
# Global Arrays User Manual

Jarek Nieplocha, Manojkumar Krishnan, Vinod Tipparaju, Bruce Palmer

# Table of Contents

# Table of Contents

# Table of 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 shared memory. All details of the data distribution, addressing, and data access are encapsulated in the global array objects. Information about the actual data distribution and locality can be easily obtained and taken advantage of whenever data locality is important. The primary target architectures for which GA was developed are massively-parallel distributed-memory and scalable shared-memory systems.

GA divides logically shared data structures into "local" and "remote" portions. It recognizes variable data transfer costs required to access the data depending on the proximity attributes. A local portion of the shared memory is assumed to be faster to access and the remainder (remote portion) is considered slower to access. These differences do not hinder the ease-of-use since the library provides uniform access mechanisms for all the shared data regardless where the referenced data is located. In addition, any processes can access a local portion of the shared data directly/in-place like any other data in process local memory. Access to other portions of the shared data must be done through the GA library calls.

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.

GA is implemented as a library with C and Fortran-77 bindings, and there have been also a Python and C++ interfaces (included starting with the release 3.2) developed. Therefore, explicit library calls are required to use the GA model in a parallel C/Fortran 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 **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 locations. At least one global array has to be created before data transfer operations can be used. These GA 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 rely on a polling operation or require inserting any other GA library calls to assure communication progress on the remote side.

A programmer in the GA program has a 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-based interface rather than addresses of the 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 -- it simply provides a global view of the data structures. 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:

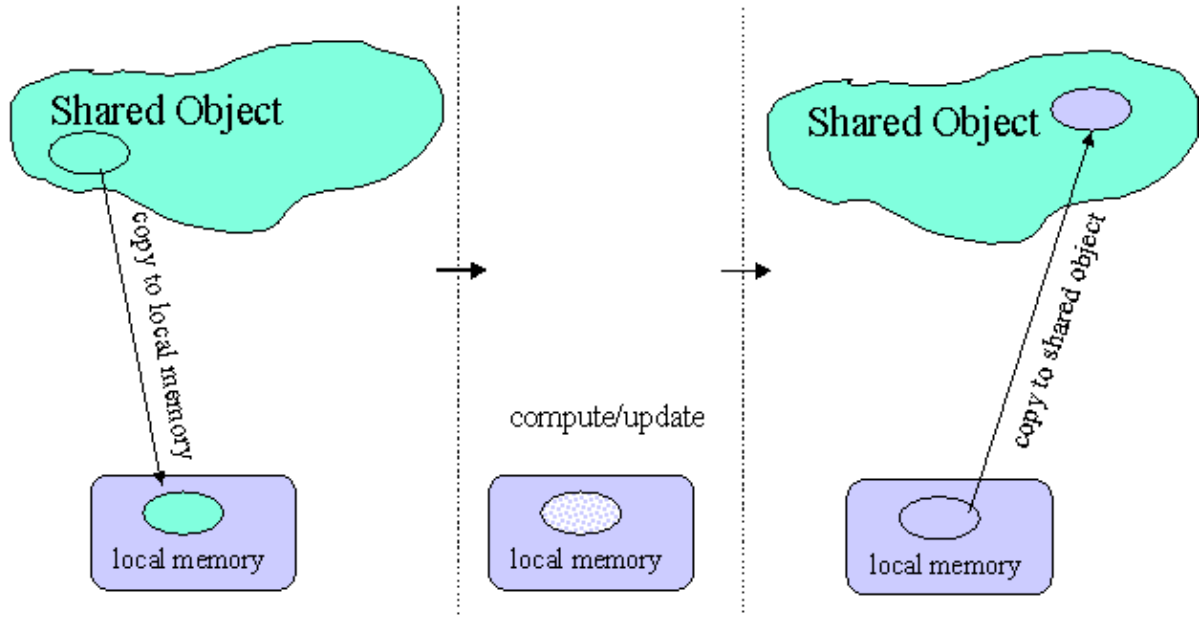
- where and an element or array section is located, and
- which process or processes own data in the specified array section.

The GA toolkit supports four data types in Fortran: integer, real, double precision, and double complex. In the C interface, int, long, float, double and struct double complex are available. Underneath, the library represents the data using C datatypes. For the Fortran users, it means that some arrays created in C for which there is no appropriate datatype mapping to Fortran (for example on the Cray T3E Fortran real is not implemented whereas C float is) might not be accessible. In all the other cases, the datatype representation is transparent.

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 Global Arrays library supports two programming styles: task-parallel and data-parallel. The GA task-parallel model of computations is based on the 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 by that process. Of course, 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 also 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.

The data-parallel model is supported by a set of collective functions that operate on global arrays or their portions. Underneath, if any interprocessor communication is required, the library uses remote memory copy (most often) or collective message-passing operations.

## 1.4 Application Guidelines

These are some guidelines regarding suitability of the GA for different types of 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 and/or for out-of-core array-based algorithms (GA + DRA)

Usability Considerations

◊ data locality must be explicitly available
◊ 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 with regular communication patters
- when synchronization associated with cooperative point-to-point message passing is needed (e.g., Cholesky factorization in Scalapack)

Usability Considerations

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

# 2. Writing, Building and Running GA Programs

The web page www.emsl.pnl.gov/docs/global/support.html contains updated information about using GA on different platforms. Please refer to this page frequently for most recent updates and platform information.

## 2.1 Platform and Library Dependencies

### 2.1.1 Supported Platforms

- IBM SP, CRAY T3E/J90/SV1, SGI Origin, Fujitsu VX/VPP, Hitachi
- Cluster of workstations: Solaris, IRIX, AIX, HPUX, Digital/Tru64 Unix, Linux, NT
- 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 newer (>3.1) versions because we do not have access to these systems. We recommend using GA 2.4 on these platforms.

For most of the platforms, there are two versions available: 32-bit and 64-bit. This table specifies valid TARGET names for various supported platforms.

| Platform | 32-bit TARGET name | 64-bit TARGET name | Remarks |
|---|---|---|---|
| Sun ultra | SOLARIS | SOLARIS64 | 64-bit version added in GA 3.1 |
| IBM BlueGene/L | | BGL | added in GA 4.0.2 (Contact your BlueGene sys admin for GA instalation). More info in support page... |
| IBM RS/6000 | IBM | IBM64 | 64-bit version added in GA 3.1 |
| IBM SP | LAPI | LAPI64 | no support yet for user-space communication in the 64-bit mode by IBM |

| | | | |
|---|---|---|---|
| Compaq/DEC alpha | not available | DECOSF | |
| HP pa-risc | HPUX | HPUX64 | 64-bit version added in GA 3.1 |
| Linux x86, ultra, powerpc | LINUX | not available | |
| Linux IA64 (Itanium), x86_64 (Opteron) | not available | LINUX64 | |
| Linux alpha | not available | LINUX64 | 64-bit version added in GA 3.1;<br>Compaq compilers rather than GNU required |
| Cray T3E | not available | CRAY-T3E | |
| Cray J90 | not available | CRAY-YMP | |
| Cray SV1 | not available | CRAY-SV1 | |
| Cray X1 | not available | cray-sv2 | In X1, by default, TARGET is defined by the operating system as cray-sv2 |
| SGI IRIX mips | SGI_N32, SGI | SGITFP | |
| Hitachi SR8000 | HITACHI | not available | |
| Fujitsu VPP systems | FUJITSU-VPP | FUJITSU-VPP64 | 64-bit version added in GA 3. |
| NEC SX series | | NEC | |
| Apple | | MACX | Running MAC X or higher |

To aid development of fully portable applications, in 64-bit mode Fortran integer datatype is 64-bits. It is motivated by 1) the need of applications to use very large data structures and 2) Fortran INTEGER*8 not being fully portable. The 64-bit representation of integer datatype is accomplished by using appropriate Fortran compiler flag.

Because of limited interest in heterogenous computing among known us GA users, the Global Array library still *does not support heterogeonous platforms*. This capability can be added if required by new applications.

### 2.1.2 Selection of the communication network for ARMCI

Some cluster installations can be equipped with a high performance network which offer instead, or in addition to TCP/IP some special communication protocol, for example GM on Myrinet network. To achieve high performance in Global Arrays, ARMCI must be built to use these protocols in its implementation of one-sided communication. Starting with GA 3.1, this is accomplished by setting an environment variable ARMCI_NETWORK to specify the protocol to be used. In addition, the it might be necessary to provide location for the header files and library path corresponding to location of s/w supporting the appropriate protocol API, see g/armci/config/makecoms.h for details.

| Network | Protocol name | ARMCI_NETWORK setting | Supported platforms |
|---|---|---|---|
| Ethernet | TCP/IP | SOCKETS (optional/default) | workstation clusters |
| Quadrics/QsNet | Elan3/Shmem | QUADRICS or ELAN3 | Linux (alpha,x86,IA64), Compaq |
| Quadrics/QsNet II | Elan4 | ELAN4 | Linux (IA64) |
| Infiniband | VAPI | MELLANOX | Linux (x86,IA64) |
| Myrinet | GM | GM | Linux (x86,ultra,IA64) |
| Giganet cLAN | VIA | VIA | Linux (x86) |

### 2.1.3 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:

1. GA with MPI (recommended): directly with MPI. In this mode, GA program should contain MPI initialization calls.

2. GA with TCGMSG-MPI (MPI and TCGMSG emulation library): TCGMSG-MPI 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*.
3. GA with TCGMSG: directly with TCGMSG. In this mode, GA program should contain TCGMSG initialization calls.

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/Mississippi 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 (starting with GA 3.1) separated by comma "," and withot the prefix, OR
- use MPI aware compiler wrappers e.g., 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 in some circumstances 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, the package might fail to build. This problem is most likely to occur on non-Linux Unix systems with non-native compilers (e.g., gcc).

On Windows NT, 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 <fanngi@ornl.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'*. Fortran source must be preprocessed as a part of compilation.

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

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
- config/makefile.h: definitions & include symbols

The user must specify TARGET as an environment variable (setenv TARGET `TARGET_name`) or in the

GNUmakefile or on the command line when calling make. For example:
setenv TARGET LAPI (for IBM/SP platform)
(or) from command line, gmake TARGET=LAPI

Valid `TARGET_name` for various supported platforms can be found in the above table. Valid TARGETs can also be listed by calling make in the top level distribution directory on UNIX family of systems when TARGET is not defined. On Windows, WIN32, CYGNUS and INTERIX (previously known as OpenNT) are supported.

Compiler Settings (optional): For various supported platforms, the default compilers and compiler options are specified in config/makefile.h. One could change the predefined default compilers and compiler flags in GA package either by specifying them on the command line or in the file config/makefile.h. Note: editing config/makefile.h for any platform requires extra care and intended for intermediate/advanced users.

- CC - name of the C compiler (e.g., gcc, cc, or ccc )
- FC - name of the Fortran compiler (e.g., g77, f90, mpif77 or fort)
- COPT - optimization or debug flags for the C compiler (e.g., -g, -O3)
- FOPT - optimization or debug flags for the Fortran compiler (e.g., -g, -O1)

For example,

```
gmake FC=f90 CC=cc FOPT=-O4 COPT=-g
```

Note that `GA` provides only Fortran-77 interfaces. To use and compile with a Fortran 90 compiler, it has to support a subset of Fortran-77.

### 2.3.1 Unix Environment

As mentioned in earlier section, there are three possible configurations for building GA.

1. GA with MPI (recommended): To build GA directly with MPI, the user needs to define environmental variables *MPI_LIB* and *MPI_INCLUDE* which should point to the location of the MPI library and include directories. Additionally, the make/environmental variable MSG_COMMS must be defined as MSG_COMMS = MPI. (In csh/tcsh, `setenv MSG_COMMS MPI`)

2. GA with TCGMSG-MPI: To build GA with the TCGMSG-MPI, 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
```

3. GA with TCGMSG: To build GA directly with TCGMSG, the user must define the environmental variable MSG_COMMS=TCGMSG.

*Note:* When MSG_COMMS=TCGMSG, make sure to unset the environment variable USE_MPI (e.g. unsetenv USE_MPI).

After chosing the configuration, to build the GA library, type

`make` or `gmake`

If the build is successfull, a test program `test.x` will be created in `global/testing` directory. Refer Section "Running GA programs" on how to run this test.

To build an application based on GA located in `g/global/testing`, 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/config/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.

Interface to ScaLAPACK: GA interface routines to ScaLAPACK are only available, when GA is build with MPI and ScaLAPACK. Before building GA, 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.

Installing GA C++ Bindings: By default, GA C++ bindings are not built. GA++ is built only if GA_C_CORE is defined as follows:
setenv GA_C_CORE y
cd GA_HOME
make clean; make (This will build GA with C core and C++ bindling).

Using GA_C_CORE: GA's internal core is implemented using Fortran and C. When GA_C_CORE is set, core Fortran functionalites are replaced by their C counterparts to eliminate the hassle involved in mixing Fortran and C with C++ bindings on certain platforms or for some compilers (like, missing Fortran symols/libraries during the linking phase). NOTE: C and C++ compilers should be from the same family. GA_C_CORE doesnot support mixing C and C++ compilers (e.g.using Intel compiler for C and GNU compiler for C++).
make FC=ifort CC=icc CXX=g++ (not supported if GA_C_CORE is set)
make FC=ifort CC=icc CXX=icpc (Intel compiler family - supported)

### 2.3.2 Windows NT

To buid GA on Windows NT, MS Power Fortran 4 or DEC Visual Fortran 5 or later, and MS Visual C 4 or later are needed. Other compilers might need the default compilation flags modified. When commercial Windows compilers are not available, one can choose to use CYGNUS or INTERIX and build it as any other Unix box using GNU compilers.

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
```

In addition, the HPVM package from UCSD offers the GA interface in the NT/Myrinet cluster environment.

GA could be built on Windows 95/98. 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 program file into the `g/global/testing` directory. The existing GNU make suffix rules would build an executable with the ".x" suffix from any C or Fortran source file. You do not have to modify makefiles in `g/global/testing` at all. For example, if your program is contained in `myfile.c` or `myfile.F` and you place it in that directory, all you need to do to create an executable called `myfile.x` is to type: `make myfile.x`.

Windows *nmake* is not as powerful as GNU make - you would need to modify the NT makefile.

This approach obviously is not feasible for large packages that contain multiple source files and directories. In that case you need to provide apropriate definitions in your makefile:

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

- follow compilation flags for the GA test programs in GNU and Windows makefiles `g/config/makefile.h`. The recommended approach is to include g/config/makefile.h in your makefile.

Starting with GA 3.1, one could simplify linking of applications by including `g/armci/config/makecoms.h` and `g/armci/config/makemp.h` that define all the necessary platform specific libraries that are required by GA.

## 2.4 Running GA Programs

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

1. On MPPs, such as Cray T3E, or IMB 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
   ```

2. 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 `'parallel'` is accessible (you might copy it into your 'bin' directory). To execute, type:

```
parallel app.x
```

3. 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 comple, 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 completes.

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 executed in a 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,*
- manages 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 the availability of shared memory. A part of GA initialization involves initialization of the ARMCI run-time library. ARMCI dynamically determines if the program can use shared memory based on the architecture type and 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, a programmer who cares about the 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 operaion 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:

```
Fortran  subroutine ga_initialize()
C        void GA_Initialize()
C++      void GA::Initialize(int argc, char **argv)
```

and

```
Fortran  subroutine ga_initialize_ltd(limit)
C        void GA_Initialize_ltd(size_t limit)
C++      void GA::Initialize(int argc, char **argv, size_t 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.

**Note:** In GA++, there is an additional functionality as follows:

```
C++      void GA::Initialize(int argc, char *argv[], unsigned long
heapSize, unsigned long stackSize, int type, size_t limit=0)
```

### 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 - any calls to allocate new arrays that would simply fail (return false). There are two ways to set the limit:

1. at initialization time by calling `ga_initialize_ltd`, or
2. after initialization by calling the function

```
Fortran  subroutine ga_set_memory_limit(limit)
C        void GA_Set_memory_limit(size_t limit)
C++      void GA::GAServices::setMemoryLimit(size_t 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

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

```
Fortran  subroutine ga_error(message, code)
C        void GA_Error(char *message, int code)
C++      void GA::GAServices::error(char *message, int code)
```

## 3.5 Creating arrays - I

There are three ways to create new arrays:

1. From scratch, for regular distribution, using

```
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,
                         chunk1, chunk2, g_a)
C            int NGA_Create(int type, int ndim, int dims[], char
*array_name,
                         int chunk[])
C++          GA::GlobalArray* GA::GAServices::createGA(int type, int
ndim,
                         int dims[], char *array_name, int chunk[])
```

or for regular distribution, using

```
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)
C            int NGA_Create_irreg(int type, int ndim, int dims[],
C++          GA::GlobalArray* GA::GAServices::createGA(int type, int
```

```
ndim,
                                    int dims[], char *array_name, int map[],
int block[])
```

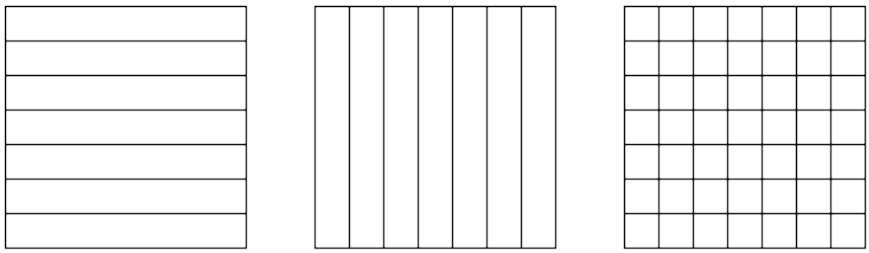
2. Based on a template (an existing array) with the function

```
   Fortran  logical function ga_duplicate(g_a, g_b, array_name)
   C        int GA_Duplicate(int g_a, char *array_name)
   C++      int GA::GAServices::duplicate(int g_a, char *array_name) - or
-
   C++      GA::GlobalArray* GA::GAServices::createGA(int g_a, char
*array_name)
```

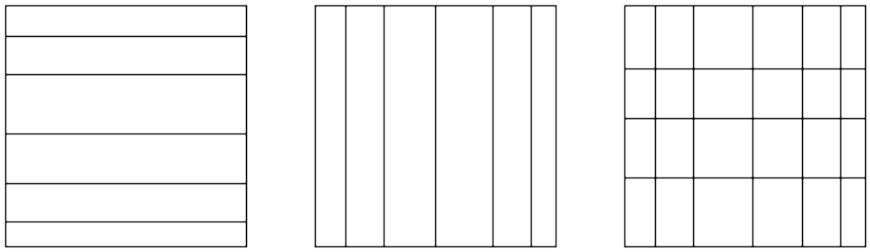
3. Refer "Creating arrays - II" section.

In this case, the new array inherits 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 using *map* array argument. The library creates an array with the overall distribution that is a Cartesian product of distributions for each dimension. A specific example is given in the documentation.



If an array cannot be created, for example due to memory shortages or an enforced memory consumption limit, these calls return failure status. Otherwise an integer handle is returned. This handle represents a global array object in all operations involving that array. This is the only piece of information the programmer needs to store for that array. 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 information explicitly in the application code.

Note that regardless of the distribution type at most one block can be owned/assigned to a processor.

### 3.5.1 Creating Arrays with Ghost Cells

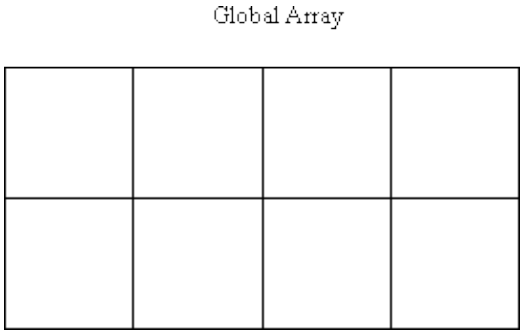
Individual processors ordinarily only hold the portion of global array data that is represent by the lo and hi index arrays returned by a call to nga_distribution or that have been set using the nga_create_irreg call. However, it is possible to create global arrays where this data is padded by a boundary region of array elements representing portions of the global array residing on other processors. These boundary regions can be updated with data from neighboring processors by a call to a single GA function. To create global arrays with these extra data elements, referred to in the following as ghost cells, the user needs to call either the functions:

```
   n-d Fortran  logical function nga_create_ghosts(type, dims, width,
array_name,
                                                  chunk, g_a)
   C            int int_NGA_Create_ghosts(int type, int ndim, int dims[],
int width[],
                                          char *array_name, int chunk[])
```

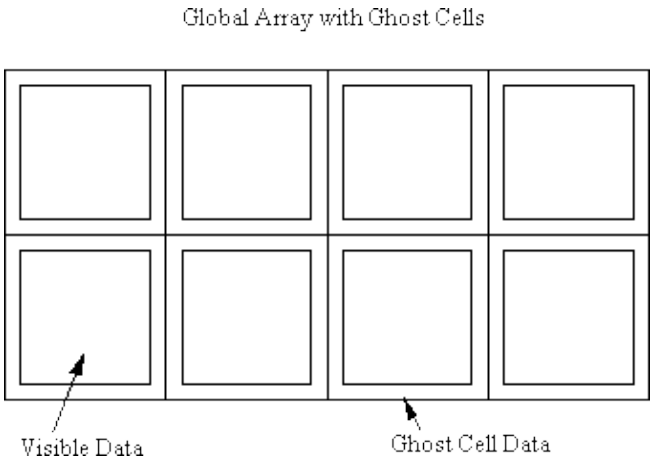
```
   C++          int GA::GAServices::createGA_Ghosts(int type, int ndim,
int dims[],
                                        int width[], char *array_name, int
chunk[])

   n-d Fortran  logical function nga_create_ghosts_irreg(type, dims,
width,
                                        array_name, map, block, g_a)
   C            int int_NGA_Create_ghosts_irreg(int type, int ndim, int
dims[],
                         int width[], char *array_name, int map[], int
block[])
   C++          int GA::GAServices::createGA_Ghosts(int type, int ndim,
int dims[],
                         int width[], char *array_name, int map[], int
block[])
```

These two functions are almost identical to the nga_create and nga_create_irreg functions described above. The only difference is the parameter array width. This is used to control the width of the ghost cell boundaries in each dimension of the global array. Different dimensions can be padded with different numbers of ghost cells, although it is expected that for most applications the widths will be the same for all dimensions. If the width has been set to zero for all dimensions, then these two functions are completely equivalent to the functionsnga_create and nga_create_irreg.
To illustrate the use of these functions, an ordinary global array is shown below. The boundaries represent the data that is held on each processor.



For a global array with ghost cells, the data distribution can be visualized as follows:




Each processor holds “visible” data, corresponding to the data held on each processor of an ordinary global array, and “ghost cell” data, corresponding to neighboring points in the global array that would ordinarily be held on other processors. This data can be updated in a single call to nga_update, described under the collective operations section of the user documentation. Note that the ghost cell data duplicates some portion of the data in the visible portion of the global array. The advantage of having the ghost cells is that this data ordinarily resides on other processors and can only be retrieved using additional calls. To access the data in the ghost cells, the user must use the nga_access_ghosts function described in Section 6.1.

## 3.6 Creating arrays - II

As mentioned in the previous section ("Creating arrays - I"), there are 3 ways to create arrays. This section describes method #3 to create arrays. Because of the increasingly varied ways that global arrays can be configured, a set of new interfaces for creating global arrays has been created. This interface supports all the configurations that were accessible via the old ga_create_XXX calls, as well as new options that can only be accessed using the new interface. Creating global arrays using the new interface starts by a call to ga_create_handle that returns the user a new global array handle. The user then calls several ga_set_XXX calls to assign properties to this handle. These properties include the dimension of the array, the data type, the size of the array, and any other properties that may be relevant. At present, the available ga_set_XXX calls largely reflect properties that are accessible via the nga_create_XXX calls, however, it is anticipated that the range of properties that can be set using these calls will expand considerably in the future. After all the properties have been set, the user calls ga_allocate on the array handle and memory is allocated for the array.

The array can now be used in exactly the same way as arrays created using the traditional ga_create_XXX calls. The calls for obtaining a new global array handle are

```
n-d Fortran  integer function ga_create_handle()
C            int GA_Create_handle()
```

Properties of the global arrays can be set using the ga_set_XXX calls. Note that the only required call is to ga_set_data. The others are all optional.

```
n-d Fortran  subroutine ga_set_data(g_a, ndim, dims, type)
C            void GA_Set_data(int g_a, int ndim, int *dims, int type)
```

The argument g_a is the global array handle, ndim is the dimension of the array, dims is an array of ndim numbers containing the dimensions of the array, and type is the data type as defined in either the macdecls.h or mafdecls.h files. Other options that can be set using these subroutines are:

```
n-d Fortran  subroutine ga_set_array_name(g_a, array_name)
C            void GA_Set_array_name(int g_a, char *array_name)
```

This subroutine assigns a character string as an array name to the global array.

```
n-d Fortran  subroutine ga_set_chunk(g_a, chunk)
C            void GA_Set_chunk(int g_a, int *chunk)
```

The chunk array contains the minimum size dimensions that should be allocated to a single processor. If the minimum size is set to -1 for some of the dimensions, then the minimum size allocation is left to the GA toolkit. The default setting of the chunk array is -1 along all dimensions.

```
n-d Fortran  subroutine ga_set_irreg_distr(g_a, map, block)
C            void GA_Set_irreg_distr(int g_a, int *map, int *block)
```

The ga_set_irreg_distr subroutine can be used to specify the distribution of data among processors. The block array contains the processor grid used to lay out the global array and the map array contains a list of the first indices of each block along each of the array axes. If the first value in the block array is M, then the first M values in the map array are the first indices of each data block along the first axis in the processor grid. Similarly, if the second value in the block array is N, then the values in the map array from M+1 to M+N are the first indices of the each data block along the second axis and so on through the D dimensions of the global array.

**n-d Fortran** subroutine ga_set_ghosts(g_a, width)
**C** void GA_Set_ghosts(int g_a, int *width)

This call can be used to set the ghost cell width along each of the array dimensions.

**n-d Fortran** subroutine ga_set_pgroup(g_a, p_group)
**C** void ga_set_pgroup(int g_a, int p_group)

This call assigns a processor group to the global array. If no processor group is assigned to the global array, it is assumed that the global array is created on the default processor group.

After all the array properties have been set, memory for the global array is allocated by a call to ga_allocate. After this call, the global array is ready for use inside the parallel application.

**n-d Fortran** logical function ga_allocate(g_a)
**C** int GA_Allocate(int g_a)

This function returns a logical variable that is true if the global array was successfully allocated and false otherwise.

## 3.7 Destroying arrays

Global arrays can be destroyed by calling the function

```
Fortran  subroutine ga_destroy(g_a)
C        void GA_Destroy(int g_a)
C++      void GA::GlobalArray::destroy()
```

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 Communication 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. Moreover, since the GA interface uses global array indices to reference nonlocal data, the calling process does not even have to know process ids and location in memory where the refernenced 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, ga_scatter_acc` |
| 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 do 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

```
   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)
   C            void NGA_Put(int g_a, int lo[], int hi[], void *buf, int
ld[])
   C++          void GA::GlobalArray::put(int lo[], int hi[], void *buf,
int 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. It is

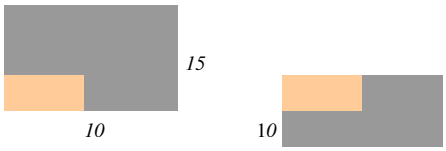
```
   n-D Fortran  subroutine nga_get(g_a, lo, hi, buf, ld)
   2-D Fortran  subroutine ga_get(g_a, ilo, ihi, jlo, jhi, buf, ld)
   C            void NGA_Get(int g_a, int lo[], int hi[], void *buf, int
ld[])
   C++          void GA::GlobalArray::get(int lo[], int hi[], void *buf,
int ld[])
```

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 ocal buffer 5 x10 array we have: (In Fortran notation)

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



## 4.2 Accumulate and read-and-increment

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

```
   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)
   C            void NGA_Acc(int g_a, int lo[], int hi[], void *buf, int
ld[],
                             void *alpha)
   C++          void NGA::GlobalArray::acc(int lo[], int hi[], void *buf,
int ld[],
                             void *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

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

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

```
   n-D Fortran  subroutine nga_read_inc(g_a, subscript, inc)
   2-D Fortran  subroutine ga_read_inc(g_a, i, j, inc)
   C            long NGA_Read_inc(int g_a, int subscript[], long inc)
   C++          long GA::GlobalArray::readInc(int subscript[], long 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 own the referenced elements in the global array.

Scatter puts array elements into a global array, which is

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

It performs (in C notation)

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

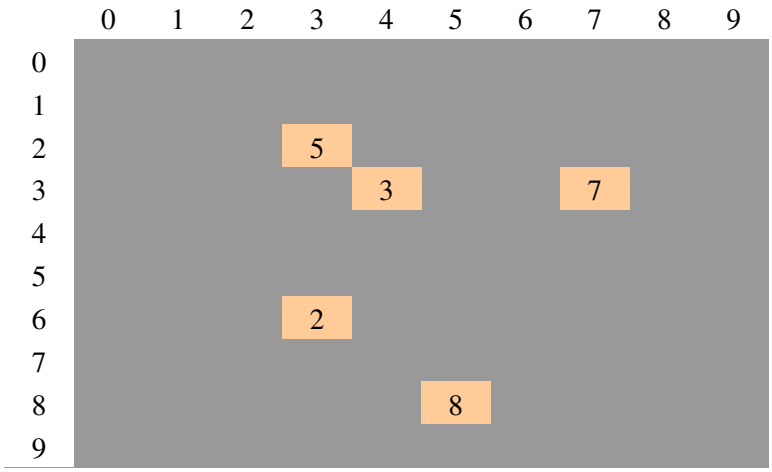
*Example:*

Scatter the 5 elements into a 10x10 global array

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

```
                                subsArray[3][1] = 7
    Element 5     v[4] = 2      subsArray[4][0] = 6
                                subsArray[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.

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

It performs (in C notation)

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

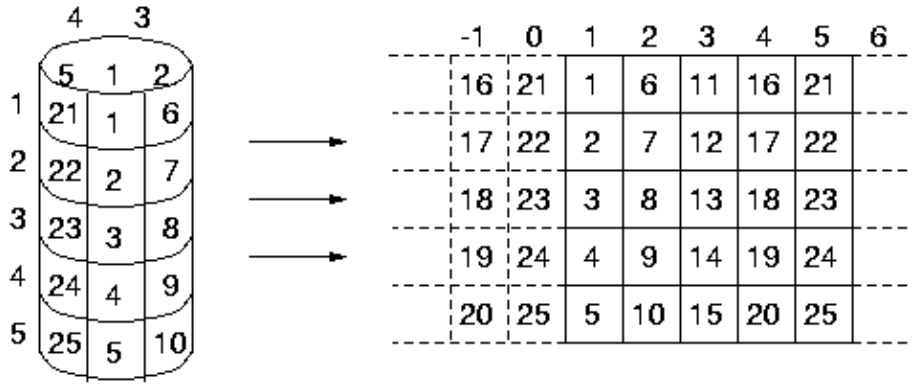
## 4.4 Periodic Interfaces

Periodic interfaces to the one-sided operations have been added to Global Arrays in **version 3.1** to support some computational fluid dynamics problems on multidimensional grids. They provide an index translation layer that allows to use put,get, and accumulate operations possibly extending beyond the boundaries of a global array. The references that are outside of the boundaries are wrapped up inside the global array. To better illustrate these operations, look the following example:

*Example:*
Assume a two dimensional global array g_a with dimensions 5 X 5.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 6 | 11 | 16 | 21 |
| 2 | 2 | 7 | 12 | 17 | 22 |
| 3 | 3 | 8 | 13 | 18 | 23 |
| 4 | 4 | 9 | 14 | 19 | 24 |
| 5 | 5 | 10 | 15 | 20 | 25 |

To access a patch [2:4,-1:3], one can assume that the array is wrapped over in the second dimension, as shown in the following figure




Therefore the patch [2:4, -1:3] is

| | | | | |
|---|---|---|---|---|
| 17 | 22 | 2 | 7 | 12 |
| 18 | 23 | 3 | 8 | 13 |
| 19 | 24 | 4 | 9 | 14 |

Periodic operations extend the boudary of each dimension in two directions, toward lower bound and torward the upper bound. For any dimension with *lo(i)* to *hi(i)*, where $1 < i < ndim$, it extends the range

from

[*lo(i) : hi(i)*]

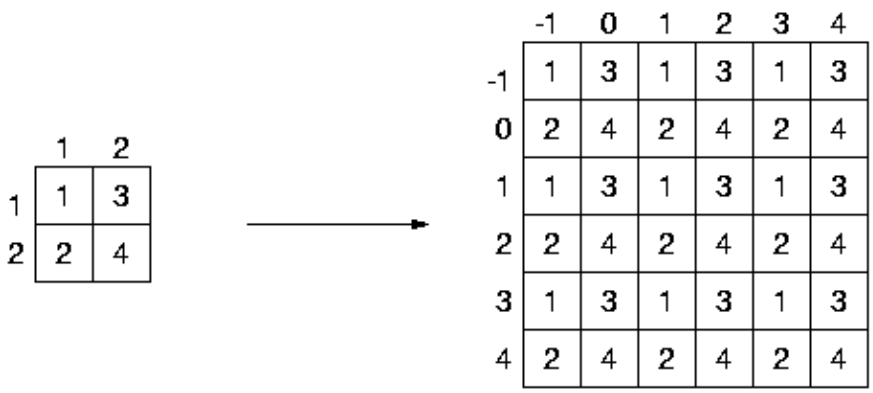
to

[*(lo(i)-1-(hi(i)-lo(i)+1)) : (lo(i)-1)*], [*lo(i) : hi(i)*], and [*(hi(i)+1) : (hi(i)+1+(hi(i)-lo(i)+1))*], or [*(lo(i)-1-(hi(i)-lo(i)+1)) : (hi(i)+1+(hi(i)-lo(i)+1))*].

Even though the patch span in a much large range, the length must always be less, or equals to *(hi(i)-lo(i)+1)).*

*Example:*

For a 2 x 2 array as shown in the following figure, where the dimensions are [1:2, 1:2], periodic operations would look the range of each dimensions as [-1:4, -1:4].




Current version of GA supports three periodic operations. They are

- periodic get,
- periodic put, and
- periodic acc.

*Periodic Get* copies data from a global array section to a local array, which is almost the same as regular *get*, except the indices of the patch can be outside the boundaries of each dimension.

```
   Fortran  subroutine nga_periodic_get(g_a, lo, hi, buf, ld)
   C        void NGA_Periodic_get(int g_a, int lo[], int hi[], void *buf,
int ld[])
   C++      void GA::GlobalArray::periodicGet(int lo[], int hi[], void
*buf, int ld[])
```

Similar to regular *get*, `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`.

*Example:*
Let us look at the first example in this section. It is 5 x 5 two dimensional global array. Assume that the local buffer is an 4x3 array.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 6 | 11 | 16 | 21 |
| 2 | 2 | 7 | 12 | 17 | 22 |
| 3 | 3 | 8 | 13 | 18 | 23 |
| 4 | 4 | 9 | 14 | 19 | 24 |
| 5 | 5 | 10 | 15 | 20 | 25 |

Also ssume that

```
lo[0] = -1, hi[0] = 2,
lo[1] = 4,  hi[1] = 6,  and
ld[0] = 4
```

After the periodic get, the local buffer `buf` would be

```
19  24  4
20  25  5
16  21  1
17  22  2
```

*Periodic Put* is the reverse operations of *Periodic Get*. It copies data from the local array to the global array section, which is

```
   Fortran  subroutine nga_periodic_put(g_a, lo, hi, buf, ld)
   C        void NGA_Periodic_put(int g_a, int lo[], int hi[], void *buf,
int ld[])
   C++      void GA::GlobalArray::periodicPut(int lo[], int hi[], void
*buf, int ld[])
```

Similar to regular *put*, `lo` and `hi` specify where the data should go in the global array; `ld` specifies the stride information of the local array `buf`.

*Periodic Put/Get* (also include the *Accumulate*, which will be discussed later in this section) divide the patch into several smaller patches. For those smaller patches that are outside the global aray, adjust the indices so that they rotate back to the original array. After that call the regular *Put/Get/Accumulate*, for each patch, to complete the operations.

*Example:*
Look at the example for periodic get. Because it is a 5 x 5 globla array, the valid indices for each dimension are

```
dimension 0: [1 : 5]
dimension 1: [1 : 5]
```

The specified `lo` and `hi` are apparently out of the range of each dimension:

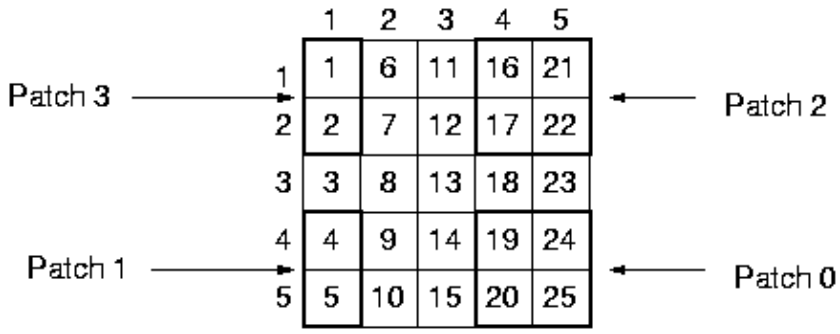
```
dimemsion 0: [-1 : 2]  -->  [-1 : 0]   -- wrap back --> [4 : 5]
                            [ 1 : 2]   ok

dimension 1: [ 4 : 6]  -->  [ 4 : 5]   ok
                            [ 6 : 6]   -- wrap back --> [1 : 1]
```

Hence, there will be four smaller patches after the adjustment. They are

```
patch 0:  [4 : 5, 4 : 5]
patch 1:  [4 : 5, 1 : 1]
patch 2:  [1 : 2, 4 : 5]
patch 3:  [1 : 2, 1 : 1]
```

as shown in the following figure




Of course the destination addresses of each samller patch in the local buffer also need to be calculated.

Similar to regular *Accumulate*, *Periodic Accumulate* combines the data from the local array with data in the global array section, which is

```
   Fortran  subroutine nga_periodic_acc(g_a, lo, hi, buf, ld, alpha)
   C        void NGA_Periodic_acc(int g_a, int lo[], int hi[], void *buf,
int ld[],
                                  void *alpha)
   C++      void GA::GlobalArray::periodicAcc(int lo[], int hi[], void
*buf, int ld[],
                                  void *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

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

*Example:*
Let us look at the same example as above. There is 5 x 5 two dimensional global array. Assume that the local buffer is an 4x3 array.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 6 | 11 | 16 | 21 |
| 2 | 2 | 7 | 12 | 17 | 22 |
| 3 | 3 | 8 | 13 | 18 | 23 |
| 4 | 4 | 9 | 14 | 19 | 24 |
| 5 | 5 | 10 | 15 | 20 | 25 |

Also ssume that

```
lo[0] = -1, hi[0] = 2,
lo[1] = 4,  hi[1] = 6,  and
ld[0] = 4.
```

The local buffer `buf` is

```
1  5  9
4  6  5
3  2  1
7  8  2
```

and the `alpha = 2.`

After the *Periodic Accumulate* operation, the global array will be

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 3 | 6 | 11 | 22 | 25 |
| 2 | 6 | 7 | 12 | 24 | 38 |
| 3 | 3 | 8 | 13 | 18 | 23 |
| 4 | 22 | 9 | 14 | 21 | 34 |
| 5 | 15 | 10 | 15 | 28 | 37 |

## 4.5 Non-blocking operations

The non-blocking 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 (e.g. nga_nbwait) routine.

The wait function completes a non-blocking one-sided operation locally. 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. Wait operation ensures only local completion. Unlike their blocking counterparts, the nonblocking operations are not ordered with respect to the destination. Performance being one reason, the other reason 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.

***Example:*** *Let us take a simple case for illustration. Say, there are two global arrays i.e. one array stores pressure and the other stores temperature. If there are two computation phases (first phase computes*

*pressure and second phase computes temperature), then we can overlap communication with computation, thus hiding latency.*

```
. . . . . .
nga_get (get_pressure_array)

nga_nbget(initiates data transfer to get temperature_array, and returns immediately)

compute_pressure() /* hiding latency - communication is overlapped with computation */

nga_nbwait(temperature_array - completes data transfer)

compute_temperature()
. . . . . .
```

The non-blocking APIs are derived from the blocking interface by adding a handle argument that identifies an instance of the non-blocking request.

```
   n-D Fortran  subroutine nga_nbput(g_a, lo, hi, buf, ld, nbhandle)
   n-D Fortran  subroutine nga_nbget(g_a, lo, hi, buf, ld, nbhandle)
   n-D Fortran  subroutine nga_nbacc(g_a, lo, hi, buf, ld, alpha,
nbhandle)
  n-D Fortran  subroutine nga_nbwait(nbhandle)

   2-D Fortran  subroutine ga_nbput(g_a, ilo, ihi, jlo, jhi, buf, ld,
nbhandle)
   2-D Fortran  subroutine ga_nbget(g_a, ilo, ihi, jlo, jhi, buf, ld,
nbhandle)
   2-D Fortran  subroutine ga_nbacc(g_a, ilo, ihi, jlo, jhi, buf, ld,
alpha, nbhandle)
  2-D Fortran  subroutine ga_nbwait(nbhandle)


   C            void NGA_NbPut(int g_a, int lo[], int hi[], void *buf,
```

```
int ld[],
                                ga_nbhdl_t* nbhandle)
   C            void NGA_NbGet(int g_a, int lo[], int hi[], void *buf,
int ld[],
                                ga_nbhdl_t* nbhandle)
   C            void NGA_NbAcc(int g_a, int lo[], int hi[], void *buf,
int ld[],
                                void *alpha, ga_nbhdl_t* nbhandle)
   C            int NGA_NbWait(ga_nbhdl_t* nbhandle)


   C++          void GA::GlobalArray::nbPut(int lo[], int hi[], void
*buf, int ld[],
                                            ga_nbhdl_t* nbhandle)
   C++          void GA::GlobalArray::nbGet(int lo[], int hi[], void
*buf, int ld[],
                                            ga_nbhdl_t* nbhandle)
   C++          void GA::GlobalArray::nbAcc(int lo[], int hi[], void
*buf, int ld[],
                                            void *alpha, ga_nbhdl_t*
nbhandle)
   C++          int GA::GlobalArray::NbWait(ga_nbhdl_t* nbhandle)
```

# 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

```
Fortran  logical function ga_create_mutexes(number)
C        int GA_Create_mutexes(int number)
C++      int GA::GAServices::createMutexes(int 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

```
Fortran  logical function ga_destroy_mutexes()
C        int GA_Destroy_mutexes()
C++      int GA::GAServices::destroyMutexes()
```

destroys the set of mutexes created with `ga_create_mutexes`.

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

The functions

```
Fortran  subroutine ga_lock(int mutex)
         subroutine ga_unlock(int mutex)
C        void GA_lock(int mutex)
         void GA_unlock(int mutex)
C++      void GA::GAServices::lock(int mutex)
         void GA::GAServices::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

```
Fortran  subroutine ga_init_fence()
C        void GA_Init_fence()
C++      void GA::GAServices::initFence()
```

Initializes tracing of completion status of data movement operations.

The function

```
Fortran  subroutine ga_fence()
C        void GA_Fence()
C++      void GA::GAServices::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

```
Fortran  subroutine ga_sync()
```

```
C       void GA_Sync()
C++     void GA::GAServices::sync()
```

Sync should be inserted as necessary. With 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

```
Fortran   subroutine ga_zero(g_a)
C         void GA_Zero(int g_a)
C++       void GA::GlobalArray::zero()
```

sets all the elements in the array to zero.

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

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

```
C++       void GA::GlobalArray::fill(void *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

```
Fortran  subroutine ga_scale(g_a, val)
C        voidGA_Scale(int g_a, void *val)
C++      voidGA::GlobalArray::scale(void *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

```
Fortran subroutine ga_copy(g_a, g_b)
C       voidGA_Copy(int g_a, int g_b)
C++     voidGA::GlobalArray::copy(const GA::GlobalArray * g_a)
```

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

For global arrays containing ghost cells, the ghost cell data can be filled in with the corresponding data from neighboring processors using the command

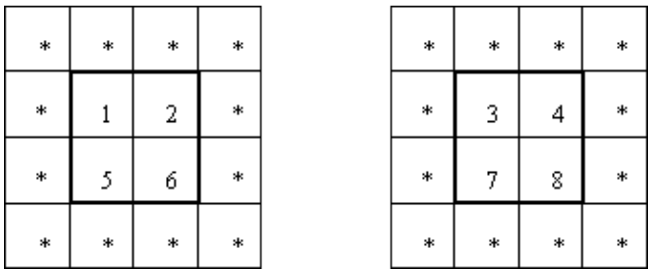
```
n-d Fortran subroutine ga_copy(g_a, g_b)
C           voidGA_Copy(int g_a, int g_b)
C++         voidGA::GlobalArray::copy(const GA::GlobalArray * g_a)
```

```
n-d Fortran subroutine ga_update_ghosts(g_a)
C           voidGA_Update_ghosts(int g_a)
```

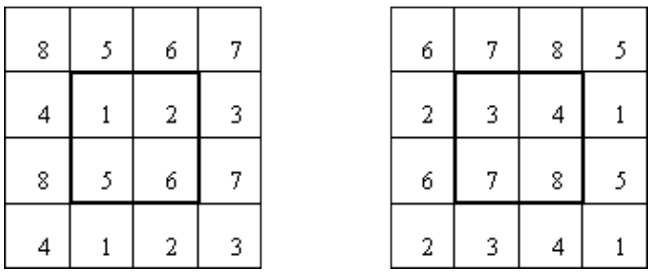
**C++** void GA::GlobalArray::updateGhosts()

This operation updates the ghost cell data by assuming periodic, or wrap-around, boundary conditions similar to those described for the nga_periodic_get operations described above. The wrap-around conditions are always applied, it is up to the individual application to decide whether or not the data in the ghost cells should be used. The update operation is illustrated below for a simple 4x2 global array distributed across two processors. The ghost cells are one element wide in each dimension.

Global Array with Unitialized Ghost Cells




Global Array after Updating Ghost Cells




**n-d Fortran** logical function nga_update_ghosts_dir(g_a, dimension, idir, flag)
**C** int NGA_Update_ghosts_dir(int g_a, int dimension, int idir, int cflag)
**C++** int GA::GlobalArray::updateGhostsDir(int dimension, int idir, int cflag)

This function can be used to update the ghost cells along individual directions.

It is designed for algorithms that can overlap updates with computation. The variable dimension indicates which coordinate direction is to be updated (e.g. dimension = 1 would correspond to the y axis in a two or three dimensional system), the variable idir can take the values +/-1 and indicates whether the side that is to be updated lies in the positive or negative direction, and cflag indicates whether or not the corners on the side being updated are to be included in the update. The following calls would be equivalent to a call to GA_Update_ghosts for a 2-dimensional system:

```
status = NGA_Update_ghost_dir(g_a,0,-1,1);
status = NGA_Update_ghost_dir(g_a,0,1,1);
status = NGA_Update_ghost_dir(g_a,1,-1,0);
status = NGA_Update_ghost_dir(g_a,1,1,0);
```

The variable cflag is set equal to 1 (or non-zero) in the first two calls so that the corner ghost cells are update, it is set equal to 0 in the second two calls to avoid redundant updates of the corners. Note that updating the ghosts cells using several independent calls to the nga_update_ghost_dir functions is generally not as efficient as using GA_Update_ghosts unless the individual calls can be effectively overlapped with computation. This is a collective operation.

### 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

```
Fortran     subroutine nga_zero_patch(g_a, alo, ahi)
C           void NGA_Zero_patch(int g_a, int lo[] int hi[])
C++         void GA::GlobalArray::zeroPatch(int lo[] int hi[])
```

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:

```
  n-DFortran  subroutine nga_fill_patch(g_a, lo, hi, val)
  2-DFortran  subroutine ga_fill_patch(g_a, ilo, ihi, jlo, jhi, val)
  C           voidNGA_Fill_patch(int g_a, int lo[] int hi[], void *val)
  C++         voidGA::GlobalArray::fillPatch(int lo[] int hi[], void
*val)
```

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

The function

```
  n-DFortran  subroutine nga_scale_patch(g_a, lo, hi, val)
  2-DFortran  subroutine ga_scale_patch(g_a, ilo, ihi, jlo, jhi, val)
  C           voidNGA_Scale_patch(int g_a, int lo[] int hi[], void *val)
  C++         voidGA::GlobalArray::scalePatch(int lo[] int hi[], void
*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

```
  n-DFortran  subroutine nga_copy_patch(trans, g_a, alo, ahi,
                                        g_b, blo, bhi)
  2-DFortran  subroutine ga_copy_patch(trans, g_a, ailo, aihi, ajlo,
                                  ajhi, g_b, bilo, bihi, bjlo, bjhi)
  C           void NGA_Copy_patch(char trans, int g_a , int alo[], int
ahi[],
                           int g_b, int blo[], int bhi[])
  C++         voidGA::GlobalArray::copyPatch(char trans, const
GA::GlobalArray* g_a,
                         int alo[], int ahi[], int blo[], int bhi[])
```

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}
```

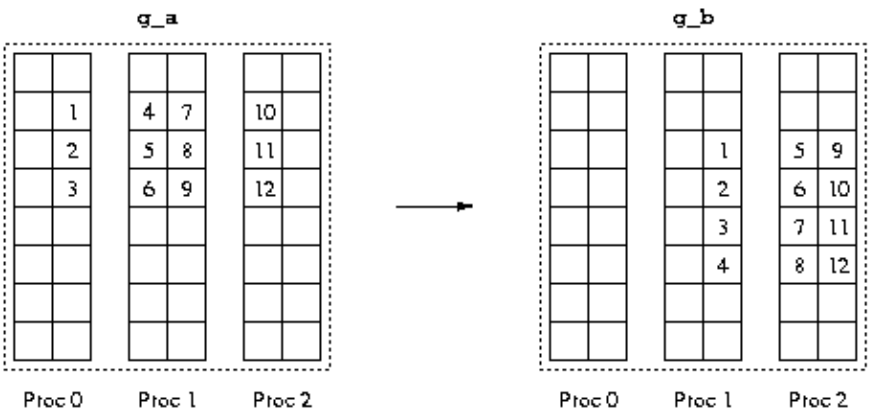
to

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

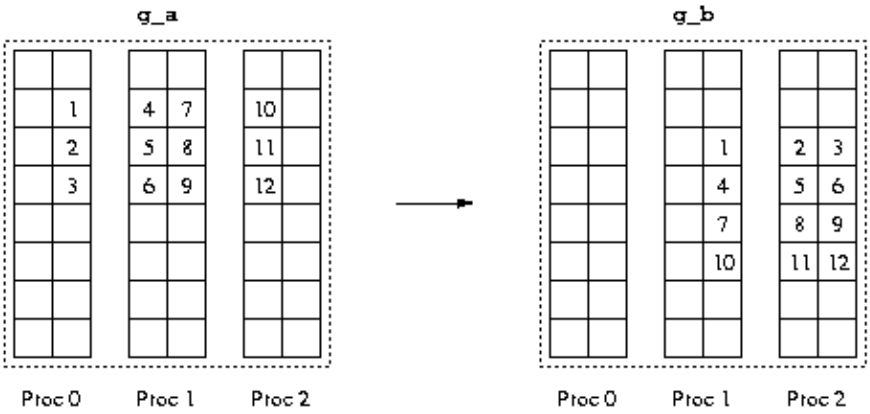
and

```
trans = 0
```

involves reshaping. Iis 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

```
   Fortran subroutine ga_add(alpha, g_a, beta, g_b, g_c)
   C       voidGA_Add(void *alpha, int g_a, void *beta,int g_b, int g_c)
   C++     voidGA::GlobalArray::add(void *alpha, const GA::GlobalArray*
g_a,
                                    void *beta, const GA::GlobalArray* g_b)
```

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_ccan 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

```
Fortran subroutine ga_dgemm(transa, transb, m, n, k,
                            alpha, g_a, g_b, beta, g_c )
C       voidGA_Dgemm(char ta, char tb, int m, int n, int k,
                     double alpha, int g_a, int g_b,
                     double beta, int g_c )
C++     voidGA::GlobalArray::dgemm(char ta, char tb, int m, int n, int
k,
                     double alpha, const GA::GlobalArray* g_a,
                     const GA::GlobalArray* g_b, double beta)
```

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

```
   Fortran  integer          function ga_idot(g_a, g_b)
            double precision functionga_ddot(g_a, g_b)
            double complex   function ga_zdot(g_a, g_b)
   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)
   C++      long           GA::GlobalArray::idot(const GA::GlobalArray*
g_a)
            double         GA::GlobalArray::ddot(const GA::GlobalArray*
g_a)
            DoubleComplex  GA::GlobalArray::zdot(const GA::GlobalArray*
g_a)
```

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

```
   Fortran   subroutine ga_symmetrize(g_a)
   C         void GA_Symmetrize(int g_a)
   C++       void GA::GlobalArray::symmetrize()
```

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

The function

```
   Fortran  subroutine ga_transpose(g_a, g_b)
   C        void GA_Transpose(int g_a, int g_b)
   C++      void GA::GlobalArray::transpose(const GA::GlobalArray* g_a)
```

transposes a matrix: $B = A'$.

### 6.2.2 Patches

The functions

```
   n-DFortran  subroutine nga_add_patch(alpha, g_a, alo, ahi,
                                        beta,  g_b, blo, bhi,
                                               g_c, clo, chi)
   2-DFortran  subroutine ga_add_patch(alpha, g_a, ailo, aihi, ajlo,
ajhi,
                                       beta,  g_b, bilo, bihi, bjlo,
bjhi,
                                              g_c, cilo, cihi, cjlo,
cjhi)
   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[])
   C++         void GA::GlobalArray::addPatch(void *alpha, const
GA::GlobalArray* g_a,
                                  int alo[], int ahi[], void *beta,  const
GA::GlobalArray* g_b,
                                  int blo[], int bhi[], int clo[], int
chi[])
```

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

```
   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)
   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)
   C++     void GA::GlobalArray::matmulPatch(char *transa, char* transb,
void* alpha, void *beta,
                            const GlobalArray * g_a, int ailo, int aihi,
int ajlo, int ajhi,
                            const GlobalArray * g_b, int bilo, int bihi,
int bjlo, int bjhi,
                            int cilo, int cihi, int cjlo, int 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

$$op( X ) = X \quad or \quad op( X ) = X',$$

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

```
n-DFortran  integer function nga_idot_patch(g_a, ta, alo, ahi,
                                            g_b, tb, blo, bhi)
            double precision functionnga_ddot_patch(g_a, ta, alo, ahi,
                                            g_b, tb, blo, bhi)
            double complex functionnga_zdot_patch(g_a, ta, alo, ahi,
                                            g_b, tb, blo, bhi)

2-DFortran  integer function ga_idot_patch(g_a, ta, ailo, aihi,
                     ajlo, ailo, g_b, tb, bilo, bihi, bjlo, bjhi)
            double precision functionga_ddot_patch(g_a, ta, ailo,
aihi,
                     ajlo, ailo, g_b, tb, bilo, bihi, bjlo, bjhi)
            double complex functionga_zdot_patch(g_a, ta, ailo, aihi,
                     ajlo, ailo, g_b, tb, bilo, bihi, bjlo, bjhi)

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[])

C++         IntegerGA::GlobalArray::idotPatch(const GA::GlobalArray*
g_a,
                                   char* ta, int alo[], int ahi[],
                                   char* tb, int blo[], int bhi[])
            double GA::GlobalArray::ddotPatch(const GA::GlobalArray*
g_a,
                                   char* ta, int alo[], int ahi[],
                                   char* tb, int blo[], int bhi[])
            DoubleComplex GA::GlobalArray::zdotPatch(const
GA::GlobalArray* g_a,
                                   char* ta, int alo[], int ahi[],
```

```
                                        char* tb, int blo[], int bhi[])
```

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.

### 6.2.3 Element-wise operations

These operations work on individual array elements rather than arrays as matrices in the sense of linear algebra operations. For example multiplication of elements stored in arrays is a completely different operation than matrix multiplication.

**Fortran** subroutine ga_abs_value(g_a)
**C** void GA_Abs_value(int g_a)
**C++** void GA::GlobalArray::absValue(int g_a)

Take element-wise absolute value of the array.

**Fortran** subroutine ga_abs_value_patch(g_a, lo, hi)
**C** void GA_Abs_value_patch(int g_a, int lo[], int hi[])
**C++** void GA::GlobalArray::absValuePatch(int lo[], int hi[])

Take element-wise absolute value of the patch.

**Fortran** subroutine ga_add_constant(g_a, alpha)
**C** void GA_Add_constant(int g_a, void* alpha)
**C++** void GA::GlobalArray::addConstant(void* alpha)

Add the contant pointed by alpha to each element of the array.

**Fortran** subroutine ga_add_constant_patch(g_a, lo, hi, alpha)
**C** void GA_Add_constant_patch(int g_a, int lo[], int hi[], void*alpha)

**C++** void GA::GlobalArray::addConstantPatch(void* alpha)

Add the contant pointed by alpha to each element of the patch.

**Fortran** subroutine ga_recip(g_a)
**C** void GA_Recip(int g_a)
**C++** void GA::GlobalArray::recip()

Take element-wise reciprocal of the array.

**Fortran** subroutine ga_recip_patch(g_a, lo, hi)
**C** void GA_Recip_patch(int g_a, int lo[], int hi[])
**C++** void GA::GlobalArray::recipPatch(int lo[], int hi[])

Take element-wise reciprocal of the patch.

**Fortran** subroutine ga_elem_multiply(g_a, g_b, g_c)
**C** void GA_Elem_multiply(int g_a, int g_b, int g_c)
**C++** void GA::GlobalArray::elemMultiply(const GA::GlobalArray * g_a,
const GA::GlobalArray * g_b)

Computes the element-wise product of the two arrays
which must be of the same types and same number of
elements. For two-dimensional arrays,

c(i, j) = a(i,j)*b(i,j)

The result (c) may replace one of the input arrays (a/b).

**Fortran** subroutine ga_elem_multiply_patch(g_a, alo, ahi, g_b, blo, bhi, g_c, clo,chi)
**C** void GA_Elem_multiply_patch(int g_a, int alo[], int ahi[], int g_b, int blo[],
int bhi[], int g_c, int clo[], int chi[])
**C++** void GA::GlobalArray::elemMultiplyPatch( const GA::GlobalArray * g_a,
int alo[], int ahi[],

const GA::GlobalArray * g_b, int blo[],
int bhi[], int clo[], int chi[])

Computes the element-wise product of the two patches
which must be of the same types and same number of
elements. For two-dimensional arrays,

c(i, j) = a(i,j)*b(i,j)

The result (c) may replace one of the input arrays (a/b).

**Fortran** subroutine ga_elem_divide(g_a, g_b, g_c)
**C** void GA_Elem_divide(Integer g_a, Integer g_b, Integer g_c)
**C++** void GA::GlobalArray::elemDivide(const GA::GlobalArray * g_a,
const GA::GlobalArray * g_b)

Computes the element-wise quotient of the two arrays
which must be of the same types and same number of
elements. For two-dimensional arrays,

c(i, j) = a(i,j)/b(i,j)

The result (c) may replace one of the input arrays (a/b). If one of the elements of array g_b is zero, the quotient for the element of g_c will be set to
GA_NEGATIVE_INFINITY.

**Fortran** subroutine ga_elem_divide_patch(g_a, alo, ahi, g_b, blo, bhi, g_c, clo, chi)
**C** void GA_Elem_divide_patch(int g_a, int alo[], int ahi[], int g_b, int blo[],
int bhi[], int g_c, int clo[], int chi[])
**C++** void GA::GlobalArray::elemDividePatch( const GA::GlobalArray * g_a,
int alo[], int ahi[],
const GA::GlobalArray * g_b, int blo[],
int bhi[], int clo[], int chi[])

Computes the element-wise quotient of the two patches
which must be of the same types and same number of
elements. For two-dimensional arrays,

c(i, j) = a(i,j)/b(i,j)

The result (c) may replace one of the input arrays (a/b).

**Fortran** subroutine ga_elem_maximum(g_a, g_b, g_c)
**C** void GA_Elem_maximum(Integer g_a, Integer g_b, Integer g_c)
**C++** void GA::GlobalArray::elemMaximum(const GA::GlobalArray * g_a,
const GA::GlobalArray * g_b)

Computes the element-wise maximum of the two arrays
which must be of the same types and same number of
elements. For two dimensional arrays,

c(i, j) = max{a(i,j), b(i,j)}

The result (c) may replace one of the input arrays (a/b).

**Fortran** subroutine ga_elem_maximum_patch(g_a, alo, ahi, g_b, blo, bhi, g_c, clo, chi)
**C** void GA_Elem_maximum_patch(int g_a, int alo[], int ahi[], int g_b, int blo[],
int bhi[], int g_c, int clo[], int chi[])
**C++** void GA::GlobalArray::elemMaximumPatch(const GA::GlobalArray * g_a,
int alo[], int ahi[],
const GA::GlobalArray * g_b, int blo[],
int bhi[], int clo[], int chi[])

Computes the element-wise maximum of the two patches
which must be of the same types and same number of
elements. For two-dimensional of noncomplex arrays,

c(i, j) = max{a(i,j), b(i,j)}

If the data type is complex, then
c(i, j).real = max{ |a(i,j)|, |b(i,j)|} while c(i,j).image = 0.

The result (c) may replace one of the input arrays (a/b).

**Fortran** subroutine ga_elem_minimum(g_a, g_b, g_c)
**C** void GA_Elem_minimum(Integer g_a, Integer g_b, Integer g_c);
**C++** void GA::GlobalArray::elemMinimum(const GA::GlobalArray * g_a,
const GA::GlobalArray * g_b)

Computes the element-wise minimum of the two arrays
which must be of the same types and same number of
elements. For two dimensional arrays,

c(i, j) = min{a(i,j), b(i,j)}

The result (c) may replace one of the input arrays (a/b).

**Fortran** subroutine ga_elem_minimum_patch(g_a, alo, ahi, g_b, blo, bhi, g_c, clo, chi)
**C** void GA_Elem_minimum_patch(int g_a, int alo[], int ahi[], int g_b,
int blo[], int bhi[], int g_c, int clo[], int chi[])
**C++** void GA::GlobalArray::elemMinimumPatch(const GA::GlobalArray * g_a,
int alo[], int ahi[],
const GA::GlobalArray * g_b, int blo[],
int bhi[], int clo[], int chi[])

Computes the element-wise minimum of the two patches
which must be of the same types and same number of
elements. For two-dimensional of noncomplex arrays,

c(i, j) = min{a(i,j), b(i,j)}

If the data type is complex, then
c(i, j).real = min{ |a(i,j)|, |b(i,j)|} while c(i,j).image = 0.

The result (c) may replace one of the input arrays (a/b).

**Fortran** subroutine ga_shift_diagonal(g_a, c)
**C** void GA_Shift_diagonal(int g_a, void *c)
**C++** void GA::GlobalArray::shiftDiagonal(void *c)

Adds this constant to the diagonal elements of the matrix.

**Fortran** subroutine ga_set_diagonal(g_a, g_v)
**C** void GA_Set_diagonal(int g_a, int g_v)
**C++** void GA::GlobalArray::setDiagonal(const GA::GlobalArray * g_v)

Sets the diagonal elements of this matrix g_a with the elements of the vector g_v.

**Fortran** subroutine ga_zero_diagonal( g_a)
**C** void GA_Zero_diagonal(int g_a)
**C++** void GA::GlobalArray::zeroDiagonal()

Sets the diagonal elements of this matrix g_a with zeros.

**Fortran** subroutine ga_add_diagonal(g_a, g_v)
**C** void GA_Add_diagonal(int g_a, int g_v)
**C++** void GA::GlobalArray::addDiagonal(const GA::GlobalArray * g_v)

Adds the elements of the vector g_v to the diagonal of this matrix g_a.

**Fortran** subroutine ga_get_diag(g_a, g_v)
**C** void GA_Get_diag(int g_a, int g_v)
**C++** void GA::GlobalArray::getDiagonal(const GA::GlobalArray * g_v)

Inserts the diagonal elements of this matrix g_a into the vector g_v.

**Fortran** subroutine ga_scale_rows( g_a, g_v)
**C** void GA_Scale_rows(int g_a, int g_v)
**C++** void GA::GlobalArray::scaleRows(const GA::GlobalArray * g_v)

Scales the rows of this matrix g_a using the vector g_v.

**Fortran** subroutine ga_scale_cols(g_a, g_v)
**C** void GA_Scale_cols(int g_a, int g_v)
**C++** void GA::GlobalArray::scaleCols(const GA::GlobalArray * g_v)

Scales the columns of this matrix g_a using the vector g_v.

**Fortran** subroutine ga_norm1(g_a, nm)
**C** void GA_Norm1(int g_a, double *nm)
**C++** void GA::GlobalArray::norm1(double *nm)

Computes the 1-norm of the matrix or vector g_a.

**Fortran** subroutine ga_norm_infinity(g_a, nm)
**C** void GA_Norm_infinity(int g_a, double *nm)
**C++** void GA::GlobalArray::normInfinity(double *nm)

Computes the 1-norm of the matrix or vector g_a.

**Fortran** subroutinega_median( g_a, g_b, g_c, g_m)
**C** void GA_Median(int g_a, int g_b, int g_c, int g_m)
**C++** void GA::GlobalArray::median(const GA::GlobalArray * g_a,
const GA::GlobalArray * g_b,
const GA::GlobalArray * g_c)

Computes the componentwise Median of three arrays g_a, g_b, and g_c, and
stores the result in this array g_m. The result (m) may replace one of the input arrays (a/b/c).

**Fortran** subroutine ga_median_patch(g_a, alo, ahi, g_b, blo, bhi, g_c,
clo, chi, g_m,mlo, mhi)
**C** void GA_Median_patch(int g_a, int alo[], int ahi[], int g_b, int blo[],
int bhi[], int g_c, int clo[], int chi[], int g_m,
int mlo[],int mhi[])
**C++** void GA::GlobalArray::medianPatch(const GA::GlobalArray * g_a, int alo[], int ahi[],
const GA::GlobalArray * g_b, int blo[], int bhi[],
const GA::GlobalArray * g_c, int clo[], int chi[],
int mlo[],int mhi[])

Computes the componentwise Median of three patches g_a, g_b, and g_c, and
stores the result in this patch g_m. The result (m) may replace one of the input patches (a/b/c).

**Fortran** subroutine ga_step_max(g_a, g_b, step)
**C** void GA_Step_max(int g_a, int g_b, double *step)
**C++** void GA::GlobalArray::stepMax(const GA::GlobalArray *g_a, double *step)

Calculates the largest multiple of a vector g_b that can be added
to this vector g_a while keeping each element of this vector
nonnegative.

**Fortran** subroutine ga_step_max2( g_xx, g_vv, g_xxll, g_xxuu, step2)
**C** void GA_Step_max2(int g_xx, int g_vv, int g_xxll, int g_xxuu, double *step2)
**C++** void GA::GlobalArray::stepMax2(const GA::GlobalArray *g_vv,
const GA::GlobalArray *g_xxll,
const GA::GlobalArray *g_xxuu, double *step2)

Calculates the largest step size that should be used in a projected bound line search.

**Fortran** subroutine ga_step_max_patch(g_a, alo, ahi, g_b, blo, bhi, step)
**C** void GA_Step_max_patch(int g_a, int *alo, int *ahi, int g_b, int *blo,

int *bhi, double *step)

**C++** void GA::GlobalArray::stepMaxPatch(int *alo, int *ahi, const GA::GlobalArray * g_b,
int *blo, int *bhi, double *step)

Calculates the largest multiple of a vector g_b that can be added
to this vector g_a while keeping each element of this vector
nonnegative.

**Fortran** subroutine ga_step_max2_patch( g_xx, xxlo, xxhi, g_vv,vvlo, vvhi, g_xxll,
xxlllo, xxllhi, g_xxuu, xxuulo, xxuuhi, step2)

**C** void GA_Step_max2_patch(int g_xx, int *xxlo, int *xxhi, int g_vv,
int *vvlo, int *vvhi, int g_xxll, int *xxlllo, int *xxllhi,
int g_xxuu, int *xxuulo, int *xxuuhi, double *step2)

**C++** void GA::GlobalArray::stepMax2Patch(int *xxlo, int *xxhi,
const GA::GlobalArray * g_vv, int *vvlo, int *vvhi,
const GA::GlobalArray * g_xxll, int *xxlllo, int *xxllhi,
const GA::GlobalArray * g_xxuu, int *xxuulo,
int *xxuuhi, double *step2)

Calculates the largest step size that should be used in a projected bound line search.

## 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

```
Fortran  integer function ga_solve(g_a, g_b)
C        int GA_Solve(int g_a, int g_b)
C++      int GA::GlobalArray::solve(const GA::GlobalArray * g_a)
```

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

```
Fortran  integer function ga_llt_solve(g_a, g_b)
C        int GA_Llt_solve(int g_a, int g_b)
C++      int GA::GlobalArray::lltSolve(const GA::GlobalArray * g_a)
```

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

```
Fortran  subroutinega_lu_solve(trans, g_a, g_b)
C        void GA_Lu_solve(char trans, int g_a, int g_b)
C++      void GA::GlobalArray::luSolve(char trans, const
GA::GlobalArray * g_a)
```

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

```
Fortran  integer function ga_spd_invert(g_a)
C        int GA_Spd_invert(int g_a)
C++      int GA::GlobalArray::spdInvert()
```

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 *fanngi@ornl.gov*. Global Arrays use this library to solve eigen-value problems.

The function

```
Fortran  subroutine ga_diag(g_a, g_s, g_v, eval)
C        void GA_Diag(int g_a, int g_s, int g_v, void *eval)
C++      void GA::GlobalArray::diag(const GA::GlobalArray*g_s,
                        const GA::GlobalArray* g_v, void *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

```
Fortran  subroutine ga_diag_reuse(control, g_a, g_s, g_v, eval)
C        void GA_Diag_reuse(int control, int g_a, int g_s, int
g_v,void *eval)
C++      void GA::GlobalArray::diagReuse(int control, const
GA::GlobalArray* g_s,
                                         const GA::GlobalArray*g_v, void
*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

```
   Fortran  subroutine ga_diag_std(g_a, g_v, eval)
   C        void GA_Diag_std(int g_a, int g_v, void *eval)
   C++      void GA::GlobalArray::diagStd( const GA::GlobalArray* g_v,
void *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. Here is the instructions for using PETSc with GA.

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. Here is the instructions for using CUMULVS with GA.

## 6.4 Synchronization Control in Collective Operations

GA collective array operations are implemented by exploiting locality information to minimize or even completely avoid interprocessor communication or data copying. Before each processor accesses its own portion of the GA data we must assure that the data is in a consistent state. That means that there are no outstanding communication operations targeting that given global array portion pending while the data owner is accessing it. To accomplish that the GA collective array operations have implicit synchronization points: at the beginning and at the end of the operation. However, in many cases when collective array operations are called back-to-back or if the user does an explicit sync just before a collective array operation, some of the

internal synchronization points could be merged or even removed if user can guarantee that the global array data is in the consistent state. The library offers a call for the user to eliminate the redundant synchronization points based on his/her knowledge of the application.

The function

```
  Fortran  subroutine ga_mask_sync(prior_sync_mask,post_sync_mask)
  C      void GA_Mask_sync(int prior_sync_mask,int post_sync_mask)
  C++    void GA::GlobalArray::maskSync(int prior_sync_mask, int
post_sync_mask)
```

This operation should be used with a lot of care and only when the application code has been debugged and the user wishes to tune its performance. Making a call to this function with prior_sync_mask parameter set to false disables the synchronization done at the beginning of first collective array operation called after a call to this function. Similarly, making a call to this function by setting the post_sync_mask parameter to false disables the synchronization done at the ending of the first collective array operation called after a call to this function.

# 7. Utility Operations

Global Arrays include some utility functions to provide process, data locality, 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

```
n-DFortran  logical functionnga_locate(g_a, subscript, owner)
2-DFortran  logical functionga_locate(g_a, i, j, owner)
C         int NGA_Locate(int g_a, int subscript[])
C++       int GA::GlobalArray::locate(int subscript[])
```

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

The function

```
n-DFortran  logical functionnga_locate_region(g_a, lo, hi,
map,proclist, np)
2-DFortran  logical functionga_locate_region(g_a, ilo, ihi, jlo,jhi,
map, np)
C             int NGA_Locate_region(int g_a, int lo[], int hi[],int
*map[], int procs[])
C++           int GA::GlobalArray::locateRegion(int lo[], int hi[],int
*map[], int procs[])
```

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:

1. find the distribution of an array, which part of the data the calling process own
2. access the data
3. operate on the date: read/write
4. release the access to the data

The function

```
   n-DFortran  subroutine nga_distribution(g_a, iproc, lo, hi)
   2-DFortran  subroutine ga_distribution(g_a, iproc, ilo, ihi, jlo, jhi)
   C            void NGA_Distribution(int g_a, int iproc, int lo[], int
hi[])
   C++          void GA::GlobalArray::distribution(int iproc, int lo[],
int hi[])
```

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

The function

```
   n-DFortran  subroutine nga_access(g_a, lo, hi, index, ld)
   2-DFortran  subroutine ga_access(g_a, ilo, ihi, jlo, jhi, index, ld)
   C            void NGA_Access(int g_a, int lo[], int hi[], void *ptr,
int ld[])
   C++          void GA::GlobalArray::access(int lo[], int hi[], void
*ptr, int 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

```
   n-DFortran  subroutine nga_release(g_a, lo, hi)
   2-DFortran  subroutine ga_release(g_a, ilo, ihi, jlo, jhi)
   C            void NGA_Release(int g_a, lo[], int hi[])
   C++          void GA::GlobalArray::release(lo[], int hi[])
```

and

```
   n-DFortran  subroutine nga_release_update(g_a, lo, hi)
   2-DFortran  subroutine ga_release_update(g_a, ilo, ihi, jlo, jhi)
   C            void NGA_Release_update(int g_a, int lo[], int hi[])
   C++          void GA::GlobalArray::releaseUpdate(int lo[], int hi[])
```

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

```
   Fortran  subroutine ga_compare_distr(g_a, g_b)
   C        void NGA_Compare_distr(int g_a, int g_b)
   C++      void GA::GlobalArray::compareDistr(const GA::GlobalArray *
g_a)
```

The only method currently available for accessing the ghost cell data for global arrays that have ghost cell data is to use the nga_access_ghosts funtion. This function is similar to the nga_access function already described, except that it returns an index (pointer) to the origin of the locally held patch of global array data. This local patch includes the ghost cells so the index (pointer) will be pointing to a ghost cell. The nga_access_ghosts function also returns the physical dimensions of the local data patch, which includes the additional ghost cells, so it is possible to access both the visible data of the global array and the ghost cells using this information. The nga_access_ghosts functions have the format

```
   n-d Fortran  subroutine nga_access_ghosts(g_a, dims, index, ld)
   C            void NGA_access_ghosts (int g_a, int dims[], void *ptr,
int ld[])
   C++          void GA::GlobalArray::accessGhosts(int dims[], void *ptr,
int ld[])
```

The array dims comes back with the dimensions of the local data patch, including the ghost cells, for each dimension of the global array, ptr is an index (pointer) identifying the beginning of the local data patch, and ld is any array of leading dimensions fpr the local data patch, which also includes the ghost cells. The array ld is actually redundant since the information in ld is also contained in dims, but is included to maintain continuity

with other GA functions.

### 7.1.1 Process Information

When developing a program, one needs to use charateristics 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

```
Fortran  integer function ga_nodeid()
C        int GA_Nodeid()
C++      int GA::GAServices::nodeid()
```

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

```
Fortran  integer function ga_nnodes()
C        int GA_Nnodes()
C++      int GA::GAServices::nodes()
```

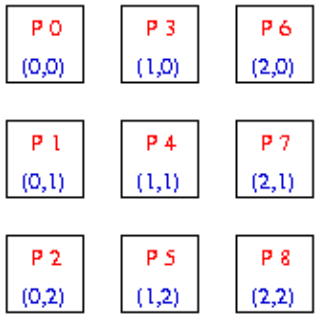
tells the number of computing processes.

The function

```
Fortran  subroutine ga_proc_topology(ga, proc, prow, pcol)
C        void NGA_Proc_topology(int g_a, int proc, int coordinates)
C++      void GA::GlobalArray::procTopology(int proc, int coordinates)
```

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.




### 7.1.2 Cluster Information

The following functions can be used to obtain information like number of nodes that the program is running on, node ID of the process, and other cluster information as discussed below:

The function

```
Fortran  integer function ga_cluster_nnodes()
C        int GA_Cluster_nnodes()
C++      int GA::GAServices::clusterNnodes()
```

returns the total number of nodes that the program is running on. On SMP architectures, this will be less than or equal to the total number of processors.

The function

```
Fortran  integer function ga_cluster_nodeid()
C        int GA_Cluster_nodeid()
C++      int GA::GAServices::clusterNodeid()
```

returns the node ID of the process. On SMP architectures with more than one processor per node, several processes may return the same node id.

The function

```
Fortran  integer function ga_cluster_nprocs(inode)
C        int GA_Cluster_nprocs(int inode)
C++      int GA::GAServices::clusterNprocs(int inode)
```

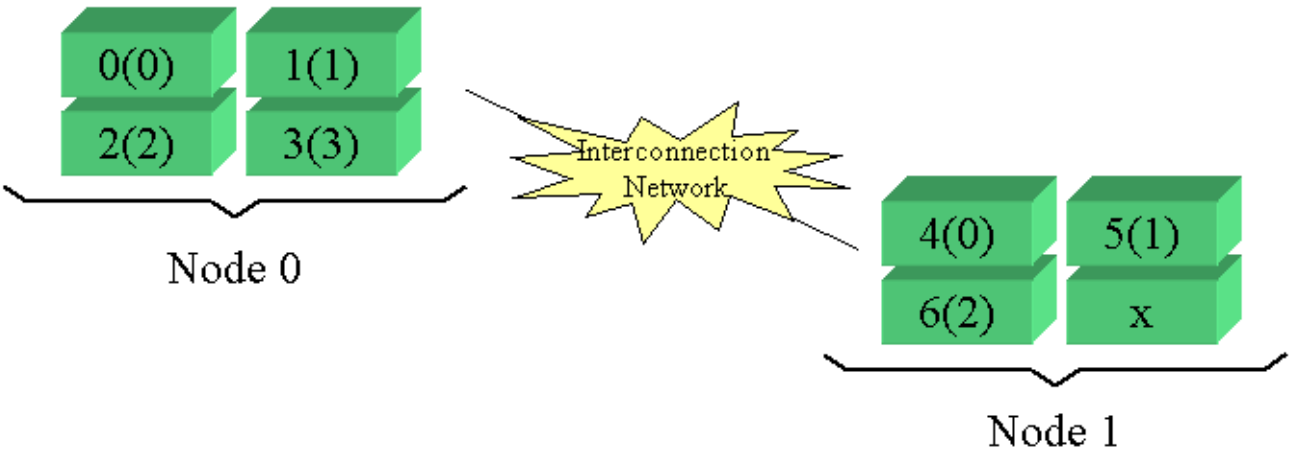
returns the number of processors available on node inode.

The function

```
Fortran  integer function ga_cluster_procid(inode, iproc)
C        int GA_Cluster_procid(int inode, int iproc)
C++      int GA::GAServices::clusterProcid(int inode, int iproc)
```

returns the processor id associated with node inode and the local processor id iproc. If node inode has N processors, then the value of iproc lies between 0 and N-1.

*Example:* 2 nodes with 4 processors each. Say, there are 7 processes created. Assume 4 processes on node 0 and 3 processes on node 1. In this case: number of nodes=2, node id is either 0 or 1 (for example, nodeid of process 2 is 0), number of processes in node 0 is 4 and node 1 is 3. The global rank of each process is shown in the figure and also the local rank (rank of the process within the node.i.e.cluster_procid) is shown in the paranthesis.




## 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:

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

The function

```
Fortran  integer function ga_inquire_memory()
C        size_t GA_Inquire_memory()
C++      size_t GA::GAServices::inquireMemory()
```

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

The function

```
Fortran  integer function ga_memory_avail()
C        size_t GA_Memory_avail()
C++      size_t GA::GAServices::memoryAvailable()
```

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

```
Fortran  logical function ga_uses_ma()
C        int GA_Uses_ma()
C++      int GA::GAServices::usesMA()
```

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

The function

```
Fortran  logical function ga_memory_limited()
C        int GA_Memory_limited()
C++      int GA::GAServices::memoryLimited()
```

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 the process is running with.

The function

```
Fortran  subroutine ga_brdcst(type, buf, lenbuf, root)
C        void GA_Brdcst(void *buf, int lenbuf, int root)
C++      void GA::GAServices::brdcst(void *buf, int lenbuf, int root)
```

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

The functions

```
Fortran  subroutine ga_igop(type, x, n, op)
         subroutine ga_dgop(type, x, n, op)
C        void GA_Igop(long x[], int n, char *op)
         void GA_Dgop(double x[], int n, char *op)
C++      void GA::GAServices::igop(long x[], int n, char *op)
         void GA::GAServices::dgop(double x[], int n, char *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_gence`operatins.

## 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

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

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

The function

```
   Fortran  subroutine ga_inquire_name(g_a, array_name)
   C        char* GA_Inquire_name(int g_a)
   C++      char* GA::GlobalArray::inquireName()
```

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

1. content of the global array
2. content of a patch of global array
3. the status of array operations
4. a summary of allocated arrays

The function

```
   Fortran  subroutine ga_print(g_a)
   C        void GA_Print(int g_a)
   C++      void GA::GlobalArray::print()
```

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

```
   Fortran  subroutine nga_print_patch(g_a, lo, hi, pretty)
   C        void NGA_Print_patch(int g_a, int lo[], int hi[], int pretty)
   C++      void GA::GlobalArray::printPatch(int lo[], int hi[], int
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

```
   Fortran  subroutine ga_print_stats()
   C        void GA_Print_stats()
   C++      void GA::GAServices::printStats()
```

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

```
   Fortran  subroutine ga_print_distribution(g_a)
   C        void GA_Print_distribution(int g_a)
   C        void GA::GlobalArray::printDistribution()
```

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

The function

```
Fortran  subroutine ga_summarize(verbose)
C        void GA_Summarize(int verbose)
C++      void GA::GAServices::summarize(int verbose)
```

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

### 7.4.3 Miscellaneous

The function

```
Fortran  subroutine ga_check_handle(g_a, string)
C        void GA_Check_handle(int g_a, char *string)
C++      void GA::GlobalArray::checkHandle(char *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.

# GA++: C++ Bindings for Global Arrays


## 8.1 Overview

GA++ provides a C++ interface to global arrays (GA) libraries. Here is the doxygen documentation of GA++: http://www.emsl.pnl.gov/docs/global/ga++/index.html The GA C++ bindings are a layer built directly on top of the GA C bindings. GA++ provides new names for the C bindings of GA functions (For example, GA_Add_patch() is renamed as addPatch()).

## 8.2 GA++ Classes

All GA classes (GAServices, GlobalArray) are declared within the scope of GA namespace.

**Namespace issue:** Although namespace is part of ANSI C++ standard, not all C++ compilers support namespaces (A non-instantiable GA class is provided for implementations using compilers without namespace).
Note: define the variable _GA_USENAMESPACE_ as 0 in ga++.h if your compiler doesnot support namespaces.

```
namespace GA {
        class GAServices;
        class GlobalArray;
};
```

Current implementation has no derived classes (no (virtual) inheritance), templates or exception handling. Eventually, more object oriented functionalities will be added, and standard library facilities will be used without affecting the performance.

## 8.3 Initialization and Termination:

GA namespace has the following static functions for initialization and termination of Global Arrays.

*GA::Initialize():*
Initialize Global Arrays, allocates and initializes internal data structures in Global Arrays. This is a collective operation.

*GA::Terminate():*
Delete all active arrays and destroy internal data structures. This is a collective operation.

```
namespace GA {
        _GA_STATIC_ void Initialize(int argc, char *argv[], size_t limit = 0);
        _GA_STATIC_ void Initialize(int argc, char *argv[], unsigned long heapSize, unsigned long
stackSize, int type, size_t limit = 0);
        _GA_STATIC_ void Terminate();
};
```

**Example:**

```
#include <iostream.h>
#include "ga++.h"

int
main(int argc, char **argv) {
        GA::Initialize(argc, argv, 0);
        cout << "Hello World\n";
```

```
        GA::Terminate();
}
```

## 8.4 GAServices:

GAServices class has member functions that does all the global operations (non-array operations) like Process Information (number of processes, process id, ..), Inter-process Synchronization (sync, lock, broadcast, reduce,..), etc,.

**SERVICES Object:**
GA namespace has a global "SERVICES" object (of type "GAServices"), which can be used to invoke the non-array operations. To call the functions (for example, sync()), we invoke them on this SERVICES object (for example, GA::SERVICES.sync()). As this object is in the global address space, the functions can be invoked from anywhere inside the program (provided the ga++.h is included in that file/program).

## 8.5 Global Array:

GlobalArray class has member functions that perform:

- Array operations
- One-sided (get/put),
- Collective array operations,
- Utility operations, etc,.

---

# Mirrored Arrays

## 9.1 Overview

Mirrored arrays use a hybrid approach to replicate data across cluster nodes and distribute data within each node. It uses shared memory for caching latency sensitive distributed data structures on Symmetric Multi-Processor nodes of clusters connected with commodity networks. The user is responsible for managing consistency of the data cached within the mirrored arrays. Instead of applying mirroring to all distributed arrays, the user can decide, depending on the nature of the algorithm and the communication requirements (number and size of messages), which arrays can or should use mirroring and which should be left fully distributed and accessed without the shared memory cache.

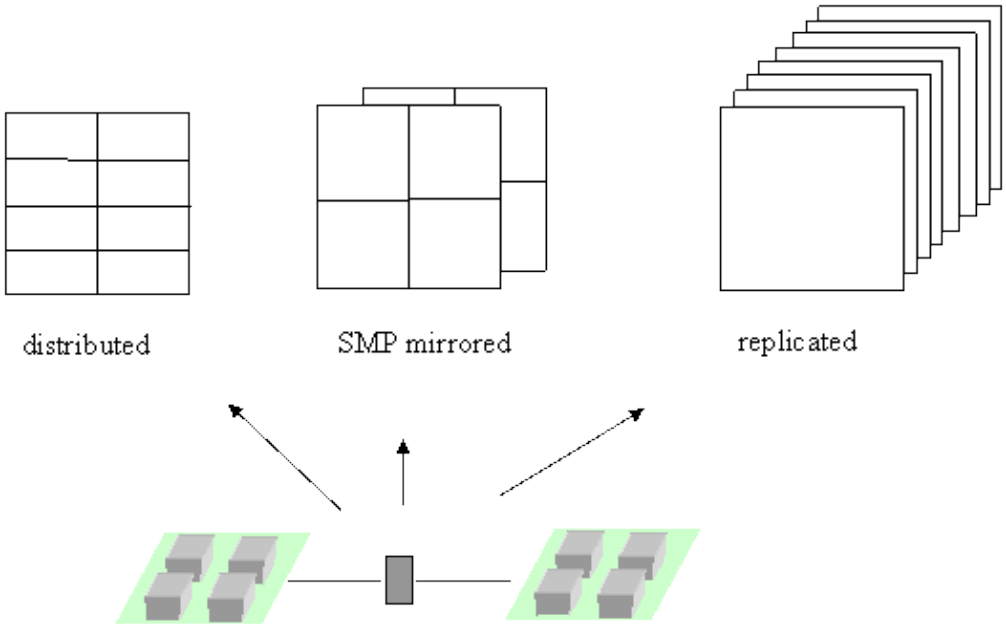



**Figure:** Example of a 2-dimensional array fully distributed, SMP mirrored, and replicated on two 4-way SMP cluster nodes.

This hybrid approach is particularly useful for problems where it is important to solve a moderate sized problem many times, such as an *ab initio* molecular dynamics simulation of a moderate size molecule. A single calculation of the energy and forces that can be run in a few minutes may be suitable for a geometry optimization, where a few tens of calculations are required, but is still too long for a molecular dynamics trajectory, which can require tens of thousands of separate evaluations. For these problems, it is still important to push scalability to the point where single energy and force calculations can be performed on the order of seconds. Similar concerns exist for problems involving Monte Carlo sampling or sensitivity analysis where it is important to run calculations quickly so that many samples can be taken.

Mirrored arrays differ from traditional replicated data schemes in two ways. First, mirrored arrays can be used in conjunction with distributed data and there are simple operations that support conversion back and forth from mirrored to distributed arrays. This allows developers maximum flexibility in incorporating mirrored arrays into their algorithms. Second, mirrored arrays are distributed within an SMP node (see the above figure). For systems with a large number of processors per node, e.g., 32 in the current generation IBM SP, this can result in significant distribution of the data. Even for systems with only 2 nodes per processor, this will result in an immediate savings of 50% over a conventional replicated data scheme.

The disadvantage of using mirrored arrays is that problems are limited in size by what can fit onto a single SMP node. This can be partially offset by the fact that almost all array operations can be supported on both mirrored and distributed arrays, so that it is easy to develop code that can switch between using mirrored arrays and conventional distributed arrays, depending on problem size and the number of available processors.

## 9.2 Mirrored Array Operations

```
Fortran  integer ga_pgroup_get_mirror()
C        int ga_pgroup_get_mirror()
C++      int GA::GAServices::pgroupGetMirror()
```

This function returns a handle to the mirrored processor list, which can then be used to create a mirrored global array using one of the NGA_Create_*_config calls.

```
Fortran  integer ga_merge_mirrored(g_a)
```

```
C         int GA_Merge_mirrored(int g_a)
C++       int GA::GlobalArray::mergeMirrored()
```

This subroutine merges mirrored arrays by adding the contents of each array across nodes. The result is that the each mirrored copy of the array represented by g_a is the sum of the individual arrays before the merge operation. After the merge, all mirrored arrays are equal. This is a collective operation.

```
Fortran   integer nga_merge_distr_patch(g_a, alo, ahi, g_b, blo, bhi)
C         int NGA_Merge_distr_patch(int g_a, int alo[], int ahi[], int
g_b, int blo[], int bhi[])
C++       int GA::GlobalArray::mergeDistrPatch(int alo[], int ahi[],
int g_b, int blo[], int bhi[])
```

This function merges all copies of a patch of a mirrored array (g_a) into a patch in a distributed array (g_b). This is same as GA_merge_mirrored, except, this function is operated on a patch rather than the whole array. This is a collective operation.

```
Fortran   integer ga_is_mirrored(g_a)
C         int GA_Is_mirrored(int g_a)
C++       int GA::GlobalArray::isMirrored()
```

This subroutine checks if the array is mirrored array or not. Returns 1 if it is a mirrored array, else returns 0. This is a local operation.

---

# Processor Groups

## 10.1 Overview

The Global Arrays toolkit has recently been extended to support global arrays defined on processor groups. The processor groups in GA programs follow the MPI approach. The MPI processor groups can be used in GA programs. However, since MPI standard does not support fault tolerance, GA provides set of APIs for process group managment which offers some opportunities for supporting environments with hardware faults.

In general, processor groups allow the programmer to subdivide the domain containing the complete set of processors ( the world group ) into subsets of processors that can act more or less independently of one another. Global arrays that are created on processor groups are only distributed amongst the processors in the group and not on all processors in the system. Collective operations executed on specific groups are also restricted to processors in the group and do not require any action from processors outside the group. A simple example is a synchronization operation. If the synchronization operation is executed on a group that is a subgroup of the world group, then only those processors in the subgroup are blocked until completion of the synchronization operation. Processors outside the subgroup can continue their operations without interruption.

The Global Arrays toolkit contains a collection of library calls that can be used to explicitly create groups, assign specific groups to global arrays, and execute global operations on groups. There is also a mechanism for setting the default group for the calculation. This is a powerful way of converting large amounts of parallel code that has already been written using the Global Arrays library to run as a subroutine on a processor group. Normally, the default group for a parallel calculation is the world group, but a call is available that can be used to change the default group to something else. This call must be executed by all processors in the subgroup. Furthermore, although it is not required, it is probably a *very* good idea to make sure that the default groups for all processors in the system (i.e. all processors contained in the original world group) represent a complete non-overlapping covering of the original world group (see figure). Once the default group has been set, all operations are implicitly assumed to occur on the default processor group unless explicitly stated otherwise. Global Arrays are only created on the default processor group and global

operations, such as synchronizations, broadcasts, and other operations, 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. Thus, a call to the ga_nodeid function will return a value of 0 for each processor designated as the zero processor within each default group. The number of processors returning 0 will be equal to the number of default groups (assuming the complete non-overlapping coverage suggested above is implemented).

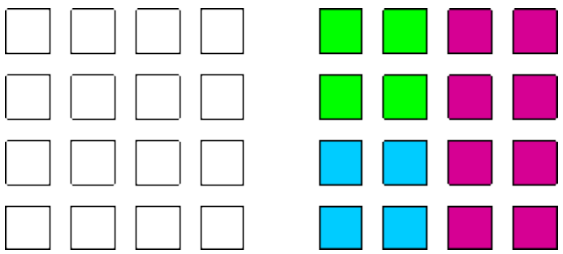


**Figure:** Original set of 16 processors decomposed into 3

non-overlapping groups

At present there are not many function calls that support operations between groups. The only calls that can be used to copy data from one group to another are the nga_copy and nga_copy_patch calls. These can be used to copy global arrays between two groups, provided that one group is completely contained in the other (this will always be the case if one of the groups is the world group). These commands will work correctly as long as they are executed only by processors contained in the smaller group. The nga_put and nga_get commands can also be used to communicate between Global Arrays on different groups (using an intermediate buffer), provided that the two groups share at least one processor (again, this will always be the case if one group is the world group).

The new functions included in the Global Arrays library are described below.

## 10.2 Creating new groups

**Fortran** integer function ga_pgroup_create(list, size)

**C** int GA_Pgroup_create(int *list, int size)

This call can be used to create a processor group of size size containing the processors in the array list. This call must be executed on all processors in the group. It returns an integer handle (for the processors group) that can be used to reference the processor group in other library calls.

Assigning groups:

**Fortran** subroutine ga_set_pgroup(g_a, p_handle)

**C** void GA_Set_pgroup(int g_a, int p_handle)

This call can be used to assign the processor group p_handle to a global array handle g_a that has been previously created using the ga_create_handle call. The processor group associated with a global array can also be set by creating the global array with one of the nga_create_XXX_config calls.

## 10.3 Setting default group

**Fortran** subroutine ga_pgroup_set_default(p_handle)

**C** void GA_Pgroup_set_default(int p_handle)

This call can be used to set the default group to something besides the world group. This call must be made on all processors contained in the group represented by p_handle. Once the default group has been set, all operations are restricted to the default group unless explicitly stated otherwise.

## 10.4 Inquiry functions

**Fortran** integer function ga_pgroup_nnodes(p_handle)

**C** int GA_Pgroup_nnodes(int p_handle)

**Fortran** integer function ga_pgroup_nodeid(p_handle)

**C** int GA_Pgroup_nodeid(int p_handle)

These functions can be used to access information about the group. The ga_pgroup_nnodes function returns the number of processors in the group specified by the handle p_handle, ga_pgroup_nodeid returns the local node ID of the processor within the group.

**Fortran** integer function ga_pgroup_get_default()

**C** int GA_Pgroup_get_default()

**Fortran** integer function ga_pgroup_get_mirror()

**C** int GA_Pgroup_get_mirror()

**Fortran** integer function ga_pgroup_get_world()

**C** int GA_Pgroup_get_world()

These functions can be used to get the handles for some standard groups at any point in the program. This is particularly useful for gaining access to the world group if the default group has been reset to a subgroup and also for gaining access to the handle for the mirror group (see section on mirrored arrays). Note that the mirror group is actually a group defined on the complete set of processors.

## 10.5 Collective operations on groups

**Fortran** subroutine ga_pgroup_sync(p_handle)

**C** void GA_Pgroup_sync(p_handle)

**Fortran** subroutine ga_pgroup_brdcst(p_handle,type,buf,lenbuf,root)

**C** void GA_Pgroup_brdcst(int p_handle, void *buf, root)

**Fortran** subroutine ga_pgroup_dgop(p_handle, type, buf, lenbuf, op)

**Fortran** subroutine ga_pgroup_sgop(p_handle, type, buf, lenbuf, op)

**Fortran** subroutine ga_pgroup_igop(p_handle, type, buf, lenbuf, op)

**C** void GA_Pgroup_dgop(int p_handle, double *buf, int lenbuf, char *op)

**C** void GA_Pgroup_fgop(int p_handle, float *buf, int lenbuf, char *op)

**C** void GA_Pgroup_igop(int p_handle, int *buf, int lenbuf, char *op)

**C** void GA_Pgroup_lgop(int p_handle, long *buf, int lenbuf, char *op)

These operations are all identical to the standard global operations, the only difference is that they have an extra argument that takes a group handle. The action of these calls is restricted to the set of processor contained in the group represented by p_handle. All processors in the group must call these subroutines.