a power grid-oriented framework is to provide distributed representations of these objects. GridPACK^{TM}contains modules for creating both distributed networks and distributed algebraic objects. In addition, GridPACK^{TM}also supplies a set of components that can be used to map between the two.

Many of the modules in GridPACK are software templates and can be used to create application-specific instances. The network class is a template that depends on user-supplied bus and branch classes. Other modules in the framework depend on the network, so once the buses and branch classes are specified, they can be used to create application-specific versions of many of the remaining modules.

In addition to the bus and branch class, the application developer is responsible for generating two other classes. The first is a factory that is responsible for operations that run over the entire network and the second is a high level application driver that controls overall program flow and implements the solution algorithm. The user factory inherits from a base factory class that contains some important initialization functions. Additional functions usually consist of operations that trigger a method on each bus and branch.

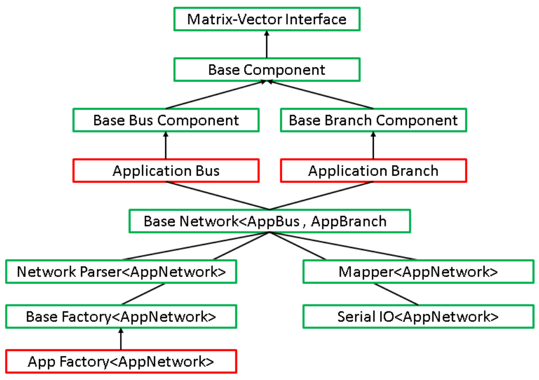

Figure 2: A schematic diagram illustrating the relationship of different GridPACK^{TM}software classes to each other. Framework classes are colored green and user-supplied supplied classes are colored blue.

The application driver instantiates the network, reads in a network configuration from an external file and runs the partitioning algorithm on it, creates the mappers that can be used to generate the matrices and vectors used in the algebraic equations that are to be solved as part of the analysis, implements the solution algorithm and writes the results to output. The actual driver code itself is fairly compact and consists primarily of calls to other modules.

A diagram showing the relationship of the basic software classes to each other is shown in Figure 2. The application developer is responsible for creating the application bus and branch classes, most of the application-specific behavior of the framework is then derived from the application buses and branches. The factory class can also be customized to support network-wide operations that are specific to a particular application. The only remaining code that requires user development is the driver, which controls the overall behavior of the application. As discussed below, this can be written at a high level and is relatively compact.

GridPACK targets three major functionalities

- Distributed graphs representing the topology of the power grid
- Distributed matrices and vectors and parallel solvers and preconditioners. The solution algorithms for power grid problems are usually expressed in terms of linear or non-linear algebraic equations.
- The mapping of objects located on the network to distributed matrices and vectors. For example, the diagonal elements of the admittance matrix are associated with buses and the off-diagonal elements are associated with branches. The mapping between the network and matrix elements can be automated to a considerable extent.

Additional functionality provided by the framework supports IO, task management, profiling, error handling, etc.

The network class manages distribution of the power grid, partitioning of the network and exchange of ghost data between processors. Ghost buses and branches represent copies of network components that are owned by other processors but are directly connected to elements on the local process. In order to update the state of local objects, it is often necessary to have current values for ghost components and this requires interprocessor communication.

The network also serves as a container for the objects that define the behavior of buses and branches in the actual power grid model. Bus and branch behaviors frequently depend on the objects immediately attached to them so that buses depend on the branches that are attached to them (and possibly on the buses attached to them via a branch) and branches depend on the buses attached at either end of the branch. Providing easy access to these attached objects is another function of the network module.

Basic algebraic objects, such as matrices and vectors, are a core part of the solution algorithms required by power grid analyses. These also tend to be large data objects that must be distributed across processors. Furthermore, the solution algorithms built around these data objects are generally the most time-consuming part of program execution, so it is necessary to ensure that the solutions are fully parallel as well. Most solution algorithms are dominated by sparse matrices but a few, such as Kalman filter analyses[17] and dynamic simulation[14], require dense matrices. Vectors are typically dense. There exists a rich set of libraries for constructing distributed matrices and vectors and these also contain preconditioner and solver capabilities. GridPACK^{TM}leverages this work heavily by creating wrappers around these libraries that can be used in solution algorithms. Wrapping these libraries instead of using them directly has two advantages 1) creating these algebraic objects can be simplified somewhat for power grid applications and 2) it allows developers to investigate new solver and algebraic libraries seamlessly, without disrupting other parts of the code. The current GridPACK^{TM}implementation is built on top of the PETSc[3] libraries but other possibilities include Hypre[9] and Trilinos[12]. All these libraries support distributed matrices and vectors, basic algebraic operations such as matrix-vector multiply, inner products, etc. and a variety of solution methods for linear and non-linear equations.

Finally, there is a need to support the generation of matrices from objects in the network and the ability to push data from solution vectors back down into network objects. This is one of the most complicated and error-prone parts of writing code, especially for parallel platforms. Much of the work involved in setting up matrices can be eliminated by having users implement a few functions that provide the individual matrix elements contributed by each bus or branch. The mapping function can then assemble these elements into a complete matrix for the entire system. The fact that developers can focus on writing code for individual matrix elements reduces the amount of programming required and fits in more intuitively with the physical models. The complicated index calculations required to evaluate the global offsets needed to set up a distributed matrix can be left to the framework.

These three capabilities are at the core of GridPACK^{TM}, but numerous additional modules are built around them. These include import modules for injesting network configuration files and using these to set up a network object, output modules that can be used to gather data from buses and branches and write them to output, a task manager that can be used to partition tasks out to either separate processors or to processor groups to support multiple levels of parallelism, a configuration module to manage input from an XML-based input deck, profiling and error handling. Many of these are also templated from the network class and some can be customized to particular networks by modifying methods in the bus and branch classes.

As already mentioned, GridPACK^{TM}is written in C++, although work on a Fortran interface is currently in progress. Communication is handled through a combination of the Message Passing Interface (MPI)[11] and Global Arrays (GA)[20] communication libraries. The matrix and solver functionality comes from the PETSc[3] libraries and network partitioning is accomplished using Parmetis[16].

3. GRIDPACK^{TM}MAPPERS

The network module is responsible for distributing the network across available processors. Figure 3 shows a partition of the Western Electric Coordinating Council (WECC) network across 16 processors. This partition results in each processor containing a relatively connected subnetwork with minimal connections to other processors. Once the network is partitioned, the problem of mapping matrix elements generated from the buses and branches becomes a formidable challenge. This is further complicated by the fact that for many applications, not all buses and branches contribute the same number of elements and some may contribute nothing at all. Assigning global indices to these elements in a consistent way that also matches the row-block partition of the distributed matrices can be a daunting challenge, even for experienced parallel programmers.

The mapper module is designed to simplify this process considerably. It works in conjunction with routines defined in the base component classes that require each bus and branch to provide a list of matrix elements contributed by that network component. For many grid applications, such as power flow and dynamic simulation, both the dependent and independent variables are associated with the buses so buses are associated with diagonal blocks and branches with off-diagonal blocks. The functions in the base network component class return the dimension of the matrix block associated with that component and the values of the matrix elements. For most applications, these blocks are relatively small, on the order of 1×1 or 2×2 for diagonal blocks. Furthermore, the values of the matrix elements contributed by any network component only depend on properties of components that are immediately connected to that component. For example, a diagonal element Y_{ii} of the Y-matrix, which is used in a great many power grid applications, can

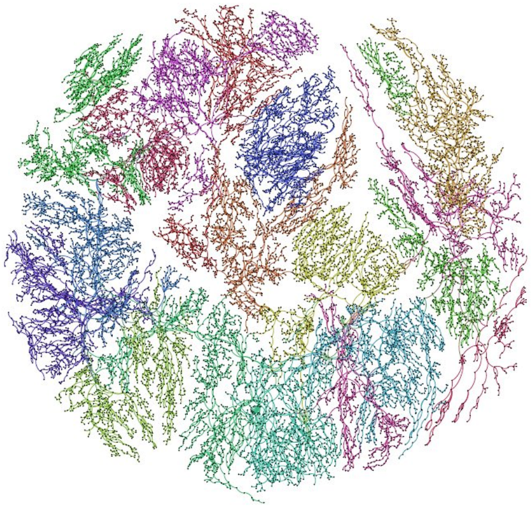

Figure 3: Partitioning of the WECC network across 16 processors. Each color represent the buses and branches associated with a different processor.

be written as a sum

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

The diagonal element Y_{ii} is evaluated on bus i and the terms Y_{ij} exist on the branches connected to bus i. The evaluation of the matrix element is relatively local and can be performed by looping over connected elements.

The way the mapper uses these elements is illustrated in Figure 4. Figure 4(a) shows a hypothetical network. Figure 4(b) shows the contributions to the matrix from all buses and branches. Note that some network components contribute nothing, and not all components contribute the same sized blocks. The mapping of the individual block from the network in Figure 4(b) to initial matrix locations based on network location is shown in Figure 4(c). This is followed in Figure 4(d) by the elimination of gaps in the matrix due to rows and columns with no values.

An example of a matrix generated using the GridPACK^{TM} mappers is shown in Figure 5. The matrix is distributed on 4 processors. The large blocks along the diagonal represent connections between buses on the same processor, matrix elements outside these blocks are generated from branches connecting buses on different processors. The block structure is the result of creating internal indices such that all the buses that are owned by a processor are indexed consecutively. The block structure should also improve the performance of preconditioners used with the solvers.

A more general matrix-vector interface has been developed that can handle systems where dependent and independent variables are associated with both buses and branches. This occurs in applications such as state estimation[1], Kalman filter analysis and market optimization. However, the generalized interface and the associated mappers still only need local information in order to implement the functions in the interface that are used to build matrices. These remain rela-

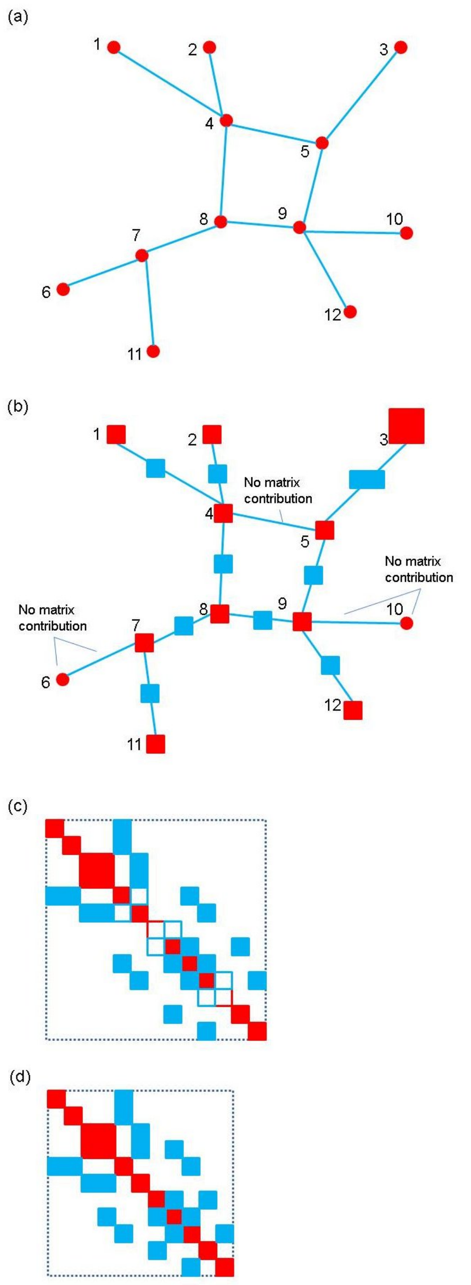

Figure 4: A schematic diagram of the matrix map function. (a) a small network (b) matrix blocks associated with branches and buses. Note that not all blocks are the same size and not all buses and branches contribute (c) initial construction of matrix based on network indices (d) final matrix after eliminating gaps The bus numbers in (a) and (b) map to approximate column locations in (c).

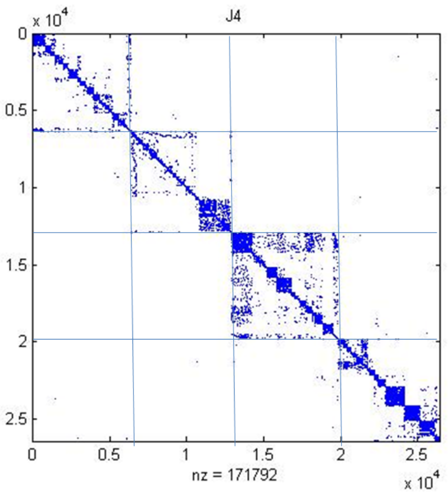

Figure 5: Fill pattern for the Jacobian matrix for a power flow calculation on the WECC network on 4 processors. The lines are a guide to the eye. The 4 blocks on the diagonal represent connections between buses on the same processor, the remaining elements are between buses on different processors

tively simple calculations and do not require complex index evaluations on the part of application developers.

4. MATH LIBRARY INTERFACE

The math module in GridPACK^{TM}relies on the PETSc libraries for its solvers and support for distributed matrices and vectors. The current math interface can be used to create both sparse and dense distributed matrices and vectors and provides access to a broad spectrum of linear and non-linear solvers. Different solvers can be accessed by using PETSc's runtime options data base, which can invoke different solvers using string arguments. These strings can be extracted from the input deck and passed through to the math module using the configuration module.

In addition to constructing matrices and vectors, the math interface supplies many basic algebraic operations. These include various types of norms (L_2, L_∞, etc.), matrix-vector multiplies, matrix transpose, scaling by a value, addition, etc. Linear and non-linear solvers are also supplied by the interface. The solver interfaces are relatively simple and most functionality is accessed by specifying options in the input deck that are passed directly to the solver. Alternatively, the interface could support different solvers and these would be accessed by instantiating different solver objects at the application level based on user input. However, this leads to a more complicated interface with likely dependencies on the underlying math library. A smaller interface that relies on runtime options is more transferable between different libraries, although it is likely that the user input file would have to change if an application was linked to a different math module implementation.

5. COMMUNICATION IN GRIDPACK^{TM}

GridPACK^{TM}relies on both MPI[11] and Global Arrays[20] for communications. To support multiple levels of parallelism and multiple-task algorithms, it was also important to guarantee that GridPACK could run on process groups beyond the world group. This meant creating a GridPACK^{TM} communicator class that contained both an MPI communicator and a GA processor group. The network class is instantiated with a communicator and this can be used to restrict the network to a subset of processors. This behavior then passes down to any objects that are instantiated from the network. Networks distributed over a subset of processors are needed for applications such as contingency analysis[5] that run many independent parallel tasks.

Many of the other modules in GridPACK^{TM}use GA for communication. The primary reason is the availability of the GA gather, scatter and scatter-accumulate calls. These functions allow random access to individual elements in a distributed array of data. The GA gather/scatter functions are one-sided and can be called from any process without a corresponding call on another process. Data consistency must be maintained by the programmer using global synchronization calls that consist of a combination of a fence, to flush out all outstanding communication, and a barrier.

The gather and scatter calls can be used to implement functions such as the ghost bus and ghost branch updates in a straightforward array. For example, each of the buses that is local to a process is given an internal index such that all the local buses on a process are indexed using a consecutive set of integers and the overall set of integers runs from 0 to $N-1$, where N is the total number of integers. A one-dimensional distributed GA array is created such that the portion of the array that is local to each process is the same size as the number of local buses. The size of the individual data elements is equal to the size of the total amount of data that each bus must exchange in a ghost bus update operation. Note that although different buses may behave in different ways, with some functionality turned on or off, the amount of data exchanged must reflect the total amount of data that *could* be needed on a remote bus. This has to be done because all data elements in a GA array must be the same size.

In the first stage of the update, the information from all local buses is scattered to the global array. This operation is fast because all data transfers are on the same process and only involve shared-memory copies. A synchronization operation is then implemented to flush out all outstanding communication and thereby guarantee that the GA is in a known state. Failing to implement a synchronization can result in a race condition where some buses receive stale data. Each process can then copy the data elements to its ghost buses using a GA gather call. The entire process is illustrated schematically in Figure 6.

The one-sided gather-scatter functionality is also used to implement the mappers. The construction of the matrices from network contributions requires a calculation to determine what the offsets in the target matrix are for the rows and columns associated with each of the locally owned buses. This information needs to be made globally available so that the off-diagonal contributions to the matrix, which come from the branches, can be properly placed in the matrix as well. The one-side gather-scatter operations, as well one-

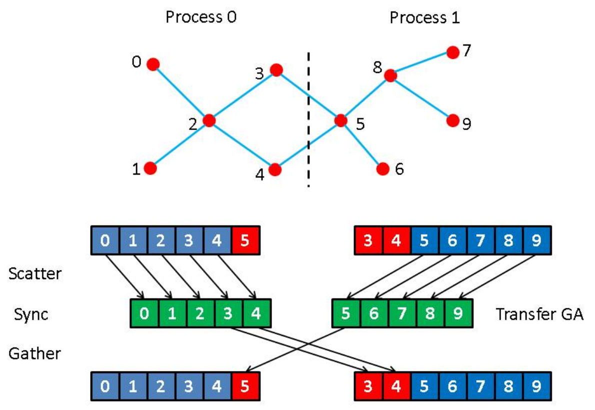

Figure 6: Schematic diagram of network bus update operation for a small network distributed on two processors. The local arrays of buses are denoted by the blue and red boxes, the global array used for transferring data is represented by the green boxes

sided get and put operations (which move blocks of data) are used to implement these calculations.

Other communication intensive parts of the code are associated with input and output. Buses are identified by a unique integer but there is no requirement on that these integers represent a monotonic consecutive sequence of integers, so there is no natural index that can be used as a starting point for organizing data. Similarly, branches are only identified by the indices of the buses at either end of the branch. This lack of initial organization in the data adds an additional layer of complexity to the problem of moving data from the input processes to the processes that actually end up holding the corresponding buses and branches. The current input and output routines use a combination of MPI and Global Arrays to map data data to the correct processes and then move it to where it can be used.

6. PROGRAMMING APPLICATIONS WITH GRIDPACK^{TM}

The top-level driver for a GridPACK^{TM}is relatively simple and comparable in complexity to scripting languages such as Python or Matlab. An example driver for a power flow application (one of the basic power grid calculations) is shown in Figure 6. Before discussing this code, it should be noted that the bulk of the effort in creating the application was in writing the bus and branch classes (in this case the PFBus and PFBranch classes). However, these classes are also the most reusable. For example, many power grid applications, including this one, require the calculation of a Y-matrix, or at least need to use its matrix elements as parameters in other calculations. Thus, a good implementation of the Y-matrix bus and branch classes can be used across many applications by inheriting from them to create bus and branch classes for more specific cases. This promotes software reuse and also makes it possible to propagate improvements or bug fixes to the Y-matrix classes easily to other applications that use it.

In Figure 1, the bus and branch classes represented by the boxes in red have all been used in more than one application.

The power flow driver shown in Figure 6 does not have many of the features that an actual application would have, including the ability to set parameters from an external input file, output of results and profiling of timing behavior, but it is sufficient to implement an actual powerflow calculation using a Newton-Raphson iterative loop to get a solution for the non-linear power flow equations. The driver starts by defining the power flow network and factory classes (lines 1 and 2). It then creates a communicator for the entire world group of processors and uses it to create a network instance (line 4). The application next creates a network configuration parser for configuration files using the PSSE/version 23 format and uses this to injest a configuration from the file "network.raw" (lines 7 and 8). The network is then distributed across processors in a form suitable for calculation by calling the network method "partition" (line 9). At this point, the network is distributed across processors and the buses and branches have been created. The parameters for each bus and branch from the network configuration file are stored in a generic data collection object.

In lines 11 through 14 a factory object is created and used to finish initializing the network. The factory load method calls a load method in the network component base class that takes the data collection object on each bus and branch as an argument. The component level load method extracts the contents of the data collection object and uses these to initialize the corresponding bus or branch. After calling the load method, the buses now have all their properties and behaviors set and can be used in computation. The setComponents method initializes internal indices that are used in the mappers. It also sets a list of pointers in the buses that point to all the neighbor branch and two pointers in the branches that point to buses at each end of the branch. This enables the buses and branches to access attached elements directly without having to go through the network object. The call to setExchange sets up the internal exchange buffers that allow ghost components to be updated with current information from remote processors.

The initBusUpdate call initializes data structures that are used in data exchanges between processors. This calculation does not need to update information on ghost branches so there is no corresponding call to initialize a branch update. The calls to the factory methods setYBus and setSBus loop over network components and call a corresponding method on each of the buses and branches. These perform internal calculations to evaluate the matrix elements of the Y-matrix and some values that are used to set up the right hand side vector in the powerflow calculation. Line 20 sets the mode to RHS. This causes an internal parameter in all the buses and branches to be set to the value RHS and controls what set of values are returned by the matrix-vector interface when a matrix mapper is instantiated (line 21). In line 22, the mapper is used to create a new distributed vector PQ that represents the right hand side of the powerflow equations. Line 24 is used to switch the mode to Jacobian. Line 25 creates a mapper that is used to create the Jacobian matrix in the powerflow equations and line 26 creates the first iteration of this matrix. Line 27 makes a copy of the right hand side vector that can be used for the solution.

Lines 29 through 33 set some control parameters and create a linear solver instance. Line 37 then solves the equations for the current values of J and PQ and returns the solution in X. The L_∞ norm is evaluated in line 38 and used to set the initial value of the tolerance variable. If this is less than the tolerance threshold, then the calculation is done, otherwise it enters the iterative Newton-Raphson loop. This is similar to the calculation of the first solution, with the exception of the call to mapToBus at line 43 and the network updateBuses at line 44. The mapToBus call pushes the values from the solution back into the network buses and updates internal bus parameters. The updateBuses call distributes these updated values to the ghost buses. The calculation then recalculates the Jacobian and right hand side vector and produces a new solution. This continues until either the convergence threshold is reached or the iteration count is exceeded.

The main feature of this code is that the solution algorithm is expressed at a fairly high level and involves abstractions representing matrices and vectors and the network. It does not require any of the internal details of these objects, so the algorithm can be written in a compact manner. Other algorithms involving the same algebraic objects could also be explored easily with minimal code development.

The results of a scaling study on the power flow code implemented using GridPACK$^{\text{TM}}$are shown in Figure 8 for an artificial power grid network consisting of 777646 buses. The results indicate reasonable scaling up to 32 processors but poor scaling in some of the setup routines (parsing the network configuration file and partitioning). Because powerflow calculations only require a single solve, external setup is a much larger factor than in many other types of numerical simulation. Work is ongoing to improve the performance of both the parsing and partitioning.

A more scalable application has been dynamic simulation, which is used to study the behavior of transients in the electric power grid[14]. This algorithm has an iterative time integration loop that amortizes the setup costs. The scaling results are shown in Figure 9 for the WECC network. This is a much smaller network than the artificial problem used in the powerflow calculation and only contains 16351 buses. The algorithm has a dense matrix solve at the start and then a sequence of matrix transpose-vector multiplies in the integration loop. The initial dense linear solve is moderately scalable (up to about 8 processors) and the transpose-multiply more so. The partitioner is not scaling but only begins to contribute significantly to the overall performance at large core counts. Scaling is seen out to 32 processors but there is still a slight decrease all the way to 64 processors.

Overall, reasonable levels of scaling are seen for the powerflow and dynamic simulation applications, but the relatively small size of today's power grid problems makes it difficult to achieve high levels of scalability. However, as HPC becomes more accessible to the power grid community, we expect to see an increase in the size and complexity of the models being studied. There is also a great deal of current interest in contingency analysis simulations, where a large number

```
typedef BaseNetwork<PFBus,PFBranch> PFNetwork;
typedef BaseFactory<PFNetwork> PFFactory;
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;
LinearSolver solver(*J);

int iter = 0;

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

Figure 7: . Top-level driver for a powerflow application using GridPACK^{TM}.

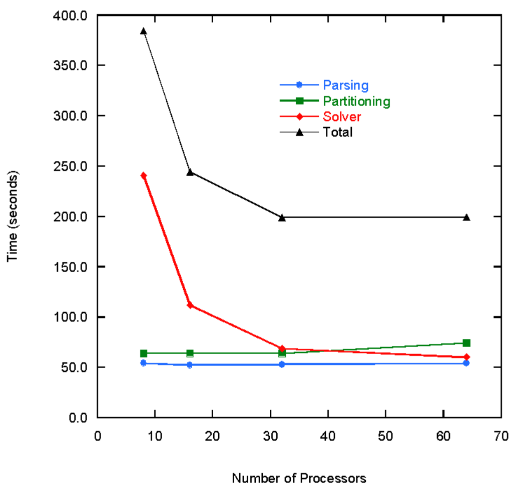

Figure 8: Strong scaling results for a powerflow calculation using on an artificial network containing 777646 buses

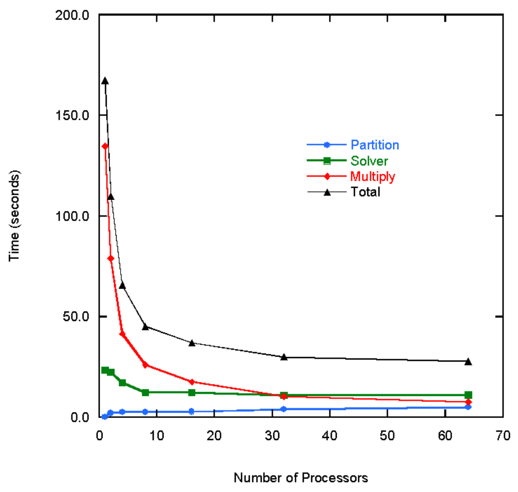

Figure 9: Strong scaling results for a dynamic simulation calculation using on the WECC network

of configurations of the grid are evaluated in separate, independent calculations. These are inherently parallel and can be scaled to very large numbers of processors.

Beyond conventional scaling to larger size problems, industry is also interested in using computation in the context of real-time control to make operational decisions about running the electric power grid. This requires reducing the time to solution for fixed sized problems. Again, there are many challenges. Most conventional HPC applications have relatively short setup phases and extended computational phases that can be used to amortize the cost of distributing and initializing the calculation. This is not the case for many power grid applications and as seen in the case of the powerflow calculation, the cost of input, setup, initialization and output can be a major fraction of the total run time. Thus, reducing overall computational time requires reduction in all phases of the computation, instead of focusing on only a few parts of it. Many of these bottlenecks tend to cluster around the areas of input and output and are aggravated by the relatively unstructured nature of the data. Mapping the data associated with particular buses and branches to the actual objects corresponding to those buses and branches tends to be an exercise in distributed hashing that is both complicated in itself and a source of significant communication overhead.

7. CONCLUSIONS

This paper has described a framework for creating power grid applications that run on HPC platforms. Most of the parallel programming has been embedded in high level abstractions and the application developer is left to write the portions of the code that express the actual detailed models and the equations that describe them. The GridPACK^{TM} framework has, to date, been used to develop a number of power grid applications, These include powerflow, dynamic simulation, and contingency analysis based on both powerflow and dynamic simulation calculations. A state estimation application is currently in development and applications based on Kalman filter analysis and optimization are planned in the next year. The existing applications have demonstrated that GridPACK^{TM}has the flexibility to implement a range of calculations using a relatively small number of modules with reasonable levels of scalability. As the scale of power grid applications increases, we anticipate that the performance gains from using parallel computing will increase as well.

GridPACK^{TM}is distributed under a BSD open-source license and is available for download at https://gridpack.org.

8. ACKNOWLEDGMENTS

We wish to thank Mahantesh Halappanavar for the visualization of the WECC partition in Figure 3. Funding for this work was provided by the U.S. Department of Energy's Office of Electricity through its Advanced Grid Modeling Program. Additional funding was provided by the Future Power Grid Initiative at Pacific Northwest National Laboratory through the Laboratory Directed Research and Development program. Pacific Northwest National Laboratory is located in Richland, WA and is operated by Battelle Memorial Institute under contract DE-AC05-76RLO1830 with the U.S. Department of Energy.

9. REFERENCES

[1] A. Abur and A. Gómez-Expósito. *Power System State Estimation: Theory and Implementation*. Marcel Dekker, Inc., New York, 2004.

[2] B. Allan, R. Armstrong, D. Bernholdt, K. Chiu, T. Dahlgren, K. Damevski, W. Elwasif, T. Epperly, D. Katz, J. Kohl, M. Krishnan, J. Larson, S. Lefantzi, M. Lewis, A. Malony, L. McInnes, J. Nieplocha, B. Norris, S. Parker, J. Ray, S. Shende, T. Windus, and S. Zhou. A component architecture for high performance scientific computing. *Int. J. High Perf. Comput. Applications*, 20(2):163–202, Summer 2006.

[3] S. Balay, J. Brown, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. McInnes, B. Smith, and H. Zhang. PETSc Web page, 2013. http://www.mcl.anl.gov/petsc.

[4] J. Chen. http://exactcodesign.org.

[5] Y. Chen, Z. Huang, and D. Chavarria-Miranda. Performance evaluation of counter-based dynamic load balancing schemes for massive contingency analysis with different computing environments. In *IEEE Power and Energy Society General Meeting PESGM*. IEEE, June 2003.

[6] Y. Chen, S. Jin, M. Rice, and Z. Huang. Parallel state estimation with practical data. In *Power and Energy Society General Meeting (PES)*, pages 1–5. IEEE, July 2013. doi:10.1109/PESMG.2013.667242.

[7] A. Craig, M. Vertenstein, and R. Jacob. A new flexible coupler for earth system modeling developed for CCSM4 and CESM1. *Int. J. High Perf. Comput. Applications*, 26(1):5–16, February 2012.

[8] D. Falcao, F. Wu, and L. Murphy. Parallel and distributed state estimation. *IEEE Trans. Power Syst.*, 10:724–730, 1995.

[9] R. Falgout, J. Jones, and U. Yang. Conceptual interfaces in *hypre*. *Fut. Gen. Comput. Systems – Int. J. Grid Comput. Theor. Meth. Applications*, 22(1–2):239–251, January 2006. Special issue on PDE software. Also available as LLNL technical report UCRL-JC-148957.

[10] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Lecture Notes in Computer Science*, pages 197–227. 5th International Conference on High Performance Computing for Computational Science (VECPAR 2002), June 2003.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1999.

[12] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, September 2005.

[13] Z. Huang, Y. Chen, and J. Nieplocha. Massive contingency analysis with high performance computing. In *Proc. IEEE Power Energy Society General Meeting*. IEEE, 2009.

[14] Z. Huang, S. Jin, and R. Diao. Predictive dynamic

simulation for large-scale power systems through high-performance computing. In *The 2nd International Workshop on High Performance Computing, Networking and Analytics for the Power Grid*. International Conference High Performance Computing Networking Storage and Analysis, SC12, November 2012.

[15] S. Jin, Z. Huang, R. Diao, D. Wu, and Y.Chen. Parallel implementation of power system dynamic simulation. In *Power and Energy Society General Meeting (PES)*, pages 1–5. IEEE, July 2013. doi:10.1109/PESMG.2013.667265.

[16] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel and Distr. Comput.*, 48:71–95, 1998.

[17] Y. Li, Z. Huang, N. Zhou, B. Lee, R. Diao, and P. Du. Application of ensemble kalman filter in power system state tracking and sensitivity analysis. In *Proceedings of the 2012 IEEE Power and Energy Society Transmission and Distribution Conference and Exposition*. IEEE, May 2012.

[18] P. Luo, F. Yang, and M. Rao. A parallel approach to computing load flow equations. *Dyanmics of Continuous, Discrete and Impulsive Systems, Series B: Applications and Algorithms*, 11:229–238, 2004.

[19] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The weather research and forecasting model: Software architecture and performance. In *Use of High Performance Computing in Meteorology*, pages 156–168. 11th WOrkshop on the Use of High Performance Computing in Meteorology, October 2005.

[20] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perf. Comput. Applications*, 20(2):203–231, Summer 2006.

[21] B. Palmer, V. Gurumoorthi, A. Tartakovsky, and T. Scheibe. A component-based framework for smoothed particle hydrodynamics simulations of reactive fluid flow in porous media. *Int. J. High Perf. Comput. Applications*, 24(2):228–239, Summer 2010.