PNNL-22620

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by Battelle Since 1965

Efficient Memory Access with NumPy Global Arrays using Local Memory Access

August 2013

JA Daily
DC Berghofer

U.S. DEPARTMENT OF ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Efficient Memory Access with NumPy Global Arrays using Local Memory Access

Dan Berghofer and Jeff Daily
Pacific Northwest National Laboratory, Washington 99352

The Global Arrays library was developed at the Pacific Northwest National Laboratory and provides an interface for easily creating arrays that exist in multi-computer systems. Part of this library includes Global Arrays in NumPy (GAiN) which allows NumPy Arrays to exist in distributed systems. Currently, whenever many binary operations are executed on two GAiN arrays, array elements from each array on several computers are copied and then operated on. Improvements were made to binary operations on GAiN arrays that reduce data copying and results in faster computation. If sections of arrays to be added are on the same computer then data copying is unnecessary for data access. Binary operation algorithms were improved so that data copying only occurs when array sections of operands exist on separate computers. While this requires more data accesses, large performance improvements were seen. This algorithm also works with array slices including stepped slices and negative slices.

I. INTRODUCTION

Today scientific data is often collected and processed by computers. This data will likely be a large set of scientific observations or data points. This data is often held in arrays and in many cases these data sets can be so large that they have to be processed by distributed networks of computers or distributed supercomputers. One problem with these distributed arrays of data is that even the simplest computations on them is very difficult because a large amount of computation is required to do operations on arrays pieces that are spread on several different computers. This computation will requires some sort of messaging routines like Message Passing Interface (MPI). The Global Arrays library solves this problem by doing this multi-computer computation with MPI automatically and allows programmers to create and manipulate arrays as if they were on a single computer [2].

Global Arrays in NumPy (GAiN) is a version of the Global Arrays library that emulates the NumPy Python library in distributed systems [2]. The work done during this internship improved the speed of GAiN when executing binary operations. When any operation involving two GAiN arrays that requires item by item computation occurs, this memory access algorithm can be used.

One concept, node overlap refers to the fact that often items in different arrays to be operated on are often on the same node. When many of these elements are on the same computer, there is a high amount of node overlap. Currently these operations are done by copying these arrays and the n doing operations on them. If there is a large degree of node overlap, then much copying can be avoided and array sections to be added together can be accessed with pointers. Others will have to be accessed by copying them from different nodes. The algorithm discussed determines which regions have to be copied and which ones don't and then executes operations on these regions. It is also designed to work on sliced (partial) arrays.

II. BACKGROUND

Distributed Global Arrays

The research completed resulted in the creation of code that enhances the speed of Global Arrays in NumPy (GAiN) functions that exist as part of the Global Arrays (GA) library. The GA library allows programmers to build distributed (multi-computer) arrays as if they were local arrays [3]. GA and GAiN code run in parallel on all the computers on a computer network. A programmer can specify that only a few computers run bits of GA code but for the most part, all the computers will be running GA code in parallel. One can create an array with a GA Create command. Global Arrays can access elements using the GA Get and GA Access commands. Arrays can have elements stored to them using the GA Put command [3]. This library certainly makes distributed functions seem as if they are occurring on local machines even if the syntax is more complex. Global Arrays in NumPy (GAiN) actually makes the look and feel of these arrays almost exactly like local NumPy arrays.

Python and GA in NumPy

GA has separate versions that are written in C, Fortran, and Python. GAiN, the module that is being improved in this internship, is written in Python and is based on the Python version of GA. Python is easy to use because it is very terse and there are many built-in operations that are helpful to the programmer. Python is used in systems programming, web design, distributed programming, GUI programming, and game programming. One problem with Python is that it is slow relative to other languages like C and Java. There are several reasons for this problem. First, lists of items in Python are allocated dynamically, and are not continuous, second Python's loops are slow, third Python is un-typed, or in other words its variables do not have to be given types by programmers. Operations on un-typed variables are slower than those on typed variables [1].

There are several solutions to Python's speed problem. First, a typed version of Python called Cython exists that does many of its operations in the much faster C language [1]. A Python library called NumPy builds arrays of contiguous items that are much faster than ordinary lists in Python. In order to avoid the slowdown that 'for loops' cause in Python, NumPy has many of its operations done in predefined functions written in the C language. For example, NumPy arrays can be transposed, added, compared, multiplied, and much more using the NumPy library [4]. One disadvantage with NumPy is that it is not distributed. GA is used to make a version of NumPy that is distributed called GA in NumPy (GAiN).

Numpy and GAiN Slicing

Our algorithm for improving GAiN works with NumPy slicing. Like generic arrays, NumPy arrays can be sliced to obtain individual elements, but unlike them it can obtain sets of individual elements that may or may not be contiguous. NumPy can access sets of elements using colon syntax [4]. For example, thisArray[4:6] will retrieve elements from index 4 up to but not including the element for index 6. One can also retrieve elements that are non-contiguous by adding a number to indicate the step of the slice. For example, thisArray[4:8:2] will retrieve elements for indexes 4,and 6. The step is two and this is why the slicing skips over element 5 and went to element 6.

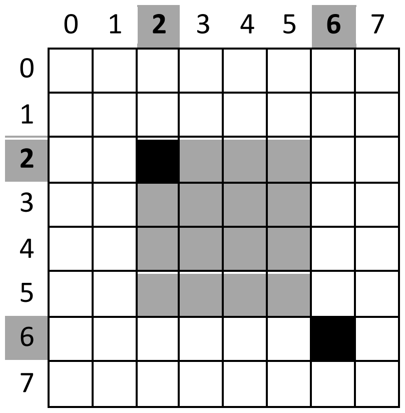
NumPy can also execute negative slicing so thisArray[5:1:-1] will start at element #5 and go backwards by steps of one until it reaches the index #1 which will not be included in the array slice . The negative sign at the end of a step indicates a negative step. NumPy is also capable of larger negative steps such as -2 or -3. NumPy can also slice multidimensional arrays using the comma syntax. An example of this is thisArray[3,5] which accesses an element at coordinate (3,5). An example of very complex multidimensional syntax is thisArray[5:2:-5, 2:9:4, 9:2:-1].

In both NumPy and GAiN, slices of arrays can be taken and assigned to other arrays [4]. For example, we can say: newArray = oldArray[1:3,6:12]. An array slice can also be thought of as a pointer to a section of an array in NumPy. GAiN implements slices too and in GAiN, slices point to the exact same memory the arrays they are sliced from point. So if oldArray and newArray were GAiN arrays, newArray could access all the elements oldArray could beyond its slicing bounds. However newArray does have a variable called global_slices that tells what elements of the entire array newArray is pointing to.

GA Distribution Syntax

If a programmer wishes to execute an operation on only a section of a GAiN array, then she can specify this section as a coordinate distribution with a low coordinate and a high coordinate.

Figure 1



The array here is having a section of itself selected using low, high syntax. The coordinates are shown in black and are (2,2), and (6,6) respectively.

In the Figure 1, we want to retrieve an array section in the shaded area. To do this we have to specify a low coordinate and a high coordinate. The low coordinate will be the coordinate on the upper left hand side of the section and in the example it is (2,2). The high coordinate is the coordinate that is to the lower right hand side of the matrix but is right outside of this matrix itself by one unit in every dimension. In this example, that coordinate is (6,6). Low, High coordinates are called **Distribution Coordinates** and are used in the functions discussed in the next section.

GA Functions Used

Major GA functions used for array manipulation are **GA Create**, **GA Distribution**, **GA Get** , **GA Access**, and **GA Put**. While they can be used on an entire global array, they can also be used on only a part of this array by specifying the distribution coordinates of this part of the array. The **GA Create** function creates a new GA array. The **GA Distribution** function returns the distribution of the part of the total array the node that called this function contains.

GA Get retrieves some or all of a global array by going to each node and copying the memory of the sections of the array each node contains and creates a NumPy array that consists of each of these copies. This operation is slow because of all the copying necessary. **GA Access** returns a pointer to some or all of the elements of the Global Array to the node that called this function. It is unable to return parts of a global array that are not in the local node. Since returning a pointer is easier than copying memory like **GA Get** does, the access function is faster.

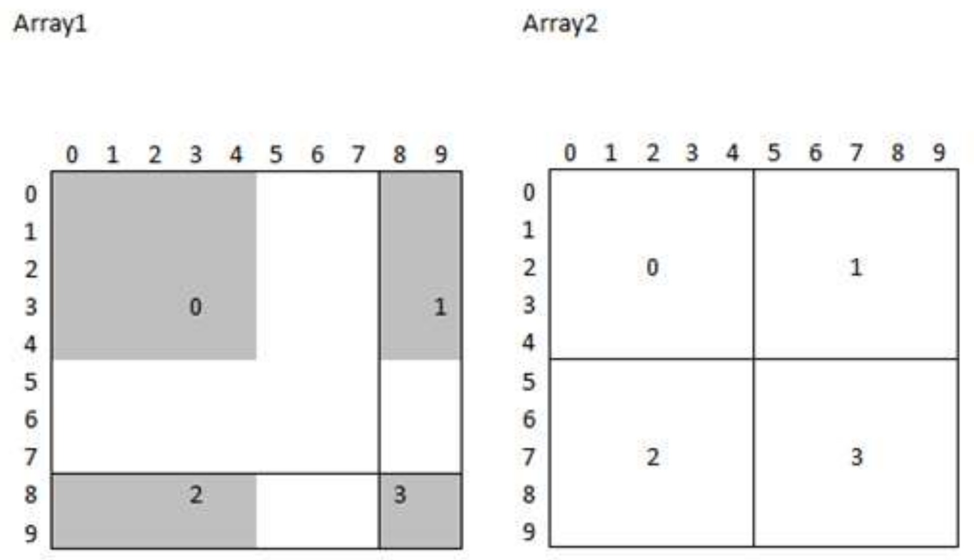
III. The EFFICIENT MEMORY ACCESS ALGORITHM (EMAA)

GAiN Addition

The GAiN Addition function accepts two GAiN arrays, adds them, and then stores the result either in one of the arrays or in a different GAiN array. This operation is very inefficient because both arrays are copied from their original locations and then added. The work done during the internship involved reducing the amount of copying done by only copying when subsections of arrays to be added exist on different nodes. These improvements also work with array slices which will be described later.

Problem with the Current Adder

Figure 2

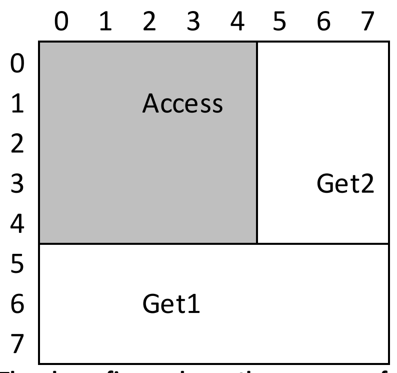


The above arrays shows which sections of them are contained in which nodes. Nodes are numbered from zero to three. Any number of nodes can be specified by GAiN. These two arrays will be added together and will be used as an example of how the EMAA algorithm works. The parts of Array1 in grey will be added to parts of Array2 in the same node.

The arrays in Figure 2 are shown with different distributions over four nodes. Notice that many of the elements in Array1 and Array2 to be added together exist in the same node. For example, all elements in Array1 node1 will be added to elements in Array2 that also exist in node1. The elements in these arrays that will be added to elements in the same node are shown in grey in Array1 in Figure 2.

Our solution

Figure 3



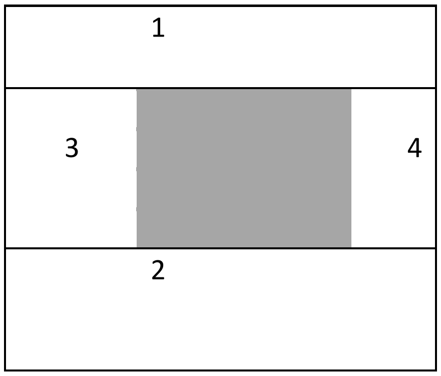
The above figure shows the memory of process 0 in Array1, and the different regions of Array2 overlaid on this process. The Access area in grey is the area of Array2 that is also held in process 0 and this area will be retrieved using GA Access. The areas in white are areas of Array2 that are not in process 0 and will be retried using GA Access.

Our solution takes into consideration that sometimes elements of two arrays to be added or operated on are in the same node, and at others they are not. Rather than executing a large GA Get function for two array addends and then adding them together, we start by making every node access local elements in Array1 and Array2. **Local elements** are the set of elements in an array that are in a specific node. For each node, the local elements of Array2 that are to be added to Array1 are determined in Figure 4 in grey. These Array1 elements are retrieved using the GA Access function. These local elements are then added to their equivalents in Array2 which were retrieved using the GA Access Function. The **equivalents** of a set of elements in this context is the set of elements they are going to be added to.

Parts of Array1 will not have local equivalents in Array2 or in other words the parts of Array2 to be added to them will not be in the same nodes as the elements in Array1. For each node, those parts of array1 without local equivalents will be retrieved using the GA Access function and their non-local equivalents will be retried using the GA Get function. They will then be added together.

Retrieval of Elements

Figure 4



This figure shows the distribution of all elements in node X in Array1. The elements in Array2 they are going to be added to are overlaid on top of them. The area in grey shows the elements in Array2 that are not in the same node as their equivalents in Array1. The area in white shows the elements in Array2 that are in the same node as their equivalents in Array1.

In Figure 4, the entire array represents the range of elements owned by Array1 in node X. The elements in Array2 to be added to them are overlaid on top of their equivalents in Array1. The grey elements are called the **overlapping elements** and represent the elements in Array1 with local equivalents in Array2. The white elements are called **non-overlapping elements** and are the elements of Array1 without local equivalents in Array2. Their equivalents are non-local.

For overlapping elements in Array1 the GA Access function is used, and for their local equivalents in Array2. This is true because our algorithm works from the perspective of each local node and not the array as a whole. If parts of Array1 exist in each node then retrieving them with GA Access makes the most sense because GA Access is fast. If they have local equivalents, then each node is able to access these equivalents with GA Access. If they have non-local equivalents, then these equivalents will be outside of each node doing the retrieving and so they will have to be retrieved using GA Get.

Grabbing every element in the non-overlapping regions in one GA Get call is not possible because Access can only grab rectangular areas. The solution to this problem is that multiple GA Gets should be executed on non-overlapping (white) element. That is why the non-overlapping region in the previous imagine is divided into four parts for separate accesses.

The algorithm for accessing non-overlapping regions involves finding the largest possible non-overlapping regions above and below the overlapping region for each dimension and doing computation on these coordinates. Once these elements have been added together, the region for these elements is then excluded from the algorithm and the algorithm then continues with the other dimensions. This algorithm is called the Efficient Memory Access Algorithm (EMAA).

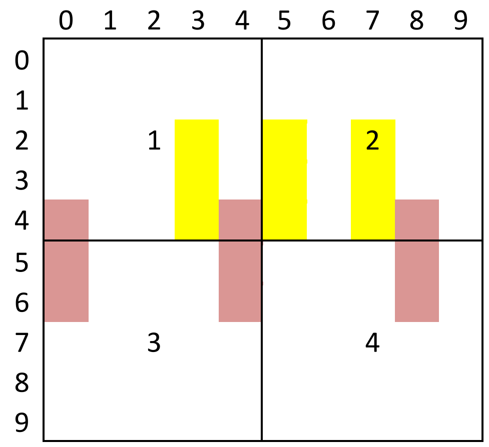
Storing Elements

Once the elements have been added together, they will need to be stored. Currently they are in NumPy format after they have been added. If one of the addends is also the target array, then the algorithm ensures that this array is stored to because it is the array that is accessed using the GA Access Function. The results of addition are stored in this array by updating the local memory. If the target array is not one of the addends, then the results of the addition is stored into the other array using the GA Put function.

Dealing with Slices

A sliced GAiN array as mentioned before points to only a part of an array it is a slice of. It does have a variable called global_slices that indicates what parts of that array is part of the slice.

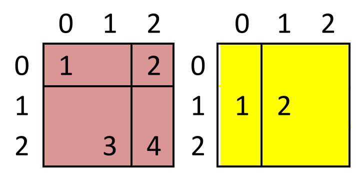
Figure 5



Two Global Array slices to be added together. The light array is sliced as [2:5:1,3:8:2]. The dark array is sliced at [4:7:1,4:9:4].

Adding sliced arrays is challenging because EMAA works with distribution coordinates that assume arrays start at zero and their steps are one. There are two ways to deal with this problem. The first is to make these algorithms work with arrays with any step or any origin. This method is not practical because working with slices adds another layer of complexity on top of the complexity EMAA already has. A better algorithm is to take the distribution coordinates of sliced arrays and translate them into a compressed form with each of their origins at 0 and step equal to 1 as shown in Figure 6. The precise regions to GA Get or GA Access is then determined, and when these operations are going to be executed on these arrays, these distribution coordinates are converted back to their original form.

Figure 6

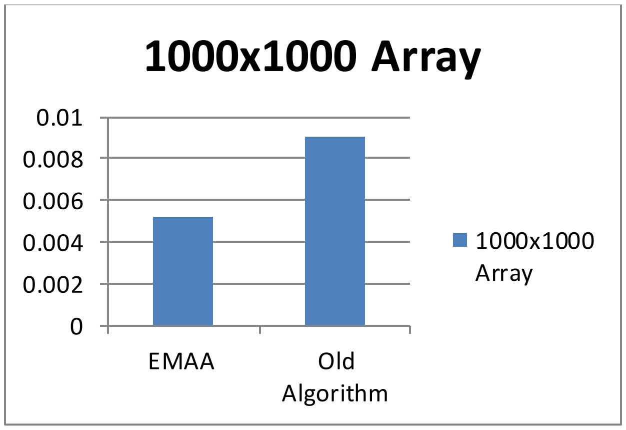


A compressed version view of the arrays in figure 5. The coordinates of each array is compressed to this version. Once this compression is complete, then the EMAA algorithm is executed.

We can see in Figure 6 the nodes the sections of each array belongs to. We then carry on our usual addition operation from the perspective of the processes in the first array detailed before this section. We use this computation to determine the distribution coordinates of each array piece to add together. These distributions are then translated back to their original form as in Figure 5, and the addition is completed.

Results

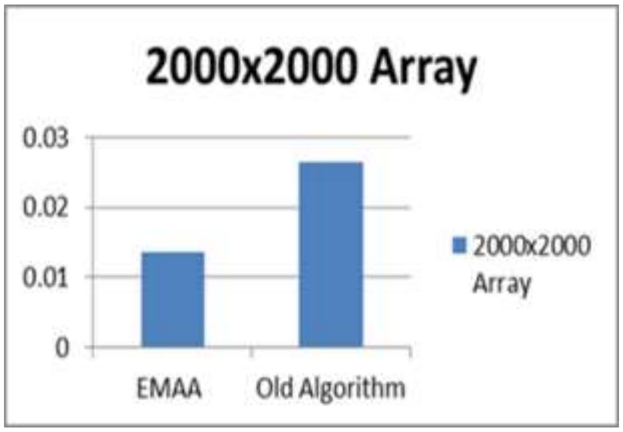
Figure 7



Array addition on 1000x1000 Arrays.

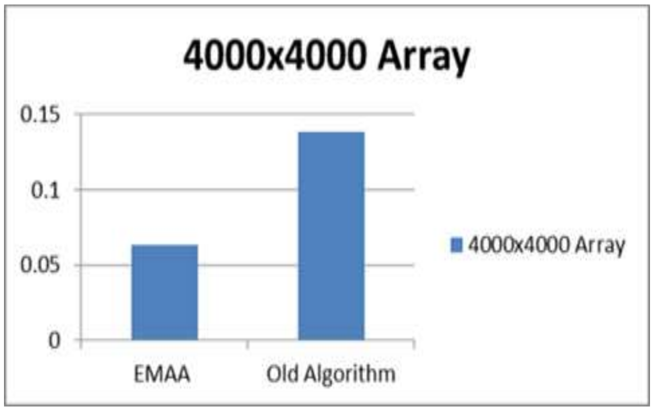
The results of this analysis show that EMAA algorithm surpasses the original algorithm. This analysis conducted array addition on arrays where there was over 90% array overlap. 32 array additions were completed on 1000x1000 arrays and the standard deviation on these trials was found to be marginal. The average time taken for the EMAA algorithm was .0052 seconds while for the average for the old algorithm was .009 seconds. The EMAA algorithm in this case is twice as fast.

Figure 8



Array addition on 2000x2000 Arrays.

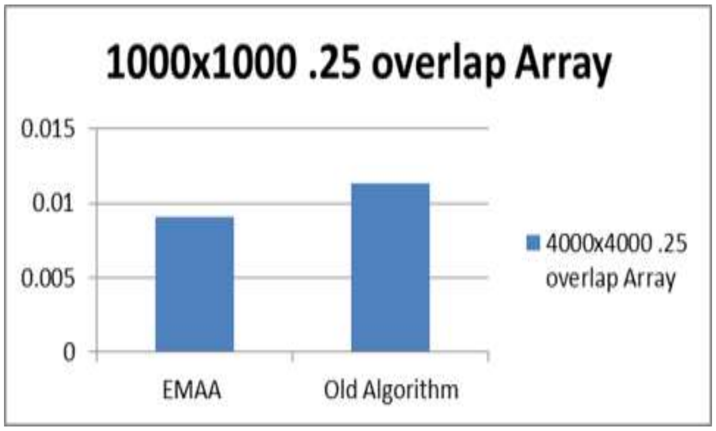
Figure 9



Array addition on 4000x4000 Arrays.

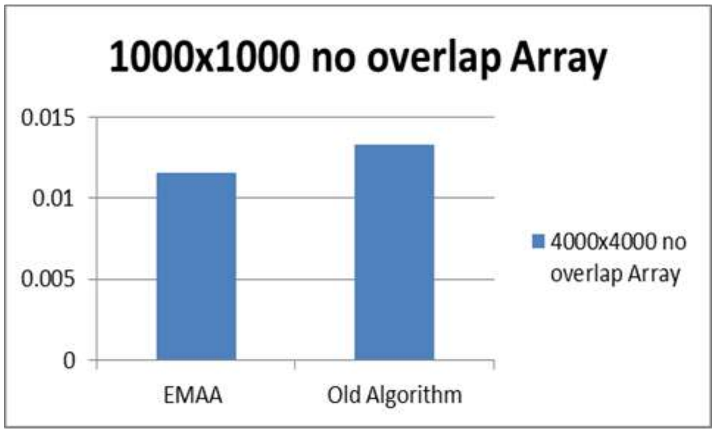
When this analysis was done with 2000x2000 arrays, the EMAA algorithm completed its computation in .014 seconds on average while the old algorithm completed its computation in an average of .026 seconds as shown figure 8. This analysis was also completed on 4000X4000 arrays as shown in figure 9. The EMAA algorithm completed its computation in .063 seconds while the old algorithm completed it in .138 seconds. As arrays get bigger, the EMAA algorithm compares better and better against the old algorithm.

Figure 10



Array addition on 1000x1000 Arrays with 25% overlap.

Figure 11



Array addition on 1000x1000 Arrays with 0% overlap.

So far the EMAA algorithm works well when there is a high degree of overlap. But what happens when this overlap is reduced? In Figure 11, two array slices with 25% node overlap are added together. Even in this situation the EMAA algorithm is faster and took .0091 seconds to complete while the old algorithm took .011 seconds as shown in figure 10. An interesting exercise was to add arrays with no node overlap. As shown in Figure 11, EMAA took .0116 seconds while the old algorithm took .133 seconds per addition.

IV. CONCLUSION AND FUTURE WORK

These trials have shown that the EMAA algorithm works well in situations with a great amount of node overlap and can also function quickly in situations with no node overlap. This can speed up a large number of binary operations in GAiN. These results make quite a bit of sense. Even when there is no node overlap, one array is being accessed using the GA Access function, and the other is being accessed using the GA Get function. The biggest disadvantage of this function is that more item accesses are occurring which can slow it down. However this slowdown is outweighed by EMAA's benefits. It is found that larger arrays work better with EMAA than smaller ones. The more array overlap, the better the EMAA algorithm did. Therefore it is recommended that more binary operations have their memory access use the EMAA algorithm.

One potential source for improvement is to minimize the use of the GA Put function by only using it for saving to memory outside of a node that calling the function. Also, this algorithm doesn't work when the same memory that is being saved to is also being called from and these memories are being sliced differently.

References:

1. Behnel, S.; Bradshaw, R.; Citro, C.; Dalcin, L.; Seljebotn, D.S.; Smith, K., "Cython: The Best of Both Worlds," Computing in Science & Engineering , vol.13, no.2, pp.31,39, March-April 2011 doi: 10.1109/MCSE.2010.118

2. Daily JA, and RR Lewis. 2011. "Automatic Parallelization of Numerical Python Applications using the Global Arrays Toolkit." In Proceedings of the 2011 Companion on High Performance Computing, Networking, Storage and Analysis, (SC 2011 Companion), November 12-18, Seattle, Washington, pp. 43-44. Association for Computing Machinery, New York, NY. doi:10.1145/2148600.2148623.

3. Manojkumar Krishnan, Bruce Palmer, Abhinav Vishnu, Sriram Krishnamoorthy, Jeff Daily, and Daniel Chavarria. 2012. "The Global Arrays User Manual." Pacific Northwest National Laboratory, Richland, WA:.–URL http://hpc.pnl.gov/globalarrays/papers/GA-UserManual-Main.pdf.

4. Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation." Computing in Science & Engineering 13, no. 2 (2011): 22-30.