Application of PGAS Programming to Power Grid Simulation

Bruce Palmer
Pacific Northwest National Laboratory
Richland, WA 99352
Email: bruce.palmer@pnnl.gov

***Abstract*—This paper will describe the application of the PGAS Global Arrays (GA) library to power grid simulations. The GridPACK™framework has been designed to enable power grid engineers to develop parallel simulations of the power grid by providing a set of templates and libraries that encapsulate most of the details of parallel programming in higher level abstractions. The communication portions of the framework are implemented using a combination of message-passing (MPI) and one-sided communication (GA). This paper will provide a brief overview of GA and describe in detail the implementation of collective hash tables, which are used in many power grid applications to match data with a previously distributed network.**

***Index Terms*– Parallel Algorithms, Table Lookup, Parallel Programming, Power Engineering Computing**

1. Introduction

Althought the power grid is central to modern life in industrialized countries and has been called the largest machine in the world, it is still mostly modeled using serial codes running on workstations. While parallel computing has begun to infiltrate power grid simulations, the application of high performance computing (HPC) to power grid problems is still relatively new and the field as a whole is only just beginning to utilize the capabilities of advanced computing architectures. As more parallel power grid applications have been developed the computational challenges of simulating the power grid on parallel architectures have also come into clearer focus. Models of power grid networks themselves remain fairly small from the perspective of HPC. Typical transmission network models contain between 15,000 and 60,000 nodes (referred to as buses in power grid terminology). The connectivity of each node is relatively low, so the number of edges (referred to as branches in power grid terminology) is comparable to the number of nodes. In spite of these small sizes, there are a number of areas where HPC is applicable. The first is in real-time simulation, where the goal is to simulate a dynamic event in less time than is required for the event to actually take place and the second area is contingency analysis where the goal is to simulate a large number of possible scenarios in a short time. Furthermore, there is currently a great deal of interest in combining simulation of transmission networks with simulations of distribution networks, which can potentially result in much larger networks (on the order of millions or more buses). There are also optimization problems associated with the power grid that have substantial possibilities for generating very large sets of equations.

The GridPACK™framework is designed to simplify the development of parallel simulations of the power grid by providing high level abstractions that eliminate most the need for explicitly programming in a distributed context. Several parts of GridPACK are implemented using Partitioned Global Address Space (PGAS) programming constructs, which are implemented as part of the Global Arrays communication library [1]. These include ghost exchanges of network information, task managers, IO modules, and distributed hash tables. These modules are written using a combination of 1-dimensional distributed arrays, gather, scatter and scatter-accumulate functions, and the atomic read-increment operations. An important component of these algorithms is that data in a global array is visible to the entire system and readily accessed by any processor.

A schematic of the GridPACK software stack is shown in Figure 1. The green boxes represent functionality supplied by the gridpack software framework and the blue boxes represent code written by the application developer. All of these components are parallel and many of them use PGAS functionality. GridPACK itself is written in C++ and makes extensive use of software templates to construct applications. Networks are represented by a template class that takes user defined bus and branch classes as template parameters. The network object is then used as a template parameter to many of the other GridPACK classes, most of these are then essentially customized to a specific application via the implementation of functions in the base bus and branch class interfaces.

The following sections will describe a number of areas in GridPACK that are implemented using GA. Section 2 provides a brief overview of the GA library, section 3 describes the implementation of collective hash tables, section 4 provides some performance results, and section 5 presents conclusions.

978-1-5090-5214-1/16 $31.00 © 2016 IEEE
DOI 10.1109/PAW.2016.10

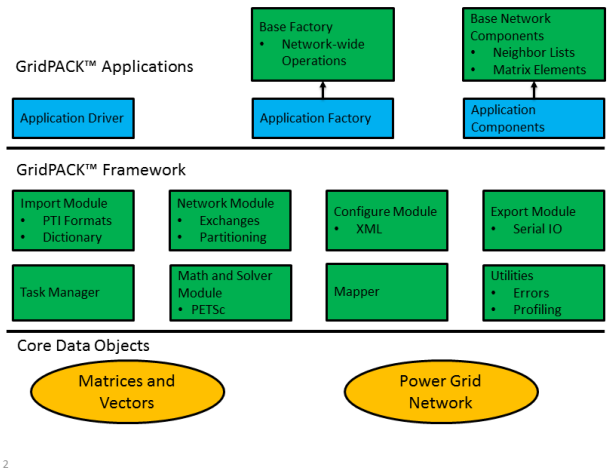

Figure 1. A schematic diagram of the GridPACK framework. Green boxes are elements supplied by the framework, blue boxes are application-specific elements created by the user.

2. Global Arrays Overview

Global Arrays is a PGAS programming model implemented as a library that supports the creation of multi-dimensional distributed rectangular data arrays that can be accessed using one-sided communication semantics. GA was originally co-designed with NWChem quantum chemistry software suite [2] to support the creation of dense 2-dimensional distributed arrays and to provide a shared memory-like programming semantics for reading and writing to these arrays. It has since been extended to support multi-dimensional arrays and includes a wide variety of operations on arrays that can be used to simplify many standard array operations. GA has been incorporated into many quantum chemistry packages in addition to NWChem and is currently used in GAMESS-UK [3], Columbus [4], MolPro [5], Molcas [6], and is being investigated for use in the CP2K chemistry modeling package [7]. A number of applications outside of quantum chemistry also use GA. These include Hydrology (STOMP [8], Smooth Particle Hydrodynamics (SPH) [9]), Atmospheric Sciences (Pagoda), hydrodynamics (TETHYS) [10], Bioinformatics (ScalaBLAST) and Machine Learning (MaTEx).

GA can be used to create multi-dimensional distributed arrays of uniformed-sized data elements by specifying the dimension of the array, the dimensions of each of the array axes and the data type of the array elements. Additional user options are to pad the arrays with ghost elements and to construct arrays on subgroups of processors. Users also have a high degree of control over data layouts and can specify how data is partitioned into blocks, whether or not a block-cyclic data layout is used and how data is mapped to processors. These attributes can all be used to improve the performance of particular algorithms. The library also provides functions for probing the data layout so users can distinguish between locally held data and remote data, again enabling additional performance optimizations by minimizing remote communication.

Global Arrays uses one-sided semantics to read and write data in a shared memory-like fashion. Data can be stored in global arrays using a put call that copies data from a local buffer to a location in the distributed array specified by the global address space. Data can be accessed using a get call that moves data from the array back into a local buffer and data from a local buffer can be added to the contents of a distributed array using an accumulate call. Put/get/accumulate all move rectangular blocks of data between local buffers and a global array, while the corresponding calls scatter/gather/scatter-accumulate can be used to move randomly ordered collections of elements back and forth to a global array. These latter calls are particularly useful in the context of unstructured distributed data sets since data accesses are unlikely to be contiguous or have any regular striding patterns to them. The one-sided nature of communication in GA can simplify programming tremendously since there is no need to coordinate communication between the processor requesting or writing data and the processor that contains the data. The fact that all data can be accessed using a global address space also simplifies programming since it eliminates many of the index calculations required to locate data on a remote processor. In addition to put/get operations, the GA library also has an atomic read-increment function that can retrieve the current value of a remote element and simultaneously increment the remote value by a user-specified amount. This is particularly useful for creating global task counters.

Data consistency is enforced by a synchronization call that consists of a fence in combination with a barrier. The fence guarantees that all outstanding communication is flushed from the system and the barrier ensures that no process proceeds until the flush is complete. The synchronization is usually put in between put and get phases of the calculation and guarantees that a global array is in a known (consistent) state.

A number of GA runtimes have been implemented on different platforms. Native ports are available for Infiniband, Portals4 and the Cray network. In addition, there are several implementations built on top of MPI [11]. Some of these depend on whether or not certain features of MPI have been implemented and may not be available on all platforms. Two runtimes depend exclusively on MPI-1 functionality and therefore can be built on almost any platform that has an MPI library. These include a two-sided runtime and a progress ranks implementation. The two-sided implementation relies on GA calls for progress and does not scale to large numbers of processors. The progress ranks implementation reserves one MPI process to manage communication and therefore reduces the number of cores available for computation but it is much higher performing and scales to large numbers of processors. Two other runtimes require that `MPI_THREAD_MULTIPLE` be implemented and are therefore not available on all platforms. A runtime based on

MPI-RMA calls in MPI-3 has also recently been developed but relies on relatively new functionality that may not be implemented optimally on all platforms. In general, GA is compatible with MPI and GridPACK contains parts based on MPI calls as well as GA. Most notably, the math module in GridPACK is based on PETSc [12], which uses MPI calls exclusively.

Global arrays and their associated functions are used in many GridPACK components. These include the use of gather/scatter functions to implement the ghost bus and ghost branch data exchanges in the network module, the use of the read-increment functionality to implement the task managers, and the use of global arrays to implement serialized IO from buses and branches. Other uses can be found in many parts of the code. In this paper, we will focus on the use of one-sided communication to implement collective hash tables that are used to match data with individual buses and branches in a distributed network. This functionality is interesting in itself and may have application beyond the power grid.

3. Collective Hash Tables

A frequent problem in power grid systems is distributing data to an already distributed network. At the start of the simulation, the network is read into the system and partitioned between processors. At a later point in the calculation, additional data is read into the processors and must be matched with the already distributed network. The association between the old and new data is a bus index that identifies a bus or a pair of bus indices that identify a branch. These indices are unique but they do not need to represent a contiguous set of integers and in practice, they almost always don't. In addition, there is no reason to assume that the new data is being ingested in a way that maps easily to the existing network decomposition and therefore it is necessary to move the data as efficiently as possible to the processors that contain the corresponding network elements. This applies irrespective of whether or not the data is read into a single processor or whether it is divided up into several blocks and then each block is read into a processor. The need to match data with elements of the network remains the same, although the performance characteristics of the different initial distributions may change. The new data can be mapped to the existing network by treating the bus index or branch index pairs as a simple key.

Distributing incoming data therefore reduces to creating a distributed hash table that can be used to discover which processor owns the bus or branch corresponding to a given key and then using this table to direct data to the appropriate processors. Since buses and branches may be replicated in the distributed network to account for branches that connect buses on different processors, the number of processors that map to a given key may be greater than one. Constructing a list of processors that own a given list of keys can be accomplished on a single processor using an STL-style multimap data structure [13] but in a distributed context the problem is more complicated.

Distributed hash tables are a key feature of the internet and many papers have been written on the subject ([14] and references therein). A more limited set of papers have been written on using distributed hash tables on HPC or multithreaded platforms [15], [16], [17]. In both cases, the focus is on extensible, flexible tables that support accesses to single values. However, some features of power grid applications can simplify the implementation of a distributed hash function capability. The first is that the request for key-value pairs are not distributed randomly but occur at discrete points during the code execution. Furthermore, the code will typically request many key-value pairs at once across all processors so requests can be aggregated together. Instead of requiring that each processor be prepared to satisfy a request at any point during the code execution, the requests occur collectively on all processors and can be handled using conventional HPC communication protocols. In particular, the collective nature of the requests greatly simplifies the issue of how to make progress. It also simplifies programming and improves efficiency since multiple requests for data can be handled in a single communication call.

GridPACK has implemented distributed hash tables using both MPI and GA. The MPI implementation is based primarily on the MPI all-to-all collectives `MPI_Alltoall` and `MPI_Alltoallv` [18], [19] and the GA implementation uses the read-increment functionality and one-sided communication capabilities available in GA. We will focus on the GA implementation in the following discussion but will present performance results comparing both the GA and MPI implementations. Both algorithms use a hash function to map keys to particular processors and are similar to the "assumed partition" algorithm proposed by Baker *et al* [20]. The hash function itself doesn't need to be specified in detail but it should have the properties of being rapidly computed and mapping keys approximately evenly to the available processors. The purpose of the hash function is to assign the keys to a particular processor. Any process requesting information about the key needs to address the query to that processor.

In the following discussion, we will focus only on the buses, but everything applies equally well to the branches. The first step in the one-sided algorithm is to create a hash table that can be used to identify which processors own a copy of a given bus. The algorithm starts by creating a vector of key-value pairs for each of the locally held buses in which the key is the bus index and the value is the processor rank. Applying the hash function to the keys divides the list of key-value pairs into a maximum of P separate bins, where P is the number of processors. The bins can be created using a simple linked-list. The next step is to send each of the P sets of pairs to the processor that owns the keys. Once each processor has gathered all its key-value pairs, it can incorporate them into a local multimap [13] data structure. If a bus appears on more than one processor then the key will appear multiple times in the multimap list.

The second part of using the hash table is to direct a list of queries to it, where a query consists of a list of keys and the response is a list of key-value pairs. For this

application, each key can map to multiple values, so the number of values in the response will likely be greater than the number of keys in the query. The basic sequence of events for answering a query is the following:

- The list of keys in the queries is sorted into groups by applying the hash function to each key. The keys in each group represent all keys in the query that map to a single processor. This is done on each processor that has a list of queries.
- Each processor sends its queries to the processor that owns the keys (based on the hash function). The processor compares the keys with its local multimap list and creates a list of key-value pairs containing the requested data.
- The key-value pairs are then sent back to the requesting processor.

The one-sided implementation of the hash table relies on the read-increment functionality in GA. Two 1-dimensional global arrays are needed to initialize the hash table. The first array contains P elements and keeps track of the total number of key-value pairs that are being assigned to a particular processor. It is initialized to zero. The second array is created after all the key-value pairs have been counted and is used to temporarily hold all key-value pairs that are being stored in the hash table.

The first step is to bin up the keys (bus indices) using the hash function. At the same time, the number of keys going to each processor can be counted. The second step is to loop over the bins of keys and call the read-increment function on the index location in the counting array corresponding to the processor rank represented by the bin. The amount the value is incremented by is the number of keys in the bin and the returned value is stored in a local array of length P. When this loop is completed each processor has a list of offsets on the remote processors that can be used to store the keys in a global array. The global array that has been used for the counting contains a list of the total number of keys that are going to each processor. This phase of the initialization is illustrated schematically in Figure 2. The important feature of this part of the algorithm is that it provides a mechanism for allocating memory on each processor that guarantees that there is space to store the key-value pairs when they are sent by the processor originally holding them.

The total number of keys being received by each process can be retrieved by having each processor copy the entire global array used for counting into a local buffer. Again, the array only contains P elements so it is small and does not consume a large amount of local memory. Each process now has a list of the total number of keys that are going to every other process and a list of offsets for the data it is sending to each of the remote processors. This allows each processor to send the data describing a set of key-value pairs to the remote processor while simultaneously guaranteeing that it is not overwritten by someone else. The total number of keys that need to be stored globally can be calculated by summing the values in the P-element counting array and using this number to create a second global array large

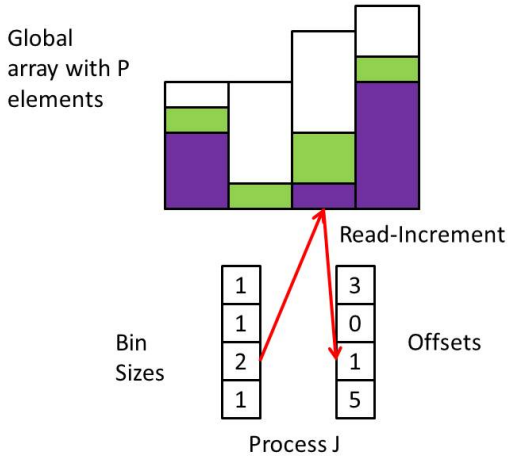

Figure 2. A schematic diagram showing how GA is used to evaluate the total number of keys being sent to each processor and how the offset onto each processor is calculated. The purple boxes represent the current value of the offset at each processor when process J tries to read it and the green boxes are the amounts that the offsets are incremented after reading.

enough to hold all data pairs. At the same time, the local copy of the counting array can be used to calculate an offset in the global array for *all* pairs going to each processor. Within each global offset, each processor has a local offset that describes the relative location to send its queries on the remote process. This can be added to the global offset derived above to get a location in the global array where the processors can safely store their data pairs. The evaluation of the offset is shown in Figure 3.

The global array holding all key values is constructed so that the segment of the array that is local to each processor is large enough to hold all the keys that the processor will be receiving. The evaluation of the offsets on the remote processor guarantees that there is a non-overlapping segment in the global array that can receive the keys that are being sent to each processor, but it does not guarantee a consistent ordering for these segments. The order of the segments will depend on the order in which the read-increment requests are processed on each processor and this can be affected by a number of unknown factors (e.g. load imbalance, network congestion). For the initialization of the distributed hash table, this does not make any difference to the algorithm, but it is an important consideration when answering queries.

After the keys have been binned, the offsets evaluated, and a global array constructed that can contain all the key-value pairs, the pairs can be copied to the global array using a standard put call. When all processors have finished writing data to the array, a synchronization call is made to guarantee data consistency. The final data layout in the global array is shown in Figure 4. All key-value pairs that map to a given processor are now located in the segment of the global array that is local to the process. The pairs

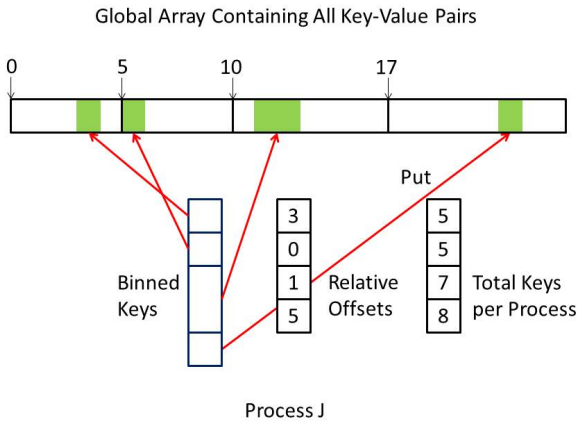

Figure 3. A schematic diagram showing how the locally held queries are added to a global array containing all queries. After each processor has copied its copies of the queries to the appropriate location in the global array, the queries can then be answered by the processor holding the corresponding data. The global offsets for each processor are shown above the global array.

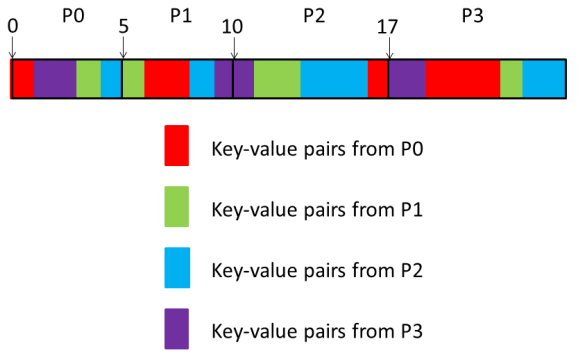

Figure 4. A diagram showing how data is arrange in a global array after all key-value pairs have been copied to it. Different colors are used to indicate key-value pairs that originate from different processors.

can be pulled from the array using a local copy and stored locally in a multimap data structure. The hash table is now initialized and the global arrays can then be destroyed and their memory freed for reuse. Each processor has a local multimap that contains all the bus indices that map to that process as the keys and the rank of the processor that owns the bus as the corresponding value.

The algorithm for answering queries is similar to the initialization algorithm and again makes use of the read-increment functionality to allocate space in a global array to store all queries. The queries are binned using the hash function and stored as index pairs, with the original key as one member of the pair and the rank of the processor *making* the query as the other. The queries are sent to the processor that can answer them using a process similar to the one described in Figures 2 and 3. The read-increment functionality is used to create a global array of index pairs, with one index representing the key for the requested value and the second index representing the rank of the processor making the request. Each processor now has a list of all queries that have been directed to it. After finding the appropriate values for the keys using the local multi-map data structure created during initialization, the process can bin up the responses (based on the index of the processor that made the request) and use a second set of read-increment calls to allocate a global array with enough memory on each processor to receive the responses. In this case, the response contains the original key (which is the bus index) and the rank of the processor that owns the bus.

Once a hash table is available, it can be used to distribute incoming data directly to the processors owning the corresponding buses. The algorithm is very similar to the algorithms for setting up and addressing queries to the hash table. The initial step is to set up a hash table, as described above, that maps all bus indices to processors that own them. Using this map, the data can be binned based on which processor it is being sent to (each data item is already associated with a bus index). The read-increment functionality is again used to determine how much data is being sent to the remote processor and what the offset for the data is. This is used to set up a global array for the index-data pairs and to copy data to it. The global array is organized so that all index-data pairs going to a particular processor form a contiguous segment in the array. After synchronization, each processor can pull the data it owns to a local buffer and match it with the corresponding local buses.

4. Results

The distributed hash table and data distribution algorithms were tested by constructing an artificial network based on a square 2-dimensional grid. This network is not typical of the power grid but it is easy to set up and is readily scaled to large sizes. The hash algorithms discussed here should not be affected by the grid topology so a square grid is a reasonable test for this particular functionality. The grid is partitioned by decomposing each axis into roughly equal segments such that the product of the number of segments on each axis is equal to the total number of processors. The buses on the grid are indexed in column-major order so the indices on each process are strided if the column axis is decomposed into more than one segment. If a branch connects buses across two processors, then a copy of the connected bus is included on each processor. The same branch may also appear on more than one processor in order to guarantee that the environment of all locally held buses is complete.

Two tests for distributing data to the buses were devised. The first represents a situation where the data to be

distributed is read in on each process from separate files (or read in using parallel I/O) and each process ends up with approximately equal amounts of data before distributing it. For this test, the total list of buses of size N is divided into P approximately equal sized segments such that process 0 will send data to the first N/P buses, process 1 will send data to the next N/P buses, etc. Because the bus indices are column-major, dividing the total set of indices in this way is roughly equivalent to dividing the grid into P vertical slabs. The grid itself is partitioned into blocks along both axes so each process will end up communicating with approximately $\sqrt{P}$ other processes. The second test mimics a situation in which only one processor reads in data from an external file. All data that is to be distributed is generated on process 0 and then sent to other processors.

The data that is being sent to the buses is a C-struct consisting of three integers. For each bus, the struct is assigned unique values so that the received data can be tested for correctness. The struct is replicated for buses using the rule

$$n = \mathrm{mod}(I, 3) + 1$$

where I is the bus index and n is the number of times this data is sent to I, so that most buses receive multiple data elements (this replicates real scenarios where different pieces of data get assigned to the same buses). Similar scenarios were created to test the distribution of data to network branches. The grid chosen for these benchmarks was a square grid consisting of 1000×1000 buses. The tests were run using from 1 to 4096 processors on a Linux cluster with an Infiniband interconnect. MPI was supplied using OpenMPI [21].

The results for the MPI implementation are shown in Figure 5. Also included in this figure is the amount of time require to initialize the hash table used in the data distribution algorithm. It is immediately worth noting that the initialization of the distributed hash table starts exhibiting very poor scaling behavior after 16 processors. The minimum in the initialization is about 0.25 seconds but the times then start climbing steeply and rise to nearly 7 minutes on 2048 processors. The code crashes on 4096 processors. The distribution of bus and branch data from a uniform distribution shows significant scaling out to 64 processors where the times drop to well under 0.5 seconds (0.14 seconds for the bus data and 0.37 seconds for the branch data). After 64 processors, the times increase gradually. There is a large performance difference for the MPI implementation between distributing data from a uniform initial distribution compared to the asymmetric distribution. The asymetric distribution flattens out at about 5 seconds for distributing bus data and 7 seconds for branch data. Although, the hash table algorithms are similar to the algorithms that are used for distributing data, the hash table initialization takes much longer to complete than the data distribution. We hypothesize that this is a consequence of the implementation of the MPI all-to-all algorithms and that the first call to an all-to-all function requires the operating system to allocate resources that are then available to subsequent calls.

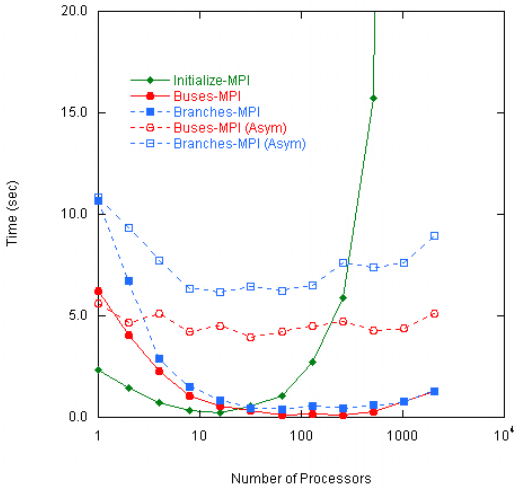

Figure 5. A plot of the time required to distribute data using an MPI implementation. Distribution times for a uniform distribution of data and for an asymmetric distribution are shown. Also shown is the time required to initialize the distributed hash table used in the algorithm.

Results for the one-sided implementation using GA are shown in Figure 6. For this implementation, the initialization is much better behaved. The time spent in initialization scales out to about 64 processors for a minimum of 0.17 seconds. The times start rising after 64 processors but stay under 1.0 seconds all the way out to 4096 processors. For uniformly distributed data, the data distribution algorithms scale out to 128-256 processors and remain low after that. The minimum distribution times are under 0.1 seconds for both bus and branch data and do not exhibit a tendency to increase dramatically at large processor counts. The asymmetric distribution exhibits behavior similar to the MPI implementation. The times average about 5 seconds for the bus data and 7 seconds for the branch data and are largely flat across the different numbers of processors. However, the algorithm does not show any tendency towards pathologies at large numbers of processors and was able to run successfully on 4096 processors, which the MPI implementation was not able to do.

A direct comparison of the MPI and one-sided implementations is shown in Figure 7 for the uniform distribution. Although the initialization times are somewhat shorter at small processor counts for the MPI implementation, at larger processor counts the one-sided algorithm is much faster. The one-sided implementation consistently matches or outperforms the MPI implementation for the actual distributing of data.

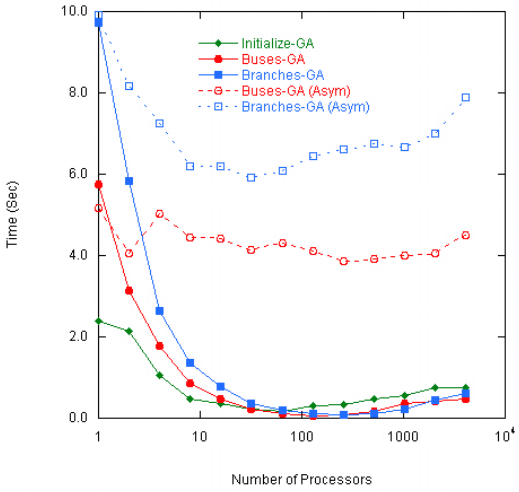

Figure 6. A plot of the time required to distribute data using a one-sided implementation. Distribution times for a uniform distribution of data and for an asymmetric distribution are shown. Also shown is the time required to initialize the distributed hash table used in the algorithm.

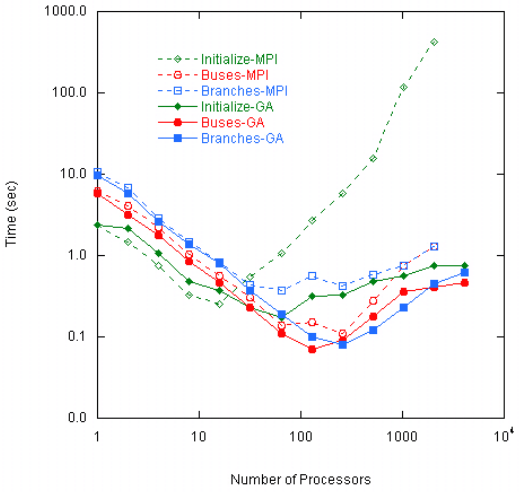

Figure 7. Comparison of MPI and one-sided implementations for the data distribution algorithm for a graph containing 1 million buses. The initial distibution of data is uniform.

5. Conclusions

This paper discusses the use of the GA PGAS library for implementing several parts of the GridPACK power grid simulation framework. It describes the use of PGAS constructs for many parts of simulation framework and provides a detailed description of a hash table module for distributing data. Performance results are presented for the distributed hash table algorithms and indicate that they are competitive with algorithms based on `MPI_Alltoall` (the poor performance seen in the initialization of the MPI implementation may be platform-specific and reflect the installation of some of the libraries on the cluster used for testing). These algorithms have been incorporated into the GridPACK™ power grid modeling framework, which is available for download at `https://gridpack.org`. GA is available for download at `http://hpc.pnl.gov/globalarrays`.

Acknowledgments

The author wishes to thank Kurt Glaesemann and Doug Baxter for useful discussions on the performance of `MPI_Alltoall`. Funding for this work was provided by the U.S. Department of Energy's Office of Electricity Delivery and Energy Reliability 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.

References

[1] 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*, vol. 20, no. 2, pp. 203–231, Summer 2006.

[2] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465510001438

[3] M. Guest, I. Bush, H. van Dam, P. Sherwood, J. Thomas, J. van Lenthe, R. Havenith, and J. Kendrick, "The gamess-uk electronic structure package: algorithms, developments and applications," *Molecular Physics*, vol. 103, no. 9, pp. 719–747, 2005.

[4] H. Lischka, R. Shepard, I. Shavitt, R. Pitzer, M. Dallos, T. Müller, P. Szalay, F. Brown, R. Ahlrichs, H. Böm, A. Chang, D. Comeau, R. Gdanitz, H. Dachsel, C. Ehrhardt, M. Ernzerhof, P. Höchtl, S. Irle, G. Kedzoria, T. Kovar, V. Parasuk, M. Pepper, P. Scharf, H. Schiffer, M. Schindler, M. Schüler, M. Seth, E. Stahlberg, J. Zhao, S. Yabushita, Z. Zhang, M. Barbatti, S. Matsika, M. Schuurmann, D. Yarkony, S. Brozell, E. Beck, J.-P. Blaudeau, M. Ruckenbauer, B. Sellner, F. Plasser, and J. Szymczak, "COLUMBUS, an ab initio electronic structure program, release 7.0," 2015.

[5] H.-J. Werner, P. Knowles, G. Knizia, F. Manby, and M. Schütz, "Molpro: a general-purpose quantum chemistry program package," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 2, pp. 242–253, 2012. [Online]. Available: http://dx.doi.org/10.1002/wcms.82

[6] F. Aquilante, T. Pedersen, V. Veryasov, and R. Lindh, "MOLCAS-a software for multiconfigurational quantum chemistry calculations," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 3, pp. 143–149, 2013. [Online]. Available: http://dx.doi.org/10.1002/wcms.1117

[7] J. VandenVondele, M. Krack, F. Mohamed, M. Parrinello, T. Chassaing, and J. Hutter, "Quickstep: Fast and accurate density functional calculations using a mixed Gaussian and plane waves approach," *Computer Physics Communications*, vol. 167, pp. 103–128, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465505000615

[8] M. White, D. Appriou, D. Bacon, Y. Fang, V. Freedman, M. Rockhold, C. Ruprech, G. Tartakovsky, S. White, and F. Zhang, "STOMP online user guide," Pacific Northwest National Laboratory, Richland, WA, Tech. Rep., 2015. [Online]. Available: http://stomp.pnl.gov/user_guide/STOMP_guide.stm

[9] B. Palmer, V. Gurumoorthi, A. Tartakovsky, and T. Sheibe, "A component-based framework for smoothed particle hydrodynamics simulations of reactive fluid flow in porous media," *Int. J. High Performance Comp. Appl.*, vol. 24, no. 2, pp. 228–239, May 2010.

[10] M. C. Richmond, W. A. Perkins, T. D. Scheibe, A. Lambert, and B. D. Wood, "Flow and axial dispersion in a sinusoidal-walled tube: Effects of inertial and unsteady flows," *Advances in Water Resources*, vol. 62, Part B, no. 0, pp. 215–226, 2013, a tribute to Stephen Whitaker. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0309170813001152

[11] J. Daily, A. Vishnu, H. van Dam, B. Palmer, and D. Kerbyson, "On the suitability of mpi as a pgas runtime," in *International Conference on High Performance Computing (HiPC)*, 2014.

[12] 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.

[13] H. Schildt, *STL Programming from the Ground Up*. Berkeley: Osborne/McGraw-Hill, 1999.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 161–172, 2001.

[15] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *J. ACM*, vol. 53, pp. 378–405, May 2006. [Online]. Available: https://doi.acm.org/10.1145/1147954.1147958

[16] A. Stivala, P. Stuckey, M. Garcia de la Banda, M. Hermenegildo, and A. Wirth, "Lock-free parallel dynamic programming," *Journal of Parallel and Distributed Computing*, vol. 70, pp. 839–848, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731510000110

[17] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing(IPDPS)*. IEEE, May 2013, pp. 775–787.

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

[19] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA: The MIT Press, 1999.

[20] A. Baker, R. Falgout, and U. Yang, "An assumed partition algorithm for determining processor inter-communication," *Parallel Computing*, vol. 32, pp. 394–414, 2006.

[21] E. Gabriel, G. Fagg, G. Bosilica, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: Goals concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, September 2004.