GAMMA: Global Arrays Meets MATLAB*

Rajkiran Panuganti [†] Muthu Manikandan Baskaran [†] David E. Hudak [‡]
Ashok Krishnamurthy [‡] Jarek Nieplocha [§] Atanas Rountev [†] P. Sadayappan [†]

[†] Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210, USA

[‡] Ohio Supercomputer Center
Columbus, OH 43212, USA

[§] Pacific Northwest National Laboratory
Richland, WA 99352, USA

[†] {panugant,baskaran,rountev,saday}@cse.ohio-state.edu
[‡] {dhudak,ashok}@osc.edu [§] jarek.nieplocha@pnl.gov

Abstract

MATLAB has become the dominant high-level language for technical computing. However, MATLAB has significant shortcomings when used for large-scale computationally intensive applications that require very high performance and/or significant amounts of memory. In such contexts, it is common for MATLAB to be first used for prototyping the application, followed by re-implementation in Fortran or C, using MPI for parallel execution. In this paper we describe GAMMA, a parallel global shared-address space framework for MATLAB, built on top of the Global Arrays library suite from Pacific Northwest National Laboratory. GAMMA enables the convenient development of large-scale computationally intensive applications directly in MATLAB, without the need to re-code them to achieve high performance. Preliminary experimental results on a Pentium cluster are provided, that demonstrate the effectiveness of the developed system.

1 Introduction

Computers have made dramatic strides in speed over the last two decades, and the trend towards increased parallelism continues with the advent of multi-core chips. However, unfortunately the difficulty of programming parallel computers has not eased. As computers have increased in achievable performance, making it feasible to accurately model more and more complex phenomena, the time and effort required to develop high-performance software has become the bottleneck in many areas of science and engineering. This is being recognized as one of the most significant challenges today in the effective use of high-performance computers and is highlighted by the DARPA High Productivity Computing Systems (HPCS) program [12] and the reports from the HECRTF (High-End Computing Revitalization Task Force) [23] and SCaLeS (Science Case for Large-scale Simulation) [36] workshops.

High-level languages like MATLAB [22] are being increasingly adopted as an attractive alternative to traditional languages (e.g., C and Fortran) for technical computing in various domains. With over 500,000 licenses sold and an estimated eight million users, MATLAB has established itself as the language of choice for technical computing in many scientific and engineering communities. The popularity of MATLAB is largely due to a programming model that is easy to use, powerful built-in visualization facilities, an integrated development environment, and a variety of domain-specific "toolboxes" (essentially, library components) for developing applications in various fields.

However, currently MATLAB does not meet the computational demands of many high-end compute-intensive scientific applications. MATLAB does not scale well to large problems [7]. As a result, for computationally demanding application domains, MATLAB is often used to rapidly develop a prototype implementation, followed by recoding of the application in a language such as Fortran or C, to improve performance and/or increase dataset sizes that can be handled. The fundamental goal of the ParaM [26] project

*Support for this project was provided to the Ohio Supercomputer Center Springfield Project through the Department of Energy ASC program.

at the Ohio Supercomputer Center (OSC) is the development of an environment that will enable MATLAB users to develop large-scale, high-performance applications without having to recode their application in other languages.

There have been several efforts to address the limitations of MATLAB. Some of these projects have sought to add parallelization facilities to MATLAB, while others have employed compilation to eliminate interpretation overhead. However, the existing work has a number of limitations that constrain its usefulness for real-world MATLAB applications; a detailed discussion of these approaches is presented in Section 4.

MPI (Message Passing Interface) is the most widely used parallel programming model today. With MPI, the same code runs in roughly synchronous fashion across all processors of the parallel computer. It is necessary to explicitly decompose each step of the parallel computation to expose sufficient parallelism to keep all processors load balanced, and to explicitly orchestrate the interaction between the parallel processes. This is particularly challenging with multi-model applications exhibiting complex interaction patterns between elements of different data structures. Parallel message-passing systems are available with MATLAB [9], but their use has been limited due to the significant effort required to develop and debug parallel programs created using the message-passing paradigm [18].

Global shared-address space (GAS) models are considerably easier to program than message-passing, but achieving scalable performance is a difficult challenge. A number of efforts have targeted the development of scalable shared-memory models for parallel computing [35]. However, unlike message-passing with MPI, that has been used successfully in developing scalable parallel applications in all domains of science and engineering, GAS programming models are yet to demonstrate effectiveness in scaling to hundreds of processors over a range of application domains. One of the few notable successes with GAS models is the Global Arrays (GA) suite [24] developed at Pacific Northwest National Laboratory, that efficiently implements a shared-memory abstraction using a "get-compute-put" model for dense matrix computations on clusters and parallel machines with physically distributed memory. For example, NWChem [16], a widely used quantum chemistry suite with over a million lines of source code, has been implemented using GA. Whereas other GAS model efforts face stiff challenges in achieving good scalability due to the generality of their model, GA has succeeded in delivering excellent performance and scalability by focusing on a constrained model for shared-space access.

In this paper we describe GAMMA, a global-shared-address space parallel programming system for MATLAB users. GAMMA is a component of the ParaM effort, and provides technical computing users with the advantages of both MATLAB and parallel computing. Although our initial focus is on MATLAB, the infrastructure is being developed to be adaptable to Octave, a popular open-source clone of MATLAB. Some important advantages of GAMMA are minimal changes or additions to parallelize existing sequential MATLAB code; ability to support a wide variety of distributions; and reuse of the extensive library support in sequential MATLAB as part of a parallel library or application.

The long term goal of our project is to provide a new generation of high productivity computing systems building upon already existing infrastructure in the high performance computing domain. Our goal is to make scalable parallel computer systems easily usable by technical experts in various domains. Our first step towards this goal is the provision of a global shared memory view of distributed MATLAB arrays. An overview of the system and the features of the programming model are described in Section 2. Section 3 discusses the details of our toolbox. A discussion of various efforts to address the performance limitations of sequential MATLAB is presented in Section 4. Section 5 presents results on various benchmarks to demonstrate that GAMMA achieves good scalability. Section 6 presents conclusions and further enhancements that are currently being pursued. Finally, the details of the entire API is presented in the appendix A.

2 Programming Model for GAMMA

GAMMA is a high productivity computing package developed to effectively address all four aspects of a high productivity computing system, namely programmability, performance, robustness, and portability. The GAMMA system has been built as a MATLAB toolbox with parallel constructs using the Global Arrays (GA) package [24] that provides a shared memory programming model on distributed memory parallel machines. The shared memory abstraction for parallel programming effectively addresses the performance limitations of sequential MATLAB, while retaining its ease of programming and robustness.

The GAMMA programming model can be characterized as follows.

- **Global shared view of the distributed data**: The model presents each user with a global shared view of the MATLAB arrays that are physically distributed across various processes. Figure 1 illustrates this model for an array that is distributed across processes $P_0, \ldots, P_3$; the required data that might span across multiple processes can be accessed by referencing it as a single logical block. The communication substrate is transparent to the user and hence the user need not be aware of the physical location of the data when the

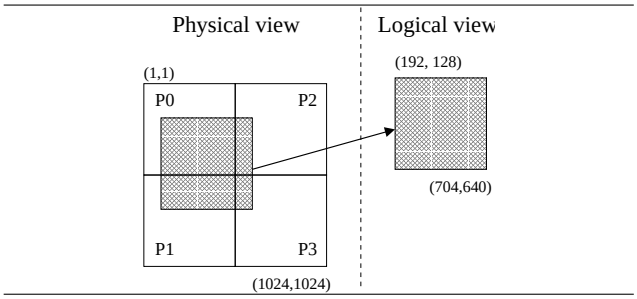

Figure 1. Global shared view of a physically distributed MATLAB array in GAMMA

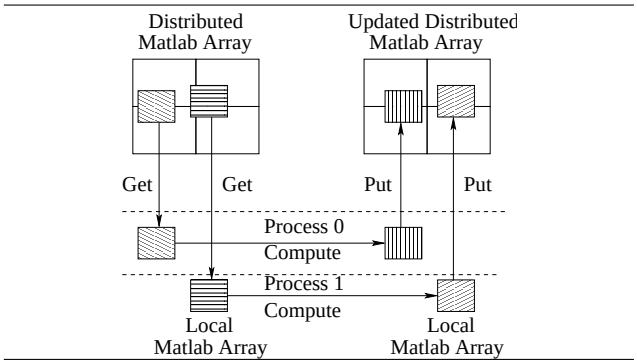

Figure 2. GAMMA Get-Compute-Put computation model

necessary data is accessed. This greatly simplifies the coding of scientific applications on distributed memory parallel computers. The efficient GA layer ensures that the performance overhead of using global shared abstraction of distributed data is very small.

- **Task parallelism and one-sided communication**: GAMMA supports task parallelism by providing operations that can be independently invoked by each process in MIMD style without any cooperation with other processes. The one-sided communication support eliminates unnecessary processor interactions and synchronization, resulting in easy coding.
- **Get-Compute-Put computation**: The model inherently supports a Get-Compute-Put computation style, as illustrated in Figure 2. The data for the computation is fetched from the distributed array independently and asynchronously using a GA_Get routine. Each process then performs computation on the local data using a sequential MATLAB computation engine. The computed data is then stored into the distributed array, again asynchronously and independently, using a GA_Put routine. Because of this model, the GAMMA user can make full utilization of the extensive set of functions provided by sequential MATLAB.
- **Synchronization**: The user is provided with various explicit synchronization primitives to ensure the consistency of the distributed data; examples of such primitives include explicit barrier and fence.
- **Data parallelism**: The model provides support for data parallel operations using collective functions that operate on the distributed data, e.g., common matrix operations such as $transpose$, sum, $scale$, etc.
- **Management of data locality**: The model provides support to control the data distribution and also to access the locality information and therefore gives explicit control to the user to exploit data locality.
- **Data replication**: GAMMA provides support to replicate near-neighbor data, i.e., data residing in the boundary of the remote neighbor process.
- **Support for MPI**: Processes can communicate non-numerical MATLAB data types using message passing primitives.

Our approach differs from other common parallel MATLAB models as follows. Star-P [11] allows only data parallel operations on distributed matrices; it needs to provide native, custom support for the extensive libraries of sequential MATLAB. Unlike pMatlab [33], our model supports one-sided communication and allows a process to access any arbitrary block of data residing in a remote process.

GAMMA operates in two workspaces, namely the MATLAB workspace and the GA workspace. When a distributed array is created in MATLAB, the memory for storing the array elements and the metadata are allocated and maintained in the GA memory manager in the GA workspace, and a handle to the distributed array is returned to the MATLAB workspace. For example, a regular 2D block-distributed array of size 1024×1024 (as shown in Figure 1) is created in MATLAB through $A = GA\_Create([1024, 1024], [1024/sq, 1024/sq])$ where sq is the square root of the number of processes over which the data is distributed, and A is the handle that is returned to the MATLAB space. (The second argument of GA_Create defines the data distributions, as described later.) A logical block of the array, for example $A[192 : 704, 128 : 640]$, can be obtained by any process using a call to GA_Get as in $block = GA\_Get(A, [192, 128], [704, 640])$ where $[192, 128]$ represents the lower indices of the logical block and $[704, 640]$ represents the higher indices of the block. A process can make this call without interaction with any other process, though the data might be local or remote or a combination of both.

Illustration of the GAMMA Model. Figure 3 shows the code for parallel 2D Fast Fourier Transform (FFT) using GAMMA. The code is a straightforward implementation of a standard parallel 2D FFT algorithm. The call to

```
    [ rank   nproc ] = GA_Begin();
% define column distribution
    dims = [ N   N ]; distr = [ N   N/nproc ];
    A = GA_Create(dims, distr);
    ATmp = GA_Create(dims, distr);
    [ loA   hiA ] = GA_Distribution(A, rank);
    GA_Fill(A, 1);
% perform fft on each column of the initial array
    tmp = GA_Get(A, loA, hiA);
    tmp = fft(tmp);
    GA_Put(A, loA, hiA, tmp);
    GA_Sync();
    GA_Transpose(A, ATmp);
    GA_Sync();
% perform fft on each column of the transposed array
    [ loATmp   hiATmp ] = GA_Distribution(ATmp, rank);
    tmp = GA_Get(ATmp, loATmp, hiATmp);
    tmp = fft(tmp);
    GA_Put(ATmp, loATmp, hiATmp, tmp);
    GA_Sync();
    GA_Transpose(ATmp, A);
    GA_Sync();
    GA_End();
```

Figure 3. MATLAB code using GAMMA to develop parallel 2D Fast Fourier Transform

GA_Begin initializes the underlying layers and returns the rank of the process and the total number of processes. The use of *distr* in the call to GA_Create defines the data distribution: each block is of size $N \times (N/nproc)$, and process P_i is assigned the block with logical indices for the upper left corner $(1, 1 + i \times N/nproc)$ and lower right corner $(N, (i + 1) \times N/nproc)$. The example assumes that the global array A is initialized with values of 1, using GA_Fill.

Figure 3 illustrates the ease of programming using GAMMA. Since the user has a global shared view of the distributed data, the code has no complex communication involving data location information. Each process gets the block of data to operate on locally through a one-sided GA_Get routine (the values of loA and hiA are 2-element vectors). A process computes its local result using the sequential built-in *fft* function in MATLAB, and puts back the computed data into the distributed array using a one-sided GA_Put call. The example also demonstrates how the programming model makes use of the functions provided by sequential MATLAB. Furthermore, the collective operation $GA\_Transpose$ does not involve unnecessary data movement between the MATLAB and GA workspaces.

3 Toolbox Details

The architecture of GAMMA is shown in Figure 4. The system has been implemented as a MATLAB toolbox using

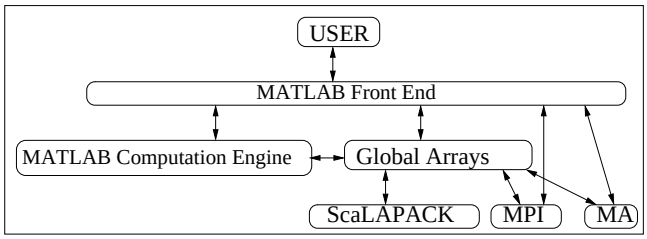

Figure 4. Architecture of GAMMA

the Global Arrays [24] and MVAPICH [20] libraries. The GA library uses Aggregate Remote Memory Copy Interface (ARMCI) [17] in its communication substrate. The memory in the GA layer is dynamically allocated and managed by the Memory Allocator (MA) [32] library. Any temporary buffer space used by a GA operation is allocated and managed in the MA stack, whereas the memory for the distributed GA data is allocated and managed in the MA heap.

The functions of the toolbox can be broadly classified into the following categories:

- **Application launch**: An application developed using the GAMMA toolbox is launched using a launcher that uses either mpirun or mpiexec. The launch mechanism starts MATLAB on each of the processors and sets the environment required for initializing the underlying MPI and ARMCI communication layers.
- **Initialization and termination**: The toolbox is initialized and terminated using GA_Begin and GA_End, respectively. These routines initialize/terminate the environment for MPI, ARMCI, MA, and GA libraries. The size of the heap and stack memory used in the GA space can be specified in a call to GA_Begin.
- **Array creation and destruction:** A distributed MATLAB array is created using the GA_Create routines. The $GA\_Destroy$ routine frees the memory used for the distributed array and its meta data. A detailed description of these routines is presented in appendix A.
- **One-sided data transfer operations:** The toolbox provides one-sided communication routines such as GA_Get, GA_Put, etc. for fetching data from and storing data into the distributed array.
- **Synchronization operations**: Due to the asynchronous nature of the data transfer routines, the operations might return even before the transfer is complete. GAMMA supports synchronization primitives to ensure the consistency of the distributed data. In the FFT example from Figure 3, GA_Sync is used to ensure the completion of the pending data transfers.
- **Utility routines**: The user is provided with routines to inquire about the location and the distribution of the data, as well as routines to fetch and store local data.

- **Collective matrix operations**: The toolbox provides numerous collective operations to perform commonly used matrix operations (e.g., $transpose$, sum, $scale$, etc. appendix A). These operations are optimized because they are implemented using the lower-level GA libraries and do not involve unnecessary data movement between MATLAB and GA. The code in Figure 3 uses $GA\_Transpose$ to perform a transpose of the matrix. A transpose can also be computed using GA_Get and GA_Put calls; however, they incur the additional overhead of data movement between MATLAB and GA.
- **Distributions**: The toolbox supports regular and irregular block distributions of distributed arrays.
- **Global indexing**: Logical blocks of physically distributed arrays are accessed using global indices. The global index translation service is implemented efficiently using the GA libraries.
- **Processor groups**: The toolbox provides a facility to divide the domain into subsets of processors that can act independently of other subsets. This functionality allows improved load balance and offers opportunities for supporting environments with hardware faults.

The implementation of the toolbox presents several challenges. First, MATLAB is an *untyped language*. Hence, the toolbox tracks dynamically the type of the data on which the operations are being performed, in order to make the appropriate GA calls. Further, in MATLAB, a user is not exposed to any explicit memory management routines. Therefore, the user space (i.e., MATLAB space) is managed automatically by the toolbox. GAMMA also handles transfer of data between the MATLAB workspace and the GA workspace. In addition, during the data transfer from the GA workspace to the MATLAB workspace, the toolbox dynamically creates a data block in the MATLAB workspace inferring the type, size, and dimensions of the block from the Get request. Furthermore, the toolbox handles the data layout incompatibility issues between MATLAB and GA and preserves the MATLAB semantics for the user. The toolbox also supports out-of-order (arbitrary) array indexing and thereby preserves an important feature of MATLAB. For example, consider a vector $A[1:100]$. A user can index the vector in an arbitrary fashion as $A([54\ 87\ 15])$.

4 Related Work

The popularity of MATLAB has motivated various research projects to address its performance limitations. These projects vary widely in their approaches and functionalities. Broadly, these efforts can be classified into the five categories described below [10].

4.1 Compilation Approach

Many projects such as Otter [29], RTExpress [31], FALCON [30], CONLAB [14], MATCH [2], Menhir [8], and Telescoping Languages [7] use a compilation-based approach to address the performance limitations of sequential MATLAB by eliminating the overhead of interpretation. Most of these efforts have sought to perform a source-to-source transformation of MATLAB into C, C++, or Fortran, and then use the compilers for these languages. Some projects have also attempted to generate parallel versions of the transformed code using automatic parallelization compilers. With the exception of the continuing work at Rice University, none of the other projects is currently active.

4.2 Embarrassingly Parallel Approach

Research projects such as PLab [19] and Parmatlab [21] provide support for embarrassingly parallel applications in MATLAB. The projects built on this approach launch multiple MATLAB processes on different nodes. Each process works only on its local data and sends the result to the parent process. However, this approach severely limits the type of applications that can be parallelized. None of these projects are currently active.

4.3 Backend Support for Parallelization

Projects such as DLAB [25], Netsolve [6], and Star-P [11] attempt to create a parallel MATLAB environment by providing backend support. This approach uses MATLAB as a front end and the computation is done using a backend parallel computation engine. The backend engine uses high-performance libraries such as ScaLAPACK [3] and FFTW [15]. Star-P is a now a commercial product, currently distributed by SGI.

The backend approach does not require the user to have any specialized knowledge of parallel computing: the high-performance benefits are achieved while retaining all MATLAB features. Thus, it is extremely attractive from the user's point of view. Star-P currently has interfaced with a number of pre-existing parallel numerical libraries such as ScaLAPACK, ARPack etc. However, the development of new functionality for Star-P takes considerable effort. Although Star-P provides a software development kit for the user to add overloaded parallel extensions of needed MATLAB functions, it requires software development with low-level communication primitives, using traditional languages such as C and Fortran. Indeed, GAMMA could be used to develop distributed parallel implementations of MATLAB functions that could be used with Star-P.

4.4 Message Passing Support

Projects such as MultiMATLAB [34], MPITB [1], and MatlabMPI [18] add message passing primitives to MATLAB. With this approach, multiple MATLAB sessions are launched on different nodes. These sessions communicate with each other using a message passing paradigm. With this approach, users have maximum flexibility to build their parallel applications using a basic set of communication primitives. MPI functionality is added to MATLAB using a file-based communication mechanism in MatlabMPI, while in MPITB it is achieved using the LAM/MPI communication library. However, the low-level abstractions of message passing cause a significant increase in the developmental effort for implementing high-performance computationally intensive MATLAB applications.

4.5 Parallel Global Address Space Model

pMatlab [33] provides a parallel global address space programming model for MATLAB users. The system defines a set of data structures for distributed arrays; a user can write an explicitly parallel application with multiple processes, that can operate on individual elements from these shared arrays. In addition, pMatlab defines a limited number of parallel functions that operate on entire distributed arrays (e.g., addition and multiplication). By performing message passing implicitly using MatlabMPI, pMatlab attempts to abstract away the communication details from the MATLAB programmer. However, access of an individual element of the distributed array (using global address space indices), requires interaction of all processes. Unlike GAMMA, pMatlab does not permit concurrent asynchronous access by the processes to arbitrary logical blocks distributed across multiple processes. Furthermore, the index translation mechanism in pMATLAB is inefficient because it is implemented in MATLAB.

5 Experimental Results

In this section, we present experimental results for four parallel algorithms used to evaluate GAMMA: NAS CG benchmark, two-dimensional FFT, two-dimensional convolution, and Jacobi iterative solver. We present data on the achieved performance as well as indicators of the ease of programming.

5.1 Experimental Test bed

The experiments were conducted on the Ohio Supercomputer Center's Intel Pentium 4 cluster constructed from commodity PC components running the Linux operating system. The hardware and software configuration of the cluster is as follows: two 2.4 GHz Intel P4 Xeon processors on each node; 4GB RAM on each node; InfiniBand interconnection network; Red Hat Linux with kernel 2.6.6; MATLAB Version 7.0.1.24704 (R14) Service Pack 1. The parallel programs built using GAMMA were launched using mpiexec [28]. All experiments were conducted such that no two processors were on the same node in the cluster, ensuring that the parallel processing environment is fully distributed.

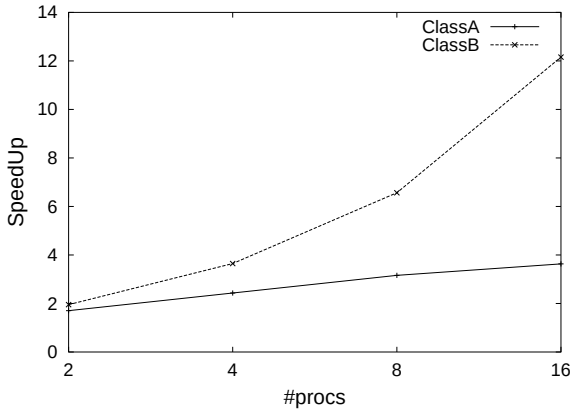

Figure 5. Speedup for NAS CG benchmark

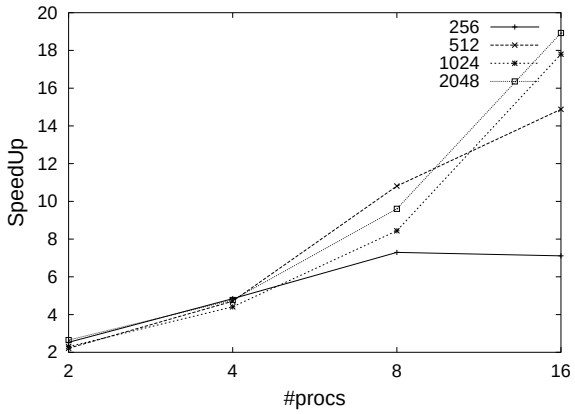

Figure 6. Speedup for parallel FFT2

5.2 Performance Analysis

We compared the performance of the parallel algorithms implemented using our toolbox with their sequential counterparts written in MATLAB Version 7.0 (with the just-in-time compilation feature enabled). In the sequential version of the algorithms, the highly efficient built-in MATLAB functions have been used wherever possible. The parallel algorithms implemented using GAMMA show good scalability as the problem size increases.

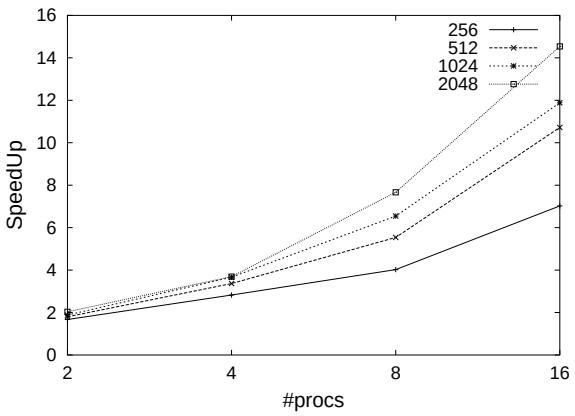

Figure 7. Speedup for parallel convolution

- **NAS benchmark CG**: The NAS Conjugate Gradient (CG) benchmark is a scientific kernel that has been used to compare the characteristics of different programming models. This benchmark uses an inverse power method to find the largest eigenvalue of a random sparse matrix. In the parallel version implemented using GAMMA, each process was assigned a block of the random sparse matrix distributed column wise. Figure 5 shows the speedup obtained for class A and class B. For class A, which has smaller problem size, the parallelization overhead dominates the execution time and therefore the speedup curve flattens. However, Class B, with much larger problem size, scales well and attains good speedup.
- **Two-dimensional FFT (FFT2)**: FFT2 is an algorithm to compute the two-dimensional fast Fourier transform. FFT is widely used in a variety of applications ranging from digital signal processing to solving partial differential equations. FFT2 is available as a built-in library function in MATLAB, implemented using the FFTW algorithm [15]. Figure 6 shows that the parallel version attains good scalability as the size of the matrix increases. For matrix sizes N $\leq$ 256, due to the parallelization overhead, the speedup does not increase proportionately as the number of processes increases.
- **Two-dimensional convolution**: Convolution is an operation that is central to many image processing and signal processing applications. It is a linear filtering technique in which the input image (matrix) is scanned by a filter (matrix of weights) or the convolution kernel. The value at each output pixel is the weighted sum of the neighboring input pixels. Convolution is a built-in library function in MATLAB. In the parallel version of the convolution algorithm, implemented using GAMMA, the input and output images (matrices) were distributed in a two dimensional block distributed fashion and the filter (kernel) was replicated on all processes. Figure 7 shows that the parallel version achieves significant scalability and as the size of the input matrix increases, the speedup obtained also increases.
- **Jacobi iterative solver**: The Jacobi iterative solver uses the Jacobi method to solve a linear system of equations arising in the solution of a discretized partial differential equation. All data were distributed using a one-dimensional block distribution. Figure 8 shows that the parallel version implemented using GAMMA achieves an almost linear speedup.

We compared our performance results with the most popularly used parallel MATLAB packages for distributed memory systems: MatlabMPI and pMatlab. MatlabMPI suggests various optimizations [13] (tuning) of the underlying system installation that can improve the performance of applications built using it. However, a user can enable only those optimizations that are permitted in the experimental environment. We have tuned the underlying system at OSC within the extent of privileges permitted to an OSC user. Applications written using GAMMA show good performance when compared to those written using MatlabMPI and pMatlab. For example, parallel 2D FFT of a matrix of size (2048×2048) implemented using GAMMA takes 0.776649, 0.407208, 0.213695, and 0.105113 seconds on 2, 4, 8, and 16 processes, respectively. However, the same problem takes 59.537, 59.679, 59.791, and 59.976 seconds in MatlabMPI and 60.125, 60.347, 60.634, and 60.982 seconds in pMatlab on 2, 4, 8, and 16 processes, respectively. A constant execution time is observed with MatlabMPI and pMatlab as they use a file-based communication mechanism and the performance is severely limited by the NFS (Network File System) latency.

5.3 Programmability

Even though there does not exist an ideal metric for evaluating the programmability of a parallel language/toolbox, the number of source lines of code (SLOC) provides some indication on the ease of programming [4]. Hence, we evaluated the ease of programming with GAMMA using SLOC measurements.

Table 9 compares the source lines of code required to implement the parallel version of the four benchmarks using GAMMA with the sequential version implemented in MATLAB. It can be observed that the parallel version can be implemented with only a small increase in the lines of code. Further, compared to the standard MPI-based F77 implementation of NAS CG benchmark, the SLOC [5] is reduced approximately by a factor of 11.

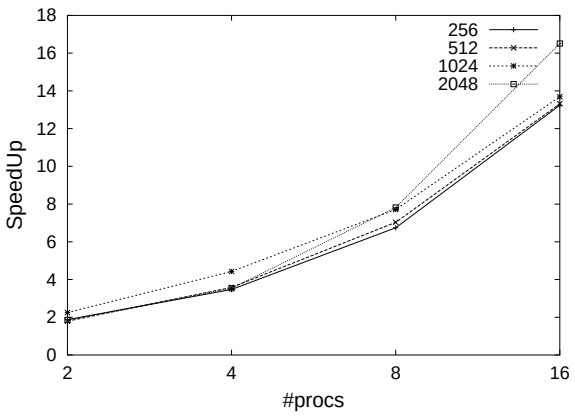

Figure 8. Speedup for parallel Jacobi iterative solver

Application	MATLAB	GAMMA
2D FFT	1 (built-in)	19
2D Convolution	1 (built-in)	26
Jacobi Solver	35	56
NAS CG	59	98

Figure 9. Lines of source code for the benchmarks

5.4 Memory scalability

The use of MATLAB for large-scale computationally intensive applications is limited by the problem sizes that it can handle on a single processor. In addition to the enhanced performance, another significant benefit of GAMMA to MATLAB users is the ability to handle larger problem sizes, with data distributed across multiple processors. For example, using sequential MATLAB, the largest matrix size that the 2D convolution routine of MATLAB (`conv2`) could handle was 9100×9100 using a filter of size 32×32. However, using GAMMA, we were able to perform two dimensional convolution on a matrix of size 20000×20000 using the same filter. Similarly, for the FFT2, 8900×8900 was the largest case sequential MATLAB was able to run, while the GAMMA version on 8 processors was able to run a 17000×17000 case.

6 Conclusions and Future Work

This paper has described GAMMA, a parallel MATLAB environment providing a global shared view of arrays that are physically distributed on clusters, and a get-compute-put model of parallel computation. Several examples were provided, illustrating the ease of programming coupled with high efficiency. Use of the get-compute-put model also facilitates effective reuse of existing sequential MATLAB libraries as part of a parallel application. An additional benefit of using the GAMMA system is the ability to run larger problems than sequential MATLAB.

The GAMMA toolbox is currently being used by the staff and users at the Ohio Supercomputer Center and will soon be made available for public release. GAMMA is part of a larger effort to develop a high-productivity environment called ParaM [26] - a Parallel MATLAB project that aims at combining compilation technology along with parallelization, to enable very high performance. Efforts are currently underway to develop a number of parallel MATLAB applications and numerical libraries using GAMMA.

References

[1] J. F. Baldomero. MPI/PVM toolbox for Matlab. http://atc.ugr.es/javier-bin/pvmtb_eng.

[2] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *Symposium on Field-Programmable Custom Computing Machines*, pages 39–49, 2000.

[3] L. Blackford, J.Choi, A.Cleary, E. D. Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Publishing, Philadelphia, PA, 1997.

[4] L. Briand, T. Langley, and I. Wieczorek. A replicated assessment and comparison of common software cost modeling techniques. In *International Conference on Software Engineering*, pages 377–386, 2000.

[5] F. Cantonnet, Y. Yao, M. M. Zahran, and T. A. El-Ghazawi. Productivity analysis of the upc language. In *IPDPS*, 2004.

[6] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *ACM/IEEE Conference on Supercomputing*, page 40, 1996.

[7] A. Chauhan. *Telescoping MATLAB for DSP Applications*. PhD thesis, Rice University, 2003.

[8] S. Chauveau and F. Bodin. Menhir: An environment for high performance Matlab. In *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LNCS 1511, pages 27–40, 1998.

[9] R. Choy. Parallel Matlab survey. http://supertech.lcs.mit.edu/∼cly/survey.

[10] R. Choy and A. Edelman. Parallel MATLAB: Doing it right. *Proceedings of the IEEE*, 93(2):331–341, Feb. 2005.

[11] R. Choy, A. Edelman, J. R. Gilbert, V. Shah, and D. Cheng. Star-P: High productivity parallel computing. In *Workshop on High Performance Embedded Computing*, 2004.

[12] DARPA. High productivity computing systems (HPCS) program. http://www.highproductivity.org.

[13] Dr. Jeremy Kepner. Parallel Programming with MatlabMPI . http://www.ll.mit.edu/MatlabMPI/.

[14] P. Drakenberg, P. Jacobsen, and B. Kåström. A CONLAB compiler for a distributed memory multicomputer. In *SIAM Conference on Parallel Processing for Scientific Computing*, pages 814–821, 1993.

[15] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 1381–1384, 1998.

[16] High Performance Computational Chemistry Group. NWChem, a computational chemistry package for parallel computers. www.emsl.pnl.gov/docs/nwchem/nwchem.html.

[17] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *IPPS/SPDP Workshops*, LNCS 1586, pages 533–546, 1999.

[18] J. Kepner and S. Ahalt. MatlabMPI. *Journal of Parallel and Distributed Computing*, 64(8):997–1005, Aug. 2004.

[19] U. Kjems. PLab: reference page, 2000. http://bond.imm.dtu.dk/plab/.

[20] J. Liu, J. Wu, S. Kini, D. Buntinas, W. Yu, B. Chandrasekaran, R. Noronha, P. Wyckoff, and D. Panda. MPI over InfiniBand: Early experiences. Technical Report OSU-CISRC-10/02-TR25, Ohio State University, Oct. 2002.

[21] Lucio Andrade. Parmatlab, 2001.

[22] Mathworks Inc. MATLAB 7 user's guide [online]. http://www.mathworks.com.

[23] NCO/NITRD. High-end computing revitalization task force. http://www.nitrd.gov/subcommittee/hec/hecrtf-outreach.

[24] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, June 1996.

[25] B. R. Norris. An environment for interactive parallel numerical computing. Technical Report 2123, Urbana, Illinois, 1999.

[26] Ohio Supercomputer Center. ParaM: Compilation of MATLAB for parallel execution. http://www.osc.edu/springfield/research/matlab.shtml.

[27] R. Panuganti, M. M. Baskaran, D. Hudak, A. Krishnamurthy, J. Nieplocha, A. Rountev, and P. Sadayappan. GAMMA: Global Arrays meets MATLAB. Technical Report OSU-CISRC-1/06-TR15, Ohio State University, Jan. 2006.

[28] Pete Wyckoff. Mpiexec. http://www.osc.edu/hpc/software/apps/mpiexec.shtml.

[29] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel MATLAB compiler. In *International Parallel Processing Symposium*, pages 81–87, 1998.

[30] L. D. Rose, K. Gallivan, E. Gallopoulos, B. A. Marsolf, and D. A. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288, 1995.

[31] RTExpress. Integrated Sensors Inc. http://www.rtexpress.com.

[32] G. Thomas. Memory Allocator. http://www.emsl.pnl.gov/docs/parsoft/ma/MAapi.html.

[33] N. Travinin, R. Bond, J. Kepner, and H. Kim. pMatlab: High Productivity, High Performance Scientific Computing. In *2005 SIAM Conference on Computational Science and Engineering*, 2005.

[34] A. E. Trefethen, V. S. Menon, C.-C. Chang, G. Czajkowski, C. Myers, and L. N. Trefethen. MultiMATLAB: MATLAB on multiple processors. Technical Report TR96-239, Cornell Theory Center, Cornell University, 1996.

[35] UC Berkeley/LBNL. Berkeley UPC - Unified Parallel C. http://upc.nersc.gov.

[36] U.S. Department of Energy, Office of Research. SCaLeS (Science Case for Large-scale Simulation) workshop, June 2003. http://www.pnl.gov/scales.

A API

A.1 Initialization

- **GA_Begin**

  Syntax:
  ```
  GA_Begin()
  GA_Begin(heap)
  GA_Begin(heap,stack)
  ```

  Description:
  Initializes the GA environment. This includes initializing the MPI layer, the memory allocator (MA), and the Global Arrays space. The user may specify the size of heap and stack space required for the entire program. The default value for heap space is 512MB and stack space is 64MB. If the application requires larger amount of memory to be managed dynamically in the global space, heap and stack values can be mentioned accordingly.

  Note:
  This is a collective operation.

  Example:
  ```
  heap = 64*1024*1024;
  stack = 8*1024*1024;
  GA_Begin(heap,stack);
  GA_End();
  ```

  See Also
  `GA_End`

- **GA_End**

  Syntax:
  ```
  GA_End()
  GA_End([ArrayHandle1,...])
  ```

  Description:
  Terminates the GA environment and destroys the Global Arrays that were created and frees all allocated memory.

  Note:
  This is a collective operation.

  Example:
  ```
  GA_Begin();
  A = GA_Create([2 2]);
  B = GA_Create([4 4]);
  GA_End(A,B);
  ```

  See Also
  `GA_Begin, GA_Create`

- **GA_Create**

  Syntax:
  ```
  [handle] = GA_Create(dims)
  [handle] = GA_Create(dims, chunk)
  ```
  dims - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the size or extent of the ith dimension of the global array
  chunk - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the minimum size that the ith dimension of the global array must be divided into among the processes

  Description:
  Creates a Global Array with distribution as specified by dims and chunk. Specifying $chunk[i] \leq 1$ will cause that dimension to be distributed evenly. If chunk is not specified, the entire array is distributed evenly, i.e. the elements are distributed such that each process gets equal number of elements along each dimension.

  Note:
  This is a collective operation.
  The handle is a numeric handle. It has to be used only with GAMMA functions. Using it with non-GAMMA functions might give undesired results.

  Example:
  ```
  GA_Begin();
  dims = [8 8]; chunk = [2 8];
  A = GA_Create(dims, chunk);
  GA_End(A);
  ```

  See Also
  `GA_Create_ghosts, GA_Duplicate`

- **GA_Create_irreg**

  Syntax:
  ```
  [handle] = GA_Create_irreg(dims,
  block, map)
  ```
  dims - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the size or extent of the ith dimension of the global array
  block - vector of 'n' elements where 'n' is the number

of dimensions of the global array to be created and the ith element in the vector denotes the number of blocks that the ith dimension of the global array must be divided into among the processes
map - vector indicating the starting index of each block

Description:
Creates an array as per the user-specified distribution information. The distribution is specified as a Cartesian product of distributions for each dimension.

Note:
This is a collective operation.

Example:
```
GA_Begin();
dims = [8 10]; block = [3 2];
map = [1,3,7,1,6];
A = GA_Create_irreg(dims, block,
map);
GA_End(A);
```
The above example demonstrates the distribution of a 2-dimensional array 8x10 on 6 (or more) processes. The distribution is nonuniform because, P1 and P4 get 20 elements each while P0,P2,P3, and P5 get only 10 elements each.

See Also
`GA_Duplicate, GA_Destroy, GA_Create`

- **GA_Destroy**

Syntax:
`GA_Destroy(ArrayHandle)`

Description:
Destroys the Global Array $ArrayHandle$ and frees the allocated memory.

Note:
This is a collective operation.

Example:
```
A = GA_Create([2 2]);
GA_Destroy(A);
```

See Also
`GA_Create`

- **GA_Duplicate**

Syntax:
`[new_handle] = GA_Duplicate(ArrayHandle)`

Description:
Creates a new array by applying all the properties of another existing array. It returns a new array handle.

Note:
This is a collective operation. Note that only the array properties are reflected and not the values of the elements.

Example:
```
GA_Begin();
A = GA_Create([2 2]);
B = GA_Duplicate(A);
GA_End(A,B);
```

See Also
`GA_Create, GA_Copy`

- **GA_Copy**

Syntax:
`GA_Copy(From_ArrayHandle, To_ArrayHandle)`

Description:
Copies elements from source array (From_ArrayHandle) to destination array (To_ArrayHandle).
Note:
This is a collective operation. Note that the arrays must be of the same shape and identically aligned.

Example:
```
GA_Begin();
A = GA_Create([2 2]);
B = GA_Duplicate(A);
GA_Fill(A,2);
%To perform the operation B = A;
GA_Copy(A,B);
GA_End(A,B);
```

See Also
`GA_Get, GA_Put, GA_Copy_patch`

- **GA_Copy_patch**

Syntax:
`GA_Copy_patch(From_ArrayHandle, From_lo, From_hi, To_ArrayHandle, To_lo, To_hi)`

Description:
Copies a patch of source array (From_ArrayHandle) to a patch of destination array (To_ArrayHandle).

Note:
This is a collective operation. Note that the array patches must be of the same shape and identically aligned.

Example:
```
GA_Begin();
A = GA_Create([8 8]);
B = GA_Duplicate(A);
GA_Fill(A,2);
GA_Copy(A, [2 4], [4 6], B, [3 5],
[5 7]);
GA_End(A,B);
```

See Also
GA_Get, GA_Put, GA_Copy

A.2 One-Sided

- **GA_Get**

Syntax:
```
[local_buffer] =
GA_Get(ArrayHandle,lo, hi)
```

Description:
Copies data asynchronously into the local array buffer of the calling process from a section of global array.

Note:
Task parallelism can be exploited using GA_Get and GA_Put functions. Perform GA_Sync before GA_Get to avoid any kind of inconsistencies. Inconsistencies might result due to load imbalance if GA_Sync is not used before GA_Get.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
GA_Fill(A,5);
% For transferring data from
[2:5,4:7] section of 2-dimensional
8x8 global array into local buffer.
buf = GA_Get(A, [2 4], [5 7]);
GA_End(A);
```

See Also
GA_Put, GA_Get_local, GA_Sync

- **GA_Put**

Syntax:
```
GA_Put(ArrayHandle, lo, hi, buf)
```

Description:
Copies data asynchronously from the local array buffer of the calling process to a section of global array.

Note:
Task parallelism can be exploited using GA_Get and GA_Put functions. Perform GA_Sync after GA_Put to avoid any kind of inconsistencies. Inconsistencies might result due to load imbalance if GA_Sync is not used after GA_Put.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
buf = ones(4,2);
% For transferring data from the
local buffer into [2:5,4:7] section
of 2-dimensional 8x8 global array.
GA_Put(A, [2 4], [5 7], buf);
GA_End(A);
```

See Also
GA_Get, GA_Put_local, GA_Fill, GA_Sync

- **GA_Get_local**
Syntax:
```
[local_buffer] =
GA_Get_local(ArrayHandle)
```

Description:
Copies data from the portion of global array owned by the calling process to the local array buffer.

Example:
```
GA_Begin();
A = GA_Create(2,[8 8], [2 8]);
GA_Fill(A,5);
buf = GA_Get_local(A);
GA_End(A);
```

See Also
GA_Put_local, GA_Get, GA_Sync

- **GA_Put_local**

Syntax:
`GA_Put_local(ArrayHandle, buf)`

Description:
Copies data from the local array buffer to the portion of global array owned by the calling process.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
buf = ones(4,2);
GA_Put_local(A, buf);
GA_End(A);
```

See Also
`GA_Get_local, GA_Put, GA_Sync`

- **GA_Acc**

Syntax:
`GA_Acc(ArrayHandle, lo, hi, buf, alpha)`

Description:
Performs the below operation: (Let A be the Global Array.)
`A(lo[1]:hi[1], lo[2]:hi[2]) = A(lo[1]:hi[1], lo[2]:hi[2]) + alpha*buf(1:hi[1]-lo[1]+1, 1:hi[2]-lo[2]+1)`
Note:
This is an atomic operation.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
buf = ones(4,2);
alpha = 5; GA_Acc(A, [2 4], [5 7], buf, alpha);
GA_End(A);
```

A.3 Synchronization

- **GA_Sync**

Syntax:
`GA_Sync()`

Description:
Synchronizes processes (a barrier) and ensures that all GA operations are complete.

Note:
This is a collective operation.

Example:
```
GA_Begin();
GA_Sync();
GA_End();
```

See Also
`GA_Fence`

- **GA_Init_fence**

Syntax:
`GA_Init_fence()`

Description:
Initializes tracing of completion status of data transfer operations.

Note:
This is a local operation. GA_Init_fence and GA_Fence must be used in pairs.

See Also
`GA_Fence`

- **GA_Fence**

Syntax:
`GA_Fence()`

Description:
Blocks the calling process until all the data transfers corresponding to GA operations called after GA_Init_fence complete.

Note:
This is a local operation. GA_Init_fence and GA_Fence must be used in pairs.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
buf = ones(4,2);
GA_Init_fence(); GA_Put(A, [2 4], [5 7], buf);
GA_Fence(); GA_End(A);
```

See Also

GA_Init_fence

A.4 Collectives

- **GA_Fill**

  Syntax:
  GA_Fill(ArrayHandle, value)

  Description:
  Assign a single value to all the elements in the array.

  Example:
  ```
  GA_Begin();
  A = GA_Create([8 8], [2 8]);
  GA_Fill(A,5);
  GA_End(A);
  ```

  See Also
  GA_Fill_patch

- **GA_Fill_patch**

  Syntax:
  GA_Fill_patch(ArrayHandle, lo, hi, value)

  Description:
  Assign a single value to all the elements in the global array patch.

  Example:
  ```
  GA_Begin();
  A = GA_Create([8 8], [2 8]);
  GA_Fill_patch(A,[3 5], [6 8], 5);
  GA_End(A);
  ```

  See Also
  GA_Fill

- **GA_Zero**

  Syntax:
  GA_Zero(ArrayHandle)

  Description:
  Sets the value of all the elements in the array to zero.

  Example:
  ```
  GA_Begin();
  A = GA_Create([8 8], [2 8]);
  GA_Zero(A);
  GA_End(A);
  ```

  See Also
  GA_Zero_patch

- **GA_Zero_patch**

  Syntax:
  GA_Zero_patch(ArrayHandle, lo, hi)

  Description:
  Sets the value of all the elements in the global array patch to zero.

  Example:
  ```
  GA_Begin();
  A = GA_Create([8 8], [2 8]);
  GA_Zero_patch(A, [3 5], [6 8]);
  GA_End(A);
  ```

  See Also
  GA_Zero

- **GA_Scale**

  Syntax:
  GA_Scale(ArrayHandle, value)

  Description:
  Scales an array by a constant.

  Example:
  ```
  GA_Begin();
  A = GA_Create([8 8], [2 8]);
  GA_Fill(A,5);
  GA_Scale(A,2);
  GA_End(A);
  ```

  See Also
  GA_Add_scale

- **GA_Add_scale**

  Syntax:
  GA_Add_scale(scalar1, SrcArrayHandle1, scalar2, SrcArrayHandle2,DestnArrayHandle)

  Description:
  Scales the source arrays, $SrcArrayHandle1$

and $SrcArrayHandle2$, by the corresponding scalar values, $scalar1$ and $scalar2$, and then performs an element-wise addition of the scaled arrays and stores the result in the destination array, $DestnArrayHandle$.

Example:

```
GA_Begin();
A = GA_Create([8 8], [2 8]);
B = GA_Duplicate(A);
C = GA_Duplicate(A);
GA_Fill(A,5);
GA_Fill(B,6);
GA_Add_scale(2,A,3,B,C);
GA_End(A,B,C);
```

See Also
`GA_Scale`

- **GA_Add**

Syntax:
`GA_Add(SrcArrayHandle1, SrcArrayHandle2,DestnArrayHandle)`

Description:
Performs an element-wise addition of the source arrays, $SrcArrayHandle1$ and $SrcArrayHandle2$, and stores the result in the destination array, $DestnArrayHandle$.

Example:

```
GA_Begin();
A = GA_Create([8 8], [2 8]);
B = GA_Duplicate(A);
C = GA_Duplicate(A);
...
GA_Add(A,B,C);
GA_End(A,B,C);
```

See Also
`GA_Minus`

- **GA_Minus**

Syntax:
`GA_Minus(SrcArrayHandle1, SrcArrayHandle2, DestnArrayHandle)`

Description:
Performs an element-wise subtraction of the source arrays, $SrcArrayHandle1$ and $SrcArrayHandle2$, and stores the result in the destination array, $DestnArrayHandle$.

Example:

```
GA_Begin();
A = GA_Create([8 8], [2 8]);
B = GA_Duplicate(A);
C = GA_Duplicate(A);
...
GA_Minus(A,B,C);
GA_End(A,B,C);
```

See Also
`GA_Add`

- **GA_Add_patch**

Syntax:
`GA_Add_patch(SrcArrayHandle1, lo1, hi1, SrcArrayHandle2, lo2, hi2, DestnArrayHandle, lo, hi)`

Description:
Performs an element-wise addition of source array patches and stores the result in a patch of destination array.

Example:

```
GA_Begin();
A = GA_Create([8 8]);
B = GA_Duplicate(A);
C = GA_Duplicate(A);
...
GA_Add_patch(A, [2 4], [4 6], B, [3 5], [5 7], C, [3 5], [5 7]);
GA_End(A,B,C);
```

See Also
`GA_Subtract_patch`

- **GA_Subtract_patch**

Syntax:
`GA_Subtract_patch(SrcArrayHandle1, lo1, hi1, SrcArrayHandle2, lo2, hi2, DestnArrayHandle, lo, hi)`

Description:
Performs an element-wise subtraction of source array patches and stores the result in a patch of destination array.

Example:

```
GA_Begin();
A = GA_Create([8 8]);
B = GA_Duplicate(A);
C = GA_Duplicate(A);
...
GA_Subtract_patch(A, [2 4], [4 6],
B, [3 5], [5 7], C, [3 5], [5 7]);
GA_End(A,B,C);
```

See Also
GA_Add_patch

- **GA_Symmetrize**

Syntax:
GA_Symmetrize(ArrayHandle)

Description:
Symmetrizes a matrix.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
...
GA_Symmetrize(A);
GA_End(A);
```

See Also
GA_Transpose

- **GA_Transpose**

Syntax:
GA_Transpose(SrcArrayHandle, DestArrayHandle)

Description:
Transposes a matrix.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
B = GA_Duplicate(A);
...
GA_Transpose(A,B);
GA_End(A,B);
```

See Also
GA_Symmetrize

- **GA_OP**

Syntax:
GA_OP(vect, op)

Description:
Performs a reduction of the elements of the vector, *vect*, across all nodes using the commutative operator, *op*. The result is broadcast to all nodes. Supported operations include '+', '$*$', 'max', 'min', 'absmax', 'absmin'.

Example:
```
GA_Begin();
A = GA_Create([8 8], [2 8]);
x = GA_Get_local(A);
...
GA_OP(x,'+');
GA_End(A);
```

A.5 ElementWise

- **GA_Abs**

Syntax:
GA_Abs(ArrayHandle)

Description:
Takes the in-place absolute value of the entire Global Array.

Note:
This is a collective operation.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Abs(A);
GA_End(A);
```

See Also
GA_Abs_patch

- **GA_Abs_patch**

Syntax:
GA_Abs_patch(ArrayHandle, lo, hi)

Description:
Takes the in-place absolute value of the Global Array patch specified.

Note:
This is a collective operation.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Abs_patch(A, [1 2], [2 3]);
GA_End(A);
```

See Also
GA_Abs

- **GA_Add_constant**

Syntax:
GA_Add_constant(ArrayHandle, value)

Description:
Adds the scalar, $value$,to each element of the Global Array.

Note:
This is a collective operation.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Add_constant(A, 3);
GA_End(A);
```

See Also
GA_Add_constant_patch, GA_Add

- **GA_Add_constant_patch**

Syntax:
GA_Add_constant_patch(ArrayHandle, lo, hi, value)

Description:
Adds the scalar, $value$,to each element of the Global Array patch.

Note:
This is a collective operation.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Add_constant_patch(A, [1 1], [2
2], 3);
GA_End(A);
```

See Also
GA_Add_constant, GA_Add

- **GA_Elem_multiply**

Syntax:
GA_Elem_multiply(ArrayHandle1, ArrayHandle2, ArrayHandleResult)

Description:
Computes the element-wise product of two arrays.

Note:
This is a collective operation.
The arrays must be of the same shape.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Fill(B,5);
GA_Elem_multiply(A, B, C);
GA_End(A, B, C);
```

See Also
GA_Elem_multiply_patch

- **GA_Elem_multiply_patch**

Syntax:
GA_Elem_multiply_patch(ArrayHandle1, lo1[], hi1[], ArrayHandle2, lo2[], hi2[], ArrayHandleResult, resultLo[], resultHi[])

Description:
Computes the element-wise product of two array patches.

Note:
This is a collective operation.
The array patches must be of the same shape.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
```

```
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Fill(B,5);
GA_Elem_multiply_patch(A, [1 1], [2
2], B, [1 1], [2 2], C, [1 1], [2
2]);
GA_End(A, B, C);
```

See Also
`GA_Elem_multiply`

- **GA_Elem_divide**

Syntax:
`GA_Elem_divide(ArrayHandle1, ArrayHandle2, ArrayHandleResult)`

Description:
Computes the element-wise quotient of two arrays.

Note:
This is a collective operation.
The arrays must be of the same shape.
The result (quotient) array may replace one of the input (dividend or divisor) arrays.
If any element of the divisor array is zero, the corresponding element of the quotient array will be set to the smallest negative integer, $GA\_NEGATIVE\_INFINITY$.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Fill(B,5);
GA_Elem_divide(A, B, C);
GA_End(A, B, C);
```

See Also
`GA_Elem_divide_patch`

- **GA_Elem_divide_patch**

Syntax:
`GA_Elem_divide_patch(ArrayHandle1, ArrayHandle2, ArrayHandleResult)`

Description:
Computes the element-wise quotient of two array patches.

Note:
This is a collective operation.
The arrays must be of the same shape.
The result (quotient) array may replace one of the input (dividend or divisor) arrays.
If any element of the divisor array is zero, the corresponding element of the quotient array will be set to the smallest negative integer, $GA\_NEGATIVE\_INFINITY$.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Fill(B,5);
GA_Elem_divide_patch(A, [1 1], [2 2],
B, [1 1], [2 2], C, [1 1], [2 2]);
GA_End(A, B, C);
```

See Also
`GA_Elem_divide`

- **GA_Elem_maximum**

Syntax: `GA_Elem_maximum(ArrayHandle1, ArrayHandle2, ArrayHandleResult)`

Description:
Computes the element-wise maximum of two arrays.

Note:
This is a collective operation.
The arrays must be of the same shape.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,2);
GA_Fill(B,5);
GA_Elem_maximum(A, B, C);
GA_End(A, B, C);
```

See Also
`GA_Elem_minimum`

- **GA_Elem_maximum_patch**

Syntax:
```
GA_Elem_maximum_patch(ArrayHandle1,
lo1[], hi1[], ArrayHandle2,
lo2[], hi2[], ArrayHandleResult,
resultLo[], resultHi[])
```

Description:
Computes the element-wise maximum of two array patches.

Note:
This is a collective operation.
The arrays must be of the same shape.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Fill(B,5);
GA_Elem_maximum_patch(A, [1 1], [2
2], B, [1 1], [2 2], C, [1 1], [2
2]);
GA_End(A, B, C);
```

See Also
GA_Elem_maximum

- **GA_Elem_minimum**

Syntax:
```
GA_Elem_minimum(ArrayHandle1,
ArrayHandle2, ArrayHandleResult)
```

Description:
Computes the element-wise minimum of two arrays.

Note:
This is a collective operation.
The arrays must be of the same shape.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,2);
GA_Fill(B,5);
GA_Elem_minimum(A, B, C);
GA_End(A, B, C);
```

See Also
GA_Elem_maximum

- **GA_Elem_minimum_patch**

Syntax:
```
GA_Elem_minimum_patch(ArrayHandle1,
lo1[], hi1[], ArrayHandle2,
lo2[], hi2[], ArrayHandleResult,
resultLo[], resultHi[])
```

Description:
Computes the element-wise minimum of two array patches.

Note:
This is a collective operation.
The arrays must be of the same shape.

Example:
```
GA_Begin();
A = GA_Create(2, [8 8], [2 8]);
B = GA_Create(2, [8 8], [2 8]);
C = GA_Create(2, [8 8], [2 8]);
GA_Fill(A,5);
GA_Fill(B,5);
GA_Elem_minimum_patch(A, [1 1], [2
2], B, [1 1], [2 2], C, [1 1], [2
2]);
GA_End(A, B, C);
```

See Also
GA_Elem_minimum

A.6 Ghosts

- **GA_Create_ghosts**

Syntax:
```
[handle] = GA_Create_ghosts(dims,
width, chunk)
```
dims - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the size or extent of the ith dimension of the global array
width - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the ghost cell width in the ith dimension of the global array
chunk - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the minimum

size that the ith dimension of the global array must be divided into among the processes

Description:
Similar to GA_Create; creates a Global Array with distribution as specified by dims and chunk. Further, the local portion of the global array residing on each processor will have a layer of ghost cells of width, $width[i]$, on either side of the visible data along the ith dimension.

Note:
This is a collective operation.

Example:
```
GA_Begin();
dims = [8 10]; width = [1 0];
chunk = [8/np 10];
A = GA_Create_ghosts(dims, width,
chunk);
GA_End(A);
```

See Also
`GA_Create`

- **GA_Create_ghosts_irreg**

Syntax:
`[handle] = GA_Create_ghosts_irreg(dims, width, block, map)`
dims - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the size or extent of the ith dimension of the global array
width - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the ghost cell width in the ith dimension of the global array
block - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the number of blocks that the ith dimension of the global array must be divided into among the processes
map - vector indicating the starting index of each block

Description:
Similar to GA_Create_irreg; creates an array as per the user-specified distribution information (specified as a Cartesian product of distributions for each dimension). Further, the local portion of the global array residing on each processor will have a layer of ghost cells of width, $width[i]$, on either side of the visible data along the ith dimension.

Note:
This is a collective operation.

Example:
```
GA_Begin();
dims = [8 10]; width = [1 0];
block = [3 2];
map = [1 3 7 1 6];
A = GA_Create_ghosts_irreg(dims,
width, block, map);
GA_End(A);
```

See Also
`GA_Create_irreg`

- **GA_Has_ghosts**

Syntax:
`[handle] = GA_Has_ghosts(ArrayHandle)`

Description:
Returns 1 if the global array, $ArrayHandle$ has some dimensions for which the ghost cell width is greater than zero, returns 0 otherwise.

Note:
This is a collective operation.

Example:
```
GA_Begin();
A = GA_Create_ghosts([4 4],[1 0],[4
1]); has_ghosts = GA_Has_ghosts(A);
GA_End(A);
```

See Also
`GA_Create_ghosts`

- **GA_Set_ghosts**

Syntax:
`[handle] = GA_Set_ghosts(ArrayHandle, width)`

Description:
Sets the ghost cell widths for a global array.

Note:
This is a collective operation.

Example:
```
GA_Begin();
```

```
A = GA_Create([4 4], [4 1]);
GA_Set_ghosts(A,[1 0]);
GA_End(A);
```

See Also
GA_Create_ghosts

A.7 Utilities

- **GA_Distribution**

  Syntax:
  `[lo, hi] = GA_Distribution(ArrayHandle)`

  Description:
  Returns the index range of the global array portion owned by the calling process. If no array elements are owned by the calling process, the range is returned as $lo[i] = 0$ and $hi[i] = -1$ $fori = 1 : ndim$ dimensions.

  Example:
  ```
  GA_Begin();
  A = GA_Create([4 4], [4 1]); [lo hi]
  = GA_Distribution(A);
  GA_End(A);
  ```

  See Also
  GA_Create

- **GA_Compare_distr**

  Syntax:
  `isSimillar = GA_Compare_distr(ArrayHandle1, ArrayHandle2)`

  Description:
  Compares the distributions of two global arrays.

  Note:
  This is a collective operation.

  Example:
  ```
  GA_Begin();
  A = GA_Create([4 4], [4 1]); B =
  GA_Create([4 4], [4 1]); isSimillar
  = GA_Compare_distr(A, B);
  GA_End(A, B);
  ```

  See Also
  GA_Create

- **GA_Nnodes**

  Syntax: `nNodes = GA_Nnodes()`

  Description:
  Returns the number of user (compute) processes.

  Note:
  This is a local operation.

  Example:
  ```
  GA_Begin();
  np = GA_Nnodes();
  GA_End();
  ```

  See Also
  GA_Nodeid

- **GA_Nodeid**

  Syntax:
  `rank = GA_Nodeid()`

  Description:
  Returns the rank or the process id (0 ... (np-1)) of the calling process.

  Note:
  This is a local operation.

  Example:
  ```
  GA_Begin();
  rank = GA_Nodeid();
  GA_End();
  ```

  See Also
  GA_Nnodes

- **GA_Check_handle**

  Syntax: `isValid = GA_Check_handle(ArrayHandle)`

  Description:
  Checks if the ArrayHandle is valid.

  Note:
  This is a collective operation.

  Example:
  ```
  GA_Begin();
  A = GA_Create([4 4]); isValid =
  ```

```
GA_Check_handle(A);
GA_End(A);
```

See Also
GA_Nnodes

A.8 Process Groups

- **GA_Create_pgroup**

  Syntax:
  `pGrpHandle = GA_Create_pgroup(list)`

  Description:
  Creates a processor group given the list of process ids and returns a process group handle.

  Note:
  This is a collective operation.

  Example:
  ```
  GA_Begin();
  pGrpHandle = GA_Create_pgroup([0
  1]); GA_End();
  ```

  See Also
  GA_Pgroup_nnodes

- **GA_Pgroup_nnodes**

  Syntax:
  `nNodes = GA_Pgroup_nnodes(pGrpHandle)`

  Description:
  Returns the number of processors contained in the group specified by $pGrpHandle$.

  Note:
  This is a local operation.

  Example:
  ```
  GA_Begin();
  pGrpHandle = GA_Create_pgroup([0
  1]); nNodes = GA_Pgroup_nnodes(pGrpHandle);
  GA_End();
  ```

  See Also
  GA_Pgroup_nodeid

- **GA_Pgroup_nodeid**

  Syntax:
  `rank = GA_Pgroup_nodeid(pGrpHandle)`

  Description:
  Returns the relative index of the calling process in the processor group specified by $pGrpHandle$.

  Note:
  This is a local operation.

  Example:
  ```
  GA_Begin();
  pGrpHandle = GA_Create_pgroup([0
  1]); rank = GA_Pgroup_nodeid(pGrpHandle);
  GA_End();
  ```

  See Also
  GA_Pgroup_nnodes

- **GA_Pgroup_sync**

  Syntax:
  `GA_Pgroup_sync(pGrpHandle)`

  Description:
  Synchronizes processes (a barrier) in the process group specified by $pGrpHandle$ and ensures that all GA operations for the process group are complete.

  Note:
  This is a collective operation.

  Example:
  ```
  GA_Begin();
  GA_Pgroup_sync();
  GA_End();
  ```

  See Also
  GA_Pgroup_nnodes