Global Arrays User's Manual

Jarek Nieplocha and Jialin Ju

Pacific Northwest National Laboratory

November 1999

This manual is intended for use with release 3.0 of Global Arrays

DISCLAIMER

ACKNOWLEDGMENT

This software and its documentation were produced with United States Government support under Contract Number DE-AC06-76RLO-1830 awarded by the United States Department of Energy. The United States Government retains a paid-up non-exclusive, irrevocable worldwide license to reproduce, prepare derivative works, perform publicly and display publicly by or for the US Government, including the right to distribute to other US Government contractors.

ABOUT THIS MANUAL

An *updated* version of this manual is available on-line at

http://www.emsl.pnl.gov:2080/docs/global/user.html

Additional information about the Global Arrays can be found at

http://www.emsl.pnl.gov:2080/docs/global, and

http://www.emsl.pnl.gov:2080/docs/global/Capi.html for C documentation, and
http://www.emsl.pnl.gov:2080/docs/global/Gaapi.html for Fortran documentation.

Most of the underlined words (primarily function names) correspond to hyperlinks in the HTML version of this document on the web. The web version is more current.

Contents

1. Introduction

1.1 Overview

The Global Arrays (GA) toolkit provides a shared memory style programming environment in the context of distributed array data structures (called "global arrays"). From the user perspective, a global array can be used as if it was stored in the shared memory. Details of the data distribution, addressing, and communication are encapsulated in the array objects. The information on the actual data distribution and locality can be obtained and taken advantage of whenever data locality is important.

The primary target architecture for which GA was developed are massively-parallel distributed memory or scalable shared memory systems. GA divides logically shared data structures into local and remote portions and it recognizes variable data transfer costs required to access the data. A "local" portion of the shared memory is assumed to be faster to access and the remainder ("remote" portion) is considered slower to access. GA is implemented as a library with C and Fortran-77 bindings. Therefore, explicit library calls are required to use the GA model in a parallel program. GA was designed to complement rather than substitute the message-passing model, and it allows the user to combine shared-memory and message-passing styles of programming in the same program. GA inherits an execution environment from a message-passing library (w.r.t. processes, file descriptors etc.) that started the parallel program.

A disk extension of the Global Array library is supported by its companion library called Disk Resident Arrays (DRA). DRA maintains array objects in secondary storage and allows transfer of data to/from global arrays.

1.2 Basic Functionality

The basic shared memory operations supported include *get*, *put*, *scatter* and *gather*. They are complemented by the **atomic** *read-and-increment*, *accumulate* (reduction operation that combines data in local memory with data in the shared memory location), and *lock* operations. However, these operations can only be used to access data in global arrays rather than arbitrary memory location. At least one global array has to be created before data transfer operations can be used. These operations are truly one-sided/unilateral and will complete regardless of actions taken by the remote process(es) that own(s) the referenced data. In particular GA does not offer or require a polling operation used in some other tools to assure communication progress.

A programmer in the GA program has full control over the distribution of global arrays. Both regular and irregular distributions are supported, see Section 3 for details.

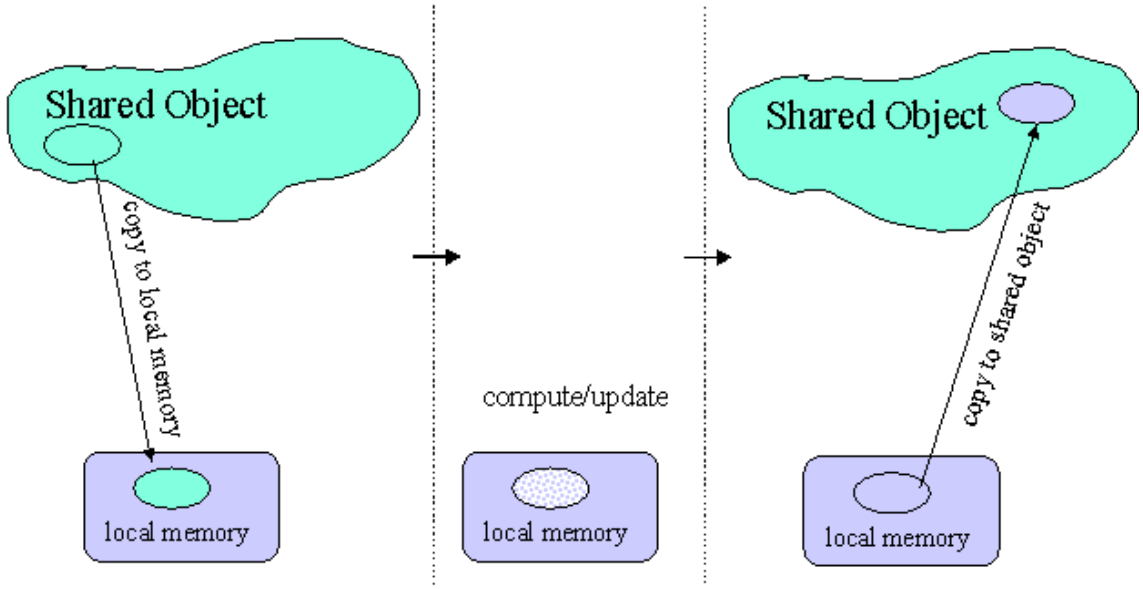
The GA data transfer operations use an array index interface rather than addresses of shared data. Unlike other systems based on global address space that support remote memory (*put/get*) operations, GA does not require the user to specify the target process/es where the referenced shared data resides. The higher level array oriented API (application programming interface) makes GA easier to use, at the same time without compromising data locality control. The library internally performs global array index-to-address translation and then transfers data between appropriate processes. If necessary, the programmer is always able to inquire:

- where and an element or array section is located, and
- which process or processes own data in the specified array section.

The GA toolkit supports three data types: integer, double precision, and double complex. The supported array dimensions range from one to seven. This limit follows the Fortran convention. The library can be reconfigured to support more than 7-dimensions but only through the C interface.

1.3 Programming Model

The GA model of computations is based on an explicit remote memory copy: The remote portion of shared data has to be copied into the local memory area of a process before it can be used in computations. However, the "local" portion of shared data can always be accessed directly thus avoiding the memory copy.



The data distribution and locality control are provided to the programmer. The data locality information for the shared data is available. The library offers a set of operations for management of its data structures, one-sided data transfer operations, and supportive operations for data locality control and queries. The GA shared memory consistency model is a result of a compromise between the ease of use and a portable performance. The load and store operations are guaranteed to be ordered with respect to each other only if they target overlapping memory locations. The store operations (*put, scatter*) and *accumulate* complete locally before returning i.e., the data in the user local buffer has been copied out but not necessarily completed at the remote side. The memory consistency is only guaranteed for:

- multiple read operations (as the data does not change) ,
- multiple accumulate operations (as addition is commutative), and
- multiple disjoint put operations (as there is only one writer for each element).

The application can manage consistency of its data structures in other cases by using *lock, barrier,* and *fence* operations available in the library.

1.4 Application Guidelines

These are some guidelines regarding suitability of the GA for applications.

When to use GA:

Algorithmic Considerations

- applications with dynamic and irregular communication patterns
- for calculations driven by dynamic load balancing
- need 1-sided access to shared data structures
- need high-level operations on distributed arrays for out-of-core array-based algorithms (GA + DRA)

Usability Considerations

- data locality important
- when coding in message passing becomes too complicated
- when portable performance is important
- need object orientation without the overhead of C++

When not to use GA:

Algorithmic Considerations

- for systolic or nearest neighbor communications
- when synchronization associated with cooperative point-to-point message passing is needed (e.g., Cholesky factorization)

Usability Considerations

- when interprocedural analysis and compiler parallelization is more effective
- existing language support is sufficient and robust compilers are available

2. Writing, Building and Running GA Programs

2.1 Platform and Library Dependencies

2.1.1 Supported Platforms

- IBM SP, CRAY T3E/J90, SGI Origin, Fujitsu VX/VPP
- Cluster of workstations: Solaris, IRIX, AIX, HPUX, Digital/True64 Unix, Linux
- Standalone uni- or multi-processor workstations or servers
- Standalone uni- or multi-processor Windows NT workstations or servers

Older versions of GA supported some additional (now obsolete) platforms such as: IPSC, KSR, PARAGON, DELTA, CONVEX. They are not supported in the current (3.0) version because we do not have access to these systems. We recommend using GA 2.4 on these platforms.

Because of limited interest in heterogenous computing from the GA users, the Global Array library still does not support heterogeonous platforms. This capability can be added if required.

2.1.2 Selection of the message-passing Library

As explained in Section 3, GA works with either MPI or TCGMSG message-passing libraries. That means that GA applications can use either of these interfaces. Selection of the message-passing library takes place when GA is built. Since the TCGMSG library is small and compiles fast, it is included with the GA distribution package and built on Unix workstations by default so that the package can be built as fast and as conveniently to the user as possible. There are three possible configurations for running GA with the message-passing libraries:

- with TCGMSG
- with MPI and TCGMSG emulation library: TCGMSG-MPI, that implements functionality of TCGMSG using MPI. In this mode, the message passing library is initialized using a TCGMSG *PBEGIN(F)* call which internally references *MPI_Initialize*. To enable this mode, define the environmental variable *USE_MPI*.
- directly with MPI. In this mode, GA program should contain MPI initialization calls instead of PBEGIN(F).

For the MPI versions, the optional environmental variables MPI_LIB and MPI_INCLUDE are used to point to the location of the MPI library and include directories if they are not in the standard system location(s). GA programs are started with the mechanism that any other MPI programs use on the given platform.

The recent versions of MPICH (an MPI implementation from ANL/Missisipi State) keep the MPI header files in more than one directory and provide compiler wrappers that implicitly point to the appropriate header files. One can :

- use MPI_INCLUDE by expanding the string with another directory component prefixed with "-I" (you are passing include directory names as a part of compiler flags), or
- use mpicc and mpif77 to build GA right out of the box on UNIX workstations:

```
make FC=mpif77 CC=mpicc
```

One disadvantage of the second approach it that GA makefile might be not able to determine which compiler (e.g., GNU or PGI) is called underneath by the MPICH compiler wrappers. Since different compilers provide different Fortran/C interface, this could cause the package to build incorrectly (test programs fail or do not compile).

One Window NTs, the current version of GA was tested with WMPI, an NT implementation derived from MPICH in Portugal.

2.1.3 Dependencies on other software

In addition to the message-passing library, GA requires:

- MA (Memory Allocator), a library for managment of local memory;
- ARMCI, a one-sided communication library that GA uses as its run-time system;
- BLAS library is required for the eigensolver and ga_dgemm;
- LAPACK library is required for the eigensolver (a subset is included with GA, which is built into *liblinalg.a*);

GA may also depend on other software depending on the functions being used.

- GA *eigensolver*, *ga_diag*, is a wrapper for the eigensolver from the PEIGS library; (Please contact George Fann <gi_fann@pnl.gov> about PEIGS)
- SCALAPACK, PBBLAS, and BLACS libraries are required for *ga_lu_solve*, *ga_cholesky*, *ga_llt_solve*, *ga_spd_invert*, *ga_solve*. If these libraries are not installed, the named operations will not be available.
- If one would like to generate trace information for GA calls, an additional library *libtrace.a* is required, and the -DGA_TRACE define flag should be specified for C and Fortran compilers.

2.2 Writing GA Programs

C programs that use Global Arrays should include files *`global.h', 'ga.h', `macdecls.h'*. Fortran programs should include the files *`mafdecls.fh', `global.fh'*.

The GA program should look like:

- When GA runs with MPI

```
      Fortran                    C

call mpi_init(..)          MPI_Init(..)     ! start MPI
call ga_initialize()       GA_Initialize()  ! start global arrays
status = ma_init(..)       MA_Init(..)      ! start memory allocator

.... do work               .... do work

call ga_terminate()        GA_Terminate()   ! tidy up global arrays
call mpi_finalize()        MPI_Finalize()   ! tidy up MPI
stop                                        ! exit program
```

- When GA runs with TCGMSG or TCGMSG-MPI

```
   Fortran                    C
call pbeginf()           PBEGIN_(..)      ! start TCGMSG
call ga_initialize()     GA_Initialize()  ! start global arrays
status = ma_init(..)     MA_Init(..)      ! start memory allocator

.... do work             .... do work

call ga_terminate()      GA_Terminate()   ! tidy up global arrays
call pend()              PEND_()          ! tidy up tcgmsg
stop                                      ! exit program
```

The *ma_init* call looks like :

```
status = ma_init(type, stack_size, heap_size)
```

and it basically just goes to the OS and gets *stack_size+heap_size* elements of size type. The amount of memory MA allocates need to be sufficient for storing global arrays on some platforms. Please refer to section 3.3.1 for the details and information on more advanced usage of MA in GA programs.

2.3 Building GA Programs

Use *GNU make* to build the GA library and application programs on Unix and Microsoft *nmake* on Windows. The structure of the available makefiles are

- `GNUmakefile`: Unix makefile
- `MakeFile`: Windows NT makefile
- `Makefile.h`: definitions & include symbols
- `MakeFiles.h`: files to be compiled depending on the TARGET machine
- `Makelib.h`: specification of libraries to be linked with when test programs are built

The user needs to specify TARGET in the GNUmakefile or on the command line when calling make. The library and test programs should be built by calling make in the current directory. The appropriate value of VERSION will be determined automatically.

Valid TARGET is one of:

`SUN, SOLARIS, FUJITSU, SGI, SGI_N32, SGITFP, DECOSF, IBM, LAPI, CRAY-T3D, CRAY-T3E, HPUX, CONVEX-SPP, LINUX` on Unix systems.

or

`WIN32, CYGNUS` on Windows NT.

2.3.1 Unix Environment

To build GA with MPI, the user needs to define environmental variables *USE_MPI*, *MPI_LIB* and *MPI_INCLUDE* which should point to the location of the MPI library and include directories.

Example: using csh/tcsh (assume using MPICH installed in /usr/local on IBM workstation)

```
setenv USE_MPI y
setenv MPI_LOC /usr/local/mpich
setenv MPI_LIB $MPI_LOC/lib/rs6000/ch_shmem
setenv MPI_INCLUDE $MPI_LOC/include
```

Additionally, if the TCGMSG-MPI library is not needed, the make/environmental variable MSG_COMMS should be defined as `MSG_COMMS = MPI`.

Interface routines to ScaLAPACK are only available with MPI, and of course with ScaLAPACK. The user is required to define the environment variables *USE_SCALAPACK*, and the location of ScaLAPACK & Co. libraries in variable SCALAPACK.

Example: using csh/tcsh

```
setenv USE_SCALAPACK y
setenv SCALAPACK '-L/msrc/proj/scalapack/LIB/rs6000
         -lscalapack -lpblas -ltools -lblacsF77cinit -lblacs'
setenv USE_MPI y
```

Since there are certain interdependencies between blacs and blacsF77cinit, some system might require specification of -lblacs twice to fix the unresolved external symbols from these libs.

To build the library, type

`make` or `gmake`

To build an application based on GA, for example, the application's name is `app.c` (or `app.F`, `app.f`), type

`make app.x` or `gmake app.x`

Please refer to compiler flags in file `g/global/Makefile.h` to make sure that Fortran and C compiler flags are consistent with flags use to compile your application. This may be critical when Fortran compiler flags are used to change the default length of the integer datatype.

2.3.2 Windows NT

To buid GA on Windows NT, one needs to have an MS Power Fortran 4 or DEC Visual Fortran 5 or 6, and MS Visual C 4, 5, or 6. When commercial Windows compilers are not available, one can choose to use CYGNUS. Set the TARGET environment variable to CYGNUS, and build it as any other Unix box.

First of all, one needs to set environment variables (same as in Unix enviroment). GA needs to know where find the MPI include files and libraries. To do this, select the *Environment* tab under the Control Panel, then set the variables to point to the location of MPI, for example for WMPI on disk D:

```
set MPI_INCLUDE as d:\Wmpi\Include
set MPI_LIB as d:\Wmpi\Console
```

Make sure that the dynamic link libraries required by the particular implementation of MPI are copied to the appropriate location for the system DLLs. For WMPI, copy `VWMPI.dll` to `\winnt`.

In the top directory do,

```
nmake
```

The GA test.exe program can be built in the `g\global\testing` directory:

```
nmake test.exe
```

Current GA works on a single PC only using shared memory and NT threads with separate address spaces. The HPVM package offers GA in the NT cluster environment.

GA could be built on Windows 95. However, due to the DOS shell limitations, the top level NTmakefile will not work. Therefore, each library has to be made separately in its own directory. The environment variables referring to MPI can be hardcoded in the NT makefiles.

2.3.3 Writing and building new GA programs

For small programs contained in a single file, the most convenient approach is to put your file into `g/global/testing` directory. The existing GNU make suffix rules would build an executable with `".x"` suffix from the C or Fortran source files. Windows *nmake* is not as powerful as GNU make - you would need to modify the NT makefile.

This approach obviously is not preferred for large packages developed with GA. In that case you need to incorporate in your makefile:

- GA/MA/... include directory, `g/include`, where all public header files are copied in the process of building GA
- add references to `libglobal.a/global.lib`, and `libma.a/ma.lib` in `g/lib/$(TARGET)` and message-passing libraries
- follow compilation flags for the GA test programs in GNU and Windows makefiles

2.4 Running GA Programs

Assume the `app.x` had already been built. To run it,

- On MPPs, such as Cray T3E, or IBM SP

  Use appropriate system command to specify the number of processors, load and run the programs.

  Example: to run on four processors on the Cray T3E, use

  ```
  mpprun -n 4 app.x
  ```

- On shared memory systems and (network of) workstations (including linux cluster)

  If the `app.x` is built based on MPI, run the program the same way as any other MPI programs.

  Example: to run on four processes on SGI workstation, use

  ```
  mpirun -np 4 app.x, or
  app.x -np 4
  ```

  If app.x is built based on TCGMSG (not including, Fujitsu, Cray J90, and Windows, because there are no native ports of TCGMSG), to execute the program on Unix workstations/servers, one should use the `'parallel'` program (built in `tcgmsg/ipcv4.0`). After building the

application, a file called '`app.x.p`' would also be generated (If there is not such a file, make it: `make app.x.p`). This file can be edited to specify how many processors and tasks to use, and how to load the executables. Make sure that the `'parallel'` is accessible (you might copy it into your `'bin'` directory). To execute, type:

```
parallel app.x
```

- On Microsoft NT, there is no support for TCGMSG, which means you can only build your application based on MPI. Run the application program the same way as any other MPI programs. For, WMPI you need to create the .pg file.

Example:

```
R:\nt\g\global\testing> start /b test.exe
```

3. Initialization and Termination

For historical reasons (the 2-dimensional interface was developed first), many operations have two interfaces, one for two dimensional arrays and the other for arbitrary dimensional (one- to seven- dimensional, to be more accurate) arrays. The latter can definitely handle two dimensional arrays as well. The supported data types are *integer*, *double precision*, and *double complex*. Global Arrays provide C and Fortran interfaces in the same (mixed-language) program to the same array objects. The underlying data layout is based on the Fortran convention.

GA programs require message-passing and Memory Allocator (MA) libraries to work. Global Arrays is an extension to the message-passing interface. GA internally does not allocate local memory from the operating system - all dynamically allocated local memory comes from MA. We will describe the details of memory allocation later in this section.

3.1 Message Passing

The first version of Global Arrays was released in 1994 before robust MPI implementations became available. At that time, GA worked only with TCGMSG, a message-passing library that one of the GA authors (Robert Harrison) had developed before. In 1995, support for MPI was added. At the present time, the GA distribution still includes the TCGMSG library for backward compatibility purposes, and because it is small, fast to compile, and provides a minimal message-passing support required by GA programs. The user can enable the MPI-compatible version of GA by defining USE_MPI environment variable before compiling the GA toolkit. On systems where vendors provide MPI with interoperable C and Fortran interfaces, there is no advantage in compiling or using TCGMSG.

The GA toolkit needs the following functionality from any message-passing library it runs with:

- initialization and termination of processes in an SPMD (single-program-multiple-data) program,
- synchronization,
- functions that return number of processes and calling process id,
- broadcast,
- reduction operation for integer and double datatypes, and
- a function to abort the running parallel job in case of an error.

The message-passing library has to be initialized before the GA library and terminated after the GA library is terminated.

GA provides two functions `ga_nnodes` and `ga_nodeid` that return the number of processes and the calling process id in a parallel program. Starting with release 3.0, these functions return the same values as their message-passing counterparts. In earlier releases of GA on clusters of workstations, the mapping between GA and message-passing process ids were nontrivial. In these cases, the `ga_list_nodeid` function (now obsolete) was used to describe the actual mapping.

Although message-passing libraries offer their own barrier (global synchronization) function, this operation does not wait for completion of the outstanding GA communication operations.

The GA toolkit offers a `ga_sync` operation that can be used for synchronization, and it has the desired effect of waiting for all the outstanding GA operations to complete.

3.2 Memory Allocation

GA uses a very limited amount of statically allocated memory to maintain its data structures and state. Most of the memory is allocated dynamically as needed, primarily to store data in newly allocated global arrays or as temporary buffers internally used in some operations and deallocated when the operation is completed.

There are two flavors of dynamically allocated memory in GA: shared memory and local memory. Shared memory is a special type of memory allocated from the operating system (UNIX and Windows) that can be shared between different user processes (MPI tasks). A process that attaches to a shared memory segment can access it as if it was local memory. All the data in shared memory is directly visible to every process that attaches to that segment. On shared memory systems and clusters of SMP (symmetritc multiprocessor) nodes, shared memory is used to store global array data and is allocated by the Global Arrays run-time system called ARMCI. ARMCI uses shared memory to optimize performance and avoid explicit interprocessor communication within a single shared memory system or an SMP node. ARMCI allocates shared memory from the operating system in large segments and then manages memory in each segment in response to the GA allocation and deallocation calls. Each segment can hold data in many small global arrays. ARMCI does not return shared memory segments to the operating system until the program terminates (calls `ga_terminate`).

On systems that do not offer shared-memory capabilities or when a program is used in serial mode, GA uses local memory to store data in global arrays.

All of the dynamically allocated local memory in GA comes from its companion library, the Memory Allocator (MA) library. MA allocates and manages local memory using *stack* and *heap* disciplines. Any buffer allocated and deallocated by a GA operation that needs temporary buffer space comes from the MA *stack*. Memory to store data in global arrays comes from *heap*. MA has additional features useful for program debugging such as:

- left and right guards: They are stamps that detect if a memory segment was overwritten by the application,
- named memory segments, and
- memory usage statistics for the entire program.

Explicit use of MA by the application to manage its non-GA, local data structures is not necessary but encouraged. Because MA is used implicitly by GA, it has to be initialized before the first global array is allocated. The *MA_init* function requires users to specify memory for *heap* and *stack*. This is because MA:

- allocates from the operating system only one segment equal in size to the sum of *heap* and *stack* ,
- manges both allocation schemes using memory coming from opposite ends of the same segment, and
- the boundary between free *stack* and *heap* memory is dynamic.

It is not important what the stack and heap size argument values are as long as the aggregate

memory consumption by a program does not exceed their sum at any given time.

3.2.1 How to determine what the values of MA stack and heap size should be?

The answer to this question depends on the run-time environment of the program including availability of shared memory. A part of GA initialization involves initialization of the ARMCI run-time library. ARMCI dynamically determines if the program can or should use shared memory based on the architecture type and the current configuration of the SMP cluster. For example on uniprocessor nodes of the IBM SP shared memory is not used whereas on the SP with SMP nodes it is. This decision is made at run-time. GA reports the information about the type of memory used with the function `ga_uses_ma()`. This function returns false when shared memory is used and true when MA is used.

Based on this information, the programmer who cares about efficient usage of memory has to consider the amount of memory per single process (MPI task) needed to store data in global arrays to set the heap size argument value in `ma_init`. The amount of stack space depends on the GA operations used by the program (for example `ga_mulmat_patch` or `ga_dgemm` need several MB of buffer space to deliver good performance) but it probably should not be less than 4MB. The stack space is only used when a GA operation is executing and it is returned to MA when it completes.

3.3 GA Initialization

The GA library is initialized after a message-passing library and before MA. It is possible to initialize GA after MA but it is not recommended: GA must first be initialized to determine if it needs shared or MA memory for storing distributed array data. There are two alternative functions to initialize GA:

```
C        void GA_Initialize()
Fortran  subroutine ga_initialize()
```

and

```
C        void GA_Initialize_ltd(size_t limit)
Fortran  subroutine ga_initialize_ltd(limit)
```

The first interface allows GA to consume as much memory as the application needs to allocate new arrays. The latter call allows the programmer to establish and enforce a limit within GA on the memory usage.

3.3.1 Limiting Memory Usage by Global Arrays

GA offers an optional mechanism that allows a programmer to limit the aggregate memory consumption used by GA for storing global array data. These limits apply regardless of the type of memory used for storing Global Array data.They do not apply to temporary buffer space GA might need to use to execute any particular operation. The limits are given per process (MPI task) in bytes. If the limit is set, GA would not allocate more memory in global arrays that would exceed the specified value - the calls to allocate new arrays that would simply fail (return false). There are two ways to set the limit:

- at initialization time by calling `ga_initialize_ltd`, or

- after initialization by calling the function

```
C         void GA_Set_memory_limit(size_t limit)
Fortran   subroutine ga_set_memory_limit(limit)
```

It is encouraged that the user choose the first option, even though the user can intialize the GA normally and set the memory limit later.

Example: Initialization of MA and setting GA memory limits

```
call ga_initialize()
if (ga_uses_ma()) then
    status = ma_init(MT_DBL, stack, heap+global)
else
    status = ma_init(MT_DBL,stack,heap)
    call ga_set_memory_limit(ma_sizeof(MT_DBL,global,MT_BYTE))
endif
if(.not. status) ... !we got an error condition here
```

In this example, depending on the value returned from `ga_uses_ma()`, we either increase the *heap* size argument by the amount of memory for global arrays or set the limit explicitly through `ga_set_memory_limit()`. When GA memory comes from MA we do not need to set this limit through the GA interface since MA enforces its memory limits anyway. In both cases, the maximum amount of memory acquired from the operating system is capped by the value *stack+heap+global*.

3.4 Termination

The normal way to terminate a GA program is to call the function

```
C         void GA_Terminate()
Fortran   subroutine ga_terminate()
```

The programmer can also abort a running program, for example, as part of handling a programmatically detected error condition by calling the function

```
C         void GA_Error(char *message, int code)
Fortran   subroutine ga_error(message, code)
```

3.5 Creating arrays

There are two way to create new arrays:

1. From scratch, for regular distribution, use the function

```
C             int NGA_Create(int type, int ndim, int dims[],
                             char *array_name, int chunk[])
n-d Fortran   logical function nga_create(type, ndim, dims,
                                          array_name, chunk, g_a)
2-d Fortran   logical function ga_create(type, dim1, dim2,
                                         array_name, chunk, g_a)
```

Or for irregular distribution, use the function

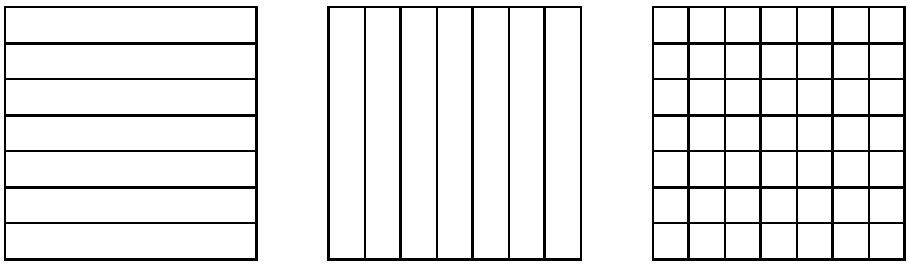
```
C            int NGA_Create_irreg(int type, int ndim, int dims[],
                                  char *array_name, int map[], int block[])
n-d Fortran  logical function nga_create_irreg(type, ndim,
                                    dims, array_name, map, nblock, g_a)
2-d Fortran  logical function ga_create_irreg(type, dim1, dim2,
                                  array_name, map1, nblock1, map2, nblock2, g_a)
```

2. Based on a template (an existing array) with the function

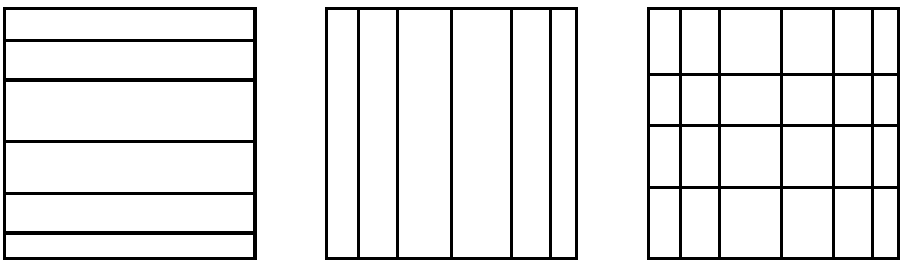
```
C        int GA_Duplicate(int g_a, char *array_name)
Fortran  logical function ga_duplicate(g_a, g_b, array_name)
```

In this case the new array inherites all the properties such as distribution, datatype, and dimensions, from the existing array.

With the regular distribution, the programmer can specify block size for none or any dimension. If block size is not specified the library will create a distribution that attempts to assign the same number of elements to each processor (for static load balancing purposes). The actual algorithm used is based on heuristics.



With the irregular distribution, the programmer specifies distribution points for every dimension. The library creates an array with the overall distribution that is a Cartesian product of distributions for each dimension.



If an array cannot be created, for example due to memory shortages or an enforced memory consumption limit, these calls return *false*. Otherwise an integer handle is returned. This handle represents a global array object. All the properties of the object (data type, distribution data, name, number of dimensions and values for each dimension) can be obtained from the library based on the handle at any time, see Section 7.4. It is not necessary to keep track of this type of

data in the application.

3.6 Destroying arrays

Global arrays can be destroyed by calling the function

```
C        void GA_Destroy(int g_a)
Fortran  subroutine ga_destroy(g_a)
```

that takes as its argument a handle representing a valid global array. It is a fatal error to call ga_destroy with a handle pointing to an invalid array.

All active global arrays are destroyed implicitly when the user calls `ga_terminate`.

4. One-sided Operations

Global Arrays provide one-sided, noncollective communication operations that allow to access data in global arrays without cooperation with the process or processes that hold the referenced data. These processes do not know what data items in their own memory are being accessed or updated by remote processes. Morevover, since the GA interface uses global array indices to reference nonlocal data, the calling process does not have to know process ids and location in memory where the data resides.

The one-sided operations that global arrays provide can be summarized into three categories:

Remote blockwise write/read:	`ga_put/ga_get`
Remote atomic update:	`ga_acc/ga_read_inc`
Remote elementwise write/read:	`ga_scatter/ga_gather`

4.1 Put/Get

Put and *get* are two powerful operations for interprocess communication, performing remote write and read. Because of their one-sided nature, they don't need cooperation from the process(es) that owns the data. The semantics of these operations does not require the user to specify which remote process or processes own the accessed portion of a global array. The data is simply accessed as if it were in shared memory.

Put copies data from the local array to the global array section, which is

```
C    void NGA_Put(int g_a, int lo[], int hi[], void *buf, int ld[])
n-D Fortran  subroutine nga_put(g_a, lo, hi, buf, ld)
2-D Fortran  subroutine ga_put(g_a,ilo,ihi,jlo,jhi,buf,ld)
```

All the arguments are provided in one call: `lo` and `hi` specify where the data should go in the global array; `ld` specifies the stride information of the local array `buf`. The local array should have the same number of dimensions as the global array; however, it is really required to present the n-dimensional view of the local memory buffer, that by itself might be one-dimensional.

The operation is transparent to the user, which means the user doesn't have to worry about where the region defined by `lo` and `hi` is located. It can be in the memory of one or many remote processes, owned by the local process, or even mixed (part of it belongs to remote processes and part of it belongs to a local process).

Get is the reverse operation of *put*. It copies data from a global array section to the local array.

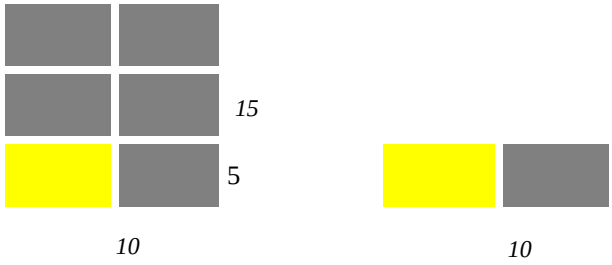
```
C    void NGA_Get(int g_a, int lo[], int hi[], void *buf, int ld[])
n-D Fortran  subroutine nga_get(g_a, lo, hi, buf, ld)
2-D Fortran  subroutine ga_get(g_a,ilo,ihi,jlo,jhi,buf,ld)
```

Similar to *put*, `lo` and `hi` specify where the data should come from in the global array, and `ld` specifies the stride information of the local array `buf`. The local array is assumed to have the

same number of dimensions as the global array. Users don't need to worry about where the region defined by `lo` and `hi` is physically located.

Example: For a `ga_get` operation transferring data from the (11:15,1:5) section of a 2-dimensional 15 x10 global array into a local buffer 5 x10 array we have: (in Fortran notation)

lo = {11,1}, hi = {15,5}, ld = {10}



4.2 Accumulate/Read_inc

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather then replacing the data there. *Accumulate* and *read_inc* perform **atomic** remote update to a patch (a section of the global array) in the global array and an element in the global array, respectively. They don't need the cooperation of the process(es) who owns the data. Since the operations are atomic, the same portion of a global array can be referenced by these operations issued by multiple processes and the GA will assure the correct and consistent result of the updates.

Accumulate combines the data from the local array with data in the global array section, which is

```
C             void NGA_Acc(int g_a, int lo[], int hi[], void *buf,
                           int ld[],void *alpha)
n-D Fortran   subroutine nga_acc(g_a, lo, hi, buf, ld, alpha)
2-D Fortran   subroutine ga_acc(g_a,ilo,ihi,jlo,jhi,buf,ld, alpha)
```

The local array is assumed to have the same number of dimensions as the global array. Users don't need to worry about where the region defined by `lo` and `hi` is physically located. The function performs (in C notation)

*global array section (lo[], hi[]) += alpha * buf*

Read_inc remotely updates a particular element in the global array, which is

```
C             void NGA_Read_inc(int g_a, int subscript[], long inc)
n-D Fortran   subroutine nga_read_inc(g_a, subscript, inc)
2-D Fortran   subroutine ga_read_inc(g_a, i, j, inc)
```

This function applies to integer arrays only. It atomically reads and increments an element in an integer array. It performs

a(subsripts) += inc

and returns the original value (before the update) of *a(subscript).*

4.3 Scatter/Gather

Scatter and *gather* transfer a specified set of elements to and from global arrays. They are one-sided: that is they don't need the cooperation of the process(es) who owns the referenced elements in the global array.

Scatter puts array elements into a global array, which is

```
C          void NGA_Scatter(int g_a, void *v, int *subsarray[], int n)
n-D Fortran  subroutine nga_scatter(g_a, v, subsarray, n)
2-D Fortran  subroutine ga_scatter(g_a, v, i, j, n)
```

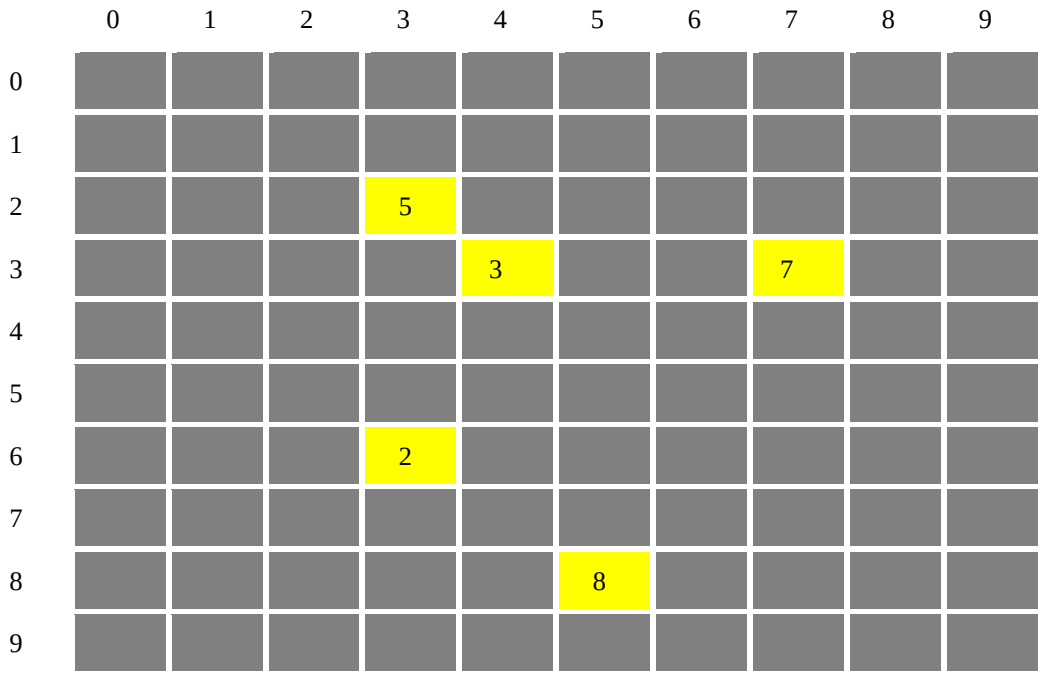
It performs (in C notation)

```
for(k=0; k<= n; k++) {
   a[subsArray[k][0]][subsArray[k][1]][subsArray[k][2]]... = v[k];
}
```

Example: Scatter the 5 elements into a 10x10 global array

```
Element 1    v[0] = 5    subarray[0][0] = 2  subarray[0][1] = 3
Element 2    v[1] = 3    subarray[1][0] = 3  subarray[1][1] = 4
Element 3    v[2] = 8    subarray[2][0] = 8  subarray[2][1] = 5
Element 4    v[3] = 7    subarray[3][0] = 3  subarray[3][1] = 7
Element 5    v[4] = 2    subarray[4][0] = 6  subarray[4][1] = 3
```

After the scatter operation, the five elements would be scattered into the global array as shown in the following figure.



Gather is the reverse operation of *scatter*. It gets the array elements from a global array into a local array.

```
C         void NGA_Gather(int g_a, void *v, int *subsarray[], int n)
n-D Fortran  subroutine nga_gather(g_a, v, subsarray, n)
2-D Fortran  subroutine ga_gather(g_a, v, i, j, n)
```

It performs (in C notation)

```
for(k=0; k<= n; k++){
    v[k] = a[subsArray[k][0]][subsArray[k][1]][subsArray[k][2]]...;
}
```

5. Interprocess Synchronization

Global Arrays provide three types of synchronization calls to support different synchronization styles.

Lock with mutex: is useful for a shared memory model. One can lock a mutex, to exclusively access a critical section.

Fence: guarantees that the Global Array operations issued from the calling process are complete. The fence operation is local.

Sync: is a barrier. It synchronizes processes and ensures that all Global Array operations completed. Sync operation is collective.

5.1 Lock and Mutex

Lock works together with mutex. It is a simple synchronization mechanism used to protect a critical section.To enter a critical section, typically, one needs to do:

1. *Create mutexes*
2. *Lock on a mutex*
3. ...
   Do the exclusive operation in the critical section
   ...
4. *Unlock the mutex*
5. *Destroy mutexes*

The function

```
C        int GA_Create_mutexes(int number)
Fortran  logical function ga_create_mutexes(number)
```

creates a set containing the *number* of mutexes. Only one set of mutexes can exist at a time. Mutexes can be created and destroyed as many times as needed. Mutexes are numbered: 0, ..., *number*-1.

The function

```
C        int GA_Destroy_mutexes()
Fortran  logical function ga_destroy_mutexes()
```

destroys the set of mutexes created with `ga_create_mutexes`.

Both `ga_create_mutexes` and `ga_destroy_mutexes` are collective operations.

The functions

```
C        void GA_lock(int mutex)
         void GA_unlock(int mutex)
Fortran  subroutine ga_lock(int mutex)
         subroutine ga_unlock(int mutex)
```

lock and unlock a mutex object identified by the `mutex` number, respectively. It is a fatal error for a process to attempt to lock a mutex which has already been locked by this process, or unlock

a mutex which has not been locked by this process.

Example 1:

Use one mutex and the lock mechanism to enter the critical section.

```
status = ga_create_mutexes(1)
if(.not.status) then
   call ga_error('ga_create_mutexes failed ',0)
endif
call ga_lock(0)

   ... do something in the critical section
   call ga_put(g_a, ...)
   ...

call ga_unlock(0)
if(.not.ga_destroy_mutexes()) then
   call ga_error('mutex not destroyed',0)
```

5.2 Fence

Fence blocks the calling process until all the data transfers corresponding to the Global Array operations initiated by this process complete. The typical scenario that it is being used is

1. *Initialize the fence*
2. ...
   Global Array operations
   ...
3. *Fence*

This would guarantee the operations between step 1 and 3 are complete.

The function

```
C        void GA_Init_fence()
Fortran  subroutine ga_init_fence()
```

Initializes tracing of completion status of data movement operations.

The function

```
C        void GA_Fence()
Fortran  subroutine ga_fence()
```

blocks the calling process until all the data transfers corresponding to GA operations called after `ga_init_fence` complete.

`ga_fence` must be called after `ga_init_fence`. A barrier, `ga_sync`, assures completion of all data transfers and implicitly cancels outstanding `ga_init_fence`. `ga_init_fence` and `ga_fence` must be used in pairs, multiple calls to `ga_fence` require the same number of corresponding `ga_init_fence` calls. `ga_init_fence/ga_fence` pairs can be nested.

Example 1:

Since `ga_put` might return before the data reaches the final destination `ga_init_fence` and

ga_fence allow the process to wait until the data is actually moved:

```
call ga_init_fence()
call ga_put(g_a, ...)
call ga_fence()
```

Example 2:

ga_fence works for multiple GA operations.

```
call ga_init_fence()
call ga_put(g_a, ...)
call ga_scatter(g_a, ...)
call ga_put(g_b, ...)
call ga_fence()
```

The calling process will be blocked until data movements initiated by two calls to ga_put and one ga_scatter complete.

5.3 Sync

Sync is a collective operation. It acts as a barrier, which synchronizes all the processes and ensures that all the Global Array operations are complete at the call.

The function is

```
C        void GA_Sync()
Fortran  subroutine ga_sync()
```

Sync should be inserted as necessary. With too many sync calls, the application performance would suffer.

6. Collective Array Operations

Global Arrays provide functions for collective array operations, targeting both whole arrays and patches (portions of global arrays). Collective operations require all the processes to make the call. In the underlying implementation, each process deals with its local data. These functions include:

- basic array operations,
- linear algebra operations, and
- interfaces to third party software packages.

6.1 Basic Array Operations

Global Arrays provide several mechanisms to manipulate contents of the arrays. One can set all the elements in an array/patch to a specific value, or as a special case set to zero. Since GA does not explicitly initialize newly created arrays, these calls are useful for initialization of an array/patch. (To fill the array with different values for each element, one can choose the one sided operation *put* or each process can initialize its local portion of an array/patch like ordinary local memory). One can also scale the array/patch by a certain factor, or copy the contents of one array/patch to another.

6.1.1 Whole Arrays

These functions apply to the entire array.

The function

```
C        void GA_Zero(int g_a)
Fortran  subroutine ga_zero(g_a)
```

sets all the elements in the array to zero.

To assign a single value to all the elements in an array, use the function

```
C        void GA_Fill(int g_a, void *val)
Fortran  subroutine ga_fill(g_a, val)
```

It sets all the elements in the array to the value *val*. The *val* must have the same data type as that of the array.

The function

```
C        void GA_Scale(int g_a, void *val)
Fortran  subroutine ga_scale(g_a, val)
```

scales all the elements in the array by factor *val*. Again the val must be the same data type as that of the array itself.

The above three functions are dealing with one global array, to set values or change all the elements together. The following functions are for copying data between two arrays.

The function

```
C       void GA_Copy(int g_a, int g_b)
Fortran subroutine ga_copy(g_a, g_b)
```

copies the contents of one array to another. The arrays must be of the same data type and have the same number of elements.

6.1.2 Patches

GA provides a set of operations on segments of the global arrays, namely patch operations. These functions are more general, in a sense they can apply to the entire array(s). As a matter of fact, many of the Global Array collective operations are based on the patch operations, for instance, the `GA_Print` is only a special case of `NGA_Print_patch`, called by setting the bounds of the patch to the entire global array. There are two interfaces for Fortran, one for two dimensional and the other for n-dimensional (one to seven). The n-dimensional interface can surely handle the two dimensional case as well. It is available for backward compatibility purposes. The functions dealing with n-dimensional patches use the "`nga`" prefix and those dealing with two dimensional patches start with the "`ga`" prefix.

The function

```
C           void NGA_Zero_patch(int g_a, int lo[] int hi[])
Fortran     subroutine nga_zero_patch(g_a, alo, ahi)
```

is similar to *ga_zero*, except that instead of applying to entire array, it sets only the region defined by *lo* and *hi* to zero.

One can assign a single value to all the elements in a patch with the function:

```
C          void NGA_Fill_patch(int g_a, int lo[] int hi[], void *val)
n-D Fortran  subroutine nga_fill_patch(g_a, lo, hi, val)
2-D Fortran  subroutine ga_fill_patch(g_a,ilo,ihi,jlo,jhi,val)
```

The `lo` and `hi` defines the patch and the `val` is the value to set.

The function

```
C        void NGA_Scale_patch(int g_a, int lo[] int hi[], void *val)
n-D Fortran  subroutine nga_scale_patch(g_a, lo, hi, val)
2-D Fortran  subroutine ga_scale_patch(g_a,ilo,ihi,jlo,jhi,val)
```

scales the patch defined by `lo` and `hi` by the factor `val`.

The copy patch operation is one of the fundamental and frequently used functions. The function

```
C           void NGA_Copy_patch(char trans, int g_a, int alo[],
                                int ahi[], int g_b, int blo[], int bhi[])
n-D Fortran  subroutine nga_copy_patch(trans, g_a, alo, ahi,
                                       g_b, blo, bhi)
2-D Fortran  subroutine ga_copy_patch(trans,g_a,ailo,aihi,ajlo,
                                      ajhi,g_b,bilo,bihi,bjlo,bjhi)
```

copies one patch defined by `alo` and `ahi` in one global array `g_a` to another patch defined by `blo` and `bhi` in another global array `g_b`. The current implementation requires that the source patch and destination patch must be on different global arrays. They must also be the same data type. The patches may be of different shapes, but the number of elements must be the same. During the process of copying, the transpose operation can be performed by specifying `trans`.

Example: Assume that there two 8x6 Global Arrays, `g_a` and `g_b`, distributed on three processes. The operation of `nag_copy_patch` (Fortran notation), from

```
g_a: alo = {2, 2}, ahi = {4, 5}
```

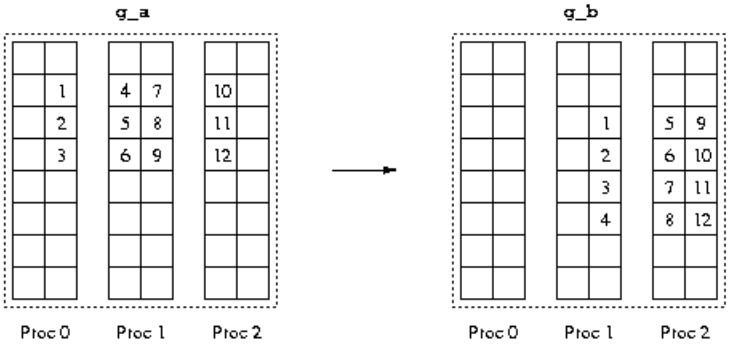
to

```
g_b: blo = {3, 4}, bhi = {6, 6}
```

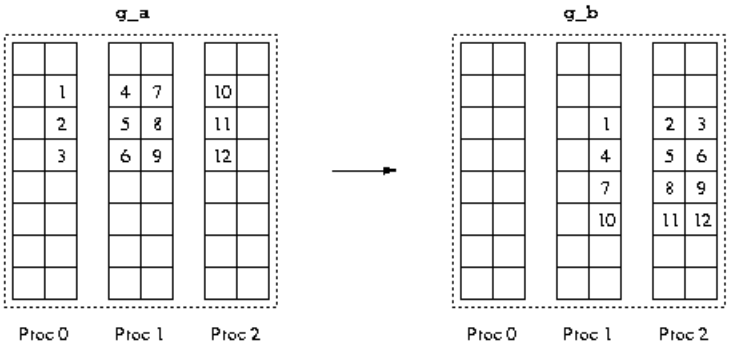
and

```
trans = 0
```

involves reshaping. It is illustrated in the following figure.



One step further, if one also want to perform the transpose operation during the copying, *i.e.* set `trans = 1`, it will look like:



If there is no reshaping or transpose, the operation can be fast (internally calling `nga_put`). Otherwise, it would be slow (internally calling `nga_scatter`, where extra time is spent on preparing the indices). Also note that extra memory is required to hold the indices if the operation involves reshaping or transpose.

6.2 Linear Algebra

Global arrays provide three linear algebra operations: addition, multiplication, and dot product. There are two sets of functions, one for the whole array and the other for the patches.

6.2.1 Whole Arrays

The function

```
C   void GA_Add(void *alpha, int g_a, void *beta, int g_b, int g_c)
Fortran  subroutine ga_add(alpha, g_a, beta, g_b, g_c)
```

adds two arrays, g_a and g_b, and saves the results to g_c. The two source arrays can be scaled by certain factors. This operation requires the two source arrays have the same number of elements and the same data types, but the arrays can have different shapes or distributions. g_c can also be g_a or g_b. It is encouraged to use this function when the two source arrays are identical in distributions and shapes, because of its efficiency. It would be less efficient if the two source arrays are different in distributions or shapes.

Matrix multiplication operates on two matrices, therefore the array must be two dimensional. The function

```
C        void GA_Dgemm(char ta, char tb, int m, int n, int k,
                       double alpha, int g_a, int g_b,
                       double beta, int g_c )
Fortran  subroutine ga_dgemm(transa, transb, m, n, k,
                             alpha, g_a, g_b, beta, g_c )
```

Performs one of the matrix-matrix operations:

*C := alpha*op(A)*op(B) + beta*C,*

where *op(X)* is one of

op(X) = X or op(X) = X',

alpha and *beta* are scalars, and *A*, *B* and *C* are matrices, with *op(A)* an *m* by *k* matrix, *op(B)* a *k* by *n* matrix and *C* an *m* by *n* matrix.

On entry, *transa* specifies the form of *op(A)* to be used in the matrix multiplication as follows:

t*a* = *'N'* or *'n'*, *op(A) = A*.
t*a* = *'T'* or *'t'*, *op(A) = A'*.

The function

```
C        long          GA_Idot(int g_a, int g_b)
         double        GA_Ddot(int g_a, int g_b)
         DoubleComplex GA_Zdot(int g_a, int g_b)
Fortran  integer          function ga_idot(g_a, g_b)
         double precision function ga_ddot(g_a, g_b)
         double complex   function ga_zdot(g_a, g_b)
```

computes the element-wise dot product of two arrays. It is available as three separate functions, corresponding to *integer*, *double precision* and *double complex* data types.

The following functions apply to the 2-dimensional whole arrays only. There are no corresponding functions for patch operations.

The function

```
C        void GA_Symmetrize(int g_a)
Fortran  subroutine ga_symmetrize(g_a)
```

symmetrizes matrix A represented with handle g_a: *A = .5 * (A+A')*.

The function

```
C        void GA_Transpose(int g_a, int g_b)
Fortran  subroutine ga_transpose(g_a, g_b)
```

transposes a matrix: *B = A'*.

6.2.2 Patches

The functions

```
C            void NGA_Add_patch(void *alpha, int g_a, int alo[],
                    int ahi[], void *beta,  int g_b, int blo[],
                    int bhi[], int g_c, int clo[], int chi[])
n-D Fortran  subroutine nga_add_patch(alpha, g_a, alo, ahi,
                                      beta,  g_b, blo, bhi,
                                             g_c, clo, chi)
2-D Fortran  subroutine ga_add_patch(alpha,g_a,ailo,aihi,ajlo,
                                     ajhi,beta,g_b,bilo,bihi,bjlo,
                                     bjhi,g_c,cilo,cihi,cjlo,cjhi)
```

add element-wise two patches and save the results into another patch. Even though it supports the addition of two patches with different distributions or different shapes (the number of elements must be the same), the operation can be expensive, because there can be extra copies which effect memory consumption. The two source patches can be scaled by a factor for the addition. The function is smart enough to detect the case that the patches are exactly the same but the global arrays are different in shapes. It handles the case as if for the arrays were identically distributed, thus the performance will not suffer.

The matrix multiplication is the only operation on array patches that is restricted to the two dimensional domain, because of its nature. It works for *double* and *double complex* data types. The prototype is

```
C        void GA_Matmul_patch(char *transa, char* transb,
                              void* alpha, void *beta,
              int g_a, int ailo, int aihi, int ajlo, int ajhi,
              int g_b, int bilo, int bihi, int bjlo, int bjhi,
              int g_c, int cilo, int cihi, int cjlo, int cjhi)
Fortran  subroutine ga_matmul_patch(transa,transb,alpha,beta,
                                    g_a,ailo,aihi,ajlo,ajhi,
                                    g_b,bilo,bihi,bjlo,bjhi,
                                    g_c,cilo,cihi,cjlo,cjhi)
```

It performs

```
C [cilo:cihi,cjlo:cjhi] := alpha* AA[ailo:aihi,ajlo:ajhi] *
        BB[bilo:bihi,bjlo:bjhi] ) + beta*C[cilo:cihi,cjlo:cjhi]
```

where *AA = op(A), BB = op(B),* and *op(X)* is one of

op(X) = X or op(X) = X',

Valid values for transpose argument: *'n', 'N', 't', 'T'*.

The dot operation computes the element-wise dot product of two (possibly transposed) patches. It is implemented as three separate functions, corresponding to *integer*, *double precision* and *Double Complex* data types. They are

```
C       Integer NGA_Idot_patch(int g_a, char* ta, int alo[],
                int ahi[], int g_b, char* tb, int blo[], int bhi[])
         double NGA_Ddot_patch(int g_a, char* ta, int alo[],
                int ahi[], int g_b, char* tb, int blo[], int bhi[])
  DoubleComplex NGA_Zdot_patch(int g_a, char* ta, int alo[],
                int ahi[], int g_b, char* tb, int blo[], int bhi[])
n-D Fortran  integer function nga_idot_patch(g_a, ta, alo, ahi,
                                             g_b, tb, blo, bhi)
    double precision function nga_ddot_patch(g_a, ta, alo, ahi,
                                             g_b, tb, blo, bhi)
      double complex function nga_zdot_patch(g_a, ta, alo, ahi,
                                             g_b, tb, blo, bhi)
2-D Fortran  integer function ga_idot_patch(g_a,ta,ailo,aihi,
                           ajlo,ailo,g_b,tb,bilo,bihi,bjlo,bjhi)
    double precision function ga_ddot_patch(g_a,ta,ailo,aihi,
                           ajlo,ailo,g_b,tb,bilo,bihi,bjlo,bjhi)
      double complex function ga_zdot_patch(g_a,ta,ailo,aihi,
                           ajlo,ailo,g_b,tb,bilo,bihi,bjlo,bjhi)
```

The patches should be of the same data types and have the same number of elements. Like the array addition, if the source patches have different distributions/shapes, or it requires transpose, the operation would be less efficient, because there could be extra copies and/or memory consumption.

6.3 Interfaces to Third Party Software Packages

There are many existing software packages designed for solving engineering problems. They are specialized in one or two problem domains, such as solving linear systems, eigen-vectors, and differential equations, etc. Global Arrays provide interfaces to several of these packages.

6.3.1 Scalapack

Scalapack is a well known software library for linear algebra computations on distributed memory computers. Global Arrays uses this library to solve systems of linear equations and also to invert matrices.

The function

```
C        int GA_Solve(int g_a, int g_b)
Fortran  integer function ga_solve(g_a, g_b)
```

solves a system of linear equations *A * X = B*. It first will call the Cholesky factorization routine

and, if successful, will solve the system with the Cholesky solver. If Cholesky is not able to factorize *A*, then it will call the LU factorization routine and will solve the system with forward/backward substitution. On exit *B* will contain the solution *X*.

The function

```
C        int GA_Llt_solve(int g_a, int g_b)
Fortran  integer function ga_llt_solve(g_a, g_b)
```

also solves a system of linear equations *A* * *X* = *B*, using the Cholesky factorization of an *NxN* double precision symmetric positive definite matrix *A* (handle g_a). On successful exit *B* will contain the solution *X*.

The function

```
C        void GA_Lu_solve(char trans, int g_a, int g_b)
Fortran  subroutine ga_lu_solve(trans, g_a, g_b)
```

solves the system of linear equations *op(A)X* = *B* based on the LU factorization. op(A) = A or A' depending on the parameter `trans`. Matrix *A* is a general real matrix. Matrix *B* contains possibly multiple *rhs* vectors. The array associated with the handle g_b is overwritten by the solution matrix *X*.

The function

```
C        int GA_Spd_invert(int g_a)
Fortran  integer function ga_spd_invert(g_a)
```

computes the inverse of a double precision matrix using the Cholesky factorization of a *NxN* double precision symmetric positive definite matrix *A* stored in the global array represented by g_a. On successful exit, *A* will contain the inverse.

6.3.2 PeIGS

The PeIGS library contains subroutines for solving standard and generalized real symmetric eigensystems. All eigenvalues and eigenvectors can be computed. The library is implemented using a message-passing model and is portable across many platforms. For more information and availability send a message to *gi_fann@pnl.gov*. Global Arrays use this library to solve eigen-value problems.

The function

```
C        void GA_Diag(int g_a, int g_s, int g_v, void *eval)
Fortran  subroutine ga_diag(g_a, g_s, g_v, eval)
```

solves the generalized eigen-value problem returning all eigen-vectors and values in ascending order. The input matrices are not overwritten or destroyed.

The function

```
C        void GA_Diag_reuse(int control, int g_a, int g_s,
                            int g_v, void *eval)
Fortran  subroutine ga_diag_reuse(control, g_a, g_s, g_v, eval)
```

solves the generalized eigen-value problem returning all eigen-vectors and values in ascending order. Recommended for REPEATED calls if `g_s` is unchanged.

The function

```
C        void GA_Diag_std(int g_a, int g_v, void *eval)
Fortran  subroutine ga_diag_std(g_a, g_v, eval)
```

solves the standard (non-generalized) eigenvalue problem returning all eigenvectors and values in the ascending order. The input matrix is neither overwritten nor destroyed.

6.3.3 Interoperability with Others

Global Arrays are interoperable with several other libraries, but do not provide direct interfaces for them. For example, one can make calls to and link with these libraries:

PETSc(the Portable, Extensible Toolkit for Scientific Computation) is developed by the Argonne National Laboratory. PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication, and is written in a data-structure-neutral manner to enable easy reuse and flexibility. The instructions for using PETSc with GA is in Appendix 1.

CUMULVS (Collaborative User Migration User Library for Visualization and Steering) is developed by the Oak Ridge National Laboratory. CUMULVS is a software framework that enables programmers to incorporate fault-tolerance, interactive visualization and computational steering into existing parallel programs. The instructions for using CUMULVS with GA is in Appendix 2.

7. Utility Operations

Global arrays provide some utility functions to get the local process/data information, check the memory availability, etc. There are also several handy functions that print array distribution information, or summarize array usage information.

7.1 Locality Information

For a given global array element, or a given patch, sometimes it is necessary to find out who owns this element or patch. The function

```
C            int NGA_Locate(int g_a, int subscript[])
n-D Fortran  logical function nga_locate(g_a, subscript, owner)
2-D Fortran  logical function ga_locate(g_a, i, j, owner)
```

tells who (process id) owns the elements defined by the array subscripts.

The function

```
C            int NGA_Locate_region(int g_a, int lo[], int hi[],
                                   int *map[], int procs[])
n-D Fortran  logical function nga_locate_region(g_a,lo,hi,map,
                                                proclist, np)
2-D Fortran  logical function ga_locate_region(g_a,ilo,ihi,jlo,
                                               jhi, map, np)
```

returns a list of GA process IDs that 'own' the patch.

The Global Arrays support an abstraction of a distributed array object. This object is represented by an integer handle. A process can access its portion of the data in the global array. To do this, the following steps need to be taken:

- find the distribution of an array, which part of the data the calling process owns
- access the data
- operate on the date: read/write
- release the access of data

The function

```
C       void NGA_Distribute(int g_a, int iproc, int lo[], int hi[])
n-D Fortran  subroutine nga_distribute(g_a, iproc, lo, hi)
2-D Fortran  subroutine ga_distribute(g_a, iproc, ilo,ihi, jlo,jhi)
```

finds out the range of the global array `g_a` that process `iproc` owns. `ipoc` can be any valid process ID.

The function

```
C void NGA_Access(int g_a, int lo[], int hi[], void *ptr, int ld[])
n-D Fortran  subroutine nga_access(g_a, lo, hi, index, ld)
2-D Fortran  subroutine ga_access(g_a,ilo,ihi,jlo,jhi,index,ld)
```

provides access to local data in the specified patch of the array owned by the calling process. The C interface gives the pointer to the patch. The Fortran interface gives the patch address as the index (distance) from the reference address (the appropriate MA base addressing array).

The function

```
C            void NGA_Release(int g_a, lo[], int hi[])
n-D Fortran  subroutine nga_release(g_a, lo, hi)
2-D Fortran  subroutine ga_release(g_a, ilo, ihi, jlo, jhi)
```

and

```
C            void NGA_Release_update(int g_a, int lo[], int hi[])
n-D Fortran  subroutine nga_release_update(g_a, lo, hi)
2-D Fortran  subroutine ga_release_update(g_a,ilo,ihi,jlo,jhi)
```

releases access to a global array. The former set is used when the data was read only and the latter set is used when the data was accessed for writing.

Global Arrays also provide a function to compare distributions of two arrays. It is

```
C        void NGA_Compare_distr(int g_a, int g_b)
Fortran  subroutine ga_compare_distr(g_a, g_b)
```

7.1.1 Process Information

When developing a program, one needs to use the characteristics of its parallel environment: process ID, how many processes are working together and what their IDs are, and what the topology of processes look like. To answer these questions, the following functions can be used.

The function

```
C        int GA_Nodeid()
Fortran  integer function ga_nodeid()
```

returns the GA process ID of the current process, and the function

```
C        int GA_Nnodes()
Fortran  integer function ga_nnodes()
```

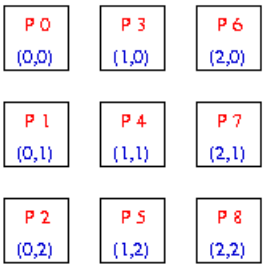
tells the number of computing processes.

The function

```
C        void NGA_Proc_topology(int g_a, int proc, int coordinates)
Fortran  subroutine ga_proc_topology(ga, proc, prow, pcol)
```

determines the coordinates of the specified processor in the virtual processor grid corresponding to the distribution of array `g_a`.

Example: An global array is distributed on 9 processors. The processors are numbered from 0 to 8 as shown in the following figure. If one wants to find out the coordinates of processor 7 in the virtual processor grid, by calling the fuction `ga_proc_topology`, the coordinates of `(2,1)` will be returned.



7.2 Memory Availability

Even though the memory management does not have to be performed directly by the user, Global Arrays provide functions to verify the memory availability. Global Arrays provide the following information:

- How much memory has been used by the allocated global arrays.
- How much memory is left for allocation of new global arrays.
- Whether the memory in global arrays comes from the Memory Allocator (MA).
- Is there any limitation for the memory usage by the Global Arrays.

The function

```
C        size_t GA_Inquire_memory()
Fortran  integer function ga_inquire_memory()
```

answers the first question. It returns the amount of memory (in bytes) used in the allocated global arrays on the calling processor.

The function

```
C        size_t GA_Memory_avail()
Fortran  integer function ga_memory_avail()
```

answers the second question. It returns the amount of memory (in bytes) left for allocation of new global arrays on the calling processor.

Memory Allocator(MA) is a library of routines that comprises a dynamic memory allocator for use by C, Fortran, or mixed-language applications. Fortran- 77 applications require such a library because the language does not support dynamic memory allocation. C (and Fortran-90) applications can benefit from using MA instead of the ordinary malloc() and free() routines because of the extra features MA provides. The function

```
C        int GA_Uses_ma()
Fortran  logical function ga_uses_ma()
```

tells whether the memory in Global Arrays comes from the Memory Allocator (MA) or not.

The function

```
C        int GA_Memory_limited()
Fortran  logical function ga_memory_limited()
```

Indicates if a limit is set on memory usage in Global Arrays on the calling processor.

7.3 Message-Passing Wrappers to Reduce/Broadcast Operations

Global Arrays provide convenient operations for broadcast/reduce regardless of the message-passing library that the process is running with.

The function

```
C        void GA_Brdcst(void *buf, int lenbuf, int root)
Fortran  subroutine ga_brdcst(type, buf, lenbuf, root)
```

broadcasts from process root to all other processes a message buffer of length `lenbuf`.

The functions

```
C        void GA_Igop(long x[], int n, char *op)
         void GA_Dgop(double x[], int n, char *op)
Fortran  subroutine ga_igop(type, x, n, op)
         subroutine ga_dgop(type, x, n, op)
```

'sum' elements of *X(1:N)* (a vector present on each process) across all nodes using the communicative operator `op`, The result is broadcasted to all nodes. Supported operations include

`+, *, Max, min, Absmax, absmin`

The integer version also includes the **`bitwise OR`** operation.

These operations unlike `ga_sync`, do not include embedded `ga_fence` operations.

7.4 Others

There are some other useful functions in Global Arrays. One group is about inquiring the array attributes. Another group is about printing the array or part of the array.

7.4.1 Inquire

A global array is represented by a handle. Given a handle, one can get the array information, such as the array name, memory used, array data type, and array dimension information, with the help of following functions.

The functions

```
C       void NGA_Inquire(int g_a, int *type, int *ndim, int dims[])
n-D Fortran  subroutine nga_inquire(g_a, type, ndim, dims)
2-D Fortran  subroutine ga_inquire(g_a, type, dim1, dim2)
```

return the data type of the array, and also the dimensions of the array.

The function

```
C        char* GA_Inquire_name(int g_a)
Fortran  subroutine ga_inquire_name(g_a, array_name)
```

finds out the name of the array.

One can also inquire the memory being used with `ga_inquire_memory` (discussed above).

7.4.2 Print

Global arrays provide functions to print

- content of the global array
- content of a patch of global array
- the status of array operations
- a summary of allocated arrays

The function

```
C        void GA_Print(int g_a)
Fortran  subroutine ga_print(g_a)
```

prints the entire array to the standard output. The output is formatted.

A utility function is provided to print data in the patch, which is

```
C     void NGA_Print_patch(int g_a, int lo[], int hi[], int pretty)
Fortran  subroutine nga_print_patch(g_a, lo, hi, pretty)
```

One can either specify a formatted output (set `pretty` to one) where the output is formatted and rows/ columns are labeled, or (set `pretty` to zero) just dump all the elements of this patch to the standard output without any formatting.

The function

```
C        void GA_Print_stats()
Fortran  subroutine ga_print_stats()
```

prints the global statistics information about array operations for the calling process, including

- number of calls to the GA create/duplicate, destroy, get, put, scatter, gather, and read_and_inc operations
- total amount of data moved in the GA primitive operations
- amount of data moved in GA primitive operations to logically remote locations
- maximum memory consumption in global arrays, the "high-water mark".

The function

```
C        void GA_Print_distribution(int g_a)
Fortran  subroutine ga_print_distribution(g_a)
```

prints the global array distribution. It shows mapping array data to the processes.

The function

```
C        void GA_Summarize(int verbose)
Fortran  subroutine ga_summarize(verbose)
```

prints info about allocated arrays. `verbose` can be either one or zero.

7.4.3 Miscellaneous

The function

```
C        void GA_Check_handle(int g_a, char *string)
Fortran  subroutine ga_check_handle(g_a, string)
```

checks if the global array handle `g_a` represents a valid array. The `string` is the message to be printed when the handle is invalid.

Appendix 1: *Instructions for using PETSc with GA*

Inter-operability of Global Arrays with PETSc

PETSc (the Portable, Extensible Toolkit for Scientific Computation) was developed by the Argonne National Laboratory. PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication, and is written in a data-structure-neutral manner to enable easy reuse and flexibility.

The following summarizes the inter-operability status of Global Arrays and PETSc:

Inter-operability

Global Arrays toolkit is inter-operable with PETSc. In an application using Global Arrays, the PETSc solvers can be called to solve PDEs that require solving large-scale, sparse nonlinear systems of equations. The primary issue is how to convert the data structures of Global Arrays to those of PETSc before calling the PETSc solvers, and how to convert the data structures of PETSc back to Global Arrays after calling the PETSc solvers. PETSc provides enough mechanisms to deal with this issue. For vector operations, there are `VecCreateMPI()`, `VecSetValues()`, `VecGetArray()`, `VecRestoreArray()`, etc. The same functions exist for matrix operation.

The packages used in the testing are:

- Global Arrays Version 3.0
- PETSc Version 2.0.24

Instructions for using PETSc in a Global Arrays application

PETSc online documentation is a well maintained site for PETSc resources. Examples can be accessed both online or from the package itself.

A typical scenario to use PETSc in a Global Arrays application is that there is a global array `x` which represents the approximate solution initialized with some initial values. It needs to call one of the PETSc solvers to solve the problem, and restore the results back to `x`.

Here are the instructions for implementing an example `Ax = b`, where `A` is the matrix defining the linear system, `b` is the right hand side, and `x` is the approximate solution and an global array.

- Initialize PETSC (`PetscInitialize()`)
- Convert the global array x to the PETSc format
  - Create a PETSc Vector `pets_x` (`VecCreateMPI()`)
  - Get the range of `pets_x` which resides in the local process(or) (`VecGetOwnershipRange()`)
  - Get access to the local portion of `pets_x` (`VecGetArray()`)
  - Get the corresponding data block (the range of `pets_x` in local process(or)) in the global array `x` (`ga_get()`)
  - Put the data block to `pets_x` (`VecRestoreArray()`)

- Create the linear solver and set various options
- Solve the linear system
- Write the solution back to Global Array.
    - Get access to the local portion of `pets_x` (`VecGetArray()`)
    - Put the local portion of solution back to global array `x` (`ga_put()`)
    - Close the access to the local portion of `pets_x` (`VecRestoreArray()`)

There are detailed instructions for setting up environment variables on different platforms with the PETSc package. For example, users on Cray T3E at NERSC only need to the load the petsc module: insert

```
module load petsc
```

into the `.login` file.

Discussion

Data conversion between the Global Arrays and PETSc is the key issue for inter-operability. PETSc provides several ways to create Vectors and Matrices and to set values to them. We found that the most efficient way to connect the Global Arrays and PETSc is to use the GetArray and RestoreArray mechanism. GetArray and RestoreArray are not intended to set values though, they open a window to access and update the local Vector/Matrix of PETSc. Global Arrays provide the one-sided operations, get and put, which are perfect match for PETSc's GetArray and RestoreArray mechanism. The array segment of Global Arrays can be sent to or received from PETSc in block fashion, instead of updating element by element.

Here is how it works:

From Global Arrays to PETSc

- Access the local portion of PETSc Vector/Matrix
- Use `ga_get()` to get the corresponding section of Global Array
- Close the access to (also update) the local portion of PETSc Vector/Matrix

From PETSc to Global Arrays

- Access the local portion of PETSc Vector/Matrix
- Use `ga_put()` to put the PETSc data in the corresponding section of Global Array
- Close the access to the local portion of PETSc Vector/Matrix

Appendix 2: *Intructions for using CUMULVS with GA*

Inter-operability of Global Arrays with CUMULVS

CUMULVS (Collaborative User Migration User Library for Visualization and Steering) is developed by the Oak Ridge National Laboratory. It is a software framework that enables programmers to incorporate fault-tolerance, interactive visualization and computational steering into existing parallel programs.

The following summarizes the interoperability status of Global Arrays and CUMULVS:

Inter-operability

Global Array is inter-operable with CUMULVS's computational steering capability. In an application using Global Arrays, steering parameters can be defined for a CUMULVS front-end viewer and manipulated by the viewer during the lifetime of execution. The packages used in the testing are:

- Global Arrays Version 3.0
- CUMULVS Version 1.0

The Global Arrays can be configured to work with one of several communication libraries. The one we used was based on MPI. The CUMULVS is based on PVM (Version 3.3.11 or later).

Instructions for using CUMULVS in a Global Arrays application

CUMULVS User's Guide provides instructions of how to use CUMULVS, and the definitions of library functions. Examples that come with the CUMULVS package servea as a good starting point, which give the user some insight of what to do, even though they are written in PVM. The following is quoted from the CUMULVS User's Guide:

> A typical statement sequence that a programmer would follow is
>
> Initialize CUMULVS data structures (`stv_init()`)
> Define data decomposition (`stv_decompDefine()`)
> Define data field with a previously defined decomposition (`stv_fieldDefine()`)
> Define steering parameters (`stv_paramDefine()`)
>
> Start main iterative loop
>
> <usual calculation
> *nchanged* = `stv_sendToFE()`
> <program response to *nchanged* steered parameters
>
> End of main iterative loop

Before doing anything, the `STV_ROOT` environment variable should be set, either in `$HOME/.cshrc` or an equivalent shell startup file. The value of `$STV_ROOT` should be the directory where CUMULVS is, as in:

```
setenv STV_ROOT /home/me/CUMULVS
```

The applications should include the header files of

`fpvm3.h` (Fortran) or `pvm3.h` (C)
`fstv.h` (Fortran) or `stv.h` (C)

Compile the application and link it with either `libfstv.a` or `libstv.a,` depending on whether the application is written in Fortran or C.

Next, start the pvm daemon and run the application.

Start the viewer to manipulate the steering parameters.

Discussion

CUMULVS is fairly easy to use in a Global Arrays application. We successfully testing the inter-operability of Global Arrays with CUMULVS's computational steering capacity. The capacity of CUMULVS's visualization needs further investigation.

Appendix 3: *List of GA functions*