Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit

Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease
Computational Sciences and Mathematics Department
Pacific Northwest National Laboratory
Richland, WA 99352

Edoardo Aprà
William R. Wiley Environmental Molecular Sciences Laboratory
Pacific Northwest National Laboratory
Richland, WA 99352

Abstract

This paper describes capabilities, evolution, performance, and applications of the Global Arrays (GA) toolkit. GA was created to provide application programmers with an interface that allows them to distribute data while maintaining the type of global index space and programming syntax similar to what is available when programming on a single processor. The goal of GA is to free the programmer from the low level management of communication and allow them to deal with their problems at the level at which they were originally formulated. At the same time, compatibility of GA with MPI enables the programmer to take advantage of the existing MPI software/libraries when available and appropriate. The variety of applications that have been implemented using Global Arrays attests to the attractiveness of using higher level abstractions to write parallel code.

1. Introduction

The two predominant classes of programming models for MIMD concurrent computing are distributed memory and shared memory. 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 modern computers this characteristic can have a negative impact on performance and scalability. Careful code restructuring to increase data reuse and replacing fine grain load/stores with block access to shared data can address the problem and yield performance for shared memory that is competitive with message-passing [1]. However, this performance comes at the cost of compromising the ease of use that the shared memory model advertises. Distributed memory models, such as message-passing or one-sided communication, offer performance and scalability but they are difficult to program.

The Global Arrays toolkit [2-4] attempts to offer the best features of both models. It implements a shared-memory programming model in which data locality is managed by the programmer. This management is achieved by calls to functions that transfer data between a global address space (a distributed array) and local storage (Figure 1). In this respect, the GA model has similarities to the distributed shared-memory models that provide an explicit acquire/release protocol e.g., [5]. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be specified by the programmer and hence managed. GA is related to the global address space languages such as UPC [6], Titanium[7], and, to a lesser extent, Co-Array Fortran[1] [8]. In addition, by providing a set of data-parallel operations, GA is also related to data-parallel languages such as HPF [9], ZPL [10], and Data Parallel C [11]. However, the Global Array programming model is implemented as a library that works with most languages used for technical computing and does not rely on compiler technology for achieving parallel efficiency. It also supports a combination of task- and data-parallelism and is available as an extension of the message-passing (MPI) model. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems [12], and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference. 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. Over time, both the number of levels and the cost (in processor cycles) of accessing deeper levels has been increasing. It is important for any scalable programming model to address memory hierarchy since it is critical to the efficient execution of scalable applications.

Before the DoE-2000 ACTS program was established [13, 14], the original GA package [2-4] offered basic one-sided communication operations, along with a limited set of collective operations on arrays in the style of BLAS [15]. Only two-dimensional arrays and two data types were supported. The underlying communication mechanisms were implemented on top of vendor specific interfaces. In the course of ten years, the package has evolved substantially and the underlying code has been completely rewritten. This included separation of the GA

[1] CAF does not provide explicit mechanisms for combining distributed data into a single shared object. It supports one-sided access to so called “co-arrays”, arrays defined on every processor in the SPMD program.

internal one-sided communication engine from the high-level data structure. A new portable, general, and GA-independent communication library called ARMCI was created [16]. New capabilities were later added to GA without the need to modify the ARMCI interfaces. The GA toolkit evolved in multiple directions:

- Adding support for a wide range of data types and virtually arbitrary array ranks (note that the Fortran limit for array rank is seven).
- Adding advanced or specialized capabilities that address the needs of some new application areas, e.g., ghost cells or operations for sparse data structures.
- Expansion and generalization of the existing basic functionality. For example, mutex and lock operations were added to better support the development of shared memory style application codes. They have proven useful for applications that perform complex transformations of shared data in task parallel algorithms, such as compressed data storage in the multireference configuration interaction calculation in the COLUMBUS package [17].
- Increased language interoperability and interfaces. In addition to the original Fortran interface, C, Python, and a C++ class library were developed. These efforts were further extended by developing a Common Component Architecture (CCA) component version of GA.
- Developing additional interfaces to third party libraries that expand the capabilities of GA, especially in the parallel linear algebra area. Examples are ScaLAPACK [18] and SUMMA [19]. More recently, interfaces to the TAO optimization toolkit have also been developed [20].
- Developed support for multi-level parallelism based on processor groups in the context of a shared memory programming model, as implemented in GA[21, 22].

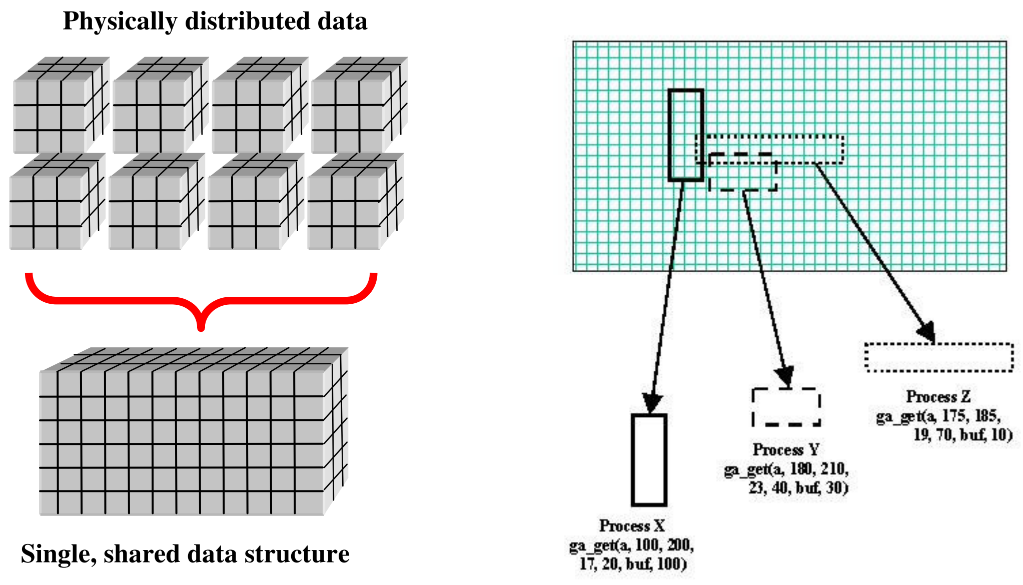

Physically distributed data
Single, shared data structure
Process Z
ga_get(a, 175, 185, 19, 70, buf, 10)
Process Y
ga_get(a, 180, 210, 23, 40, buf, 30)
Process X
ga_get(a, 100, 200, 17, 20, buf, 100)

Figure 1: Dual view of GA data structures (left). Any part of GA data can be accessed independently by any process at any time (right).

These advances generalized the capabilities of the GA toolkit and expanded its appeal to a broader set of applications. At the same time the programming model, with its emphasis on a shared memory view of the data structures in the context of distributed memory systems with a hierarchical memory, is as relevant today as it was in 1993 when the project started. This paper describes the characteristics of the Global Arrays programming model, capabilities of the toolkit, and discusses its evolution. In addition, performance and application experience are presented.

2. The Global Arrays Model

The classic message-passing paradigm of parallel programming not only transfers data but also synchronizes the 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. The MPI-2 [23] one-sided communication relaxes the synchronization requirements between sender and receiver while imposing new constraints on progress and remote data access rules that make the programming model more complicated than with other one-sided interfaces [24, 25]. Despite programming difficulties, the message-passing memory paradigm maps well to the distributed-memory architectures deployed 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 but is very hard to use and often offers little or no advantages over the MPI-only approach [26, 27].

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 [12]. 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 the programming effort involved in application development tends to be much lower than in the message-passing approach, the performance is usually less competitive.

The shared memory model based on Global Arrays combines the advantages of a distributed memory model with the ease of use of shared memory. It is able to exploit SMP locality and deliver peak performance within the SMP by placing user's data in shared memory and allowing direct access rather than through a message-passing protocol. This is achieved by function calls that provide information on which portion of the distributed data is held locally and the use of explicit calls to functions that transfer data between a shared address space and local storage. The combination of these functions allows users to make use of the fact that remote data is slower to access than local data and to optimize data reuse and minimize communication in their algorithms. Another advantage is that GA, by optimizing and moving only the data requested by the user, avoids issues such as false sharing, data coherence overheads, and redundant data transfers present in some software-based distributed shared memory (DSM) solutions [28-30]. These issues also affect performance of OpenMP programs compiled to use DSM [31].

GA allows the programmer to control data distribution and makes the locality information readily available to be exploited for performance optimization. For example, global arrays can be created by 1) allowing the library to determine the array distribution, 2) specifying the decomposition for only one array dimension and allowing the library to determine the others, 3) specifying the distribution block size for all dimensions, or 4) specifying an irregular distribution as a Cartesian product of irregular distributions for each axis. The distribution and locality information is always available through interfaces to query 1) which data portion is held by a given process, 2) which process owns a particular array element, and 3) a list of 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 block copy operations that transfer data between layers of memory hierarchy, namely global memory (distributed array) and local memory. Further extending the benefits of using blocked data accesses, copying remote locations into contiguous local memory can improve uniprocessor cache performance by reducing both conflict and capacity misses [32]. In addition, each process is able to access directly data held in a section of a Global Array that is locally assigned to that process and on

SMP clusters sections owned by other processes on the same node. 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 is extensible as well. New operations can be defined exploiting the low level interfaces dealing with distribution, locality and providing direct memory access (**`nga_distribution`**, **`nga_locate_region`**, **`nga_access`**, **`nga_release`**, **`nga_release_update`**) [33]. These, for example, were used to provide additional linear algebra capabilities by interfacing with third party libraries e.g., ScaLAPACK [18].

2.1 Memory consistency model

In shared memory programming, one of the issues central to performance and scalability is memory consistency. Although the sequential consistency model [34] is straightforward to use, weaker consistency models [35] can offer higher performance on modern architectures and they have been implemented on actual hardware. The GA approach is to use a weaker than sequential consistency model that is still relatively straightforward to understand by an application programmer. The main characteristics of the GA approach include:

- GA distinguishes two types of completion of the store operations (i.e., put, scatter) targeting global shared memory: local and remote. The blocking store operation returns after the operation is completed locally, i.e., the user buffer containing the source of the data can be reused. The operation completes remotely after either a memory fence operation or a barrier synchronization is called. The fence operation is required in critical sections of the user code, if the globally visible data is modified.
- The blocking operations (load/stores) are ordered only if they target overlapping sections of global arrays. Operations that do not overlap or access different arrays can complete in arbitrary order.
- The nonblocking load/store operations complete in arbitrary order. The programmer uses wait/test operations to order completion of these operations, if desired.

3. The Global Array Toolkit

There are three classes of operations in the Global Array toolkit: core operations, task parallel operations, and data parallel operations. These operations have multiple language bindings, but provide the same functionality independent of the language. The GA package has grown considerably in the course of ten years. The current library contains approximately 200 operations that provide a rich set of functionality related to data management and computations involving distributed arrays.

3.1 Functionality

The basic components of the Global Arrays toolkit are function calls to create global arrays, copy data to, from, and between global arrays, and identify and access the portions of the global array data that are held locally. There are also functions to destroy arrays and free up the memory originally allocated to them. The basic function call for creating new global arrays is **`nga_create`**. The arguments to this function include the dimension of the array, the number of indices along each of the coordinate axes, and the type of data (integer, float, double, etc.) that each array element represents. The function returns an integer handle that can be used to reference the array in all subsequent calculations. The allocation of data can be left completely to the toolkit, but if it is desirable to control the distribution of data for load balancing or other reasons, additional versions of the **`nga_create`** function are available that allow the user to specify in detail how data is distributed between processors. Even the basic **`nga_create`** call contains an array that can be used to specify the minimum dimensions of a block of data on each processor.

One of the most important features of the Global Arrays toolkit is the ability to easily move blocks of data between global arrays and local buffers. The data in the global array can be referred to using a global indexing scheme and data can be moved in a single function call, even if it represents data distributed over several processors. The **`nga_get`** function can be used to move a block of distributed data from a global array to a local buffer and has a relatively simple argument list. The arguments consist of the array handle for the array that data is being taken from, two integer arrays representing the lower and upper indices that bound the block of distributed data that is going to be moved, a pointer to the local buffer or a location in the local buffer that is to receive the data, and an array of strides for the local data. The **`nga_put`** call is similar and can be used to move data in the opposite direction. For a distributed data paradigm with message-passing, this kind of operation is much more complicated. The block of distributed data that is being accessed must be decomposed into separate blocks, each residing on different processors, and separate message-passing events must be set up between the processor containing the buffer and the processors containing the distributed data. A conventional message-passing interface will also require concerted actions on each pair of processors that are communicating, which contributes substantially to program complexity.

The one-sided communications used by Global Arrays eliminate the need for the programmer to account for responses by remote processors. Only the processor issuing the data request is involved, which considerably reduces algorithmic complexity compared to the programming effort required to move data around in a two-sided communication model. This is especially true for applications with dynamic or irregular communication patterns. Even for other programming models that support onesided communications, such as MPI-2, the higher level abstractions supported by GA can reduce programming complexity. To copy data from a local buffer to a distributed array requires only a single call to **`nga_put`**. Based on the data distribution, the GA library handles the decomposition of the put into separate point-to-point data transfers to each of the different processors to which the data must be copied and implements each transfer. The corresponding **`MPI_Put`**, on the other hand, only supports point-to-point transfers, so all the decomposition and implementation of the separate transfers must be managed by the programmer.

To allow the user to exploit data locality, the toolkit provides functions identifying the data from the global array that is held locally on a given processor. Two functions are used to identify local data. The first is the **`nga_distribution`** function, which takes a processor ID and an array handle as its arguments and returns a set of lower and upper indices in the global address space representing the local data block. The second is the **`nga_access`** function, which returns an array index and an array of strides to the locally held data. In Fortran, this can be converted to an array by passing it through a subroutine call. The C interface provides a function call that directly returns a pointer to the local data.

In addition to the communication operations that support task-parallelism, the GA toolkit includes a set of interfaces that operate on either entire arrays or sections of arrays in the data parallel style. These are collective data-parallel operations that are called by all processes in the parallel job. For example, movement of data between different arrays can be accomplished using a single function call. The **`nga_copy_patch`** function can be used to move a patch, identified by a set of lower and upper indices in the global index space, from one global array to a patch located within another global array. The only constraints on the two patches are that they contain equal numbers of elements. In particular, the array distributions do not have to be identical and the implementation can perform as needed the necessary data reorganization (so called "MxN" problem [36]). In addition, this interface supports an optional transpose operation for the transferred data. If the copy is from one patch to another on the same global array, there is an additional constraint that the patches do not overlap.

3.2 Example

A simple code fragment illustrating how these routines can be used is shown in Figure 2. A 1-dimensional array is created and initialized and then inverted so that the entries are running in the opposite order. The locally held piece of the arrays is copied to a local buffer, the local data is inverted, and then it is copied back to the inverted location in the global array. The chunk array specifies minimum values for the size of each locally held block and in this example guarantees that each local block is a 100 integer array. Note that global indices are used throughout and that it is unnecessary to do any transformations to find the local indices of the data on other processors.

```
integer ndim, nelem
parameter (ndim=1, nelem=100)
integer dims, chunk, nprocs, me, g_a, g_b
integer a(nelem), b(nelem)
integer i, lo, hi, lo2, hi2, ld

me = ga_nodeid()       ! rank of the process
nprocs = ga_nnodes() ! total # of processes

dims = nprocs*nelem
chunk(1) = nelem
ld = nelem

call nga_create(MT_INT, ndim, dims,
                  'array A', chunk, g_a)
call nga_duplicate(g_a, g_b, 'array B')
!  INITIALIZE DATA IN GA (NOT SHOWN)
call nga_distribution(g_a, me, lo, hi)

call nga_get(g_a, lo, hi, a, ld)
! INVERT LOCAL DATA
do i = 1, nelem
   b(i) = a(nelem+1-i)
end do
! INVERT DATA GLOBALLY
lo2 = dims + 1 – hi
hi2 = dims + 1 – lo
call nga_put(g_b, lo2, hi2, b, ld)
```

```
#define   NDIM      1
#define   NELEM   100
int dims, chunk, nprocs, me, g_a;
int a[NELEM],b[NELEM];
int i, lo, hi, lo2, hi2, ld;
GA::GlobalArray *g_a, *g_b;

me       = GA::SERVICES.nodeid();
nprocs = GA::SERVICES.nodes();

dims   = nprocs*NELEM;
chunk = ld = NELEM;

// create a global array
g_a = GA::SERVICES.createGA(C_INT, NDIM,
                              dims, "array A", chunk);
g_b = GA::SERVICES.createGA(g_a, "array B");
// INITIALIZE DATA IN GA (NOT SHOWN)
g_a->distribution(me, lo, hi);
g_a->get(lo, hi, a, ld);
// INVERT DATA LOCALLY
for (i=0; i<nelem; i++)    b[i] = a[nelem-1 – i];
// INVERT DATA GLOBALLY
lo2 = dims – 1 – hi;
hi2 = dims – 1 – lo;
g_b->put(lo2,hi2,b,ld);
```

Figure 2: Example Fortran (left) and C++ (right) code for transposing elements of an array

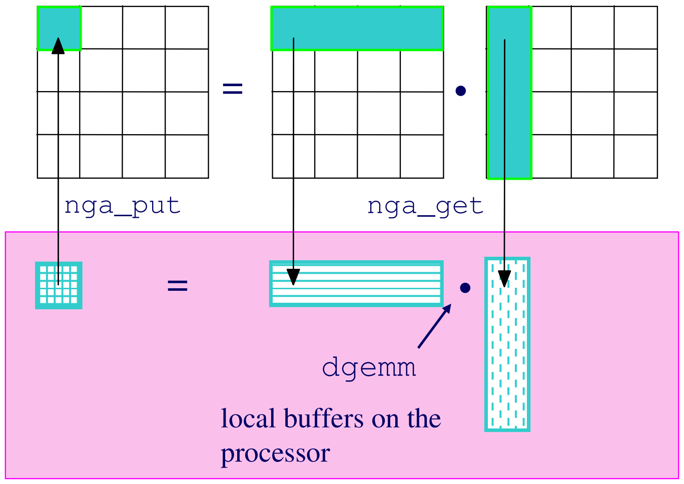


Figure 3: Schematic representation of distributed matrix multiply, C = A·B.

A more complicated example is a distributed matrix multiply of two global arrays, which is illustrated schematically in Figure 3. (This is not an optimal algorithm and is used primarily to illustrate how the toolkit can be used.) The matrix multiply requires three global arrays, A, B, and the product array, C.

The **nga_distribution** function is used to identify the indices of the locally held block from the product array C; this then determines what portions of the arrays A and B need to be moved to each processor to perform the calculation. The two data strips required to produce the target block are then obtained using a pair of **nga_get** calls. These calls will, in general, get data from multiple processors. Once the data from the A and B arrays has been copied to local buffers, the multiplication can be performed locally using an optimized scalar matrix multiplication algorithm, such as the LAPACK **dgemm** subroutine. The product patch is then copied back to the C array using an **nga_put** call.

The Global Array toolkit also contains a broad spectrum of elementary functions that are useful in initializing data or performing calculations. These include operations that zero all the data in a global array, uniformly scale the data by some value, and support a variety of (BLAS-like) basic linear algebra operations including addition of arrays, matrix multiplication, and dot products. The global array matrix multiplication has been reimplemented over the years, without changing the user interface. These different implementations ranged from a wrapper to the SUMMA [19] algorithm to the more recently introduced high-performance SRUMMA algorithm [37, 38] that relies on a collection of protocols (shared memory, remote memory access, nonblocking communication) and techniques to optimize performance depending on the machine architecture and matrix distribution.

3.3 Interoperability with other packages

The GA model is compatible with and extends the distributed memory model of MPI. The GA library relies on the execution/run-time environment provided by MPI. In particular, the job startup and interaction with the resource manager is left to MPI. Thanks to the compatibility and interoperability with MPI, GA 1) can exploit the existing rich set of software based on MPI to implement some of the capabilities such as linear algebra and 2) provides the programmer with the ability to mix and match different communication styles and capabilities provided by GA and MPI. For example, the molecular dynamics module of NWChem uses an MPI-based FFT library combined with a GA-based implementation of different molecular dynamics algorithms and data management strategies.

The GA package by itself relies on the linear algebra functionality provided by ScaLAPACK. For example, the **ga_lu_solve** is interfaced with the appropriate ScaLAPACK factorization **pdgetrf** and the forward/backward substitution interface **pdgetrs**. Because the global arrays already contain all the information about array dimensionality and layout, the GA interface is much simpler (see Figure 4).

```
call ga_lu_solve(g_a, g_b)
                instead of
call pdgetrf(n,m, locA, p, q, dA, ind, info)
call pdgetrs(trans, n, mb, locA, p, q, dA, dB, info)
```

Figure 4: `ga_lu_solve` operation is much easier to use than the ScaLAPACK interfaces that are invoked beneath this GA operation.

Other important capabilities integrated with GA and available to the programmer include numerical optimization. These capabilities are supported thanks to the Toolkit for Advanced Optimization (TAO) [20, 39]. TAO offers these capabilities while relying on the GA distributed data management infrastructure and linear algebra operations.

Examples of other toolkits and packages that are interoperable and have been used with GA include the Petsc PDE solver toolkit [40], CUMULVS toolkit for visualization and computational steering [41], and PEIGS parallel eigensolver library [42].

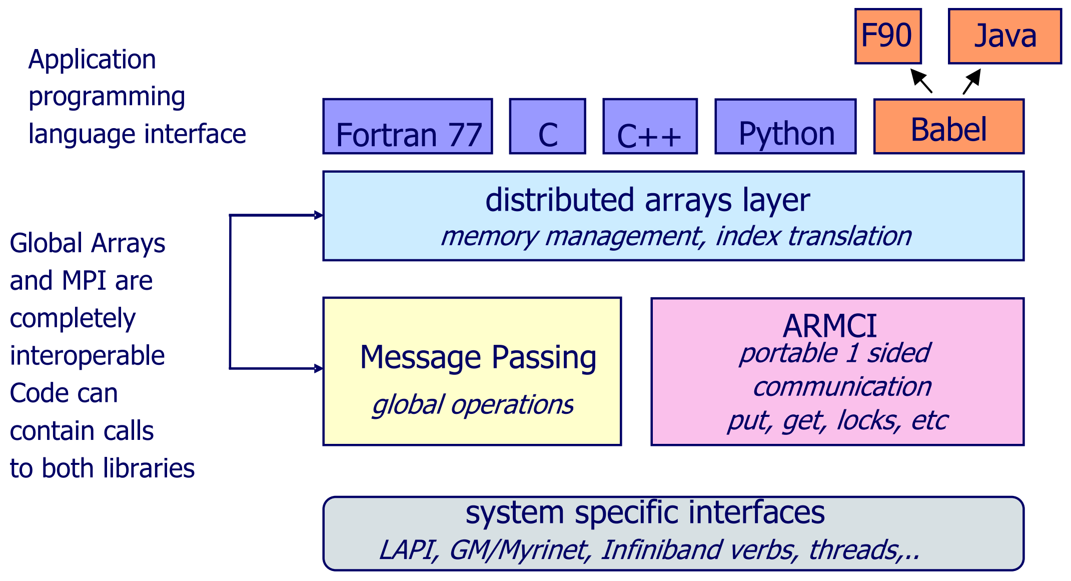

F90
Java
Application programming language interface
Fortran 77
C
C++
Python
Babel
distributed arrays layer
memory management, index translation
Global Arrays and MPI are completely interoperable Code can contain calls to both libraries
Message Passing
global operations
ARMCI
portable 1 sided communication
put, get, locks, etc
system specific interfaces
LAPI, GM/Myrinet, Infiniband verbs, threads,..

Figure 5: Diagram of Global Arrays showing the overall structure of the toolkit and emphasizing different language bindings

3.4 Language Interfaces

One of the strengths of the GA toolkit is its ambivalence toward the language that the end user application is developed in. Interfaces in C, C++, Fortran, and Python to GA have been developed (Figure 5). Mixed language applications are supported as well. For example, a global array created from a Fortran interface can be used within a C function, provided that the corresponding data types exist in both languages. The view of the underlying data layout of the array is adjusted, depending on the language bindings, to account for the preferred array layout native to the language (column-based in Fortran and row-based in C/C++/Python). A 50x100 array of double precision data created from the Fortran interface is available as a 100x50 array of doubles through the C bindings.

Recently, Babel [43] interfaces to GA have been developed. Babel supports additional translation of the GA interfaces to Fortran 90 and Java.

4. Efficiency and Portability

GA uses ARMCI (Aggregate Remote Memory Copy Interface) [16] as the primary communication layer. Collective operations, if needed by the user program, can be handled by MPI. Neither GA nor ARMCI can work without a message-passing library that provides the essential services and elements of the execution environment (job control, process creation, interaction with the resource manager).The Single Program Multiple Data (SPMD) model of computations is inherited from MPI, along with the overall execution environment and services provided by the operating system to the MPI programs. ARMCI is currently a component of the run-time system in the Center for Programming Models for Scalable Parallel Computing project [44]. In addition to being the underlying communication interface for Global Arrays, it has been used to implement communication libraries and compilers [16, 45-47]. ARMCI offers an extensive set of functionality in the area of RMA communication: 1) data transfer operations; 2) atomic operations; 3) memory management and synchronization operations; and 4) locks. Communication in most of the non-collective GA operations is implemented as one or more ARMCI communication operations. ARMCI was designed to be a general, portable, and efficient one-sided communication interface that is able to achieve high performance [48-51]. It also avoided the complexity of the progress rules and increased synchronization in the MPI-2 one-sided model (introduced in 1997), that contributed to its delayed implementations and still rather limited adoption (as of 2004).

During the ACTS project, development of the ARMCI library represented one of the most substantial tasks associated with advancements of GA. It implements most of the low level communication primitives required by GA. High performance implementations of ARMCI were developed under the ACTS project within a year for the predominant parallel systems used in the US in 1999 [16, 51] and it has been expanded and supported since then on most other platforms, including massively parallel scalar and vector supercomputers [52] as well as clusters [48, 50].

GA's communication interfaces combine the ARMCI interfaces with a global array index translation layer, see Figure 6. Performance of GA is therefore proportionate to and in-line with ARMCI performance. GA achieves most of its portability by relying on ARMCI. This reduces the effort associated with porting GA to a new platform to porting ARMCI. GA relies on the global memory management provided by ARMCI, which requires that remote memory be allocated via the memory allocator routine, ARMCI_Malloc (similar to MPI_Win_malloc in MPI-2). On shared memory systems, including SMPs, this approach makes it possible to 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 [51]. Similarly, on clusters with networks based on physical memory RDMA, registration of allocated memory maks it suitable for zero-copy communication. To support efficient communication in the context of multi-dimensional arrays, GA requires and utilizes the non-contiguous vector and multi-strided primitives provided by ARMCI [16].

Many GA operations achieve good performance by utilizing the low-overhead high-overlap non-blocking operations in ARMCI to overlap computation with communication. Even blocking one-sided GA operations internally use non-blocking ARMCI operations to exploit available concurrency in the network communication. Figure 6 shows how a GA_Get call is implemented and eventually translated to ARMCI Get call(s). The left side represents a flow chart and the right side shows the corresponding example for the flowchart. A GA Get call first requires determination of data locality: the physical location of the data in the distributed/partitioned address space needs to be determined. Then indices corresponding to where the data is located (on that process) need to be found. When this information is available, multiple ARMCI non-blocking Get calls are made, one for each remote destination that holds a part of the data. After all the calls are issued, they are waited upon until completed. By issuing all the calls first and then waiting on their completion, a significant amount of overlap can be achieved when there is more than one remote destination. When the wait is completed, the data is in the buffer and the control is returned to the user program.

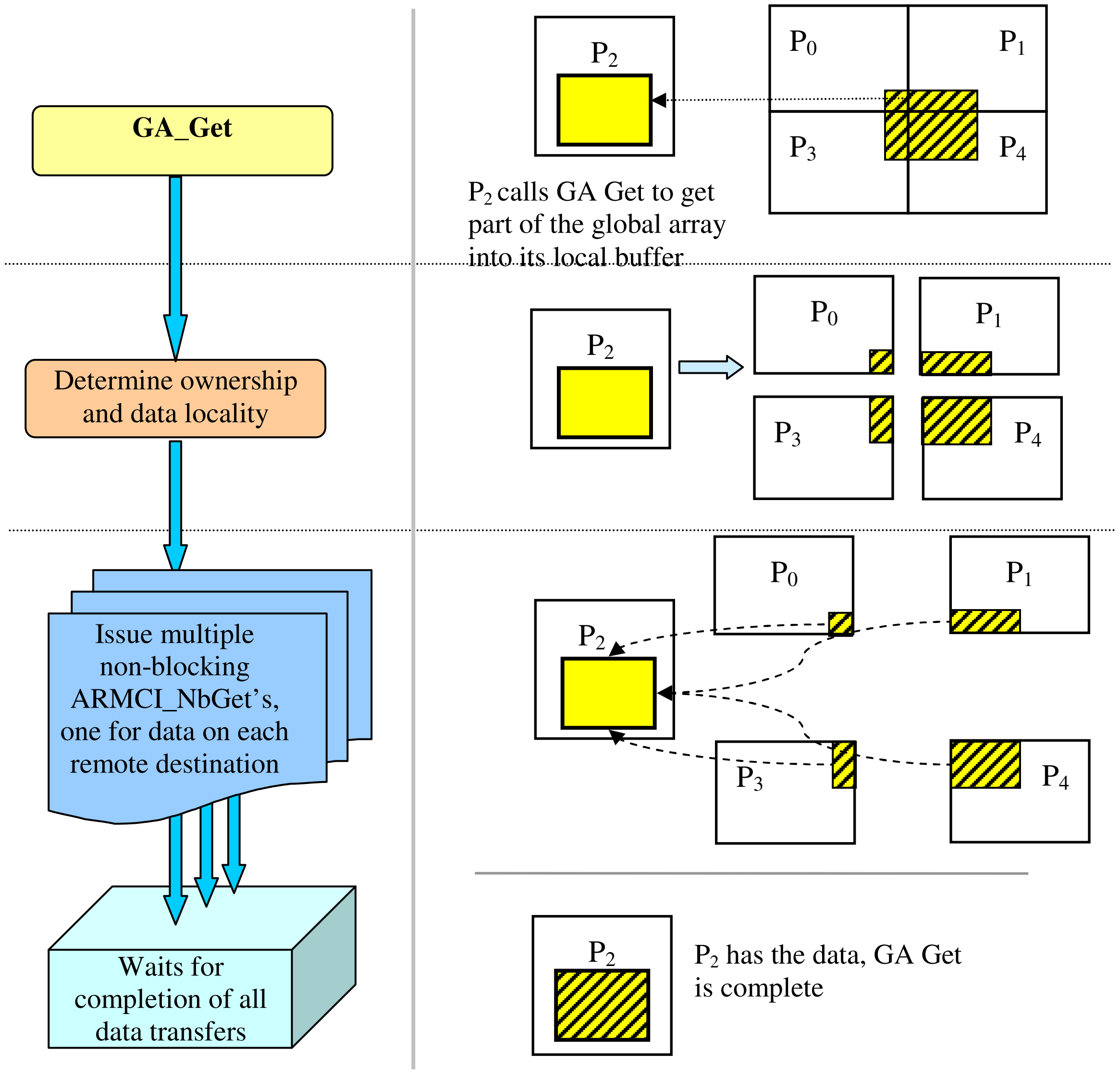
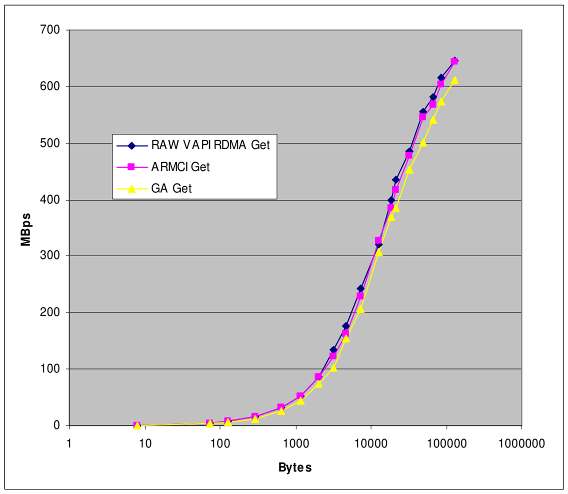

GA_Get
Determine ownership and data locality
Issue multiple non-blocking ARMCI_NbGet's, one for data on each remote destination
Waits for completion of all data transfers
P2
P0
P1
P3
P4
P2 calls GA Get to get part of the global array into its local buffer
P2 has the data, GA Get is complete

Figure 6: Left: *GA_Get* flow chart. Right: An example: Process P_2 issues *GA_Get* to get a chunk of data, which is distributed (partially) among P_0, P_1, P_3 and P_4 (owners of the chunk).

On cluster interconnects, ARMCI achieves bandwidth close to the underlying network protocols [50], see Figure 7. The same applies to latency if the native platform protocol supports the equivalent remote memory operation (e.g., **`elan_get`** on Quadrics). For platforms that do not support remote get (VIA) the latency sometimes includes the cost of interrupt processing that is used in ARMCI to implement the get operation. Although, relying on interrupt may increase the latency over the polling-based approaches, progress in communication is guaranteed regardless of whether or not the remote process is computing or communicating. The performance of inter-node operations in GA closely follows the performance of ARMCI. Thanks to its very low overhead implementation, ARMCI is able to achieve performance close to that of native communication protocols, see Figure 7. These benefits are carried over to GA, making its performance very close to native communication protocols on many platforms. This can be seen in Figure 7, which compares GA Get performance with ARMCI Get performance and raw native network protocol performance on the Mellanox Infiniband and Elan4 (both are popular high speed current generation cluster interconnects). Figure 8 shows the performance of GA Get and Put on Linux clusters. Figure 9 shows the performance of GA Get/Put strided calls for square sections of a two-dimensional array, which involve non-contiguous data transfers. Latencies in GA and ARMCI operations are compared in Table 1. The small difference between performances of these two interfaces is due to the extra cost of the global array index translation, see Figure 6. The differences are considerable in the shared memory version because simple load/store operation is faster than the index translation.

Table 1: Latency (in microseconds) in GA and ARMCI operations

Operation/platform	Linux 1.5GHz IA64 Elan-4	Linux 1GHz IA64 4X Infiniband	Linux 2.4GHz IA32 Myrinet-2000/C card	Linux 2.4GHz IA32 shared memory
ARMCI Get	4.54	16	17	0.162
GA Get	6.59	22	18	1.46
ARMCI Put	2.45	12	12	0.17
GA Put	4.71	16	13	1.4



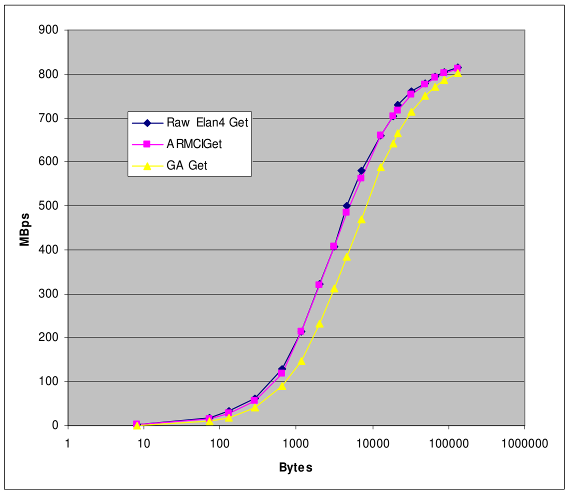
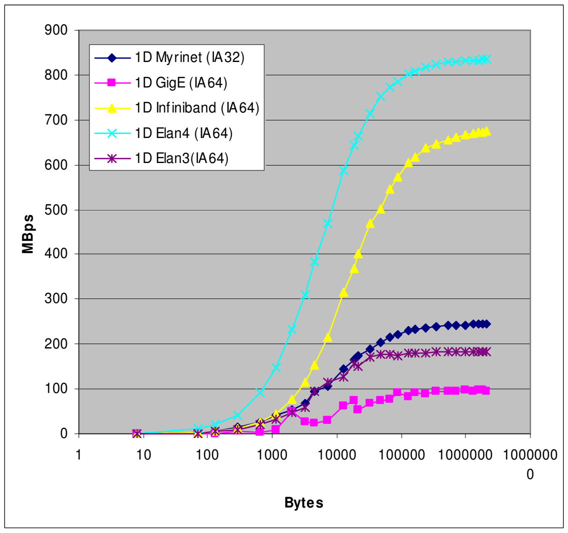


Figure 7: Comparison of GA Get with ARMCI Get and native protocols performance Left: InfiniBand (IA64), Right: Quadrics Elan4 (IA64)



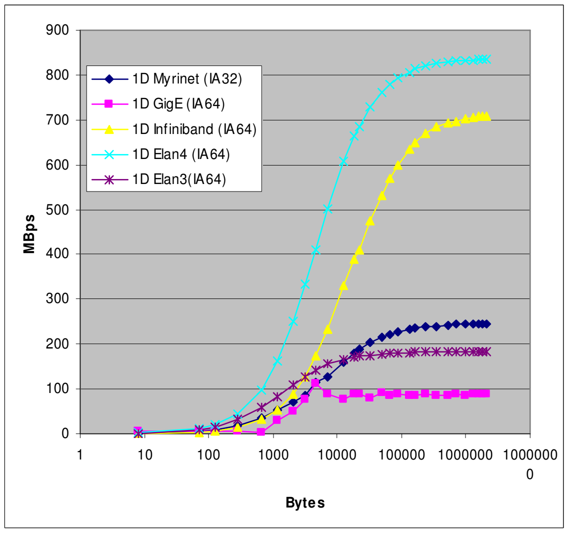
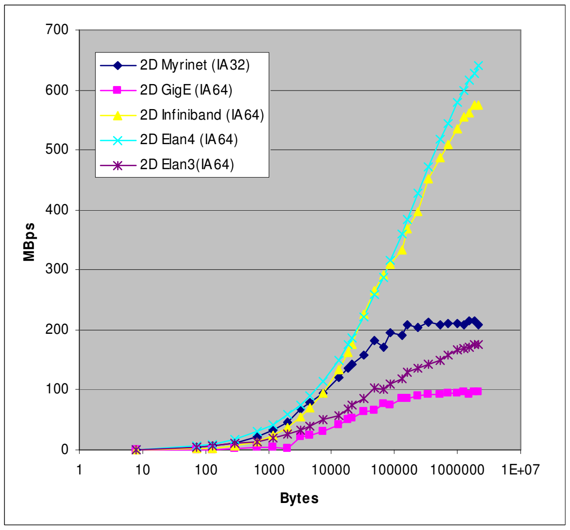


Figure 8: Performance of basic GA 1D operations on Linux clusters: get (left), put (right)



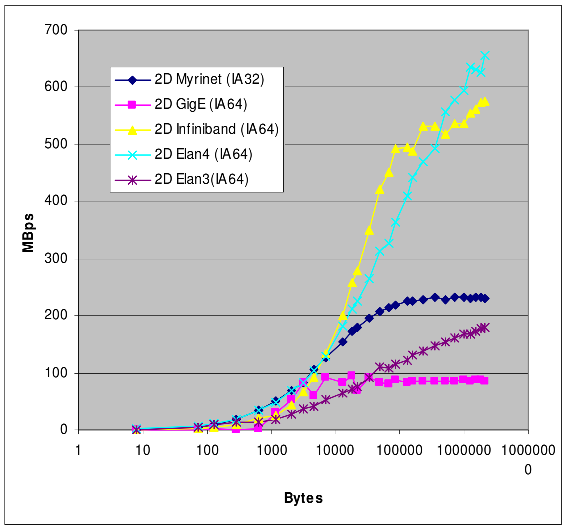


Figure 9: Performance of basic GA 2D operations: get (left), put (right)

5. Advanced Features

The GA model was defined and implemented ten years ago [2], and then ported to the leading parallel machines of that time. In addition to ports and optimizations that have been introduced since then, the GA toolkit has evolved dramatically in terms of its capabilities, generality, and interoperability. In this section, advanced capabilities of the GA toolkit are discussed.

5.1 Ghost Cells and Periodic Boundary Conditions

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 [53]. Examples of other packages that include support for ghost cells include POOMA [54], Kelp [55], Overture [56], and Zoltan [57]. 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 10. 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 best. The implementation of the update 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 several 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 function that cancels any synchronization points in the update operation, see Section 5.5. This can be used to eliminate unnecessary synchronizations in codes where other parts of the algorithm guarantee consistency of the data.

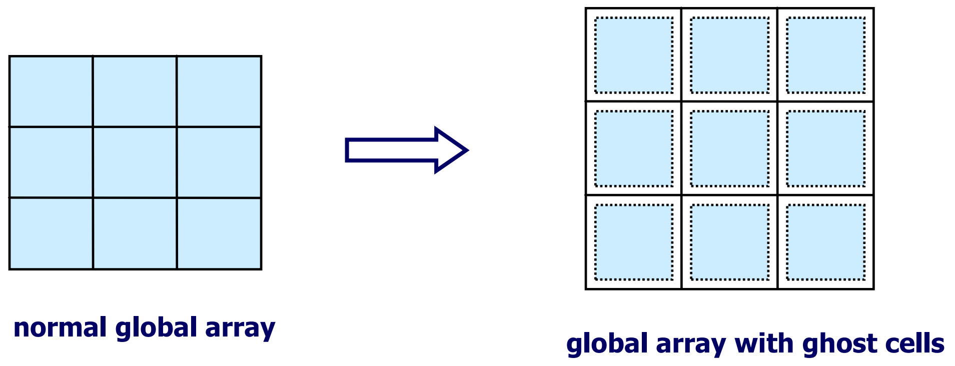


Figure 10: An ordinary global array distributed on 9 processors (left) and the corresponding global array with ghost cells (right).

Along with ghost cells, additional operations are provided that can be used to implement periodic boundary conditions on periodic or partially periodic grids. All the onesided operations (put/get/accumulate) in GA are available in versions that support periodic boundary conditions. The syntax for using these commands is that the user requests a block of data using the usual global index space. If one of the dimensions of the requested block exceeds the dimension of the global array, that portion of the request is automatically wrapped around the array edge and the data from the other side of the array is used to complete the request. This simplifies coding of applications using periodic boundary conditions, since the data can be copied into a local buffer that effectively "pads" the original array to include the wrapped data due to periodicity. This eliminates the need to explicitly identify data at the edge of the array and makes coding much simpler.

5.2 Sparse Data Management

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 developed to operate on distributed, compressed, sparse matrix data structures built on top of one-dimensional global arrays [58, 59]. These functions have been patterned after similar functions in CMSSL, developed for Thinking Machines CM-2 and CM-5 massively parallel computers in the late 80's and early 90's. Some of them were also included in the set of HPF intrinsic functions [9]. The types of functions that have been designed, implemented and tested include: 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 structures. Numerical operators defined on unstructured meshes typically have sparse matrix representations that are stored as one-dimensional data structures. The Global Array functions for manipulating one-dimensional arrays have been adopted in mesh generation (NWGrid [60]) and computational biophysics (NWPhys [61]) codes.

The use of some of these functions can be seen by considering a sparse matrix multiplying a sparse vector. Sparse data structures are generally stored by mapping them onto dense, one-dimensional arrays. A portion of a sparse matrix stored in column major form is shown in Figure 11 along with its mapping to a corresponding dense vector. Only non-zero values are stored, along with enough information about indices to reconstruct the original matrix.

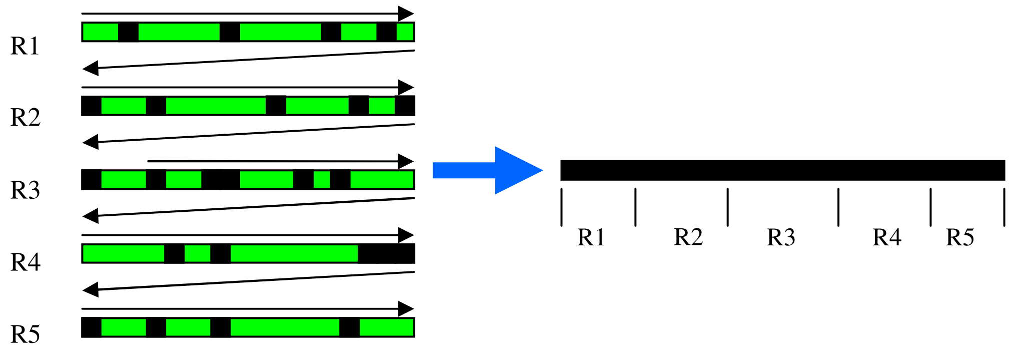


Figure 11. A portion of a sparse matrix in column major form is remapped to a dense, one-dimensional array. Rows 1-5 are converted to short segments in the one-dimensional array.

Generally, only compressed arrays are created, construction of the full sparse matrix is avoided at all steps in the algorithm. The steps of a sparse matrix-vector multiply are illustrated in Figure 12. The original sparse matrix and sparse vector are shown in Figure 12(a). The sparse vector is then mapped onto a matrix with the same sparsity pattern as the original sparse matrix (Figure 12(b)). This mapping is partially accomplished using the enumerate command. Note that the remapped sparse vector may have many zero entries (shown as hatched elements in the figure). Both the original matrix and the sparse matrix representation of the vector are written as dense one-dimensional arrays (Figure 12(c)). The multiplication can be completed by multiplying together each component of the two dense vectors and then performing a segmented_scan_with_OP, where OP is addition, to get the final product vector. The segmented_scan_with_OP adds together all elements within a segment and can be used to add together all elements in the product matrix corresponding to an individual row (Figure 12(d)).

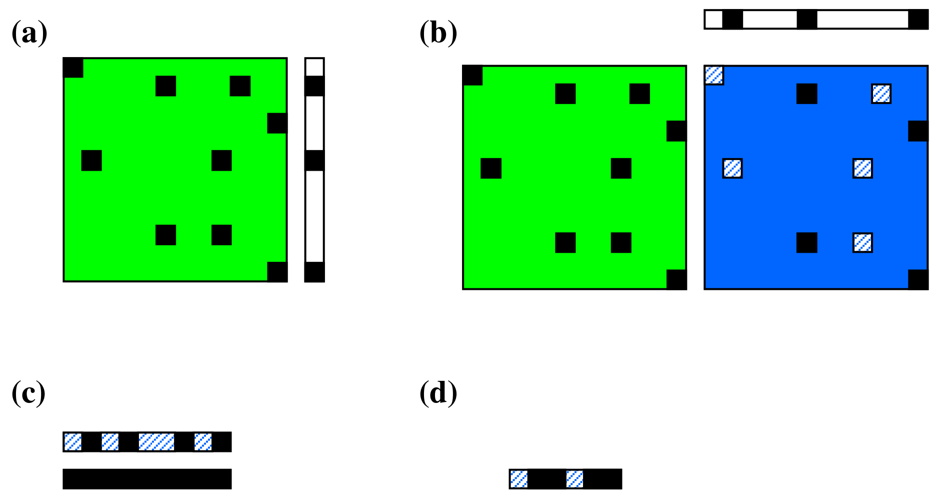


Figure 12. (a) original sparse matrix-vector multiply (b) sparse vector has been expanded to a sparse matrix (transpose of original sparse vector is included to show how mapping is accomplished) (c) Compressed versions of sparse matrix and vector (d) product vector after element-wise multiplication and segmented scan with addition.

The pack/unpack functions can be used to work on portions of the sparse data structure by masking portions of the data structure and copying it into another array. These can be used to implement the equivalent of the HPF **`where`** statement. The binning routines can be used to partition and manipulate structures. For example, the N-to-M binning function can be used to spatially partition an unstructured mesh using a regular mesh superimposed on top of it.

5.3 Nonblocking Communication

Nonblocking communication is a mechanism for latency hiding where a programmer attempts to overlap communication with computation. In some applications, by pipelining communication and computation, the overhead of transferring data from remote processors can be overlapped with calculations. The nonblocking operations (get/put/accumulate) are derived from the blocking interface by adding a handle argument that identifies an instance of the non-blocking request. Nonblocking operations initiate a communication call and then return control to the application. A return from a nonblocking operation call indicates a mere initiation of the data transfer process and the operation can be completed locally by making a call to the wait (**`ga_wait`**) routine. Waiting on a nonblocking put or an accumulate operation assures that data was injected into the network and the user buffer can be now be reused. Completing a get operation assures data has arrived into the user memory and is ready for use. The wait operation ensures only local completion. Unlike their blocking counterparts, the nonblocking operations are not ordered with respect to the destination. Performance is one reason, another is that by ensuring ordering we incur additional and possibly unnecessary overhead on applications that do not require their operations to be ordered. For cases where ordering is necessary, it can be done by calling a fence operation. The fence operation is provided to the user to confirm remote completion if needed.

It should be noted that most users of nonblocking communication implicitly assume that progress in communication can be made concurrently in a purely computational phase of the program execution. However, this assumption is often not satisfied in practice -- the availability of a nonblocking API does not guarantee that the underlying system hardware and native protocols support overlapping communication with computation [62].

A simple benchmark was performed in the context of GA and MPI to demonstrate the overlap of communication with computation. We measure the overlap as follows: we assume the time to issue a non-blocking call is a constant, which can be represented by t_i. The time to wait for a non-blocking call (or the time taken to issue a wait for the non-blocking call) is $t_d + t_w$, where t_d represents time spent waiting for the data to arrive and t_w represents the time taken to complete the wait call when the data has already arrived. $t_i + (t_d + t_w)$ is the total time taken when the non-blocking call is issued and waited on immediately. This time is typically the same as the time taken to issue a blocking version of the same call. In the non-blocking call, t_d represents the time that can be effectively be utilized in doing computation. A very good measure for the effectiveness of a non-blocking call is to see what percentage of the total time t_d represents $(t_d * 100) / (t_i + (t_d + t_w)$. A higher percentage indicates more overlap is possible.

We performed an experiment on two nodes with one node issuing a nonblocking get for data located on the other, and then waiting for the transfer to be completed in the wait call. We also implemented an MPI version of the above benchmark; our motivation was to compare the overlap in GA with the overlap in the MPI nonblocking send/receive operations. In MPI, if one process (A) needs a portion of data from another one (B), it sends a request and waits on a nonblocking receive for the response. Process A's calling sequence is as follows: 1) MPI_Isend, 2) MPI_Irecv, 3) MPI_wait (waits for MPI_Isend to complete), 4) MPI_Wait (waits for MPI_Irecv to complete). Process B's calling sequence is: 1) MPI_Recv and 2) MPI_Send. In process A, computation is gradually inserted between the initiating nonblocking Irecv call (i.e after step 3) and the corresponding wait completion call. We measured the computation overlap for both the GA and MPI versions of the benchmark on a Linux cluster with dual 2.4GHz Pentium-4 nodes and Myrinet-2000 interconnect. The results are plotted in Figure 13. The percentage overlap (represented by $(t_d * 100) / (t_i + (t_d + t_w)))$ is effectively a measure of the amount of time that a nonblocking (data transfer) call can be overlapped with useful computation. We observe that GA offers a higher degree of overlap than MPI. For larger messages (>16K) where the MPI implementation switches to the Rendezvous protocol (which involves synchronization between sender and receiver), we were able to overlap almost the entire time (>99%) in GA, where as in MPI, it is less than 10%.

The experimental results illustrating limited opportunities for overlapping communication and computation are consistent with findings reported for multiple MPI implementations in [62-64]. Since the GA represents a higher–level abstraction model and, in terms of data transfer, simpler than MPI (e.g., it does not involve message tag matching or dealing with early arrival of messages), more opportunities for effective implementation of overlapping communication with computation are available.

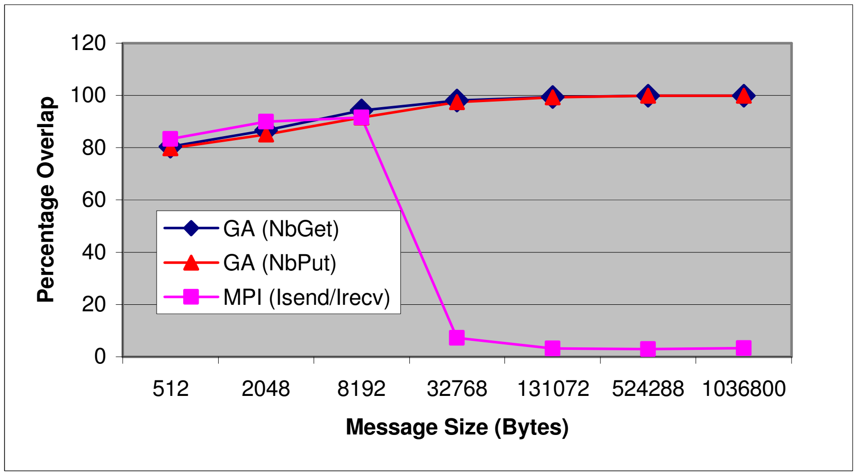


Figure 13: Percentage computation overlap for increasing message sizes for MPI and GA on Linux/Myrinet

5.4 Mirroring: Shared Memory Cache for Distributed Data

Caching distributed memory data in shared memory is another mechanism for latency hiding supported in the GA toolkit [65]. It has been primarily developed for clusters with SMP nodes; however earlier it was used for grid computing [66, 67]. Compared to most custom supercomputer designs, where the CPU power is balanced with a high speed memory subsystem and high-performance network, commodity clusters are often built based on very fast processors using relatively low-performance networks. Mirrored arrays are designed to address these configurations by replicated data across nodes but distributing it and storing in shared memory within the SMP nodes (Figure 14). This technique has several potential advantages on clusters of SMP nodes, particularly if the internode communication is slow relative to computation. Work can be done on each "mirrored" copy of the array independently of copies on other nodes. Within the node the work is distributed, which saves some memory (on systems with many processors per node, this savings can be quite substantial relative to strict replication of data). Intranode communication is via shared memory and is therefore very fast. One-sided operations such as put, get, and accumulate are only between the local buffer and the mirrored copy on the same node as the processor making the request. Most operations that are supported for regular global arrays are also supported for mirrored arrays, so the amount of user code modification in transitioning from fully distributed to mirrored schemes is minimal. Operations between two or more global arrays are generally supported if both arrays are mirrored or both arrays are distributed. Mirrored arrays can be used in situations where the array is being exclusively accessed for either reading or writing. For the DFT application described below, data is read from one mirrored array using the **`nga_get`** operation and accumulated to another mirrored array using **`nga_acc`**. Mirrored arrays also limit the problem size to systems that can be contained on a single node. In fact, mirrored arrays are essentially using memory to offset communication, but because data is distributed within the node they are more efficient than strict replicated data schemes.

At some point the different copies of the array on each node must be combined so that all copies are the same. This is accomplished with the **`ga_merge_mirrored`** function. This function adds together all copies of the mirrored array across all nodes. After merging, all copies of the mirrored array are the same and equal to the sum of all individual copies of the array. This function allows programmers to combine the work that is being done on separate SMP nodes together to get a single answer that is easily available to all processors. In addition to creating the new merge operation, the copy operations have been augmented so that they work between a mirrored array and a regular distributed array. The copy operation does not implicitly perform a merge if the mirrored array is copied into distributed array. The availability of an easy conversion between mirrored and distributed arrays allows programmers to convert some parts of their code to use mirrored arrays and leave other parts of their code using distributed arrays. Code that is limited by communication can be converted to use mirroring while the remaining code can be left using distributed arrays, thereby saving memory.

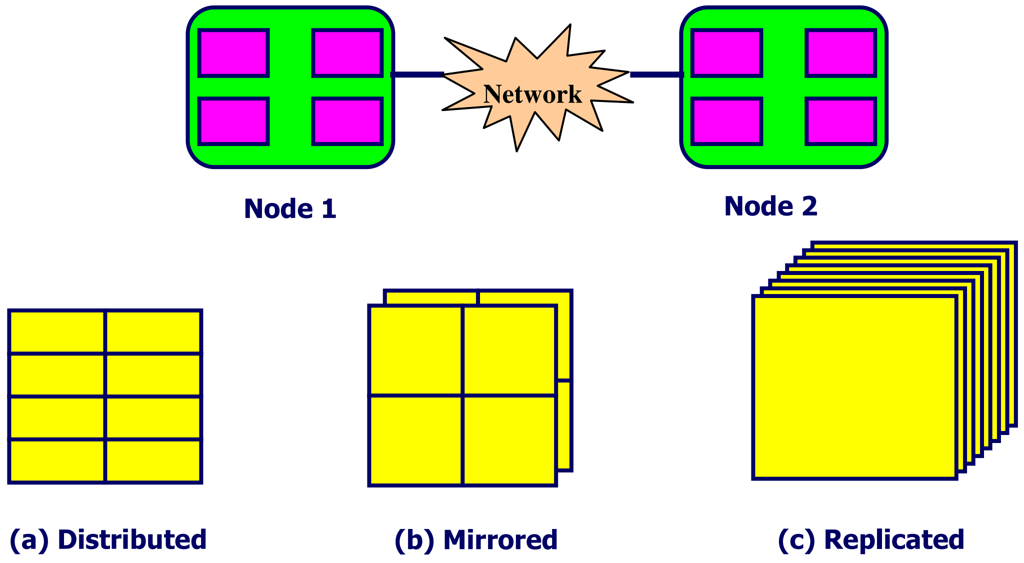


Figure 14: Example of a 2-dimensional array fully distributed (a), SMP mirrored (b), and replicated (c) on a two 4-way SMP cluster nodes

5.5. Synchronization Control

GA includes a set of data-parallel interfaces that operate on global arrays, including BLAS-like linear algebra operations. As a convenience to the programmer (especially novice users), to simplify management of memory consistency, most data parallel operations include at the beginning and at the end a global barrier operation. The role of the initial barrier is to assure safe transition from the task parallel to the data parallel phase of computations. Specifically, before the data in a global array is accessed in the data parallel operation, the barrier call synchronizes the processors and completes any outstanding store operations that could modify the data in the global array. Similarly, the final barrier assures that all processors committed their changes to the global array before it can be accessed remotely. Although the barrier operation is optimized for performance, and, where possible, uses hardware barriers, it is a source of overhead, especially in fine-grain sections of the applications. The importance of reducing barrier synchronization has been recognized and studied extensively in the context of data-parallel computing [68-70]. In order to eliminate this overhead, the toolkit offers an optional **`ga_mask_sync`** operation that allows the programmer to eliminate either of the two barriers before calling a data parallel operation. This operation updates the internal state of the synchronization flags so that when the actual operation is called, one or both barriers can be eliminated. The temporal scope of the mask operation is limited to the next data parallel operation and when that is completed the status of the synchronization flags is reset. The availability of this mechanism enables the programmer, after debugging and analyzing dependencies in his/her code, to improve performance by eliminating redundant barriers.

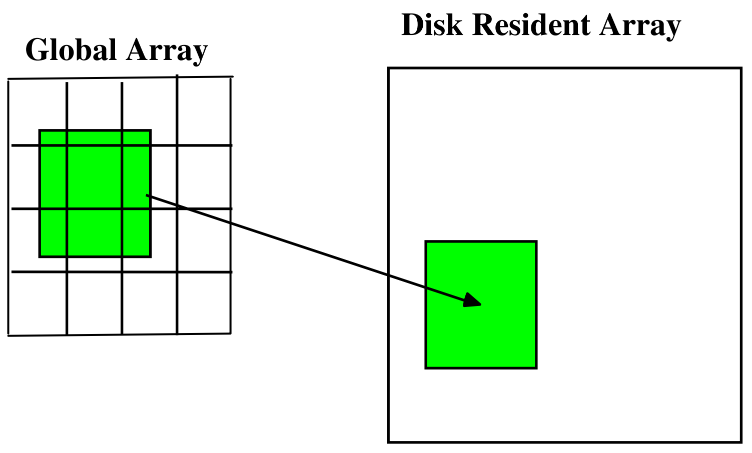
5.6 Locks and Atomic Operations

Atomic operations such as fetch-and-add can be used to implement dynamic load balancing (see Section 6.1) or mutual exclusion. In addition, GA through ARMCI offers explicit lock operations that help the programmer to protect critical sections of the code. ARMCI lock operations are optimized to deliver better performance than the user would otherwise be able to implement based on fetch-and-add [48, 71].

Moreover, GA offers an atomic reduction operation, **`accumulate`**, that has built in atomicity and thus does not require explicit locking. This operation is one of the key functionalities required in quantum chemistry applications, and makes the use of locks in this application area rare. This operation and its use in chemistry was the primary motivation for including **`mpi_accumulate`** in the MPI-2 standard. One important difference is that GA, unlike MPI with its distributed memory model, does not specify explicitly the processor location where the modified data is located. In addition, GA provides an additional scaling parameter that increases generality of this operation. This is similar to the BLAS **daxpy** operation.

5.7. Disk Resident Arrays

Global Arrays provide a convenient mechanism for programs to store data in a distributed manner across processors and can be considered as a level in a memory hierarchy. This particular level can represent the memory of the entire system and is therefore much larger than the memory on any single processor. However, many applications still require more memory than is available in core, even when data is distributed. For example, in computational chemistry there are two strategies for dealing with this situation [72]. The first is to reorganize the calculation so that intermediate results are no longer stored in memory but are recomputed when needed. This approach results in excess calculations and the number of computations increase as the memory requirements are decreased. The cost of these excess calculations can be reduced by designing hybrid algorithms that store some results and recompute the remainder with the goal of choosing the partition in a way that simultaneously minimizes recomputation and memory. The other approach is to write intermediate results to disk, which typically can store much more data than memory. The tradeoff here is the cost of recomputing results compared to the I/O cost of writing and reading the data to a file. The advantage to this approach is that the amount of data that can be stored to disk is usually orders of magnitude higher than that can be stored in core.



`ndra_write_section(transp,g_a,glo,ghi,d_a,dlo,dhi,rid)`

Figure 15: Schematic of a write operation from a patch of a global array g_a(glo:ghi) to a patch of a disk resident array d_a(dlo:dhi)

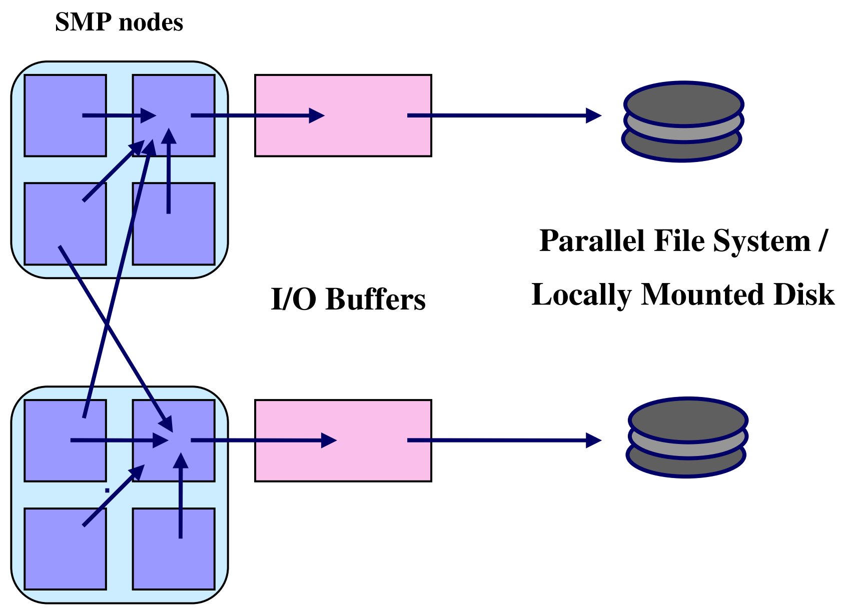


Figure 16: Write operation between a patch of a global array to a patch of a disk resident array. Data flows from the global array to the I/O processors and then is written to disk.

Disk Resident Arrays (DRAs) are designed to extend the concept of Global Arrays to the file system, in effect, treating disk as another level in the memory hierarchy [72, 73]. A DRA is essentially a file, or collection of files, that represent a global array stored on disk. Data stored in a disk resident array can be copied back and forth between global arrays using simple read and write commands that are similar to the syntax of the global arrays **`nga_copy_patch`** commands. This operation is illustrated schematically in Figure 15. The collective mode of operation in DRA increases the opportunities for performance optimizations [74]. Data in the DRA can be referred to using a global index space, identical to that used in Global Arrays. The details of how data is partitioned within a single file or divided between multiple files are hidden from the user, however, the underlying code is designed to partition data on the disk to optimize I/O. The use of multiple files allows the system to read and write data to disk from multiple processors. If the local scratch space mounted on each node or a parallel file system is used for these files, this can greatly increase the bandwidth for reading and writing to the DRAs. The flow of data for a write statement to a DRA distributed on separate disks is illustrated schematically in Figure 16.

This example represents a write request from a patch of a global array to a patch of the DRA. The DRA patch is first partitioned between the different files on disk. Each file is controlled by a single processor, typically one I/O processor per SMP node. Once the DRA patch has been decomposed between files, the global array patch is also decomposed so that each portion of the global array is mapped onto its corresponding portion of the DRA patch. Each piece of the global array data is then moved to the I/O processor and copied into the I/O buffer. Once the data is in the I/O buffer, it is then written to disk. This operation can occur independently on each I/O processor, allowing multiple read/write operations to occur in parallel. The partitioning of data on the disk is also chosen to optimize I/O for most data requests. In addition to computational chemistry applications [72], DRAs have been used to temporarily store large data sets to disk in image processing applications [75].

5.8 GA Processor Groups

GA supports creating and managing arrays on processor groups for the development of multi-level parallel algorithms [21]. Due to the required compatibility of GA with MPI, the MPI approach to the processor group management was followed as closely as possible. However, in shared memory programming management of memory and shared data rather than management of processor groups itself is the primary focus. More specifically we need to determine how to create, efficiently access, update, and destroy shared data in the context of the processor management capabilities that MPI already provides. One of the fundamental group-aware GA operations involves the ability to create shared arrays on subsets of processors. Every global array has only one associated processor group specifying the group that created the array. Another useful operation is the data-parallel copy operation that works on arrays (or subsections) defined on different processor groups as long as the intersection of these groups is a non-empty set. In the GA programming model, data distributed in a processor group (containing *M* processors) can be redistributed to another processor group (containing *N* processors) regardless of the number of processors in each group and the data layout. This can be done as a collective call across processors in both the groups or as a non-collective one-sided operation. This feature enabled development of applications with nontrivial relationships between processor groups.

The concept of the *default processor group* is a powerful capability added to enable rapid development of new group-based codes and simplify conversion of the existing non-group aware codes. Under normal circumstances, the default group for a parallel calculation is the MPI “world group” (contains the complete set of processors user allocated), but a call is available that can be used to change the default group to a processor subgroup. This call must be executed by all processors in the subgroup. Once the default group has been set, all operations are implicitly assumed to occur on the default processor group unless explicitly stated otherwise. By default, GA shared arrays are created on the default processor group and global operations by default are restricted to the default group. Inquiry functions, such as the number of nodes and the node ID, return values relative to the default processor group.

5.9 Common Component Architecture (CCA) GA Component

The Common Component Architecture (CCA) is a component model specifically designed for high performance computing. Components encapsulate well-defined units of reusable functionality and interact through standard interfaces [36]. The GA component, an object-oriented CCA based component, provides interfaces to full capabilities of GA. This component supports both classic and SIDL interfaces, and it provides three ports: *GlobalArrayPort*, *DADFPort* and *LinearAlgebraPort*. These ports are the set of public interfaces that the GA component implements and can be referenced and used by other components. The *GlobalArrayPort* provides interfaces for creating and accessing distributed arrays. The *LinearAlgebraPort* provides core linear algebra support for manipulating vectors, matrices, and linear solvers. The *DADFPort* offers interfaces for defining and querying array distribution templates and distributed array descriptors, following the API proposed by the CCA Scientific Data Components Working Group [76]. The GA component is currently used in applications involving molecular dynamics and quantum chemistry, as discussed in [39].

Figure 17 illustrates an example of CCA components in action in a CCA (e.g. CCAFFEINE) Framework [77]. The GA component adds the “provides” ports, which is visible to other components to the CCA *Services* object. TAO component registers the ports that it will need with the CCA *Services* object. The CCAFEINE framework connects GA and TAO components and transfers the *LinearAlgebraPort(LA)* to the TAO component, using the GA Component’s *Services* object.

In [22], experimental results for numerical Hessian calculation show that multilevel parallelism expressed and managed through the CCA component model and GA processor groups can be very effective for improving performance and scalability of NWChem. For example, the numerical Hessian calculation using three levels of parallelism outperformed the original version of the NWChem code based on single level parallelism by a factor of 90% when running on 256 processors.

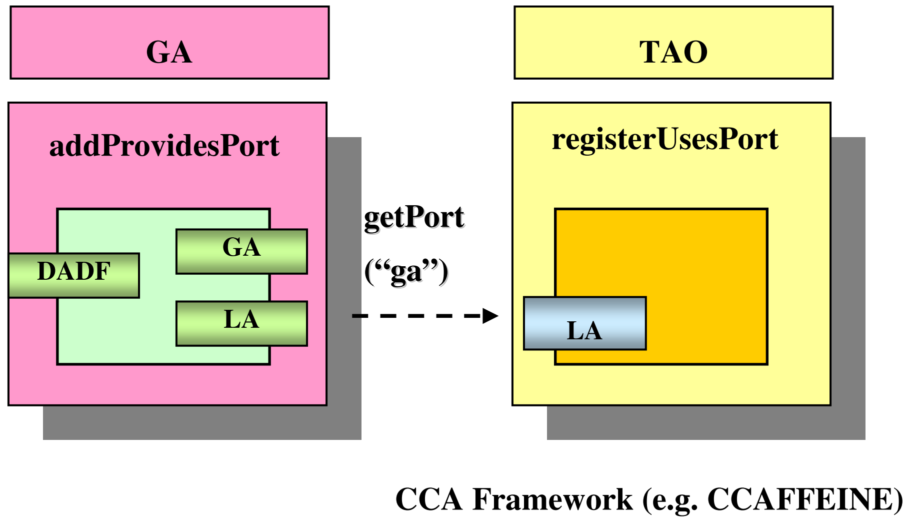


Figure 17: CCA Components in action. TAO Component uses the LA Port (Linear Algebra) provided by GA Component for manipulating vectors and matrices.

6 Applications of Global Arrays

The original application for GA was to support electronic structure codes and it remains the de facto standard for managing data transfer in most programs that perform large, scalable electronic structure calculations. Electronic structure calculations involve the construction of large, dense matrices. Once constructed, these are subsequently manipulated using standard linear algebra operations to produce the final answer. The construction of the matrices is highly parallel in that it can be divided into a large number of smaller tasks. Each task can be assigned to a processor, which is then responsible for working on a portion of the matrix and accumulating the results into a product matrix. The tasks typically require copying a portion of one matrix into a local buffer, doing some work on it, and then copying and accumulating the result back into another matrix. The natural decomposition of these tasks is easily formulated in terms of the dimensions of the original matrices but typically results in copying portions of the matrix that are distributed over several processors to local buffers. The copy operations between the local buffer and the distributed matrices therefore require communication with multiple processors and would be quite complicate if coded using a standard message-passing interface. Global Arrays can accomplish each copy with a single function call using the global index space. The matrix operations that must be performed to get the final answer are also highly non-trivial for distributed data. GA supports many of these operations directly and also provides interfaces to other linear algebra packages, such as SCALAPACK. Because all the information about the matrix and how it is distributed are already contained in the global array, these interfaces are quite simple and reflect the algebra of the original problem, rather than the details of how the data is distributed.

The ability of Global Arrays to manage large distributed arrays has made them useful in many other areas beyond electronic structure. Any application that requires large, dense, multi-dimensional data grids can make use of toolkit. Examples are algorithms and applications built around dense matrices (e.g. electronic structure) and algorithms on multi-dimensional grids (hydrodynamics and other continuum simulations). Even algorithms that are based on sparse data structures (unstructured grids in finite element codes) often convert the original sparse data to a dense 1-dimensional array. For most of these applications it remains attractive to treat these data structures as single arrays using a global index space that maps directly onto the original problem. However, the large amount of data typically involved means that the data must be distributed, which makes the concept of a single local data structure impractical. Data must now be accessed by referring to a local index that identifies the data within a single processor, as well as an index identifying the processor that the data is stored on. The connection between the original problem and the data is lost and must now be managed by the application programmer. GA provides higher level abstractions that are designed to restore the connection between the global index space of the original problem and the distributed data by providing an interface that manages all the necessary transformations between the global index space and the local indices that specify where data is actually located. This approach vastly simplifies programming and thus improves productivity [78]. The toolkit also provides mechanisms for identifying what data is held locally on a processor, allowing programmers to make use of data locality when designing their programs, and even provides direct access to the data stored in the global array, which saves on the time and memory costs associated with duplication.

6.1 Molecular Dynamics

Molecular dynamics (MD) is a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating their equations of motion. For this application, the force between two atoms is approximated by a Lennard-Jones potential energy function *U(r)*, where *r* is the distance between two atoms. Good performance and scalability in the application require an efficient parallel implementation of the objective function and gradient evaluation. These routines were implemented by using GA to decompose the atoms over the processors and distribute the computation of forces in an equitable manner. The decomposition of forces between atoms is based on a block decomposition of the forces distributed among processors, where each processor computes a fixed subset of inter-atomic forces [79]. The entire array of forces (N x N) is divided into multiple blocks (m x m), where m is the block size and N is the total number of atoms. Each process owns N/P atoms, where P is the total number of processors. Exploiting the symmetry of forces between two particles halves the amount of computation. The force matrix and atom coordinates are stored in a global array. A centralized task list is maintained in a global array, which stores the information of the next block that needs to be computed.

To address the potential load imbalance in our test problem, we use a simple and effective dynamic load-balancing technique called fixed-size chunking [80]. This is a good example illustrating the power of shared memory style management of distributed data that makes the GA implementation both simple and scalable. Initially, all the processes get a block from the task list. Whenever a process finishes computing its block, it gets the next available block from the task list. Computation and communication are overlapped by issuing a nonblocking get call to the next available block in the task list, while computing a block [81]. This implementation of the dynamic load-balancing technique takes advantage of the atomic and one-sided operations in the GA toolkit (see Figure 18). The GA one-sided operations eliminate explicit synchronization between the processor that executes a task and the processor that has the relevant data. Atomic operations reduce the communication overhead in the traditional message-passing implementations of dynamic load balancing based on the master-slave strategy. This master-slave strategy has associated scalability issues because with the increased number of processors, management of the task list by a single master processor becomes a bottleneck. Hierarchical master-slave implementations (with multiple masters) [82] address that part of the problem; however, they introduce synchronization between multiple masters that degrades performance. Moreover, the message-passing implementation of this strategy can be quite complex. On the other hand, the implementation of dynamic load balancing using GA atomics (*fetch-and-increment* operation) involves only a couple of lines of code, while the overall performance of the simulation is competitive with the MPI-1 version [81].

```
Get task (i.e., block info) to be computed ( atomic fetch-and-add)
Issue nonblocking get call for the first block
do (until last block/task)
   determine what the next block/task is
   issue nonblocking get call for the next block
   wait for previously issued get call
   compute Function-Gradient
            (overlapping communication with computation.
          i.e., receiving next block while computing previous block)
   accumulate function and gradient into respective Arrays
done
```

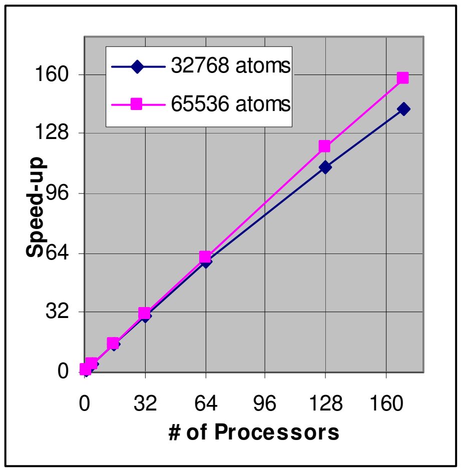


Figure 18: Function gradient evaluation using GA (left). Speedup in the Lennard-Jones potential energy optimization for 32,768 and 65,536 atoms (right).

The experimental results of the molecular dynamics benchmark on a Linux cluster with Myrinet indicate that using GA resulted in improved application performance over message-passing, see Figure 19 [81]. This benchmark problem scales well when the number of processors and/or the problem size is increased, thus proving the solution is cost-optimal. In best cases, the performance improvement over MPI is greater than 40%.

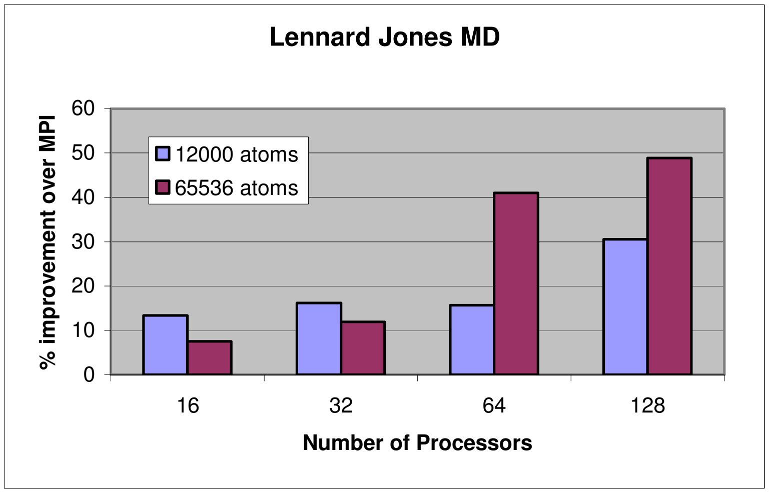


Figure 19: Performance improvement in the molecular dynamics simulation involving 12000 and 65536 atoms.

6.2 Lattice Boltzmann Simulation

A scalar lattice Boltzmann code was converted to use the Global Array libraries [53]. The lattice Boltzmann algorithm is a method for simulating hydrodynamic flows based on a discretized version of the Boltzmann equation and is distinguished by its simplicity and stability[83]. The lattice Boltzmann algorithm is typically implemented on a regular square or cubic lattice (other lattices, such as the hexagonal lattice [84], are occasionally used) and is composed of two basic steps. The first is an equilibration step that can be completed at each lattice site by using only data located at that site, the second is a streaming step that requires data from all adjacent sites (depending on the particular implementation, this can include corner and edge sites). The streaming step requires communication because sites corresponding to the boundaries of the locally held data will need data from other processors. This is accomplished by padding the locally held data with ghost cells and using the GA update operation to refresh the data in these ghost cell regions at each time step. A graph of speedup versus processors is shown below in Figure 20 for a simulation on a 1024x1024 lattice on an IBM SP. The timings show good speedups until quite large numbers of processors.

An earlier version of this code was created that just used onesided put/get calls to copy a suitably padded piece of the global lattice to a local buffer. The update was then performed and the result (minus the padded lattice points) was copied back to the global array. If present, periodic boundary conditions were handled using the periodic versions of the put/get operations. This approach is also quite easy to implement, but has some disadvantages relative to ghost cells. First, more memory is required since the lattice must effectively be duplicated (once in the global array and again in the local buffers) and second, no advantage is taken of the potential for optimizing the communication involved in updating the boundary data.

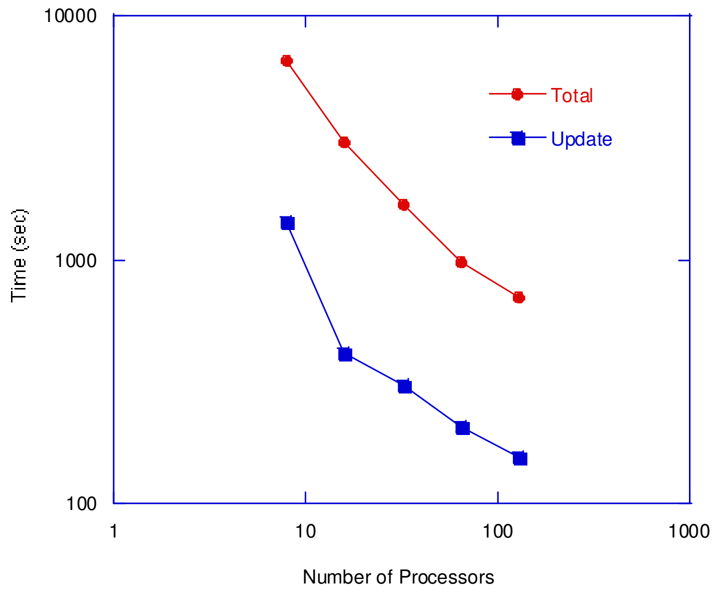


Figure 20: Timings for a lattice Boltzmann simulation on a 1024x1024 lattice on an IBM SP using ghost cells

6.3 Electronic structure

As already mentioned, developers of electronic structure codes have elected the Global Arrays toolkit as a de facto standard as far as communication libraries are concerned. Some of the most widely used electronic structure codes make use of GA: NWChem [85], Columbus [86], MOLPRO [87], MOLCAS [88], QChem [89] and GAMESS-UK [90]. Developers of GAMESS-US [91] have developed their own distributed memory management layer that implements a subset of GA functionality. Another scalable chemistry code MPQC [92], has adopted ARMCI. Parallelization of methods implemented in these codes involves the distribution of dense matrices among processing elements; if N is defined as the number of basis functions used, methods like Hartree-Fock (HF) [93, 94] or Density-Functional Theory (DFT) [95, 96] make use of matrices of size N^2, whereas correlated methods such as MP2 [97] or Coupled Cluster [98] use quantities whose size scales as the fourth of higher power of N (leading to larger storage requirements). Typically, N is also proportional to the size of the system, so larger molecular systems lead to rapid increases in both memory and computational requirements. This makes it essential to distribute the data associated with each of the matrices. The task based nature of the algorithms also implies that there is extensive communication involved in copying back and forth between the distributed arrays and local buffers.

As an example of the performance of GA for these theoretical methods, some benchmark numbers are reported for the DFT and MP2 methods (Figures 21 and 22). While the DFT benchmark scaling requires low interconnect latency, the MP2 runs necessitate high interconnect bandwidth and high performing disk I/O; therefore, the lower latency of Elan3 vs. Elan4 and of Infiniband vs. Myrinet can be noticed in Figures 21 and 22.

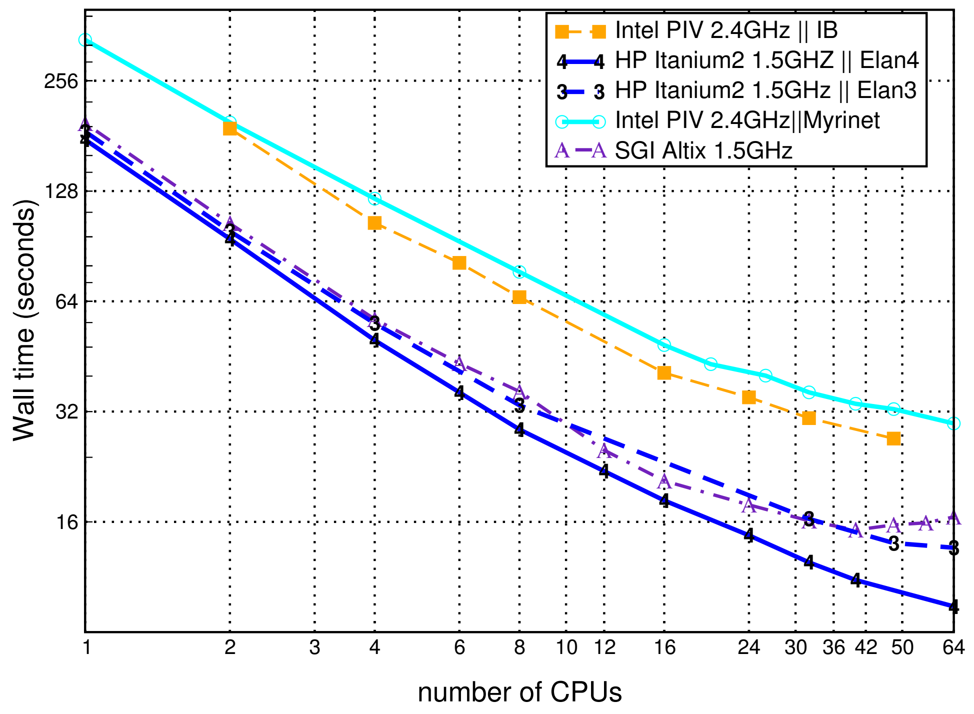


Figure 21: DFT LDA energy calculation on a $Si_8O_7H_{18}$ zeolite fragment, 347 basis functions

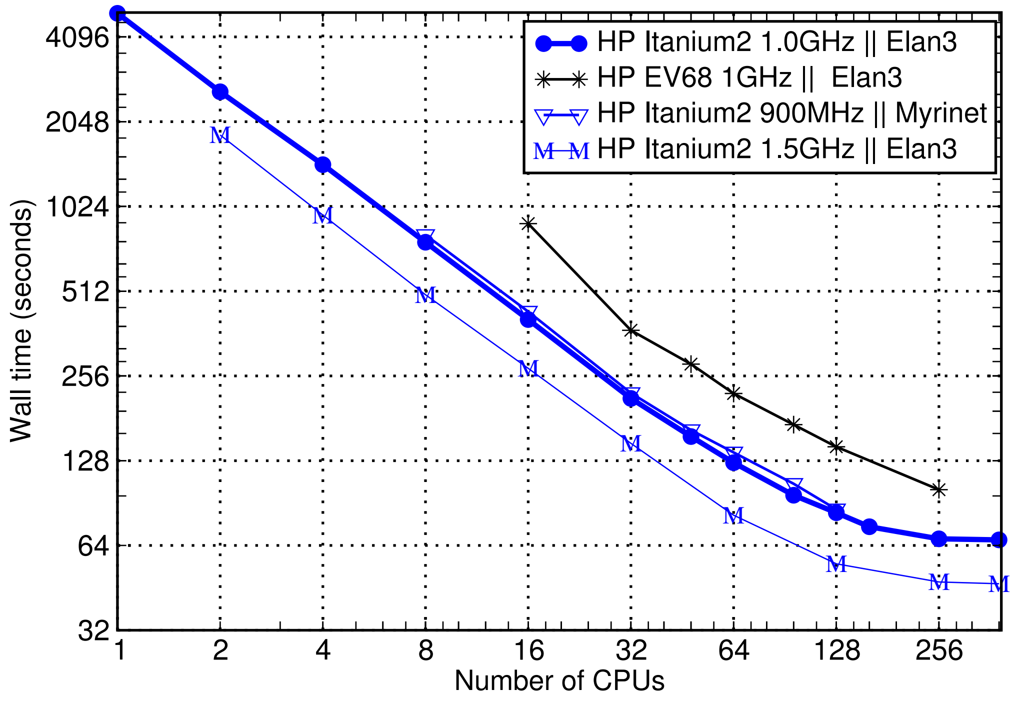


Figure 22: MP2 Energy + gradient calculation on a $(H_2O)_7$ cluster, 287 basis functions

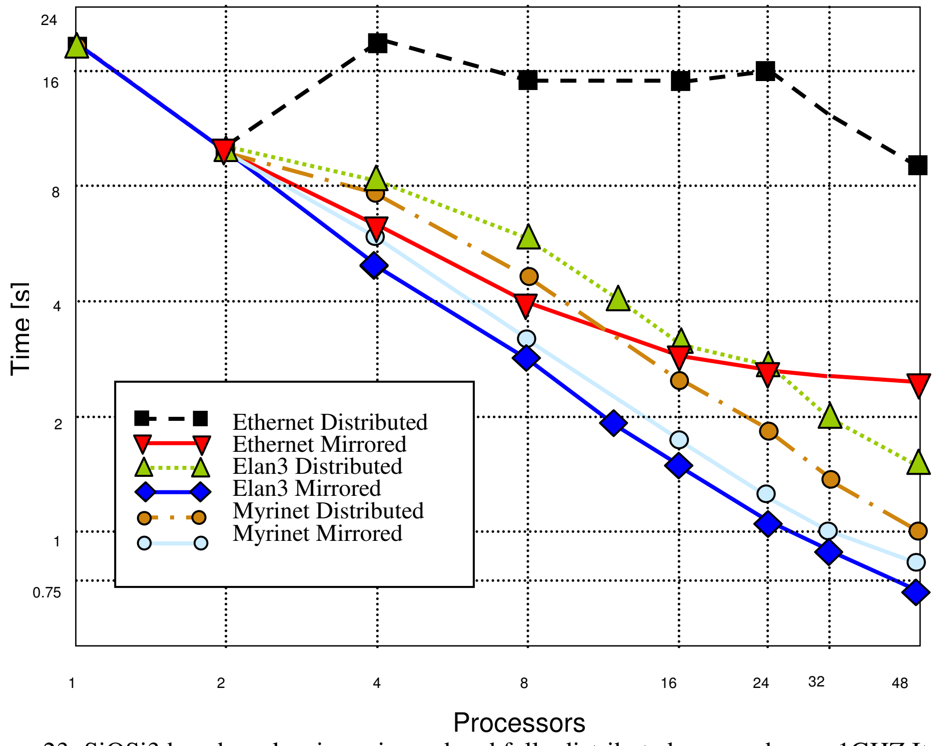


Figure 23: SiOSi3 benchmark using mirrored and fully distributed approach on a 1GHZ Itanium2 dual processor system with three different interconnects: Ethernet, Myrinet, or Elan3 (Quadrics)

6.3.1 Mirrored Arrays in Density Functional Theory

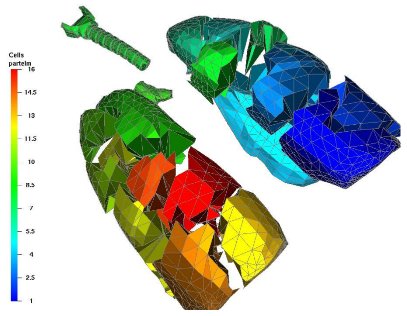
The mirrored arrays functionality has been implemented in the Gaussian function-based DFT module of NWChem [65, 85]. More precisely, it has been implemented in the evaluation of the matrix representation of Exchange-Correlation (XC) potential on a numerical grid [99]. Prior to the current work, this quantity was evaluated using a distributed data approach, where the main arrays were distributed among the processing elements by using the GA library.

This algorithm is very similar to the Hartree-Fock (a.k.a. SCF) algorithm, since both methods are characterized by the utilization of two main 2-dimensional arrays. The major steps of this algorithm require the generation of a density matrix from a parallel matrix multiply into a distributed global array representing the density matrix. Portions of this matrix must be copied to a local buffer where they are used to evaluate the density, which is then used to evaluate a portion of the Kohn-Sham matrix. This is copied back out to another distributed array. The construction of the Kohn-Sham matrix requires repeated use of the **`nga_get`**, and **`nga_acc`** methods and involves significant communication. Doing this portion of the calculation on mirrored arrays guarantees that all this communication occurs via shared memory and results in a significant increase in scalability. Results for a DFT calculation using the mirrored arrays on a standard chemical system are shown in Figure 23. The system is a 1GHZ Itanium2 dual processor system with three different interconnects: Ethernet, Myrinet, or Elan3 (Quadrics). The results show that scalability for the mirrored calculation is improved on all three networks over the fully distributed approach, with especially large improvements for the relatively slow Ethernet.

Surprisingly, Myrinet, which represents a network with intermediate performance, shows the smallest overall improvement on going from distributed to mirrored arrays. The expectation would be that Elan3 would show the least amount of improvement, since this is the fastest network and latency would not be expected to contribute as significantly to overall performance.

6.4 AMR-based Computational Physics

Grid generation is a fundamental part of any mesh-based computational physics problem. The NWGrid/NWPhys package integrates automated grid generation, time-dependent adaptivity, applied mathematics, and numerical analysis for hybrid grids on distributed parallel computing systems. This system transforms geometries into computable hybrid grids upon which computational physics problems can then be solved. NWGrid is used as the preprocessing grid generator [100] for NWPhys, setting up the grid, applying boundary and initial conditions, and defining the run-time parameters for the NWPhys calculations. NWGrid provides the grid partitioning functions and the time-dependent grid generation functions for adaptive mesh refinement (AMR), reconnection, smoothing, and remapping. The main tool used by NWGrid to perform partitioning is METIS [101]. To make use of METIS the multi-dimensional, hybrid, unstructured mesh is transformed into a two-dimensional graph, where nodes (or elements) form the diagonal entries of the graph and node-node (element-element) connections form the off-diagonal entries. NWPhys moves the grid according to forcing functions in non-linear physics drivers and NWGrid fixes it up based on grid topology and grid quality measures. Extensions of NWPhys include incorporating new packages for fluid-solid interactions, computational electromagnetics, particle transport, chemistry, and aerosol transport.



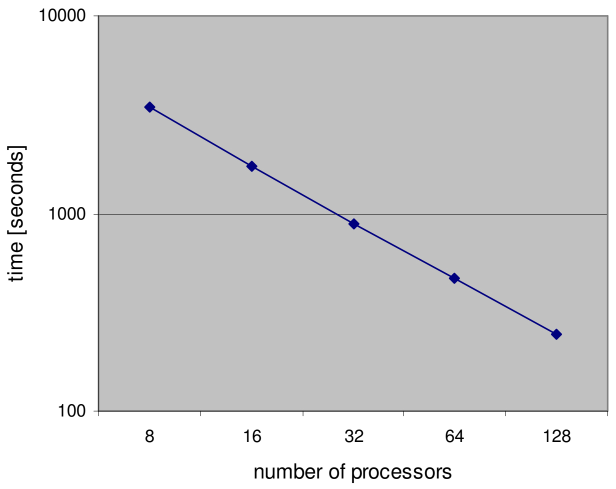


Figure 24: Human lung modeling using NWPhys/NWGrid – mesh discretization on the left for 16 procesoors and the parallel execution timings on a Linux cluster on the right

# of processors	Time(sec)
1	1690
2	1974
4	2222
8	2293
16	2343
32	2355
64	2384
128	2390

Table 2: Timing Results for a problem with 10,000 elements per processor and 1320 cycles. The problem size increases proportionally to the number of processors.

All of this functionality relies heavily on one dimensional representation of the grid data and operators defined on the grid. The package is implemented on top of GA, and makes extensive use of the operations supporting sparse data management, described above in Section 5.2. Figure 24 demonstrates performance of human lung modeling on a Linux cluster, indicating excellent scaling for this application. This problem involves one million grid elements and the simulation involved 360 cycles. The scaling of the absolute time and grind-time (time/cycle/element) is approximately linear, mainly because of the (near) optimal partitioning of the data and work per processor. Table 2 shows the timing results of a problem that grows as the number of processors grows. The problem was defined to have 10,000 elements per processor. So, 32 processors had 320,000 elements and 64 processors had 640,000 elements. The scaling is relatively constant as the problem size and number of processors grow. 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 use optimal data partitioning algorithms should.

7. Conclusions

This paper gives an overview of the functionality and performance of the Global Arrays toolkit. GA was created to provide application programmers with an interface that allows them to distribute data while maintaining the type of global index space and programming syntax similar to what is available when programming on a single processor. The details of identifying data location and mapping indices can be left to the toolkit, thereby reducing programming effort and the possibility of error. For many problems, the overall volume of code that must be created to manage data movement and location is significantly reduced.

The Global Array toolkit has been designed from the start to support shared memory style communication, which offers numerous possibilities for further code optimizations beyond what are available in traditional message-passing models. The shared communication model of GA also maps closely to current hardware and the low level communication primitives on which most communication libraries are built. In GA, the shared memory model is supported by ARMCI, which is an explicitly one-sided communication library. The availability of non-blocking one-sided protocols provides additional mechanisms for increasing the scalability of parallel codes by allowing programmers to overlap communication with computation. By "pipelining" communication and computation, the overhead of transferring data from remote processors can be almost completely overlapped with calculations. This can substantially reduce the performance penalty associated with remote data access on large parallel systems.

The Global Array toolkit also offers many high-level functions traditionally associated with arrays, eliminating the need for programmers to write these functions themselves. Examples are standard vector operations such as dot products and matrix multiplication, scaling an array or initializing it to some value, and interfaces to other parallel libraries that can solve linear equations or perform matrix diagonalizations. Again, this drastically cuts down on the effort required from the application programmer and makes it less error prone.

The widespread availability and vendor support for MPI has lead to a corresponding assumption that the message-passing paradigm is the best way to implement parallel algorithms. However, practical experience suggests that even relatively simple operations involving the movement of data between processors can be difficult to program. The goal of the Global Array toolkit is to free the programmer from the low level management of communication and allow them to deal with their problems at the level at which they were originally formulated. At the same time, compatibility of GA with MPI enables the programmer to take advantage of the existing MPI software/libraries when available and appropriate. The variety of applications that have been implemented using Global Arrays attests to the attractiveness of using higher level abstractions to write parallel code.

Acknowledgements

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. It was supported by the DoE-2000 ACTS project, Center for Programming Models for Scalable Parallel Computing, both sponsored by the Mathematical, Information, and Computational Science Division of DOE's Office of Computational and Technology Research, and the Environmental Molecular Sciences Laboratory. PNNL is operated by Battelle for the U.S. DOE under Contract DE-AC06-76RLO-1830.The Global Arrays toolkit would not exist without the invaluable contributions from computational scientists who provided requirements, feedback, and encouragement for our efforts, and in some cases directly became involved in the toolkit development.

References

[1] H. Shan and J. P. Singh, "A Comparison of Three Programming Models for Adaptive Applications on the Origin2000," in proceedings of Supercomputing, 2000.

[2] 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.

[3] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *Journal of Supercomputing*, vol. 10, pp. 169-189, 1996.

[4] J. Nieplocha, R. J. Harrison, M. Krishnan, B. Palmer, and V. Tipparaju, "Combining shared and distributed memory models: Evolution and recent advancements of the Global Array Toolkit," in proceedings of POHLL'2002 workshop of ICS-2002, NYC, 2002.

[5] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," in proceedings of Operating Systems Design and Implementation Symposium, 1996.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," Center for Computing Sciences CCS-TR-99-157, 1999.

[7] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," *Concurrency Practice and Experience*, vol. 10, pp. 825-836, 1998.

[8] R. W. Numrich and J. K. Reid, "Co-Array Fortran for parallel programming," *ACM Fortran Forum*, vol. 17, pp. 1-31, 1998.

[9] "High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.0," in *Scientific Programming*, vol. 2, 1993.

[10] L. Snyder, *A programmer's guide to ZPL*: MIT Press, 1999.

[11] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming on MIMD Computers*: The MIT Press, 1991.

[12] J. Nieplocha, R. Harrison, J., and I. Foster, "Explicit Management of Memory Hierarchy," *Advances in High Performance Computing*, pp. 185-200, 1996.

[13] ACTS -- Advanced Computational Testing and Simulation. http://www-unix.mcs.anl.gov/DOE2000/acts.html.

[14] The DOE ACTS Collection. http://acts.nersc.gov/.

[15] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, "Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, pp. 1-17, 1990.

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

[17] H. Dachsel, J. Nieplocha, and R. Harrison, J., "An out-of-core implementation of the COLUMBUS massively-parallel multireference configuration interaction program," in proceedings of High Performance Networking and Computing Conference, SC98, 1998.

[18] L. S. 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: A Linear Algebra Library for Message-Passing Computers," in proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, 1997.

[19] R. A. VanDeGeijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency-Practice and Experience*, vol. 9, pp. 255-274, 1997.

[20] S. Benson, L. McInnes, and J. J. Moré. Toolkit for Advanced Optimization (TAO) Web page. http: //www.mcs.anl.gov/tao.

[21] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and Y. Zhang, "Exploiting Processor Groups to Extend Scalability of the GA Shared Memory Programming Model," in proceedings of ACM Computing Frontiers, Italy, 2005.

[22] M. Krishnan, Y. Alexeev, T. L. Windus, and J. Nieplocha, "Multilevel Parallelism in Computational Chemistry using Common Component Architecture and Global Arrays," in proceedings of Supercomputing, Seattle, WA, USA, 2005.

[23] MPI-2. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. www.mpi-forum.org.

[24] R. Bariuso and A. Knies, *SHMEM's User's Guide*: Cray Research, Inc., 1994.

[25] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and experience with LAPI-a new high-performance communication library for the IBM RS/6000 SP," in proceedings of International Parallel Processing Symposium IPPS/SPDP, 1998.

[26] R. D. Loft, S. J. Thomas, and J. M. Dennis, "Terascale spectral element dynamical core for atmospheric general circulation models," in proceedings of 2001 ACM/IEEE conference on Supercomputing (CDROM), Denver, Colorado, 2001.

[27] D. S. Henty, "Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling," in proceedings of Supercomputing, 2000.

[28] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, "Evaluating the performance of software distributed shared memory as a target for parallelizing compilers," in proceedings of 1997 11th International Parallel Processing Symposium, IPPS 97, Apr 1-5 1997, Geneva, Switz, 1997.

[29] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "Midway distributed shared memory system," in proceedings of 38th Annual IEEE Computer Society International Computer Conference - COMPCON SPRING '93, Feb 22-26 1993, San Francisco, CA, USA, 1993.

[30] V. W. Freeh and G. R. Andrews, "Dynamically controlling false sharing in distributed shared memory," in proceedings of 1996 5th IEEE International Symposium on High Performance Distributed Computing, Aug 6-9 1996, Syracuse, NY, USA, 1996.

[31] A. Basumallik, S.-J. Min, and R. Eigenmann, "Towards OpenMP execution on software distributed shared memory systems," in proceedings of Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI'02), 2002.

[32] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "Cache performance and optimizations of blocked algorithms," in proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Apr 8-11 1991, Santa Clara, CA, USA, 1991.

[33] J. Nieplocha, J. Ju, M. Krishnan, B. Palmer, and V. Tipparaju, "The Global Arrays User's Manual," Pacific Northwest National Laboratory PNNL-13130, 2002.

[34] C. Scheurich and M. Dubois, "Correct memory operation of cache-based multiprocessors," in proceedings of 14th annual international symposium on Computer architecture, Pittsburgh, Pennsylvania, United States, 1987.

[35] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in proceedings of 13th annual international symposium on Computer architecture, Tokyo, Japan, 1986.

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

[37] M. Krishnan and J. Nieplocha, "SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems," in proceedings of Parallel and Distributed Processing Symposium, 2004.

[38] M. Krishnan and J. Nieplocha, "Optimizing Parallel Multiplication Operation for Rectangular and Transposed Matrices," in proceedings of 10th IEEE International Conference on Parallel and Distributed Systems (ICPADS'04). 2004.

[39] S. Benson, M. Krishnan, L. McInnes, J. Nieplocha, and J. Sarich, "Using the GA and TAO toolkits for solving large-scale optimization problems on parallel computers,"

Trans. on Mathematical Software, submitted to ACTS Collection special issue, Preprint ANL/MCS-P1084-0903, 2003.

[40] S. Balay. PETSc home page. http://www.mcs.anl.gov/petsc.

[41] CUMULVS. CUMULVS Home Page. http://www.csm.ornl.gov/cs/cumulvs.html.

[42] PeIGS. PeIGS Home Page. http://www.emsl.pnl.gov/docs/nwchem/doc/peigs/docs/peigs3.html.

[43] T. Dahlgren, T. Epperly, and G. Kumfert, "Babel/SIDL Design-by-Contract: Status," Lawrence Livermore National Laboratory UCRLPRES-152674, 2003.

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

[45] K. Parzyszek, J. Nieplocha, and R. A. Kendall, "Generalized Portable SHMEM Library for High Performance Computing," in proceedings of IASTED Parallel and Distributed Computing and Systems, Las Vegas, Nevada, 2000.

[46] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey, "Co-Array Fortran Performance and Potential: An NPB Experimental Study," in proceedings of 16th International Workshop on Languages and Compilers for Parallel Computing, 2003.

[47] B. Carpenter, "Adlib: A distributed array library to support HPF translation," in proceedings of 5th International Workshop on Compilers for Parallel Computers, University of Malaga, Malaga, Spain, 1995.

[48] J. Nieplocha, V. Tipparaju, A. Saify, and D. K. Panda, "Protocols and strategies for optimizing performance of remote memory operations on clusters," in proceedings of Communication Architecture for Clusters (CAC'02) Workshop, held in conjunction with IPDPS '02, 2002.

[49] J. Nieplocha, E. Apra, J. Ju, and V. Tipparaju, "One-Sided Communication on Clusters with Myrinet," *Cluster Computing*, vol. 6, pp. 115-124, 2003.

[50] V. Tipparaju, G. Santhmaraman, J. Nieplocha, and D. K. Panda, "Host-assised zero-copy remote memory access communication on Infiniband," in proceedings of International Parallel and Distributed Computing Symposium (IPDPS), Santa Fe, NM, USA, 2004.

[51] J. Nieplocha, J. L. Ju, and T. P. Straatsma, "A multiprotocol communication support for the global address space programming model on the IBM SP," in *Euro-Par 2000 Parallel Processing, Proceedings*, vol. 1900, *Lecture Notes in Computer Science*, 2000, pp. 718-728.

[52] R. H. Nobes, A. P. Rendell, and J. Nieplocha, "Computational chemistry on Fujitsu vector-parallel processors: Hardware and programming environment," *Parallel Computing*, vol. 26, pp. 869-886, 2000.

[53] B. Palmer and J. Nieplocha, "Efficient Algorithms for Ghost Cell Updates on Two Classes of MPP Architectures," in proceedings of Parallel and Distributed Computing and Systems (PDCS 2002), Cambridge, USA, 2002.

[54] J. A. Crotinger, J. Cummings, S. Haney, W. Humphrey, S. Karmesian, J. Reynders, S. Smith, and T. J. Williams, "Generic Programming in POOMA and PETE," *Programming Lecture Notes in Computational Science*, vol. 1766, pp. 218, 2000.

[55] S. Baden, P. Collela, D. Shalit, and B. Van Straalen, "Abstract Kelp," in proceedings of International Conference on Computational Science, San Francisco, CA, 2001.

[56] D. L. Brown, W. D. Henshaw, and D. J. Quinlan, "Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids," in proceedings of SIAM Conference on Object-Oriented Methods for Scientific Computing, 1999.

[57] C. Douglas, J. Hu, J. Ray, D. Thorne, and R. Tuminaro, "Fast, Adaptively Refined Computational Elements in 3D," in proceedings of International Conference on Scientific Computing, 2002.

[58] S. Chatterjee, G. E. Blelloch, and M. Zagha, "Scan primitives for vector computers," in proceedings of Supercomputing, New York, New York, United States, 1990.

[59] G. E. Blelloch, M. A. Heroux, and M. Zagha, "Segmented Operations for Sparse Matrix Computation on Vector Multiprocessor," Carnegie Mellon University CMU-CS-93-173, 1993.

[60] NWGrid. NWGrid Home Page. http://www.emsl.pnl.gov/nwgrid.

[61] NWPhys. The NWPhys homepage. http://www.emsl.pnl.gov/nwphys.

[62] J. B. White and S. W. Bova, "Where's the overlap? Overlapping communication and computation in several popular mpi implementations," in proceedings of Third MPI Developers' and Users' Conference, 1999.

[63] B. Lawry, R. Wilson, A. B. Maccabe, and R. Brightwell, "COMB: A Portable Benchmark Suite for Assessing MPI Overlap," in proceedings of IEEE Cluster'02, 2002.

[64] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. P. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda, "Performance Comparison of MPI implementations over Infiniband, Myrinet and Quadrics," in proceedings of Int'l Conference on Supercomputing, (SC'03), 2003.

[65] B. Palmer, J. Nieplocha, and E. Apra, "Shared memory mirroring for reducing communication overhead on commodity networks," in proceedings of International Conference on Cluster Computing, 2003.

[66] J. Nieplocha and R. J. Harrison, "Shared memory NUMA programming on I-WAY," in proceedings of High Performance Distributed Computing, 1996.

[67] J. Nieplocha and R. J. Harrison, "Shared memory programming in metacomputing environments: The global array approach," *Journal of Supercomputing*, vol. 11, pp. 119-136, 1997.

[68] M. Gupta and E. Schonberg, "Static Analysis to Reduce Synchronization Costs of Data-Parallel Programs," in proceedings of ACM Symposium on Principles of Programming Languages (POPL), 1996.

[69] U. Legedza and W. E. Weihl, "Reducing synchronization overhead in parallel simulation," in proceedings of 10th Workshop on Parallel and Distributed Simulation, Philadelphia, Pennsylvania, 1996.

[70] M. O'Boyle and E. Stöhr, "Compile Time Barrier Synchronization Minimization," *IEEE Transactions on Parallel and Distributed System*, vol. 13, 2002.

[71] D. Buntinas, A. Saify, D. K. Panda, and J. Nieplocha, "Optimizing synchronization operations for remote memory communication systems," in proceedings of Parallel and Distributed Processing Symposium, 2003.

[72] J. Nieplocha, I. Foster, and R. A. Kendall, "ChemIO: High performance parallel I/O for computational chemistry applications," *International Journal of High Performance Computing Applications*, vol. 12, pp. 345-363, 1998.

[73] J. Nieplocha and I. Foster, "Disk resident arrays: an array-oriented I/O library for out-of-core computations," in proceedings of Frontiers of Massively Parallel Computing, 1996.

[74] Y. Chen, J. Nieplocha, I. Foster, and M. Winslett, "Optimizing collective I/O performance on parallel computers: a multisystem study," in proceedings of 11th International Conference on Supercomputing, Vienna, Austria, 1997.

[75] D. R. Jones, E. R. Jurrus, B. D. Moon, and K. A. Perrine, "Gigapixel-size Real-time Interactive Image Processing with Parallel Computers," in proceedings of Workshop on Parallel and Distributed Image Processing, Video Processing, and Multimedia, PDIVM 2003, IPDPS 2003 Workshops, Nice, France, 2003.

[76] CCA-DCWG. Comparison of distributed array descriptors (DAD) as proposed and implemented for SC01 demos. http://www.csm.ornl.gov/~bernhold/cca/data.

[77] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing," in proceedings of Eighth International Symposium on High Performance Distributed Computing, 1999.

[78] D. E. Bernholdt, J. Nieplocha, and P. Sadayappan, "Raising the Level of Programming Abstraction in Scalable Programming Models," in proceedings of HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004), Madrid, Spain, 2004.

[79] S. Plimpton and G. Heffelfinger, "Scalable parallel molecular dynamics on MIMD supercomputers," in proceedings of Scalable High Performance Computing Conference, 1992.

[80] C. P. Kruskal and A. weiss, "Allocating independent subtasks on parallel processors," *IEEE Trans. Softw. Eng.*, vol. 11, pp. 1001-1016, 1985.

[81] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, and D. Panda, "Exploiting non-blocking remote memory access communication in scientific benchmarks," in *High Performance Computing - HiPC*, vol. 2913, *Lecture Notes in Computer Science*, 2003, pp. 248-258.

[82] T. Matthey and J. A. Izaguirre, "ProtoMol: A Molecular Dynamics Framework with Incremental Parallelization," in proceedings of Tenth SIAM Conf. on Parallel Processing for Scientific Computing (PP01), 2001.

[83] U. Frisch, d'Humieres, D., Hasslacher, B., Lallemand, P., Pomeau, Y., and Rivet, J-P., "Lattice Gas Hydrodynamics in Two and Three Dimensions," *Complex Systems*, vol. 1, pp. 649, 1987.

[84] U. Frisch, Hasslacher, P., and Pomeau, Y., "Lattice-Gas Automata for the Navier-Stokes Equation," *Phys. Rev. Lett.*, vol. 56, pp. 1505, 1986.

[85] R. A. Kendall, E. Apra, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. L. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, "High performance computational chemistry: An overview of NWChem a distributed parallel application," *Computer Physics Communications*, vol. 128, pp. 260-283, 2000.

[86] H. Dachsel, H. Lischka, R. Shepard, J. Nieplocha, and R. J. Harrison, "A massively parallel multireference configuration interaction program: The parallel COLUMBUS program," *Journal of Computational Chemistry*, vol. 18, pp. 430-448, 1997.

[87] A. J. Dobbyn, P. J. Knowles, and R. J. Harrison, "Parallel internally contracted multireference configuration interaction," *Journal of Computational Chemistry*, vol. 19, pp. 1215-1228, 1998.

[88] G. Karlstrom, R. Lindh, P. A. Malmqvist, B. O. Roos, U. Ryde, V. Veryazov, P. O. Widmark, M. Cossi, B. Schimmelpfennig, P. Neogrady, and L. Seijo, "MOLCAS: a program package for computational chemistry," *Computational Materials Science*, vol. 28, pp. 222-239, 2003.

[89] J. Kong, C. A. White, A. I. Krylov, D. Sherrill, R. D. Adamson, T. R. Furlani, M. S. Lee, A. M. Lee, S. R. Gwaltney, T. R. Adams, C. Ochsenfeld, A. T. B. Gilbert, G. S. Kedziora, V. A. Rassolov, D. R. Maurice, N. Nair, Y. H. Shao, N. A. Besley, P. E. Maslen, J. P. Dombroski, H. Daschel, W. M. Zhang, P. P. Korambath, J. Baker, E. F. C. Byrd, T. Van Voorhis, M. Oumi, S. Hirata, C. P. Hsu, N. Ishikawa, J. Florian, A. Warshel, B. G. Johnson, P. M. W. Gill, M. Head-Gordon, and J. A. Pople, "Q-chem 2.0: A high-performance ab initio electronic structure program package," *Journal of Computational Chemistry*, vol. 21, pp. 1532-1548, 2000.

[90] M. F. Guest, J. H. V. Lenthe, J. Kendrick, K. Schoffel, and P. Sherwood, "GAMESS - UK: Version 6.3."

[91] G. D. Fletcher, M. W. Schmidt, B. M. Bode, and M. S. Gordon, "The Distributed Data Interface in GAMESS," *Computer Physics Communications*, vol. 128, pp. 190-200, 2000.

[92] C. L. Janssen, E. T. Seidl, and M. E. Colvin, "Object-oriented implementation of a Parallel Ab-initio Program," *Parallel Computing in Computational Chemistry ACS Symposium Series, American Chemical Society, Washington,DC, 1995*, vol. 592, pp. 47, 1995.

[93] V. A. Fock, "Naherungsmethode zur Losung des quantenmechanischen Mehrkorperproblems," *Zeit. für Phys*, vol. 61, pp. 126, 1930.

[94] D. R. Hartree, "The Wave Mechanics of an Atom in a Non-Coulomb Central Field," *Proc. Camb. Phil. Soc.*, vol. 24, pp. 89, 1928.

[95] P. Hohenberg and W. Kohn, "Inhomogenoeous electron gas," *Phys. Rev.*, vol. 136, 1964.

[96] S. Kohn and L. Sham, "Self-consistent equations including exchange and correlation effects," *Phys. Rev.*, vol. 140, 1965.

[97] C. Møller and S. Plesset, "Note on an Approximation Treatment for Many-Electron Systems," *Phys. Rev*, vol. 46, 1934.

[98] J. Cìzek, "On the use of the cluster expansion and the technique of diagrams in calculations of the correlations effects in atoms and molecules," *Adv. Chem. Phys*, vol. 14, pp. 35, 1969.

[99] A. D. Becke, "A Multicenter Numerical-Integration Scheme for Polyatomic-Molecules," *Journal of Chemical Physics*, vol. 88, pp. 2547-2553, 1988.

[100] H. E. Trease and e. al., "Grid Generation Tools for Performing Feature Extraction, Image Reconstruction, and Mesh Generation on Digital Volume Image Data for Computational Biology Applications," in proceedings of 8th International Conference On Grid Generation and Scientific Applications, Honolulu, Hawaii, 2002.

[101] metis. METIS. http://www-users.cs.umn.edu/~karypis/metis.

Appendix A

Figure A2 illustrates that programming based on GA is relatively simple. For the parallel transposition of 1-dimensional array (Figure A1) thanks to the high-level interfaces for array management provided by GA, the code size reduces by a factor of three when compared to the MPI version. In the MPI version, each task has to identify where (tasks ranks) to send the data. Say in Figure A1, task P_0 owns first 50 elements (i.e. 0-49) of the distributed array and after transposition, the data owned by task P_0 is moved to P_2 and P_3 (P_0 sends elements 0-46 to task P_3 and 47-49 to task P_2). Similarly P_3 sends the last 47 elements to P_0, and P_2 sends its last 3 elements to P_0. Thus the programmer has to identify how many receives (MPI_Recv) each task has to post, to obtain the corresponding data. Each task should also send the global indices of the data to the receiving task. The MPI code would become more complicated to handle two-dimensional arrays. In case of GA, the programmer would only have to specify the indices of the 2-dimensional array block to be transposed.

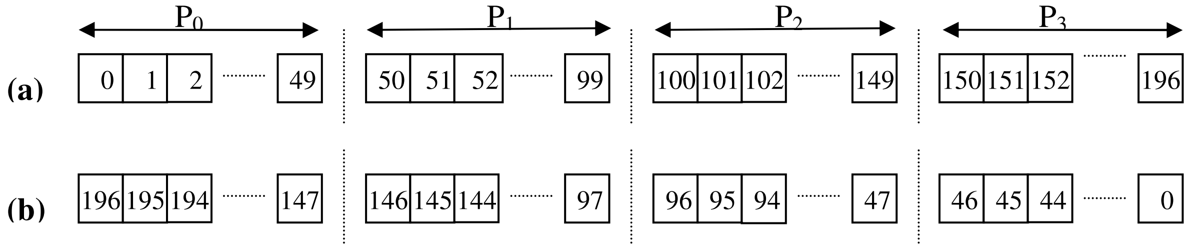


Figure A1: Example of parallel transposing of 1-d array (197 elements) on 4 processors. (a) Distributed integer array with 50 elements each, except processor P_3 with 47 elements. Array values initialized from 0 to 196. (b) Final result.

```
/************  GA VERSION  *************/
#define   NDIM         1
#define   TOTALELEMS   197

int main(int argc, char **argv) {
    int dims,chunk,nprocs,me,i,lo,hi,lo2,hi2,ld;
    int g_a, g_b, a[TOTALELEMS],b[TOTALELEMS];

    GA_Initialize();
    me     = GA_Nodeid();
    nprocs = GA_Nnodes();
    dims   = nprocs*TOTALELEMS;
    chunk  = ld = TOTALELEMS;

    /* create a global array  */
    g_a = GA_Create(C_INT, NDIM, dims, "array A", chunk);
    g_b = GA_Duplicate(g_a, "array B");

    /* INITIALIZE DATA IN GA */
    GA_Enumerate(g_a, 0);

    GA_Distribution(g_a, me, lo, hi);
    GA_Get(g_a, lo, hi, a, ld);
    // INVERT DATA LOCALLY
    for (i=0; i<nelem; i++)    b[i] = a[nelem-1-i];
    // INVERT DATA GLOBALLY
    lo2 = dims - hi -1;
    hi2 = dims - lo -1;
    GA_Put(g_a,lo2,hi2,b,ld);

    GA_Terminate();
}
```

```
/************  MPI VERSION  *************/
#define TOTALELEMS 197
#define MAXPROC 128
#define MIN(a,b) ((a) < (b) ? (a) : (b))

int main(int argc, char **argv) {
    int *a, *b, me, nprocs, i, j, np=0,start=-1,lo2, hi2;
    int global_idx,local_idx, rem, local_count, position,
    bytes;
    char *send_buf, **recv_buf;
    int lo[MAXPROC],hi[MAXPROC], count[MAXPROC];
    int nrecv[MAXPROC], nrecv2[MAXPROC], to[MAXPROC],
    elems_per_proc[MAXPROC];
    MPI_Status status;
    MPI_Request request[MAXPROC];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    /* distributed array, where each process has
     elems_per_proc elements */
    rem = TOTALELEMS;
    elems_per_proc[0] = MIN(rem, TOTALELEMS/nprocs+1);
    lo[0]=0; hi[0]=elems_per_proc[0]-1;
    rem -= elems_per_proc[0];
    for(i=1; i<nprocs; i++) {
       elems_per_proc[i] = MIN(rem, TOTALELEMS/nprocs+1);
       lo[i]=hi[i-1]+1; hi[i]=lo[i]+elems_per_proc[i]-1;
       rem -= elems_per_proc[i];
    }
    /* initialize */
    a = (int*)malloc(sizeof(int)*elems_per_proc[me]);
    b = (int*)malloc(sizeof(int)*elems_per_proc[me]);
    for(i=0; i<elems_per_proc[me]; i++)
       a[i]=elems_per_proc[0]*me+i;
```

```
   for(i=0; i<elems_per_proc[me]; i++)  b[i]=-1;
   for(i=0; i<nprocs; i++) nrecv[i]=0;

   /* INVERT DATA LOCALLY */
   for(i=0; i<elems_per_proc[me]; i++)
      b[i]=a[elems_per_proc[me]-1-i];

   /* identify where to send the data */
   lo2 = TOTALELEMS-1-hi[me];
   hi2 = TOTALELEMS-1-lo[me];

   /* find process(es) rank, where data has to be sent */
   for(i=0; i<nprocs; i++)
      if(lo2>=lo[i] && lo2<=hi[i])
   i=start; np=0;
   do {
      nrecv[i]=1;
      to[np]=i;
      ++np;
   }while (hi2>hi[i++]);

   /* count # of elems to be sent for each destination
  processes */
   count[0] = hi[start]-lo2+1;
   if(np>0) {
      for(i=start+1,j=1; i<start+np-1; i++,j++)
         count[j]=hi[i]-lo[i]+1;
      count[np-1] = hi2-lo[start+np-1]+1;
   }

   /* broadcast the number of recv's for each process */
   MPI_Allreduce(nrecv, nrecv2, nprocs, MPI_INT, MPI_SUM,
   MPI_COMM_WORLD);

   /* INVERT DATA GLOBALLY */
   global_idx=lo2; local_idx=0;
   bytes = sizeof(int)*(elems_per_proc[me]+1);
   send_buf = (char*)malloc(bytes);
   recv_buf = (char**)malloc(nrecv2[me]*sizeof(char*));

   /* Post the receive's */
   for(i=0; i<nrecv2[me]; i++) {
      recv_buf[i] = (char*)malloc(bytes);

      MPI_Irecv(recv_buf[i], bytes, MPI_PACKED,
      MPI_ANY_SOURCE, 555, MPI_COMM_WORLD, &request[i]);
   }

   for(i=0; i<np; i++) { /*pack global idx actual data*/
      position = 0;
      MPI_Pack(&global_idx, 1, MPI_INT, send_buf, bytes,
        &position, MPI_COMM_WORLD);
      MPI_Pack(&b[local_idx],  count[i], MPI_INT,
      send_buf, bytes, &position, MPI_COMM_WORLD);
      MPI_Send(send_buf, position, MPI_PACKED, to[i],
      555, MPI_COMM_WORLD);
      local_idx += count[i];
      global_idx = lo2+count[i];
   }

   for(i=0; i<nrecv2[me]; i++) {
      MPI_Wait(&request[i], &status);
      MPI_Get_count(&status, MPI_INT, &local_count);
      position = 0;
      MPI_Unpack(recv_buf[i],      bytes,      &position,
      &global_idx, 1, MPI_INT, MPI_COMM_WORLD);
      local_idx = global_idx - me*elems_per_proc[0];
      MPI_Unpack(recv_buf[i], bytes, &position,
      &a[local_idx], local_count-1, MPI_INT,
      MPI_COMM_WORLD);
   }

   MPI_Finalize();
}
```

Figure A2: Parallel implementation of 1-dimensional array transpose: GA version shown on the left and MPI version on the right.