Combining Distributed and Shared Memory Models: Approach and Evolution of the Global Arrays Toolkit

J. Nieplocha, R.J. Harrison, M.K. Kumar, B. Palmer, V. Tipparaju, H. Trease
Pacific Northwest National Laboratory

Introduction

Both shared memory and distributed memory models have advantages and shortcomings. Shared memory model is much easier to use but it ignores data locality/placement. Given the hierarchical nature of the memory subsystems in the modern computers this characteristic might have a negative impact on performance and scalability. Various techniques, such as code restructuring to increase data reuse and introducing blocking in data accesses, can address the problem and yield performance competitive with message passing [Singh], however at the cost of compromising the ease of use feature. Distributed memory models such as message passing or one-sided communication offer performance and scalability but they compromise the ease-of-use. In this context, the message-passing model is sometimes referred to as "assembly programming for the scientific computing".

The Global Arrays toolkit [GA1, GA2] attempts to offer the best features of both models. It implements a shared-memory programming model in which data locality is managed explicitly by the programmer. This management is achieved by explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to the distributed shared-memory models that provide an explicit acquire/release protocol. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be explicitly specified and hence managed. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference.

This paper describes the characteristics of the Global Arrays programming model, capabilities of the toolkit, and discusses its evolution.

The Global Arrays Approach

Virtually all the scalable architectures possess non-uniform memory access characteristics that reflect their multi-level memory hierarchies. These hierarchies typically comprise processor registers, multiple levels of cache, local memory, and remote memory. In future systems, both the number of levels and the cost (in processor cycles) of accessing deeper levels can be expected to increase. It is important for programming models to address memory hierarchy since it is critical to the efficient execution of scalable applications. The two dominant programming models for MIMD concurrent computing are message passing and shared memory.

A message-passing operation not only transfers data but also synchronizes sender and receiver. Asynchronous (nonblocking) send/receive operations can be used to diffuse the synchronization point, but cooperation between sender and receiver is still required. The synchronization effect is beneficial in certain classes of algorithms such as parallel linear algebra where data transfer usually indicates completion of some computational phase; in these algorithms, the synchronizing messages can often carry both the results and a required dependency. For other algorithms, this synchronization can be unnecessary and undesirable, and a source of performance degradation and programming complexity. Despite programming difficulties, the message-passing paradigm's memory model maps well to the distributed-memory architectures used in scalable MPP systems. Because the programmer must explicitly control data distribution and is required to address data-locality issues, message-passing applications tend to execute efficiently on such systems. However, on systems with multiple levels of remote memory, for example networks of SMP workstations or computational grids, the message-passing model's classification of main memory as local or remote can be inadequate. A hybrid model that extends MPI with OpenMP attempts to address this problem is very hard to use and often offers little advantage over the MPI only approach.

In the shared-memory programming model, data is located either in "private" memory (accessible only by a specific process) or in "global" memory (accessible to all processes). In shared-memory systems, global memory

is accessed in the same manner as local memory. Regardless of the implementation, the shared-memory paradigm eliminates the synchronization that is required when message passing is used to access shared data. A disadvantage of many shared-memory models is that they do not expose the NUMA memory hierarchy of the underlying distributed-memory hardware. Instead, they present a flat view of memory making it hard for programmers to understand how data access patterns affect the application performance or how to exploit data locality. Hence, while programming effort involved in application development tends to be much lower than in the message-passing approach, achieved performance is usually less competitive. The shared memory model based on Global Arrays combines advantages of distributed memory model with the ease of use of shared memory. It is able to exploit SMP locality and deliver peak performance with the SMP by placing user's data in shared memory that allows accessing it directly rather than through a message-passing protocol.

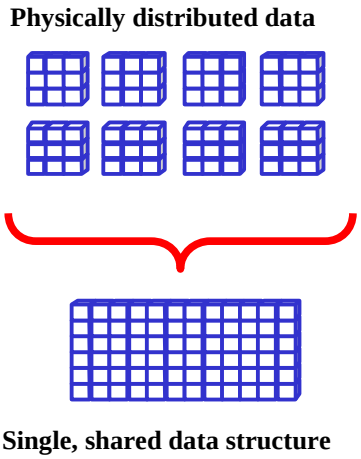

Figure 1: Dual view of GA data structures

The Global Arrays toolkit attempts to offer the best features of the shared and distributed memory models. It implements a shared-memory programming model in which data locality is managed explicitly by the programmer. This management is achieved by explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to distributed shared-memory models that provide an explicit acquire/release protocol. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be explicitly specified and hence managed. Another advantage is that GA, by optimizing and moving only the data requested by the user, avoids issues such as false sharing or redundant data transfers present in some DSM solutions. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference.

The GA provides extensive support for controlling array distribution and accessing locality information. Global arrays can be created by 1)allowing the library to determine array distribution, 2) specifying decomposition only for one array dimension and allowing the library to determine the others, 3)specifying the distribution block size for all dimensions, or 4) specifying irregular distribution as a Cartesian product of irregular distributions for each axis. The distribution and locality information is available through library operations that 1) specify the array section held by a given process, 2) specify which process owns a particular array element, and 3) return list of

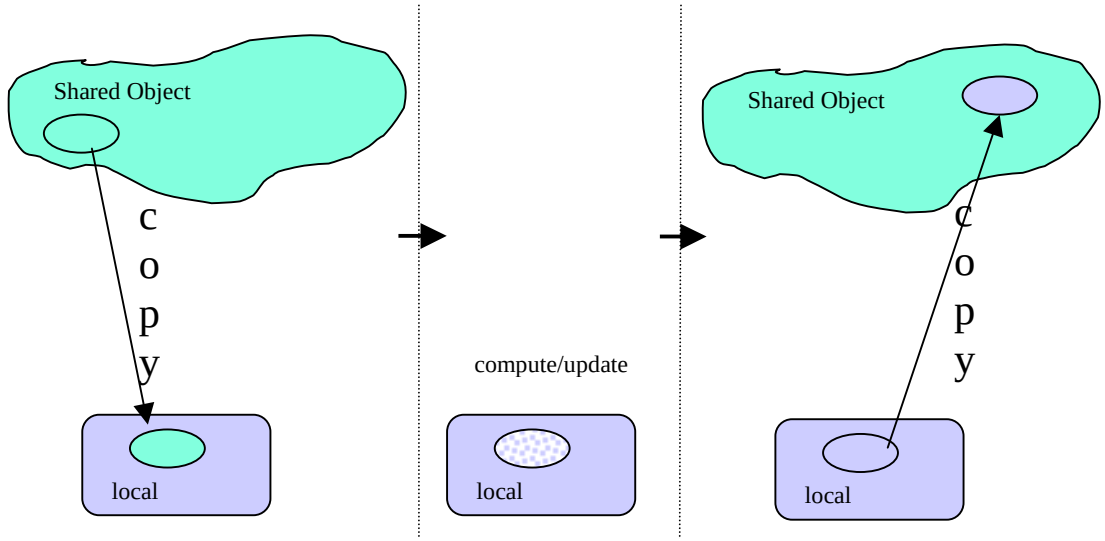

Figure 2: Typical model of computations in GA applications

processes and the blocks of data owned by each process corresponding to a given section of an array.

The primary mechanisms provided by GA for accessing data are copy operations that transfer data between layers of memory hierarchy, namely global memory (distributed array) and local memory. In addition, each process is able to access directly data held in a section of a global array that is logically assigned to that process. Atomic operations are provided that can be used to implement synchronization and assure correctness of an accumulate operation (floating-point sum reduction that combines local and remote data) executed concurrently by multiple processes and targeting overlapping array sections.

GA extends its capabilities in the area of linear algebra by offering interfaces to third party libraries e.g., standard and generalized real symmetric eigensolvers (PeIGS), and linear equation solvers (ScaLAPACK). The library can be used in C, C++, Fortran 77, Fortran 90 and Python programs.

Evolution of the Package and New Capabilities

The original GA package [GA1, GA2] offered basic one-sided communication operations along with a limited set of collective operations on arrays in the style of BLAS. Only two-dimensional arrays and two data types were supported. The underlying communication mechanisms were implemented on top of the vendor specific interfaces. In the course of eight years, the package evolved substantially and the underlying code was completely rewritten. Separation of the GA internal one-sided communication engine from the data structure specific high-level operation was necessary. A new portable, general, and independent of GA communication library called ARMCI was created. The new capabilities that were later added to GA simply relied on the existing ARMCI interfaces. The toolkit evolved in multiple directions:

- Eliminating some of the restrictions in the original package such as limited set of data types and generalizing the arrays to support arbitrary dimensions.
- Adding specialized capabilities that address needs of some the new application areas, e.g., ghost cells or operations for sparse data structures.
- Expansion and generalization of the existing interfaces. For example, mutex and lock operations were added to better support development of shared memory style application codes.
- Increased language interoperability and interfaces. For example, in addition to the original Fortran interfaces, C, Python, and C++ class library were developed. These efforts were further continued by developing a Common Component Architecture (CCA) component version of GA.
- Developing additional interfaces to the third party libraries, especially in the parallel linear algebra area. Examples are ScaLAPACK [scalapack] and SUMMA [RVG]. Morerecently, interfaces to Tao [TAO] are being developed.

ARMCI

Development of the ARMCI (Aggregate Remote Memory Copy Interface) library represents the most substantial task associated with the GA project. ARMCI was developed to be a general, portable, and efficient one-sided communication interface that is able to achieve high performance without modification of the semantics or API on each vendor platform [ARMCI]. Another design requirement was for noncontiguous data transfers to be optimized to deliver performance levels as close to the contiguous data transfers as possible. This requirement has been possible to meet, thanks to the non-contiguous data interfaces available in the ARMCI data transfer operations: multi-strided and generalized UNIX I/O vector interfaces [ARMCI2]. ARMCI supports up to eight stride levels corresponding to eight-dimensional arrays. The library provides three classes of operations: 1) data transfer operations including put, get, and accumulate (operations also available in MPI-2 but not in any vendor specific remote memory interface); 2) synchronization operations— atomic read-modify-write, locks/mutex operations, and 3) operations for memory management, local and global fence, and error handling. ARMCI only targets remote memory allocated via the provided memory allocator routine, ARMCI_Malloc (similar to MPI_Win_malloc in MPI-2). On shared memory, systems including SMPs, this approach makes it possible allocate shared memory for the user data and consecutively map remote memory operations to direct memory references, thus achieving sub-microsecond latency and a full memory bandwidth [ARMCI3].
ARMCI offers full portability to the Global Arrays package w.r.t. the communication interfaces. It also provides powerful interfaces for developing new capabilities and other data structures. ARMCI is currently a component of the run-time system in the Center for Programming Models for Scalable Parallel Computing project [pmodels].

N-dimensional arrays

The original version of the GA package has offered explicit support for two-dimensional arrays only. This characteristic was related to the original application domain of GA – numerical linear algebra and matrix related problems. It also influenced the API of most operations in the package. In the process of generalizing the capabilities of GA, this constraint had to be eliminated. It happened when the package was rewritten to use ARMCI. In the new version, only n-dimensional array capabilities are supported. The limit for the maximum number of dimensions is a compile time option with the default set to seven, the maximum number of dimensions supported by Fortran. For backwards compatibility, the original 2-dimensional interfaces are supported as wrappers to the new n-dimensional operations.

Ghost Cells

Many applications simulating physical phenomena defined on regular grids benefit from explicit support for ghost cells. These capabilities have been added recently to Global Arrays, along with the corresponding update and shift operations that operate on ghost cell regions. The update operation fills in the ghost cells with the visible data residing on neighboring processors. Once the update operation is complete, the local data on each processor contains the locally held "visible" data plus data from the neighboring elements of the global array, which has been used to fill in the ghost cells. Thus, the local data on each processor looks like a chunk of the global array that is slightly bigger than the chunk of locally held visible data, see Figure 3. The update operation to fill in the ghosts cells can be treated as a collective operation, enabling a multitude of optimization techniques. It was found that depending on the platform, different communication algorithms (message-passing, one-sided communication, shared memory) work the bestThe implementation of this operation makes use of the optimal algorithm for each platform. GA also allows ghost cell widths to be set to arbitrary values in each dimension, thereby allowing programmers to improve performance by combining multiple fields into one global array and using multiple time steps between ghost cell updates. The GA update operation offers a multitude of embedded synchronization semantics: no synchronization whatsoever, synchronization at the beginning of the operation, at the end or both. They are selected by the user by calling an optional separate function that cancels any unnecessary synchronization points in the following update operation depending on the consistency of the data.

Sparse Data Storage

Unstructured meshes are typically stored in a compressed sparse matrix form where the arrays that represent the data structures are one-dimensional. Computations on such unstructured meshes often lead to irregular data access and communication patterns. They also map to a distributed, shared memory, parallel programming model. Developing high-level abstractions and data structures that are general and applicable to a range of problems and applications is a challenging task. Therefore, our plan was to identify a minimal set of lower level interfaces that facilitate operations on sparse data format first and then try to define higher level data structures and APIs after gaining some experience in using these interfaces.

A set of functions was designed to operate on distributed, compressed, sparse matrix data structures built on top of one-dimensional global arrays. These functions have been patterned after similar functions in CMSSL on the Thinking Machines CM-2 and CM-5 massively parallel computers in the late 80's and early 90's. The types of

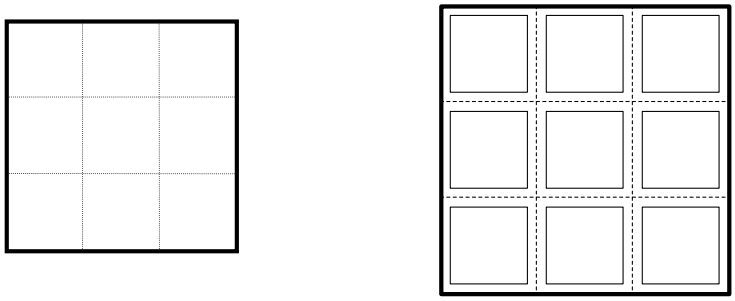

Figure 3: Schematic illustration of extension of ordinary global array (left) to global array with ghost cells (right). Heavy solid lines are global array boundaries, light solid lines are boundaries of visible data on each processor, and dotted lines are boundaries of ghost cell data.

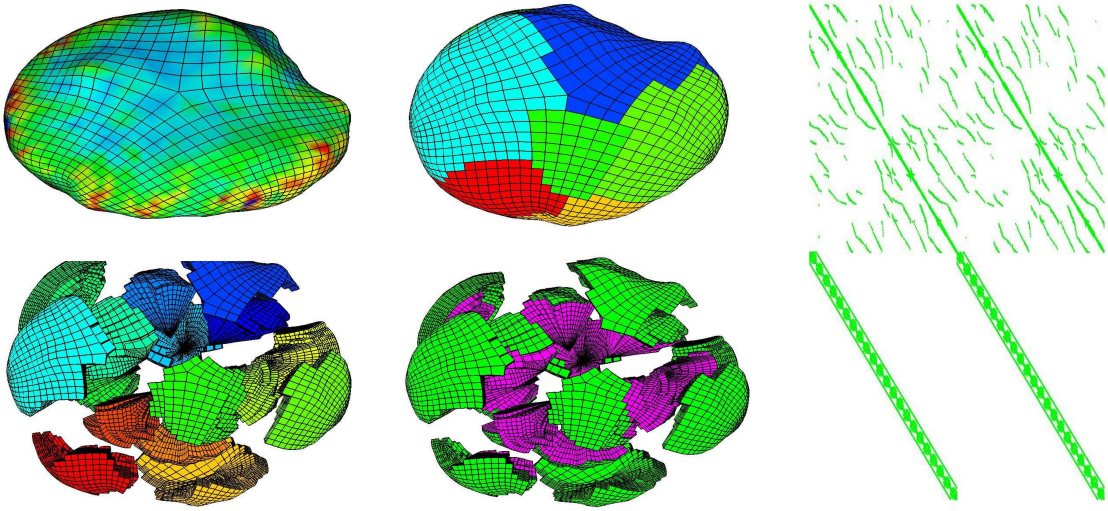

Figure 4: Example of domain decomposition of an unstructured mesh of a biological cell for 16 parallel processors. This series of images show the mesh, partitioned mesh, exploded mesh, along with the before (right/top) and after (right/bottom) partitioning of the sparse matrix representation of the unstructured mesh.

functions that have been designed, implemented and tested includes: 1) enumerate; 2) pack/unpack; 3) scatter_with_OP, where OP can be plus, max, min; 4) segmented_scan_with_OP, where OP can be plus, max, min, copy; 5) binning (i.e., N-to-M mapping); and 6) a 2-key binning/sorting function. All the functions operate on one-dimensional global arrays and can form a foundation for building unstructured mesh data structure. They were adopted in mesh generation (NWGrid [NWG]) and computational biophysics (NWPhys [NWP]) codes. In numerous applications, the performance has been demonstrated to scale linearly with the number of processors and problem size, as most unstructured mesh codes that do optimal data partitioning algorithms should. Figure 4 shows a graphic of the mesh, partitioning, and sparse matrix structure [HET].

Disk Resident Arrays

The disk resident arrays (DRA) model extends the GA model to another level in the storage hierarchy, namely, secondary storage [DRA]. It introduces the concept of a disk resident array—a disk-based representation of an array–and provides functions for transferring blocks of data between global arrays and disk arrays. Hence, it allows programmers to access data located on disk via a simple interface expressed in terms of arrays rather than files. The benefits of global arrays (in particular, the absence of complex index calculations and the use of optimized array communication) can be extended to programs that operate on arrays that are too large to fit into memory. By providing distinct interfaces for accessing objects located in main memory (local and remote) and on the disk, GA and DRA render visible the different levels of the memory hierarchy in which objects are stored. Hence, programs can take advantage of the performance characteristics associated with access to these levels. In modern computers, memory hierarchies consist of multiple levels, but are managed between two adjacent levels at a time. For example, a page fault causes the transfer of a data block (page) to main memory while a cache miss transfers a cache line. Similarly, GA and DRA allow data transfer only between adjacent levels of memory. In particular, data transfer between disk resident arrays and local memory is not supported.

Disk resident arrays have a number of uses. They can be used to checkpoint global arrays. Implementations of out-of-core computations can use disk arrays to implement user-controlled virtual memory, locating arrays that are too big to fit in aggregate main memory in disk arrays, and then transferring sections of these disk arrays into main memory for use in the computation. DRA functions are used to stage the disk array into a global array; individual processors then use GA functions to transfer global array components into local storage for computation. If the global array is updated, a DRA write operation may be used to write the global array back to the appropriate component of the disk array. DRA has been designed to support collective transfers of large data blocks. No attempts are made to optimize performance for small (<0.5MB) requests.

The DRA library evolved similarly to the GA. The original version shared the same limitations as GA regarding the number of dimensions and data types supported. The current version supports n-dimensional arrays, a range

of C, and Fortran data types. Performance and scalability have not been sacrificed when developing these capabilities, see Figure 5 (coming from Bruce). The concept of chunked data layout has been adopted for n-dimensional arrays to increase locality of reference and concurrency of I/O [DRPA].

Multi-language Interfaces and Interoperability

The original GA package, although developed in C, supported primarily Fortran applications. However, the package evolved to become more language independent. The core library is still written in C, however, user interfaces are now available in multiple languages(see Figure 6). The primary languages are Fortran and C. In addition, a C++ class library and Python interfaces have been developed. GA attempts to provide as much compatibility between different language interfaces as possible. For example, arrays crated in Fortran with Fortran data types are accessible through the C interface. The same applies the other way around if a Fortran data type corresponding to C exists. Moreover, for even greater compatibility, the C interface supports either the Fortran or C view of multidimensional arrays in terms of data layout (column- or row-major based) and indexing.

Common Component Architecture (CCA) GA component

High performance scientific applications are assembled from large blocks of hand crafted code into monolithic applications, which also includes many generic support routines. A major disadvantage of this traditional approach is that software boundaries (function interfaces and global symbols) are frequently not well defined throughout the code [CCA]. The component approach attempts to address this problem. Components encapsulate well-defined units of reusable functionality and they interact through standard interfaces. Components are protected from changes in the software environment outside their boundaries. The Common Component Architecture (CCA) is a component model specifically designed for high performance computing. Components are peers, i.e. they are viewed as equal participants rather than as elements in an inheritance hierarchy. The CCA consists of three types of entities: components, ports and frameworks. Components are the basic units of software that are composed together at run-time to form applications. Ports are the fully abstract well-defined interfaces on components, which are managed by the framework in the composition process. Frameworks provide the means to hold the components and compose them into applications [CCA]. CCAFFEINE, a CCA-compliant framework composes single program multiple data (SPMD) applications from components. A peer component communicates via ports with other components in the same address space and communicates via a process-to-process protocol (e.g. MPI) within its SCMD (Single Component Multiple Data) set of corresponding components on all P processors [Allan]. We developed an object-oriented global arrays (GA) peer component for high performance computing using the CCA standard.

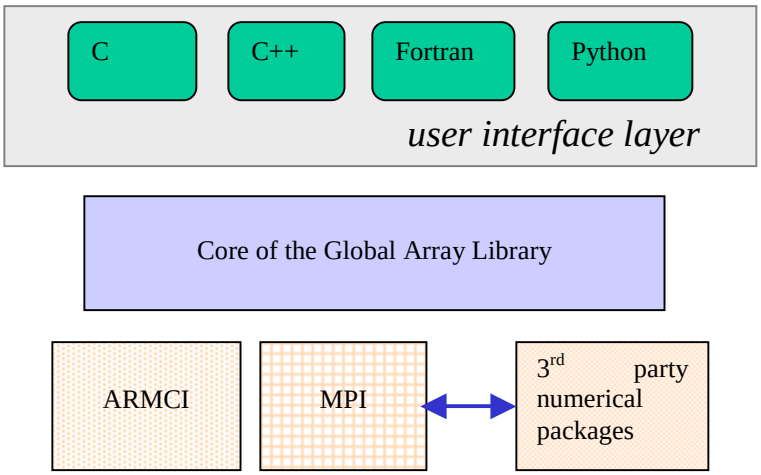

Figure 6: Structure of the GA package

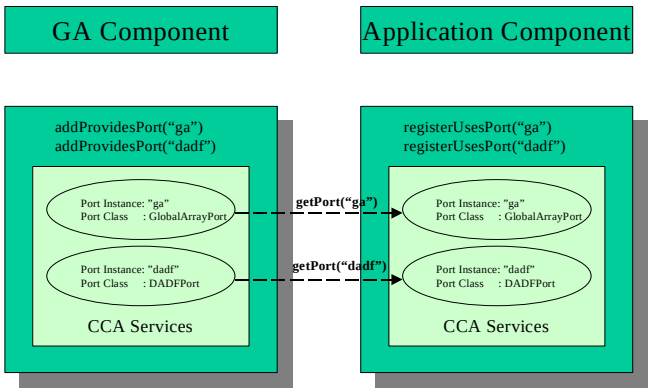

Figure 7: Peer-to-peer connection mechanism in GA component

GA Component provides a shared-memory programming interface for distributed-memory computers to access dense multi-dimensional arrays. It provides two ports: *GlobalArrayPort* and *DADFPort*. These ports are the set of public interfaces that the GA component implements, which can be referenced and used by other components. *GlobalArrayPort* provides public interfaces for creating and accessing distributed arrays. These interfaces are intended to support the collection of the global information and creation of *GlobalArray* objects. All details of the data distribution, addressing, and data access are encapsulated in the *GlobalArray* objects. The *GlobalArray* object offers a set of operations for one-sided data transfer operations (get, put, scatter, gather, etc), collective array operations, and supportive operations for data locality control and queries.

DADFPort provides public interfaces for defining and querying array distribution templates and distributed array descriptors. These array templates and descriptors, when combined, provide a uniform means to describe the parallel data distribution of dense multi-dimensional rectangular arrays. An array template is a virtual multidimensional array to which one or more actual distributed arrays may be aligned and an array descriptor is the association of real data (pointers, strides, etc.) to the distribution defined by a particular template. *DADFPort* interfaces are intended to support the creation, cloning, and destruction of *DistArrayTemplate* and *DistArrayDescriptor* objects, defined by the data working group of the CCA forum [DDADF]. *DistArrayTemplate* objects are used to describe the distribution template for a distributed array. These objects can also be used to create an actual data object. The *DistArrayDescriptor* object is constructed from size and type information, pointers to the local data on each process, and a mapping onto a distribution template. It describes any distributed array sufficiently to allow the construction of parallel communications schedules and other data movement-related operations. It is used together with *DistArrayTemplate* object to provide a complete description (i.e. an array descriptor) of the data distribution of a distributed array. These objects are primarily intended to low-level use within new components.

Figure 7 illustrates the mechanism for making a peer-to-peer connection, which connects a GA component with an application component using CCAFFEINE [Armstrong]. The GA component adds the "provides" ports, which is visible to other components to the CCA *Services* object. The application component registers the ports that it will need with the CCA *Services* object. The CCAFEINE framework connects two components and transfers the *GlobalArrayPort* and *DADFPort* to the application component (or any component) using GA Component's *Services* object. The application component uses its Services object to retrieve the ports provided by GA Component. The connected components are set in motion by the go() function in *GoPort* , which is provided by the application component.

It is apparent that the latency overhead for using components is equivalent to one virtual function call when using C++ [Allan]. We are also proceeding with plans to provide ports that employ interfaces developed by the Equation Solver Interface (ESI) Forum group.

Performance

GA's communication interfaces form a thin wrapper around ARMCI interfaces. Performance of GA is hence proportionate to and in-line with ARMCI performance. Table-1 gives the GA and ARMCI bandwidth and latency on CRAY-T3E and Linux cluster for intra-SMP node (local on Cray) and remote get operations. The very narrow difference in both bandwidth and latency numbers for GA and ARMCI can be seen in Table-1. On Linux, performance of remote operations has been shown with both GM and VIA as underlying networks. The Linux systems used for measurements are SGI 1100 servers with dual 1GHz Intel Pentium III processors. GM readings were obtained using Myrinet-2000 cards and a version 1.5pre4 of the GM software. VIA readings were obtained on a CLAN network running version 1.3 of the CLAN driver.
Performance of inter-node operations in ARMCI follows very closely performance of the memory copy operation on that system. Similarly, ARMCI achieves bandwidth close to the underlying network protocols. The same applies to latency if the native platform protocol supports the equivalent remote memory operation (e.g., shmem_get on the Cray T3E). In cases platforms which do not support remote get (GM,VIA) the latency includes cost of interrupt processing that is used in ARMCI to implement get operation.

Platform	GA Latency (us)		ARMCI Latency(us)		GA Bandwidth(MBps)		ARMCI Bandwidth(MBps)	
	SMP/Local	Remote	SMP/Local	Remote	SMP/Local	Remote	SMP/Local	Remote
Cray-T3E	6.4	8.05	0.81	3.01	215	329	225	330
Linux-GM	1.56	37.30	0.302	37.2	409	168	412	168
Linux-VIA		38.30		38.10		104		104

Table-1: Perfomance results of GA and ARMCI on CRAY-T3E and Linux (with GM and VIA)

Conclusions

Since it was introduced, the GA toolkit has proved itself an effective tool for developing scalable parallel applications. The applications of GA span quantum chemistry, molecular dynamics, image processing, electron microscopy data processing, financial security forecasting, computational fluid dynamics, computational biology, and other areas. The library has become the de facto standard parallel programming tool within the electronic structure computational chemistry community. Of the ten most widely used packages that execute in parallel (NWChem, Gamess-UK, Gamess-US, Molpro, Molcas, COLUMBUS, Qchem, Gaussian, Cadpac, ADF) five of them have adopted GA (NWChem, Gamess-UK, Molpro, Molcas, COLUMBUS), one is in a process of adopting it (QChem, a commercial s/w package) and one developed a library emulating a subset of the Global Arrays capabilities (Gamess-US). The remaining ones that do not use GA (Gaussian, Cadpac, ADF) also do not emphasize scalability.

Over the years, the toolkit has evolved substantially by expanding its capabilities to serve new application areas and it has been ported to many new architectures. However, despite the many changes in h/w and the toolkit itself, the basic concepts behind GA have been proven successful. The current GA toolkit is 100% fully backward compatible with the interfaces in the original package. The package will be advanced in the future. Some of the recent capabilities described in this paper are only first steps. For example, sparse data structures and GA component efforts will surely bring new concepts and capabilities into the package.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) operated for DOE by Battelle Memorial Institute. This work was supported by the Center for Programming Models for Scalable Parallel Computing and DoE-2000 ACTS project, both sponsored by the Mathematical, Information, and Computational Science Division of DOE's Office of Computational and Technology Research. The Molecular Science Computing Facility at PNNL, National Energy Research Supercomputing Center, and University of Buffalo provided the high-performance computational resources for this work.

References

[Singh] J.P. Singh, A Comparison of Three Programming Models for Adaptive Applications on the Origin2000, Proc. SC2000.

[GA1] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A portable 'shared-memory' programming model for distributed memory computers. In *Proceedings of Supercomputing 1994*, pages 340–349, 1994.

[GA2] 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*, 10:197–220, 1996.

[ARMCI] J. Nieplocha, B. Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, *Proc. RTSPP of IPPS/SDP'99*, 1999.

[ARMCI2] J. Nieplocha, J. Ju, ARMCI: A Portable Aggregate Remote Memory Copy Interface, Version 1.1, October 30, 2000, http://www.emsl.pnl.gov:2080/docs/parsoft/armci/armci1-1.pdf.

[ARMCI3] J. Nieplocha, J. Ju, T.P. Straatsma, A multiprotocol communication support for the global address space programming model on the IBM SP, Proc. EuroPar-2000, Springer Verlag LNCS-1900, 2000.

[pmodels] Center for Programming Models for Scalable Parallel Computing www.pmodels.org

[DRA] J. Nieplocha and I. Foster. Disk Resident Arrays: An array-oriented library for out-of-core computations. In *Proc. Frontiers of Massively Parallel Computation*, pages 196–204, 1996.

[DRPA] Y. Chen, I. Foster, J. Nieplocha, M.Winslett, Optimizing Collective I/O Performance on Parallel Computers: A Multisystem Study, *Proc. 11th ACM Intl. Conf. on Supercomputing*, ACM Press, 1997.

[NWG] http://www.emsl.pnl.gov/nwgrid

[NWP] http://www.emsl.pnl.gov/nwphys

[HET] Trease, H.E., etal, Grid Generation Tools for Performing Feature Extraction, Image Reconstruction, and Mesh Generation on Digital Volume Image Data for Computational Biology Applications, 8th International Conference On Grid Generation and Scientific Applications, Honolulu, Hawaii, June 2-5, 2002.

[Allan] Allan, Benjamin A., Robert C. Armstrong, Alicia P. Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The CCA core specification in a distributed memory SPMD framework. http://www.cca-forum.org/old/ccafe03a/ccafe03a.pdf

[Armstrong] Armstrong, Rob, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. 1999. Toward a common component architecture for high-performance scientific computing. http://www-unix.mcs.anl.gov/%7Ecurfman/cca/web/cca_paper.html

[DADF] Bernholdt, David E. Comparison of distributed array descriptors (DAD) as proposed and implemented for SC01 demos. http://www.csm.ornl.gov/~bernhold/cca/data

[CCA] CCA Forum. Common Component Architecture Forum. http://www.cca-forum.org

[TAO] TAO: Toolkit for Advanced Optimization, http://www-fp.mcs.anl.gov/tao/

[RVG] R. van de Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, Dept of Computer Sciences, The University of Texas, 1995.

[scalapack] The ScaLAPACK Project http://www.netlib.org/scalapack/index.html