GA_ABS_VALUE
GA_ABS_VALUE_PATCH
GA_ACC
GA_ACCESS
GA_ACCESS_BLOCK
GA_ACCESS_BLOCK_GRID
GA_ACCESS_BLOCK_SEGMENT
GA_ACCESS_GHOST_ELEMENT
GA_ACCESS_GHOSTS
GA_ADD
GA_ADD_PATCH
GA_ADD_CONSTANT
GA_ADD_CONSTANT_PATCH
GA_ADD_DIAGONAL
GA_ALLOCATE
GA_BRDCST
GA_CHECK_HANDLE
GA_CLUSTER_NNODES
GA_CLUSTER_NODEID
GA_CLUSTER_PROC_NODEID
GA_CLUSTER_NPROCS
GA_CLUSTER_PROCID
GA_COMPARE_DISTR
GA_COPY
GA_COPY_PATCH
GA_CREATE
GA_CREATE_CONFIG
GA_CREATE_GHOSTS
GA_CREATE_GHOSTS_CONFIG
GA_CREATE_HANDLE
GA_CREATE_IRREG
GA_CREATE_IRREG_CONFIG
GA_CREATE_GHOSTS_IRREG
GA_CREATE_GHOSTS_IRREG_CONFIG
GA_CREATE_MUTEXES
GA_DDOT
GA_DDOT_PATCH
GA_DESTROY
GA_DESTROY_MUTEXES
GA_DGEMM
GA_DGOP
GA_DIAG
GA_DIAG_REUSE
GA_DIAG_STD
GA_DISTRIBUTION
GA_DUPLICATE
GA_ELEM_MULTIPLY
GA_ELEM_MULTIPLY_PATCH
GA_ELEM_DIVIDE
GA_ELEM_DIVIDE_PATCH
GA_ELEM_MAXIMUM
GA_ELEM_MAXIMUM_PATCH
GA_ELEM_MINIMUM
GA_ELEM_MINIMUM_PATCH
GA_FENCE
GA_FILL
GA_FILL_PATCH
GA_GATHER
GA_GET
GA_GET_BLOCK_INFO
GA_GET_DIAG
GA_GET_DEBUG
GA_HAS_GHOSTS
GA_IDOT
GA_IDOT_PATCH
GA_IGOP
GA_INIT_FENCE
GA_INITIALIZE
GA_INITIALIZE_LTD
GA_INQUIRE
GA_INQUIRE_MEMORY
GA_INQUIRE_NAME

- GA_IS_MIRRORED
- GA_LGOP
- GA_LIST_NODEID
- GA_LLT_SOLVE
- GA_LOCATE
- GA_LOCATE_REGION
- GA_LOCK
- GA_LU_SOLVE
- GA_MASK_SYNC
- GA_MATMUL_PATCH
- GA_MATMUL_PATCH(N)
- GA_MEDIAN
- GA_MEDIAN_PATCH
- GA_MEMORY_AVAIL
- GA_MEMORY_LIMITED
- GA_MERGE_DISTR_PATCH
- GA_MERGE_MIRRORED
- GA_MPI_COMMUNICATOR
- GA_NBACC
- GA_NBGET
- GA_NBGET_GHOST_DIR
- GA_NBLOCK
- GA_NBPUT
- GA_NBWAIT
- GA_NDIM
- GA_NNODES
- GA_NODEID
- GA_NORM1
- GA_NORM_INFINITY
- GA_PATCH_ENUM
- GA_PACK
- GA_PERIODIC_ACC
- GA_PERIODIC_GET
- GA_PERIODIC_PUT
- GA_PGROUP_BRDCST
- GA_PGROUP_CREATE
- GA_PGROUP_DESTROY
- GA_PGROUP_DGOP
- GA_PGROUP_FGOP
- GA_PGROUP_GET_DEFAULT
- GA_PGROUP_GET_MIRROR
- GA_PGROUP_GET_WORLD
- GA_PGROUP_IGOP
- GA_PGROUP_LGOP
- GA_PGROUP_NNODES
- GA_PGROUP_NODEID
- GA_PGROUP_SET_DEFAULT
- GA_PGROUP_SPLIT
- GA_PGROUP_SPLIT_IRREG
- GA_PGROUP_SYNC
- GA_PRINT
- GA_PRINT_DISTRIBUTION
- GA_PRINT_FILE
- GA_PRINT_PATCH
- GA_PRINT_STATS
- GA_PROC_TOPOLOGY
- GA_PUT
- GA_READ_INC
- GA_RECIP
- GA_RECIP_PATCH
- GA_RELEASE
- GA_RELEASE_BLOCK
- GA_RELEASE_BLOCK_GRID
- GA_RELEASE_BLOCK_SEGMENT
- GA_RELEASE_UPDATE
- GA_RELEASE_UPDATE_BLOCK
- GA_RELEASE_UPDATE_BLOCK_GRID
- GA_RELEASE_UPDATE_BLOCK_SEGMENT
- GA_SCALE
- GA_SCALE_PATCH
- GA_SCALE_ROWS
- GA_SCALE_COLS

GA_INITIALIZE

```
void GA_Initialize()
```

Allocate and initialize internal data structures in Global Arrays.

This is a collective operation.

GA_INITIALIZE_LTD

```
void GA_Initialize_ltd(size_t limit)

limit      amount of memory in bytes per process              [input]
```

Allocate and initialize internal data structures and set limit for memory used in global arrays. The limit is per process: it is the amount of memory that the given processor can contribute to collective allocation of global arrays. It does not include temporary storage that GA might be allocating (and releasing) during execution of a particular operation.

*limit < 0 means "allow unlimited memory usage" in which case this operation is equivalent to GA_initialize.

This is a collective operation.

GA_PGROUP_CREATE

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

list[size]                     list of processor IDs in group  [input]
size                           number of processors in group   [input]
```

This command is used to create a processor group. At present, it must be invoked by all processors in the current default processor group. The list of processors use the indexing scheme of the default processor group. If the default processor group is the world group, then these indices are the usual processor indices. This function returns a process group handle that can be used to reference this group by other functions.

This is a collective operation on the default processor group.

GA_PGROUP_DESTROY

```
int GA_Pgroup_destroy(int p_handle)

p_handle                       processor group handle          [input]
```

This command is used to free up a processor group handle. It returns 0 if the processor group handle was not previously active.

This is a collective operation on the default processor group.

GA_PGROUP_SET_DEFAULT

```
void GA_Pgroup_set_default(int p_handle)

p_handle                       processor group handle          [input]
```

This function can be used to reset the default processor group on a collection of processors. All processors in the group referenced by p_handle must make a call to this function. Any standard global array call that is made after resetting the default processor group will be restricted to processors in that group. Global arrays that are created after resetting the default processor group will only be defined on that group and global operations such as GA_Sync or GA_Igop will be restricted to processors in that group. The GA_Pgroup_set_default call can be used to rapidly convert large applications, written with GA, into routines that run on processor groups.

The default processor group can be overridden by using GA calls that require an explicit group handle as one of the arguments.

This is a collective operation on the group represented by the handle p_handle.

NGA_CREATE

```
int NGA_Create(int type, int ndim, int dims[], char *array_name, int chunk[])

array_name       - a unique character string                   [input]
type             - data type (MT_F_DBL,MT_F_INT,MT_F_DCPL)      [input]
ndim             - number of array dimensions                   [input]
dims[ndim]       - array of dimensions                          [input]
chunk[ndim]      - array of chunks, each element specifies minimum size that
                   given dimensions should be chunked up into   [input]
```

Creates an ndim-dimensional array using the regular distribution model and returns integer handle representing the array.

The array can be distributed evenly or not. The control over the distribution is accomplished by specifying chunk (block) size for all or some of array dimensions. For example, for a 2-dimensional array, setting chunk[0]=dim[0] gives distribution by vertical strips (chunk[0]*dims[0]); setting chunk[1]=dim[1] gives distribution by horizontal strips (chunk[1]*dims[1]). Actual chunks will be modified so that they are at least the size of the minimum and each process has either zero or one chunk. Specifying chunk[i] as <1 will cause that dimension to be distributed evenly.

As a convenience, when chunk is specified as NULL, the entire array is distributed evenly.

Return value: a non-zero array handle means the call was succesful.
This is a collective operation.

NGA_CREATE_CONFIG

```
int NGA_Create_config(int type, int ndim, int dims[], char *array_name,
                      int chunk[], int p_handle)

array_name       - a unique character string                   [input]
type             - data type (MT_F_DBL,MT_F_INT,MT_F_DCPL)      [input]
ndim             - number of array dimensions                   [input]
dims[ndim]       - array of dimensions                          [input]
chunk[ndim]      - array of chunks, each element specifies minimum size that
                   given dimensions should be chunked up into   [input]
p_handle         - processor list handle                        [input]
```

Creates an ndim-dimensional array using the regular distribution model but with an explicitly specified processor list handle and returns an integer handle representing the array.

This call is essentially the same as the NGA_Create call, except for the processor list handle p_handle. It can be used to create mirrored arrays.

Return value: a non-zero array handle means the call was succesful.
This is a collective operation.

NGA_CREATE_GHOSTS

```
int NGA_Create_ghosts(int type, int ndim, int dims[], int width[],
                      char *array_name, int chunk[])

array_name   - a unique character string                   [input]
type         - data type (MT_DBL,MT_INT,MT_DCPL)            [input]
ndim         - number of array dimensions                   [input]
dims[ndim]   - array of dimensions                          [input]
```

```
width[ndim]  - array of ghost cell widths                   [input]
chunk[ndim]  - array of chunks, each element specifies
               minimum size that given dimensions should be
               chunked up into                              [input]
```

Creates an ndim-dimensional array with a layer of ghost cells around the visible data on each processor using the regular distribution model and returns an integer handle representing the array.

The array can be distributed evenly or not evenly. The control over the distribution is accomplished by specifying chunk (block) size for all or some of the array dimensions. For example, for a 2-dimensional array, setting chunk(1)=dim(1) gives distribution by vertical strips (chunk(1)*dims(1)); setting chunk(2)=dim(2) gives distribution by horizontal strips (chunk(2)*dims(2)). Actual chunks will be modified so that they are at least the size of the minimum and each process has either zero or one chunk. Specifying chunk(i) as **<1** will cause that dimension (i-th) to be distributed evenly. The width of the ghost cell layer in each dimension is specified using the array width(). The local data of the global array residing on each processor will have a layer width[n] ghosts cells wide on either side of the visible data along the dimension n.

Return value: a non-zero array handle means the call was successful. This is a collective operation.

NGA_CREATE_GHOSTS_CONFIG

```
int NGA_Create_ghosts_config(int type, int ndim, int dims[],
            int width[], char *array_name, int chunk[], int p_handle)
```

```
array_name   - a unique character string                    [input]
type         - data type (MT_DBL,MT_INT,MT_DCPL)            [input]
ndim         - number of array dimensions                   [input]
dims[ndim]   - array of dimensions                          [input]
width[ndim]  - array of ghost cell widths                   [input]
chunk[ndim]  - array of chunks, each element specifies
               minimum size that given dimensions should be
               chunked up into                              [input]
p_handle     - processor list handle
```

Creates an ndim-dimensional array with a layer of ghost cells around the visible data on each processor using the regular distribution model and an explicitly specified processor list and returns an integer handle representing the array.

This call is essentially the same as the NGA_Create_ghosts call, except for the processor list handle p_handle. It can be used to create mirrored arrays.

Return value: a non-zero array handle means the call was successful. This is a collective operation.

NGA_CREATE_IRREG

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

```
array_name    - a unique character string                     [input]
type          - MA data type (MT_F_DBL,MT_F_INT,MT_F_DCPL)    [input]
ndim          - number of array dimensions                    [input]
dims          - array of dimension values                     [input]
nblock[ndim]  - no. of blocks each dimension is divided into  [input]
map[s]        - starting index for for each block; the size
                  s is a sum all elements of nblock array     [input]
```

Creates an array by following the user-specified distribution and returns integer handle representing the array.

The distribution is specified as a Cartesian product of distributions for each dimension. The array indices start at 0. For example, the following figure demonstrates distribution of a 2-dimensional array 8x10 on 6 (or more) processors. nblock[2]={3,2}, the size of map array is s=5 and array map contains the following elements map={0,2,6, 0, 5}. The distribution is nonuniform because, P1 and P4 get 20 elements each and processors P0,P2,P3, and P5 only 10 elements each.

5 5

P0 P3 2

P1 P4 4

P2 P5 2

Return value: a non-zero array handle means the call was succesful.
This is a collective operation.

NGA_CREATE_IRREG_CONFIG

```
int NGA_Create_irreg_config(int type, int ndim, int dims[], char *array_name,
                            int block[], int map[], int p_handle)
```

```
array_name    - a unique character string                   [input]
type          - MA data type (MT_F_DBL,MT_F_INT,MT_F_DCPL)    [input]
ndim          - number of array dimensions                   [input]
dims          - array of dimension values                    [input]
nblock[ndim]  - no. of blocks each dimension is divided into  [input]
map[s]        - starting index for for each block; the size
                 s is a sum all elements of nblock array     [input]
p_handle      - processor list handle
```

Creates an array by following the user-specified distribution and an explicitly specified processor list handle and returns an integer handle representing the array.

This call is essentially the same as the NGA_Create_irreg call, except for the processor list handle p_handle. It can be used to create mirrored arrays.

Return value: a non-zero array handle means the call was succesful.
This is a collective operation.

NGA_CREATE_GHOST_IRREG

```
int NGA_Create_ghost_irreg(int type, int ndim, int dims[], width[],
                     char *array_name, map[], nblock[])
```

```
array_name   - a unique character string                  [input]
type         - data type (MT_DBL,MT_INT,MT_DCPL)           [input]
ndim         - number of array dimensions                 [input]
dims[ndim]   - array of dimensions                        [input]
width[ndim]  - array of ghost cell widths                 [input]
nblock[ndim] - no. of blocks each dimension is divided into[input]
map[s]       - starting index for for each block; the size
               s is a sum of all elements of nblock array  [input]
```

Creates an array with ghost cells by following the user-specified distribution and returns integer handle representing the array.

The distribution is specified as a Cartesian product of distributions for each dimension. For example, the following figure demonstrates distribution of a 2-dimensional array 8x10 on 6 (or more) processors. nblock(2)={3,2}, the size of map array is s=5 and array map contains the following elements map={1,3,7, 1, 6}. The distribution is nonuniform because, P1 and P4 get 20 elements each and processors P0,P2,P3, and P5 only 10 elements each.

5 5

P0 P3 2

P1	P4	4
P2	P5	2

The array width[] is used to control the width of the ghost cell boundary around the visible data on each processor. The local data of the global array residing on each processor will have a layer width[n] ghosts cells wide on either side of the visible data along the dimension n.

Return value: a non-zero array handle means the call was succesful.
This is a collective operation.

NGA_CREATE_GHOSTS_IRREG_CONFIG

```
int NGA_Create_ghost_irreg_config(int type, int ndim, int dims[],
          width[], char *array_name, map[], nblock[], int p_handle)
```

```
array_name   - a unique character string                [input]
type         - data type (MT_DBL,MT_INT,MT_DCPL)         [input]
ndim         - number of array dimensions                [input]
dims[ndim]   - array of dimensions                       [input]
width[ndim]  - array of ghost cell widths                [input]
nblock[ndim] - no. of blocks each dimension is divided into[input]
map[s]       - starting index for for each block; the size
               s is a sum of all elements of nblock array  [input]
p_handle     - processor list handle                     [input][
```

Creates an array with ghost cells by following the user-specified distribution and returns integer handle representing the array.

This call is essentially the same as the NGA_Create_ghosts_irreg call, except for the processor list handle p_handle. It can be used to create mirrored arrays.

Return value: a non-zero array handle means the call was succesful.

This is a collective operation.

GA_CREATE_HANDLE

```
int GA_Create_handle()
```

This function returns a global array handle that can then be used to create a new global array. This is part of a new API for creating global arrays that is designed to replace the old interface built around the NGA_Create_xxx calls. The sequence of operations is to begin with a call to GA_Greate_handle to get a new array handle. The attributes of the array, such as dimension, size, type, etc. can then be set using successive calls to the GA_Set_xxx subroutines. When all array attributes have been set, the GA_Allocate subroutine is called and the global array is actually created and memory for it is allocated.

This is a collective operation.

GA_SET_ARRAY_NAME

```
void GA_Set_array_name(int g_a, char *name)
```

```
g_a                                              [input]
name       - array name                          [input]
```

This function can be used to assign a unique character string name to a global array handle that was obtained using the GA_Create_handle function.

This is a collective operation.

GA_SET_DATA

```
void GA_Set_data(int g_a, int ndim, int dims[], int type)

g_a                                                [input]
ndim     - dimension of global array               [input]
dims[]   - dimensions of global array              [input]
type     - data type of global array               [input]
```

This function can be used to set the array dimension, the coordinate dimensions, and the data type assigned to a global array handle obtained using the GA_Create_handle function.

This is a collective operation.

GA_SET_IRREG_DISTR

```
void GA_Set_irreg_distr(int g_a, int mapc[], int nblock[])

g_a                                                      [input]
mapc[s]       - starting index for each block; the size
                s is the sum of all elements of the array
                nblock                                   [input]
nblock[ndim]  - number of blocks that each dimension is
                divided into                             [input]
```

This function can be used to partition the array data among the individual processors for a global array handle obtained using the GA_Create_handle function.

The distribution is specified as a Cartesian product of distributions for each dimension. For example, the following figure demonstrates distribution of a 2-dimensional array 8x10 on 6 (or more) processors. nblock(2)={3,2}, the size of mapc array is s=5 and array mapc contains the following elements mapc={1,3,7, 1, 6}. The distribution is nonuniform because, P1 and P4 get 20 elements each and processors P0,P2,P3, and P5 only 10 elements each.

5	5	
P0	P3	2
P1	P4	4
P2	P5	2

The array width() is used to control the width of the ghost cell boundary around the visible data on each processor. The local data of the global array residing on each processor will have a layer width(n) ghosts cells wide on either side of the visible data along the dimension n.

This is a collective operation.

GA_SET_PGROUP

```
void GA_Set_pgroup(int g_a, int p_handle)

g_a                                     [input]
p_handle      processor group handle    [input]
```

This function can be used to set the processor configuration assigned to a global array handle that was obtained using the GA_Create_handlefunction. It can be used to create mirrored arrays by using the mirrored array processor configuration in this function call. It can also be used to create an array on a processor group by using a processor group handle in this call.

This is a collective operation.

GA_SET_GHOSTS

```
void GA_Set_ghosts(int g_a, int width[])

g_a                                          [input]
width[ndim] - array of ghost cell widths     [input]
```

This function can be used to set the ghost cell widths for a global array handle that was obtained using the GA_Create_handle function. The ghosts cells widths indicate how many ghost cells are used to pad the locally held array data along each dimension. The padding can be set independently for each coordinate dimension.

This is a collective operation.

GA_SET_CHUNK

```
void GA_Set_chunk(int g_a, int chunk[])

g_a                                            [input]
chunk[]  - array of chunk widths               [input]
```

This function is used to set the chunk array for a global array handle that was obtained using the GA_Create_handle function. The chunk array is used to determine the minimum number of array elements assigned to each processor along each coordinate direction.

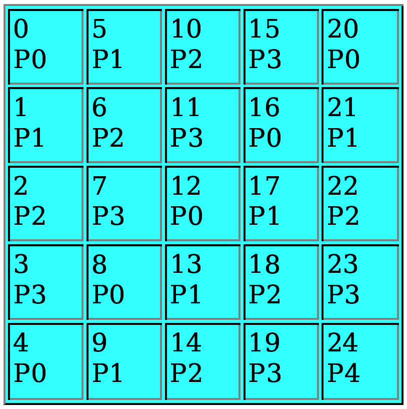
This is a collective operation.

GA_SET_BLOCK_CYCLIC

```
void GA_Set_block_cyclic(int g_a, int dims[])

g_a           - global array handle           [input]
dims[]        - array of block dimensions     [input]
```

This subroutine is used to create a global array with a simple block-cyclic data distribution. The array is broken up into blocks of size dims and each block is numbered sequentially using a column major indexing scheme. The blocks are then assigned in a simple round-robin fashion to processors. This is illustrated in the figure below for an array containing 25 blocks distributed on 4 processors. Blocks at the edge of the array may be smaller than the block size specified in dims. In the example below, blocks 4,9,14,19,20,21,22,23, and 24 might be smaller thatn the remaining blocks. Most global array operations are insensitive to whether or not a block-cyclic data distribution is used, although performance may be slower in some cases if the global array is using a block-cyclic data distribution. Individual data blocks can be accessesed using the block-cyclic access functions.



0 P0	5 P1	10 P2	15 P3	20 P0
1 P1	6 P2	11 P3	16 P0	21 P1
2 P2	7 P3	12 P0	17 P1	22 P2
3 P3	8 P0	13 P1	18 P2	23 P3
4 P0	9 P1	14 P2	19 P3	24 P4

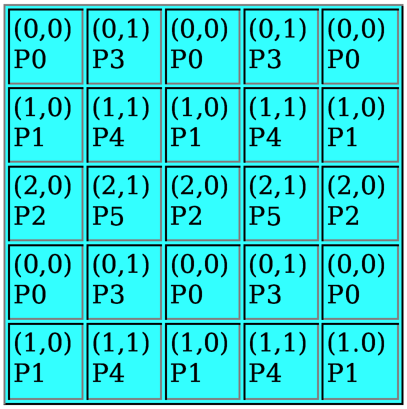
This is a collective operation.

GA_SET_BLOCK_CYCLIC_PROC_GRID

```
void GA_Set_block_cyclic(int g_a, int dims[], int proc_grid[])

g_a             - global array handle         [input]
dims[]          - array of block dimensions   [input]
proc_grid[]     - processor grid dimensions   [input]
```

This subroutine is used to create a global array with a SCALAPACK-type block cyclic data distribution. The user specifies the dimensions of the processor grid in the array proc_grid. The product of the processor grid dimensions must equal the number of total number of processors and the number of dimensions in the processor grid must be the same as the number of dimensions in the global array. The data blocks are mapped onto the processor grid in a cyclic manner along each of the processor grid axes. This is illustrated below for an array consisting of 25 data blocks disributed on 6 processors. The 6 processors are configured in a 3 by 2 processor grid. Blocks at the edge of the array may be smaller than the block size specified in dims. Most global array operations are insensitive to whether or not a block-cyclic data distribution is used, although performance may be slower in some cases if the global array is using a block-cyclic data distribution. Individual data blocks can be accessesed using the block-cyclic access functions.



(0,0) P0	(0,1) P3	(0,0) P0	(0,1) P3	(0,0) P0
(1,0) P1	(1,1) P4	(1,0) P1	(1,1) P4	(1,0) P1
(2,0) P2	(2,1) P5	(2,0) P2	(2,1) P5	(2,0) P2
(0,0) P0	(0,1) P3	(0,0) P0	(0,1) P3	(0,0) P0
(1,0) P1	(1,1) P4	(1,0) P1	(1,1) P4	(1.0) P1

This is a collective operation.

GA_ALLOCATE

```
int GA_Allocate(int g_a)

g_a                                            [input]
```

This function allocates the memory for the global array handle originally obtained using the GA_Create_handle function. At a minimum, the GA_Set_data function must be called before the memory is allocated. Other GA_Set_xxx functions can also be called before invoking this function.

This is a collective operation.

GA_UPDATE_GHOSTS

`void GA_Update_ghosts(int g_a)`

This call updates the ghost cell regions on each processor with the corresponding neighbor data from other processors. The operation assumes that all data is wrapped around using periodic boundary data so that ghost cell data that goes beyound an array boundary is wrapped around to the other end of the array. The GA_Update_ghosts call contains two GA_Sync calls before and after the actual update operation. For some applications these calls may be unecessary, if so they can be removed using the GA_Mask_sync subroutine.
This is a collective operation.

NGA_UPDATE_GHOST_DIR

`int NGA_Update_ghost_dir(int g_a, int dimension, int idir, int cflag)`

```
g_a                                                 [input]
dimension    - array dimension that is to be updated   [input]
idir         - direction of update (+/- 1)             [input]
cflag        - flag (0/1) to include corners in update [input]
```

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.

GA_HAS_GHOSTS

`int GA_Has_ghosts(int g_a)`

This function returns 1 if the global array has some dimensions for which the ghost cell width is greater than zero, it returns 0 otherwise.
This is a collective operation.

NGA_ACCESS_GHOSTS

`void NGA_Access_ghosts(int g_a, int dims[], void *ptr, int ld[])`

```
g_a                                                      [input]
dims[ndim]  - array of dimensions of local patch, including
              ghost cells                                 [output]
ptr         - returns an index corresponding to the origin
              the global array patch held locally on the
              processor                                   [output]
ld[ndim-1]  - physical dimenstions of the local array patch,
              including ghost cells                       [output]
```

Provides access to the local patch of the global array. Returns leading dimension ld and and pointer for the data. This routine will provide access to the ghost cell data residing on each processor. Calls to NGA_Access_ghosts should normally follow a call to NGA_Distribution that returns coordinates of the visible data patch associated with a processor. You need to make sure that the coordinates of the patch are valid (test values returned from NGA_Distribution).

You can only access local data.
This is a local operation.

NGA_ACCESS_GHOST_ELEMENT

```
void NGA_Access_ghost_element(int g_a, void *ptr, int subscript[],
                              int ld[])
```

```
g_a                                                      [input]
index           - index pointing to location of element
                  indexed by subscript[]                 [output]
subscript[ndim] - array of integers that index desired
                  element                                [input]
ld[ndim-1]      - array of strides for local data patch.
                  These include ghost cell widths.       [output]
```

This function can be used to return a pointer to any data element in the locally held portion of the global array and can be used to directly access ghost cell data. The array subscript refers to the local index of the element relative to the origin of the local patch (which is assumed to be indexed by (0,0,...)).
This is a local operation.

GA_TOTAL_BLOCKS

```
int GA_Total_blocks(int g_a)
```

```
g_a                                       [input]
```

This function returns the total number of blocks contained in a global array with a block-cyclic data distribution. This is a local operation.

GA_GET_BLOCK_INFO

```
void GA_Get_block_info(int g_a, int num_blocks[], int block_dims[])
```

```
g_a                                                 [input]
num_blocks[ndim]   - number of blocks along each axis [output]
block_dims[ndim]   - dimensions of block              [output]
```

This subroutine returns information about the block-cyclic distribution associated with global array g_a. The number of blocks along each of the array axes are returned in the array num_blocks and the dimensions of the individual blocks, specified in the GA_Set_block_cyclic or GA_Set_block_cyclic_proc_grid subroutines, are returned in block_dims. This is a local function.

GA_DUPLICATE

```
int GA_Duplicate(int g_a, char* array_name)
```

```
array_name    - a character string                 [input]
g_a           - integer handle for reference array [input]
```

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

Return value: a non-zero array handle means the call was succesful.
This is a collective operation.

GA_DESTROY

```
void GA_Destroy(int g_a)
```

```
g_a   - array handle      [input]
```

Deallocates the array and frees any associated resources.

This is a collective operation.

GA_TERMINATE

```
void GA_Terminate()
```

Delete all active arrays and destroy internal data structures.

This is a collective operation.

GA_SYNC

```
void GA_Sync()
```

Synchronize processes (a barrier) and ensure that all GA operations completed.

This is a collective operation.

GA_MASK_SYNC

```
void GA_Mask_sync(int first,int last)
```

```
first   - mask (0/1) for prior internal synchronization [input]
last    - mask (0/1) for post internal synchronization  [input]
```

This subroutine can be used to remove synchronization calls from around collective operations. Setting the parameter first = .false. removes the synchronization prior to the collective operation, setting last = .false. removes the synchronization call after the collective operation. This call is applicable to all collective operations. It most be invoked before each collective operation.
This is a collective operation.

GA_ZERO

```
void GA_Zero(int g_a)
```

```
g_a - array handle       [input]
```

Sets value of all elements in the array to zero.

This is a collective operation.

GA_FILL

```
void GA_Fill(int g_a, void *value)
```

```
g_a - array handle       [input]
value   - pointer to the value of appropriate type (double/DoubleComplex/long)
          that matches array type
```

Assign a single value to all elements in the array.

This is a collective operation.

GA_DOT

```
int GA_Idot(int g_a, int g_b)

long GA_Ldot(int g_a, int g_b)
float GA_Fdot(int g_a, int g_b)
double GA_Ddot(int g_a, int g_b)
DoubleComplex GA_Zdot(int g_a, int g_b)
g_a, g_b   - array handles                 [input]
```

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

```
return value = SUM_ij a(i,j)*b(i,j)
```

This is a collective operation.

GA_SCALE

```
void GA_Scale(int g_a, void *value)

g_a     - array handle                                                     [input]
value   - pointer to the value of appropriate type (double/DoubleComplex/long)
          that matches array type                                          [input]
```

Scales an array by the constant s. Note that the library is unable to detect errors when the pointed value is of different type than the array.

This is a collective operation.

GA_ADD

```
void GA_Add(void *alpha, int g_a, void* beta, int g_b, int g_c)

g_a, g_b, g_c                         - array handles       [input]
double/complex/int      *alpha        - scale factor        [input]
double/complex/int      *beta         - scale factor        [input]
```

The arrays (which must be the same shape and identically aligned) are added together element-wise

```
c = alpha * a  +  beta * b.
```

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

This is a collective operation.

GA_COPY

```
void GA_Copy(int g_a, int g_b)

g_a, g_b - array handles       [input]
```

Copies elements in array represented by g_a into the array represented by g_b. The arrays must be the same type, shape, and identically aligned.

This is a collective operation.

GA_SET_MEMORY_LIMIT

```
void GA_Set_memory_limit(size_t limit)

limit    - the amount of memory in bytes per process    [input]
```

Sets the amount of memory to be used (in bytes) per process

This is a local operation.

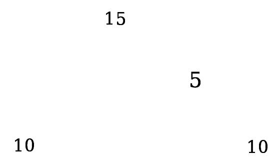
GA_GET

`void NGA_Get(int g_a, int lo[], int hi[], void* buf, int ld[])`

```
g_a        - global array handle                                         [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section          [input]
hi[ndim]   - array of ending indices for global array section            [input]
buf        - pointer to the local buffer array where the data goes       [output]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
```

Copies data from global array section to the local array buffer. The local array is assumed to be have the same number of dimensions as the global array. Any detected inconsitencies/errors in the input arguments are fatal.

Example:
For ga_get operation transfering data from the [10:14,0:4] section of 2-dimensional 15x10 global array into local buffer 5x10 array we have:
lo={10,0}, hi={14,4}, ld={10}



This is a one-sided operation.

GA_PERIODIC_GET

`void NGA_Periodic_get(int g_a, int lo[], int hi[], void* buf, int ld[])`

```
g_a        - global array handle                                         [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section          [input]
hi[ndim]   - array of ending indices for global array section            [input]
buf        - pointer to the local buffer array where the data goes       [output]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
```

Same as nga_get except the indices can extend beyond the array boundary/dimensions in which case the library wraps them around.
This is a one-sided operation.

GA_STRIDED_GET

`void NGA_Strided_get(int g_a, int lo[], int hi[], int skip[], void* buf, int ld[])`

```
g_a        - global array handle                                         [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section          [input]
```

```
hi[ndim]   - array of ending indices for global array section                 [input]
skip[ndim] - array of strides for each dimension                              [input]
buf        - pointer to the local buffer array where the data goes            [output]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
```

This operation is the same as NGA_Get, except that the values corresponding to dimension n in buf correspond to every skip[n] values of the global array g_a. This is a one-sided operation.

GA_PUT

```
void NGA_Put(int g_a, int lo[], int hi[], void* buf, int ld[])
```

```
g_a        - global array handle                                              [output]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section               [input]
hi[ndim]   - array of ending indices for global array section                 [input]
buf        - pointer to the local buffer array where the data is              [input]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
```

Copies data from local array buffer to the global array section . The local array is assumed to be have the same number of dimensions as the global array.
Any detected inconsitencies/errors in input arguments are fatal.

This is a one-sided operation.

GA_PERIODIC_PUT

```
void NGA_Periodic_put(int g_a, int lo[], int hi[], void* buf, int ld[])
```

```
g_a        - global array handle                                              [output]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section               [input]
hi[ndim]   - array of ending indices for global array section                 [input]
buf        - pointer to the local buffer array where the data is              [input]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
```

Same as nga_put except the indices can extend beyond the array boundary/dimensions in which case the library wraps them around.
This is a one-sided operation.

GA_STRIDED_PUT

```
void NGA_Strided_put(int g_a, int lo[], int hi[], int skip[], void* buf, int ld[])
```

```
g_a        - global array handle                                              [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section               [input]
hi[ndim]   - array of ending indices for global array section                 [input]
skip[ndim] - array of strides for each dimension                              [input]
buf        - pointer to the local buffer array where the data goes            [output]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
```

This operation is the same as NGA_Put, except that the values corresponding to dimension n in buf are copied to every skip[n] values of the global array g_a. This is a one-sided operation.

GA_ACC

```
void NGA_Acc(int g_a, int lo[], int hi[], void* buf, int ld[], void* alpha)
```

```
g_a        - global array handle                                              [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for array section                      [input]
hi[ndim]   - array of ending indices for array section                        [input]
buf        - pointer to the local buffer array                                [input]
```

```
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
double/DoubleComplex/long *alpha     scale factor                                 [input]
```

Combines data from local array buffer with data in the global array section. The local array is assumed to be have the same number of dimensions as the global array.

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

This is a one-sided and atomic operation.

GA_PERIODIC_ACC

```
void NGA_Periodic_acc(int g_a, int lo[], int hi[], void* buf, int ld[], void* alpha)

g_a        - global array handle                                                  [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for array section                          [input]
hi[ndim]   - array of ending indices for array section                            [input]
buf        - pointer to the local buffer array                                    [input]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
double/DoubleComplex/long *alpha     scale factor                                 [input]
```

Same as nga_acc except the indices can extend beyond the array boundary/dimensions in which case the library wraps them around.
This is a one-sided and atomic operation.

GA_STRIDED_ACC

```
void NGA_Strided_acc(int g_a, int lo[], int hi[], int skip[], void* buf, int ld[])

g_a        - global array handle                                                  [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section                   [input]
hi[ndim]   - array of ending indices for global array section                     [input]
skip[ndim] - array of strides for each dimension                                  [input]
buf        - pointer to the local buffer array where the data goes                [output]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
double/DoublComplex/long *alpha     scale factor                                  [input]
```

This operation is the same as NGA_Acc, except that the values corresponding to dimension n in buf are accumulated to every skip[n] values of the global array g_a. This is a one-sided operation.

GA_DISTRIBUTION

```
void NGA_Distribution(int g_a, int iproc, int lo[], int hi[])

g_a        - array handle                               [input]
iproc      - process number                             [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for array section [input]
hi[ndim]   - array of ending indices for array section   [input]
```

If no array elements are owned by process iproc, the range is returned as **lo[ ]=0** and **hi[ ]= -1** for all dimensions.
This operation is local.

GA_COMPARE_DISTR

```
int GA_Compare_distr(int g_a, int g_b)
g_a, g_b    - array handles               [input]
```

Compares distributions of two global arrays. Returns 0 if distributions are identical and 1 when they are not.

This is a collective operation.

GA_ACCESS

```
void NGA_Access(int g_a, int lo[], int hi[], void *ptr, int ld[])

g_a        - global array handle                           [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for array section     [input]
hi[ndim]   - array of ending indices for array section       [input]
ptr        - points to location of first element in patch    [output]
ld[ndim-1] - leading dimensions for the pacth elements       [output]
```

Provides access to the specified patch of a global array. Returns array of leading dimensions ld and a pointer to the first element in the patch. This routine allows to access directly, in place elements in the local section of a global array. It useful for writing new GA operations. A call to ga_access normally follows a previous call to ga_distribution that returns coordinates of the patch associated with a processor. You need to make sure that the coordinates of the patch are valid (test values returned from ga_distribution).

Each call to ga_access has to be followed by a call to either ga_release or ga_release_update. You can access in this fashion only local data. Since the data is shared with other processes, you need to consider issues of mutual exclusion.
This operation is local.

GA_ACCESS_BLOCK_SEGMENT

```
void NGA_Access_block_segment(int g_a, int proc, void *ptr, int len)

g_a              - array handle                    [input]
proc             - processor ID                    [input]
ptr              - pointer to locally held data    [output]
len              - length of data on processor     [output]
```

This function can be used to gain access to the all the locally held data on a particular processor that is associated with a block-cyclic distributed array. Once the index has been returned, local data can be accessed as described in the documentation for NGA_Access. The parameter len is the number of data elements that are held locally. The data inside this segment has a lot of additional structure so this function is not generally useful to developers. It is primarily used inside the GA library to implement other GA routines. Each call to ga_access_block_segment should be followed by a call to either NGA_Release_block_segment or NGA_Release_update_block_segment.

This is a local operation.

GA_ACCESS_BLOCK

```
void NGA_Access_block(int g_a, int idx, int index, int ld[])

g_a              - array handle                 [input]
ndim             - number of array dimensions
idx              - block index                  [input]
index            - pointer to locally held block [output]
ld[ndim-1]       - array of leading dimensions  [output]
```

This function can be used to gain direct access to the data represented by a single block in a global array with a block-cyclic data distribution. The index idx is the index of the block in the array assuming that blocks are numbered sequentially in a column-major order. A quick way of determining whether a block with index idx is held locally on a processor is to calculate whether mod(idx,nproc) equals the processor ID, where nproc is the total number of processors. Once the index has been returned, local data can be accessed as described in the documentation for NGA_Access. Each call to ga_access_block should be followed by a call to either NGA_Release_block or NGA_Release_update_block.

This is a local operation.

GA_ACCESS_BLOCK_GRID

```
void NGA_Access_block_grid(int g_a, int subscript[], void *ptr, int ld[])
```

```
g_a              - array handle                       [input]
ndim             - number of array dimensions
subscript[ndim]  - subscript of block in array        [input]
ptr              - pointer to locally held bloc       [output]
ld[ndim-1]       - array of leading dimensions        [output]
```

This function can be used to gain direct access to the data represented by a single block in a global array with a SCALAPACK block-cyclic data distribution that is based on an underlying processor grid. The subscript array contains the subscript of the block in the array of blocks. This subscript is based on the location of the block in a grid, each of whose dimensions is equal to the number of blocks that fit along that dimension. Once the index has been returned, local data can be accessed as described in the documentation for NGA_Access. Each call to ga_access_block_grid should be followed by a call to either NGA_Release_block_grid or NGA_Release_update_block_grid.

This is a local operation.

GA_RELEASE

```
void NGA_Release(int g_a, int lo[], int hi[])
```

```
g_a        - global array handle                            [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for array section      [input]
hi[ndim]   - array of ending indices for array section        [input]
```

Releases access to a global array when the data was read only.

Your code should look like:

```
NGA_Distribution(g_a, myproc, lo,hi);
NGA_Access(g_a, lo, hi, &ptr, ld);

     <operate on the data referenced by ptr>

GA_Release(g_a, lo, hi);
```

NOTE: see restrictions specified for ga_access
This operation is local.

GA_RELEASE_UPDATE

```
void NGA_Release_update(int g_a, int lo[], int hi[])
```

```
g_a        - global array handle                            [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for array section      [input]
hi[ndim]   - array of ending indices for array section        [input]
```

Releases access to the data. It must be used if the data was accessed for writing. NOTE: see restrictions specified for ga_access.
This operation is local.

GA_RELEASE_BLOCK

```
void NGA_Release_block(int g_a, int index)
```

```
g_a        - array handle                        [input]
index      - block index                         [input]
```

Releases access to the block of data specified by the integer index when data was accessed as read only. This is only applicable to block-cyclic data distributions created using the simple block-cyclic distribution. This is a local operation.

GA_RELEASE_UPDATE_BLOCK

```
void NGA_Release_update_block(int g_a, int index)

g_a         - array handle                      [input]
index       - block index                       [input]
```

Releases access to the block of data specified by the integer index when data was accessed in read-write mode. This is only applicable to block-cyclic data distributions created using the simple block-cyclic distribution. This is a local operation.

GA_RELEASE_BLOCK_GRID

```
void NGA_Release_block_grid(int g_a, int subscript[])

g_a             - array handle                            [input]
ndim            - number of dimensions of the global array
subscript[ndim] - indices of block in array               [input]
```

Releases access to the block of data specified by the subscript array when data was accessed as read only. This is only applicable to block-cyclic data distributions created using the SCALAPACK data distribution. This is a local operation.

GA_RELEASE_UPDATE_BLOCK_GRID

```
void NGA_Release_update_block_grid(int g_a, int subscript[])

g_a             - array handle                            [input]
ndim            - number of dimensions of the global array
subscript[ndim] - indices of block in array               [input]
```

Releases access to the block of data specified by the subscript array when data was accessed in read-write mode. This is only applicable to block-cyclic data distributions created using the SCALAPACK data distribution. This is a local operation.

GA_RELEASE_BLOCK_SEGMENT

```
void NGA_Release_block_segment(int g_a, int iproc)

g_a           - array handle                    [input]
iproc         - processor ID                    [input]
```

Releases access to the block of locally held data for a block-cyclic array, when data was accessed as read-only. This is a local operation.

GA_RELEASE_UPDATE_BLOCK_SEGMENT

```
void NGA_Release_block_segment(int g_a, int iproc)

g_a           - array handle                    [input]
iproc         - processor ID                    [input]
```

Releases access to the block of locally held data for a block-cyclic array, when data was accessed as read-only. This is a local operation.

GA_READ_INC

```
long NGA_Read_inc(int g_a, int subscript[], long inc)
```

```
g_a             - global array handle                          [input]
ndim            - number of dimensions of the global array
subscript[ndim] - subscript array for the referenced element       [input]
```

Atomically read and increment an element in an integer array.

```
*BEGIN CRITICAL SECTION*
old_value = a(subscript)
a(subscript) += inc
*END CRITICAL SECTION*
return old_value
```

This is a one-sided and atomic operation.

GA_SCATTER

```
void NGA_Scatter(int g_a, void *v, int* subsArray[], int n)
```

```
g_a                  - global array handle                     [input]
n                    - number of elements                      [input]
v[n]                 - array containing values                 [input]
ndim                 - number of array dimensions
subsArray[n][ndim]   - array of subscripts for each element    [input]
```

Scatters array elements into a global array. The contents of the input arrays (v,subscrArray) are preserved, but their contents might be (consistently) shuffled on return.

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

This is a one-sided operation.

GA_GATHER

```
void NGA_Gather(int g_a, void *v, int* subsArray[], int n)
```

```
g_a                  - global array handle                     [input]
n                    - number of elements                      [input]
v[n]                 - array containing values                 [input]
ndim                 - number of array dimensions
subsArray[n][ndim]   - array of subscripts for each element    [input]
```

Gathers array elements from a global array into a local array. The contents of the input arrays (v, subscrArray) are preserved, but their contents might be (consistently) shuffled on return.

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

This is a one-sided operation.

GA_SCATTER_ACC

```
void NGA_Scatter_acc(int g_a, void *v, int* subsArray[], int n, void *alpha)
```

```
g_a                  - global array handle                     [input]
n                    - number of elements                      [input]
v[n]                 - array containing values                 [input]
ndim                 - number of array dimensions
subsArray[n][ndim]   - array of subscripts for each element    [input]
alpha                - multiplicative factor                   [input]
```

Scatters array elements from a local array into a global array. Adds values from the local array to existing values in the global array after multiplying by alpha. The contents of the input arrays (v, subscrArray) are preserved, but their contents might be (consistently) shuffled on return.

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

This is a one-sided operation.

GA_ERROR

```
void GA_Error(char *message, int code)

message  - string to print          [input]
code     - code to print            [input]
```

To be called in case of an error. Print an error message and an integer value that represents error code. Releases some system resources. This is the required way of aborting the program execution. This operation is local.

GA_LOCATE

```
int NGA_Locate(int g_a, int subscript[])

g_a              array handle         [input]
subscript[ndim]  element subscript    [output]
```

Return in owner the GA compute process id that 'owns' the data. If any element of subscript[] is out of bounds "-1" is returned.
This operation is local.

GA_LOCATE_REGION

```
int NGA_Locate_region(int g_a, int lo[], int hi[], int map[], int procs[])

g_a            - global array handle                          [input]
ndim           - number of dimensions of the global array
lo[ndim]       - array of starting indices for array section   [input]
hi[ndim]       - array of ending indices for array section     [input]
map[][2*ndim]  - array with mapping information                [output]
procs[nproc]   - list of processes that own a part of array section[output]
```

Return the list of the GA processes id that 'own' the data. Parts of the specified patch might be actually 'owned' by several processes. If lo/hi are out of bounds "0" is returned, otherwise return value is equal to the number of processes that hold the data .

```
map[i][0:ndim-1]         - lo[i]
map[i][ndim:2*ndim-1]    - hi[i]
procs[i]                 - processor id that owns data in patch lo[i]:hi[i]
```

This operation is local.

GA_INQUIRE

```
void NGA_Inquire(int g_a, int *type, int *ndim, int dims[])

g_a  - array handle                [input]
type - data type                   [output]
ndim - number of dimensions        [output]
dims - array of dimensions         [output]
```

Returns data type and dimensions of the array.
This operation is local.

GA_INQUIRE_MEMORY

```
size_t GA_Inquire_memory()
```

Returns amount of memory (in bytes) used in the allocated global arrays on the calling processor.
This operation is local.

GA_INQUIRE_NAME

```
char* GA_Inquire_name(int g_a)

g_a  - array handle                [input]
```

Returns the name of an array represented by the handle g_a.
This operation is local.

GA_NDIM

```
int GA_Ndim(int g_a)

g_a  - array handle                [input]
```

Returns the number of dimensions in array represented by the handle g_a.
This operation is local.

GA_NBLOCK

```
void GA_Nblock(int g_a, int nblock[])

g_a           - array handle                            [input]
nblock[ndim]  - number of partitions for each dimension [output]
```

Given a distribution of an array represented by the handle g_a, returns the number of partitions of each array dimension.
This operation is local.

GA_MEMORY_AVAIL

```
size_t GA_Memory_avail()
```

Returns amount of memory (in bytes) left for allocation of new global arrays on the calling processor.

Note: If GA_uses_ma returns true, then GA_Memory_avail returns the lesser of the amount available under the GA limit and the amount available from MA (according to ma_inquire_avail operation). If no GA limit has been set, it returns what MA says is available.

If (! GA_Uses_ma() && ! GA_Memory_limited()) returns < 0, indicating that the bound on currently available memory cannot be determined.
This operation is local.

GA_USES_MA

```
int GA_Uses_ma()
```

Returns "1" if memory in global arrays comes from the Memory Allocator (MA). "0"means that memory comes from another source, for example System V shared memory is used.
This operation is local.

GA_MEMORY_LIMITED

```
int GA_Memory_limited()
```

Indicates if limit is set on memory usage in Global Arrays on the calling processor. "1" means "yes", "0" means "no".
This operation is local.

GA_PROC_TOPOLOGY

```
void NGA_Proc_topology(int g_a, int proc, int coordinates[])
```

```
g_a               array handle                 [input]
ndim              number of array dimensions
proc              process id                   [input]
coordinates[ndim] coordinates in processor grid [output]
```

Based on the distribution of an array associated with handle g_a, determines coordinates of the specified processor in the virtual processor grid corresponding to the distribution of array g_a. The numbering starts from 0. The values of -1 means that the processor doesn't 'own' any section of array represented by g_a.

This operation is local.

GA_PRINT_FILE

```
void GA_Print_file(FILE *file, int g_a)
```

```
file   - file pointer             [input]
g_a    - array handle             [input]
```

Prints an entire array to a file.

This is a collective operation.

GA_PRINT_PATCH

```
void NGA_Print_patch(int g_a, int lo[],int hi[],int pretty)
```

```
g_a                - array handle              [input]
lo[],hi[]          - coordinates of the patch  [input]
int pretty         - formatting flag           [input]
```

Prints a patch of g_a array to the standard output. If pretty has the value 0 then output is printed in a dense fashion. If pretty has the value 1 then output is formatted and rows/columns labeled.

This is a collective operation.

GA_PRINT

```
void GA_Print(int g_a)
```

```
g_a    - array handle             [input]
```

Prints an entire array to the standard output.

This is a collective operation.

GA_PRINT_STATS

```
void GA_Print_stats()
```

This non-collective (MIMD) operation prints information about:

- 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 logicaly remote locations
- maximum memory consumption in global arrays, and
- number of requests serviced in the interrupt-driven implementations by the calling process.

This operation is local.

GA_PRINT_DISTRIBUTION

```
void GA_Print_distribution(int g_a)
```

```
g_a    - array handle             [input]
```

Prints the array distribution.

This is a collective operation.

GA_CHECK_HANDLE

```
void GA_Check_handle(int g_a, char* string)
```

```
g_a    - array handle               [input]
string - message string             [input]
```

Check that the global array handle g_a is valid ... if not call ga_error with the string provided and some more info.
This operation is local.

GA_INIT_FENCE

```
void GA_Init_fence()
```

Initializes tracing of completion status of data movement operations.
This operation is local.

GA_FENCE

```
void GA_Fence()
```

Blocks the calling process until all the data transfers corresponding to GA operations called after ga_init_fence complete. For example, since ga_put might return before the data reaches the final destination, ga_init_fence and ga_fence allow process to wait until the data tranfer is fully completed:

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

ga_fence must be called after ga_init_fence. A barrier, ga_sync, assures completion of all data transfers and implicitly cancels all outstanding ga_init_fence calls. 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.

ga_fence works for multiple GA operations. For example:

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

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

GA_CREATE_MUTEXES

```
int GA_Create_mutexes(int number)
```

```
number  - number of mutexes in mutex array   [input]
```

Creates a set containing the number of mutexes. Returns 0 if the opereation succeeded or 1 when failed. Mutex is a simple synchronization object used to protect Critical Sections. Only one set of mutexes can exist at a time. Array of mutexes can be created and destroyed as many times as needed.

Mutexes are numbered: 0, ..., number -1.

This is a collective operation.

GA_DESTROY_MUTEXES

```
int GA_Destroy_mutexes()
```

Destroys the set of mutexes created with ga_create_mutexes. Returns 0 if the operation succeeded or 1 when failed.

This is a collective operation.

GA_LOCK

```
void GA_Lock(int mutex)

mutex - mutex object id  [input]
```

Locks a mutex object identified by the mutex number. It is a fatal error for a process to attempt to lock a mutex which was already locked by this process.

GA_UNLOCK

```
void GA_Unlock(int mutex)

mutex  - mutex object id [input]
```

Unlocks a mutex object identified by the mutex number. It is a fatal error for a process to attempt to unlock a mutex which has not been locked by this process.

GA_NODEID

```
int GA_Nodeid()
```

Returns the GA process id (0, ..., ga_Nnodes()-1) of the requesting compute process.
This operation is local.

GA_NNODES

```
int GA_Nnodes()
```

Returns the number of the GA compute (user) processes.
This operation is local.

GA_GEMM

```
void GA_Dgemm(char ta, char tb, int m, int n, int k, double alpha,
              int g_a, int g_b, double beta, int g_c )
void GA_Sgemm(char ta, char tb, int m, int n, int k, float alpha,
              int g_a, int g_b, float beta, int g_c )
void GA_Zgemm(char ta, char tb, int m, int n, int k, DoubleComplex alpha,
              int g_a, int g_b, DoubleComplex beta, int g_c )

g_a, g_b,     - handles to input arrays                        [input]
g_c           - handles to output array                        [output]
ta, tb        - transpose operators                            [input]
m             - number of rows of op(A) and of matrix  C       [input]
```

```
n            - number of columns of op(B) and of matrix  C         [input]
k            - number of columns of op(A) and rows of matrix op(B) [input]
alpha, beta  - scale factors                                       [input]
```

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:
ta = 'N' or 'n', op(A) = A.
ta = 'T' or 't', op(A) = A'.

This is a collective operation.

GA_COPY_PATCH

```
void NGA_Copy_patch(char *trans, int g_a, int alo[], int ahi[],
                                int g_b, int blo[], int bhi[])
```

```
trans                    - transpose operator    [input]
g_a, g_b                 - array handles         [input]
alo[], ahi[]             - g_a patch coordinates [input]
blo[], bhi[]             - g_b patch coordinates [input]
```

Copies elements in a patch of one array into another one. The patches of arrays may be of different shapes but must have the same number of elements. Patches must be nonoverlapping (if g_a=g_b).

trans = 'N' or 'n' means that the transpose operator should **not** be applied.
trans = 'T' or 't' means that transpose operator should be applied.

This is a collective operation.

GA_DOT_PATCH

```
double NGA_Ddot_patch(int g_a, char ta, int alo[], int ahi[],
                      int g_b, char tb, int blo[], int bhi[])
long   NGA_Idot_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[])
```

```
g_a, g_b                - array handles            [input]
alo[], ahi[]            - g_a patch coordinates    [input]
blo[], bhi[]            - g_b patch coordinates    [input]
ta, tb                  - transpose flags          [input]
```

Computes the element-wise dot product of the two (possibly transposed) patches which must be of the same type and have the same number of elements.

This is a collective operation.

GA_MATMUL_PATCH

```
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)
```

```
g_a, g_b, g_c             array handles       [input]
ailo, aihi, ajlo, ajhi    patch of g_a        [input]
```

```
bilo, bihi, bjlo, bjhi    patch of g_b         [input]
cilo, cihi, cjlo, cjhi    patch of g_c         [input]
alpha, beta               scale factors        [input]
transa, transb            transpose operators  [input]
```

ga_matmul_patch is a patch version of ga_dgemm:

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

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

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

Valid values for transpose arguments: 'n', 'N', 't', 'T'. It works for both double and DoubleComplex data tape.

This is a collective operation.

NGA_MATMUL_PATCH

```
void NGA_Matmul_patch(char transa, char transb, void* alpha, void *beta,
                      int g_a, int alo[], int ahi[],
                      int g_b, int blo[], int bhi[],
                      int g_c, int clo[], int chi[])
```

```
g_a, g_b, g_c             array handles        [input]
alo, ahi    patch of g_a         [input]
blo, bhi    patch of g_b         [input]
clo, chi    patch of g_c         [input]
alpha, beta               scale factors        [input]
transa, transb            transpose operators  [input]
```

nga_matmul_patch is a n-dimensional patch version of ga_dgemm:

```
C[clo[]:chi[]] := alpha* AA[alo[]:ahi[]] *
                          BB[blo[]:bhi[]] ) + beta*C[clo[]:chi[]],
```

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

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

Valid values for transpose arguments: 'n', 'N', 't', 'T'. It works for both double and DoubleComplex data tape.

This is a collective operation.

GA_ADD_PATCH

```
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[])
```

```
g_a, g_b, g_c             array handles        [input]
alo[], ahi[]              patch of g_a         [input]
blo[], bhi[]              patch of g_b         [input]
clo[], chi[]              patch of g_c         [input]
alpha, beta               scale factors        [input]
```

Patches of arrays (which must have the same number of elements) are added together element-wise.

c[ ][ ] = alpha * a[ ][ ] + beta * b[ ][ ].

This is a collective operation.

GA_FILL_PATCH

```
void NGA_Fill_patch (int g_a, int lo[], int hi[], void *val)
```

```
g_a                     array handles       [input]
lo[], hi[]              patch of g_a        [input]
val                     value to fill       [input]
```

Fill the patch of g_a with value of 'val'

This is a collective operation.

GA_ZERO_PATCH

```
void NGA_Zero_patch (int g_a, int lo[], int hi[])
```

```
g_a                     array handles       [input]
lo[], hi[]              patch of g_a        [input]
```

Set all the elements in the patch to zero.

This is a collective operation.

GA_SCALE_PATCH

```
void NGA_Scale_patch (int g_a, int lo[], int hi[], void *val)
```

```
g_a                     array handles       [input]
lo[], hi[]              patch of g_a        [input]
val                     scale factor        [input]
```

Scale an array by the factor 'val'

This is a collective operation.

GA_BRDCST

```
void GA_Brdcst(void *buf, int lenbuf, int root)
```

```
lenbuf       - length of buffer (bytes)  [input]
buf[lenbuf]  - data                      [input/output]
root         - root process              [input]
```

Broadcast from process root to all other processes a message of length lenbuf.

This is operation is provided only for convenience purposes: it is available regardless of the message-passing library that GA is running with.

This is a collective operation.

GA_DGOP

```
void GA_Dgop(double x[], int n, char *op)
```

```
n     - number of elements      [input]
x[n]  - array of elements       [input/output]
op    - operator                [input]
```

Double Global OPeration.

X(1:N) is a vector present on each process. DGOP 'sums' elements of X accross all nodes using the commutative operator OP. The result is broadcast to all nodes. Supported operations include '+', '*', 'max', 'min', 'absmax', 'absmin'. The use of lowerecase for operators is necessary.

This is operation is provided only for convenience purposes: it is available regardless of the message-passing library that GA is running with.

This is a collective operation.

GA_IGOP

```
void GA_Igop(long x[], int n, char *op)

n     - number of elements      [input]
x[n]  - array of elements       [input/output]
op    - operator                [input]
```

Integer Global OPeration. The integer (more precisely long) version of ga_dgop described above, also include the bitwise OR operation.

This is operation is provided only for convenience purposes: it is available regardless of the message-passing library that GA is running with.

This is a collective operation.

GA_LGOP

```
void GA_Lgop(long x[], int n, char *op)

n     - number of elements      [input]
x[n]  - array of elements       [input/output]
op    - operator                [input]
```

Long Global OPeration. The long version of ga_dgop described above, also include the bitwise OR operation.

This is operation is provided only for convenience purposes: it is available regardless of the message-passing library that GA is running with.

This is a collective operation.

GA_CLUSTER_NNODES

```
int GA_Cluster_nnodes()
```

This functions 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.

This is a local operation.

GA_CLUSTER_NODEID

```
int GA_Cluster_nodeid()
```

This function 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.

This is a local operation.

GA_CLUSTER_PROC_NODEID

```
int GA_Cluster_proc_nodeid(int proc)
```

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

This is a local operation.

GA_CLUSTER_NPROCS

```
int GA_Cluster_nprocs(int inode)
```

```
inode               [input]
```

This function returns the number of processors available on node inode.

This is a local operation.

GA_CLUSTER_PROCID

```
int GA_Cluster_procid(int inode, int iproc)
```

```
inode,iproc         [input]
```

This function 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.

This is a local operation.

GA_DIAG

```
void GA_Diag(int g_a, int g_s, int g_v, void *eval)
```

```
g_a     - Matrix to diagonalize          [input]
g_s     - Metric                         [input]
g_v     - Global matrix to return evecs  [output]
eval    - Local array to return evals    [output]
```

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

This is a collective operation.

GA_DIAG_REUSE

```
void GA_Diag_reuse(int control, int g_a, int g_s, int g_v, void *eval)
```

```
control - Control flag                   [input]
g_a     - Matrix to diagonalize          [input]
g_s     - Metric                         [input]
g_v     - Global matrix to return evecs  [output]
eval    - Local array to return evals    [output]
```

Solve the generalized eigen-value problem returning all eigen-vectors and values in ascending order. Recommended for REPEATED calls if g_s is unchanged. Values of the control flag:

value	action/purpose
0	indicates first call to the eigensolver
>0	consecutive calls (reuses factored g_s)
<0	only erases factorized g_s; g_v and eval unchanged (should be called after previous use if another eigenproblem, i.e., different g_a and g_s, is to be solved)

The input matrices are not destroyed.

This is a collective operation.

GA_DIAG_STD

```
void GA_Diag_std(int g_a, int g_v, void *eval)

g_a     - Matrix to diagonalize          [input]
g_v     - Global matrix to return evecs  [output]
eval    - Local array to return evals    [output]
```

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

This is a collective operation.

GA_LLT_SOLVE

```
int GA_Llt_solve(int g_a, int g_b)

g_a     - coefficient matrix        [input]
g_b     - rhs/solution matrix       [output]
```

Solves a system of linear equations

A * X = B

using the Cholesky factorization of an NxN double precision symmetric positive definite matrix A (epresented by handle g_a). On successful exit B will contain the solution X.

It returns:

= 0 : successful exit
> 0 : the leading minor of this order is not positive
    definite and the factorization could
    not be completed

This is a collective operation.

GA_LU_SOLVE

```
void GA_Lu_solve(char trans, int g_a, int g_b)

trans   - transpose or not transpose [input]
g_a     - coefficient matrix         [input]
g_b     - rhs/solution matrix        [output]
```

Solve the system of linear equations op(A)X = B based on the LU factorization.

op(A) = A or A' depending on the parameter trans:
  trans = 'N' or 'n' means that the transpose operator should not be applied.
  trans = 'T' or 't' means that the transpose operator should be applied.

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.

This is a collective operation.

GA_SOLVE

```
int GA_solve(int g_a, int g_b)

g_a     - coefficient matrix         [input]
g_b     - rhs/solution matrix        [output]
```

Solves a system of linear equations

A * X = B

It first will call the Cholesky factorization routine and, if sucessfully, will solve the system with the Cholesky solver. If Cholesky will be not be 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.

It returns

= 0 : Cholesky factoriztion was succesful
> 0 : the leading minor of this order
    is not positive definite, Cholesky factorization
    could not be completed and LU factoriztion was used

This is a collective operation.

GA_SPD_INVERT

```
int GA_Spd_invert(int g_a)
```

```
g_a     - matrix         [input/output]
```

It computes the inverse of a double precision 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.

It returns

= 0 : successful exit
> 0 : the leading minor of this order is not positive
    definite and the factorization could not be completed
< 0 : it returns the index i of the (i,i)
    element of the factor L/U that is zero and
    the inverse could not be computed

This is a collective operation.

GA_SELECT_ELEM

```
void NGA_Select_elem(int g_a, char *op, void* val, int index[])
```

```
g_a             - array handle Control                [input]
op              - operator {"min","max"}              [input]
val             - address where value should be stored [output]
index[ndim]     - array index for the selected element [output]
```

Returns the value and index for an element that is selected by the specified operator in a global array corresponding to g_a handle.
This is a collective operation.

GA_SUMMARIZE

```
void GA_Summarize(int verbose)
```

```
verbose     - If true print distribution info [input]
```

Prints info about allocated arrays.

GA_SYMMETRIZE

```
void GA_Symmetrize(int g_a)
```

```
g_a             [input]
```

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

This is a collective operation.

GA_TRANSPOSE

```
void GA_Transpose(int g_a, int g_b)

g_a     - original matrix         [input]
g_b     - solution matrix         [output]
```

Transposes a matrix: B = A', where A and B are represented by handles g_a and g_b.

This is a collective operation.

GA_ABS_VALUE

```
void GA_Abs_value(int g_a)

g_a    - array handle                  [input]
```

Take element-wise absolute value of the array.
This is a collective operation.

GA_ABS_VALUE_PATCH

```
void GA_Abs_value_patch(int g_a, int lo[], int hi[])

g_a          - array handle                        [input]
lo[], hi[]   - g_a patch coordinates               [input]
```

Take element-wise absolute value of the patch.
This is a collective operation.

GA_ADD_CONSTANT

```
void GA_Add_constant(int g_a, void *alpha)

g_a                                   - array handle    [input]
double/complex/int/long/float *alpha  - added value     [input]
```

Add the constant pointed by alpha to each element of the array.
This is a collective operation.

GA_ADD_CONSTANT_PATCH

```
void GA_Add_constant_patch(int g_a, int lo[], int hi[], void *alpha)

g_a                                  - array handle        [input]
lo[], hi[]                           - patch coordinates   [input]
double/complex/int/long/float *alpha - added value         [input]
```

Add the constant pointed by alpha to each element of the patch.
This is a collective operation.

GA_RECIP

```
void GA_Recip(int g_a)

g_a         - array handle                  [input]
```

Take element-wise reciprocal of the array.
This is a collective operation.

GA_RECIP_PATCH

```
void GA_Recip_patch(int g_a, int lo[], int hi[])

g_a               - array handle              [input]
lo[], hi[]        - patch coordinates         [input]
```

Take element-wise reciprocal of the patch.
This is a collective operation.

GA_ELEM_MULTIPLY

```
void GA_Elem_multiply(int g_a, int g_b, int g_c)

g_a, g_b          - array handles                [input]
g_c               - array handle                 [output]
```

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).
This is a collective operation.

GA_ELEM_MULTIPLY_PATCH

```
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[])

g_a, g_b           - array handles                   [input]
g_c                - array handle                    [output]
alo[], ahi[]       - g_a patch coordinates           [input]
blo[], bhi[]       - g_b patch coordinates           [input]
clo[], chi[]       - g_c patch coordinates           [output]
```

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).
This is a collective operation.

GA_ELEM_DIVIDE

```
void GA_Elem_divide(int g_a, int g_b, int g_c)

g_a, g_b           - array handles                   [input]
g_c                - array handle                    [output]
```

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.

This is a collective operation.

GA_ELEM_DIVIDE_PATCH

```
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[])

g_a, g_b            - array handles                 [input]
g_c                 - array handle                  [output]
alo[], ahi[]        - g_a patch coordinates         [input]
blo[], bhi[]        - g_b patch coordinates         [input]
clo[], chi[]        - g_c patch coordinates         [output]
```

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).
This is a collective operation.

GA_ELEM_MAXIMUM

```
void GA_Elem_maximum(int g_a, int g_b, int g_c)

g_a, g_b            - array handles                 [input]
g_c                 - array handle                  [output]
```

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).
This is a collective operation.

GA_ELEM_MAXIMUM_PATCH

```
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[])

g_a, g_b            - array handles                 [input]
g_c                 - array handle                  [output]
alo[], ahi[]        - g_a patch coordinates         [input]
blo[], bhi[]        - g_b patch coordinates         [input]
clo[], chi[]        - g_c patch coordinates         [output]
```

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).
This is a collective operation.

GA_ELEM_MINIMUM

```
void GA_Elem_minimum(int g_a, int g_b, int g_c)

g_a, g_b           - array handles                    [input]
g_c                - array handle                     [output]
```

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).
This is a collective operation.

GA_ELEM_MINIMUM_PATCH

```
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[])

g_a, g_b           - array handles                    [input]
g_c                - array handle                     [output]
alo[], ahi[]       - g_a patch coordinates            [input]
blo[], bhi[]       - g_b patch coordinates            [input]
clo[], chi[]       - g_c patch coordinates            [output]
```

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).
This is a collective operation.

GA_SHIFT_DIAGONAL

```
void GA_Shift_diagonal(int g_a, void *c)

g_a                             - array handle     [input]
double/complex/int/long/float   - shift value      [input]
```

Adds this constant to the diagonal elements of the matrix.
This is a collective operation.

GA_SET_DIAGONAL

```
void GA_Set_diagonal(int g_a, int g_v)

g_a, g_v          - array handles                [input]
```

Sets the diagonal elements of this matrix g_a with the elements of the vector g_v.
This is a collective operation.

GA_ZERO_DIAGONAL

```
void GA_Zero_diagonal(int g_a)

g_a                - array handle                  [input]
```

Sets the diagonal elements of this matrix g_a with zeros.
This is a collective operation.

GA_ADD_DIAGONAL

```
void GA_Add_diagonal(int g_a, int g_v)

g_a, g_v            - array handles                [input]
```

Adds the elements of the vector g_v to the diagonal of this matrix g_a.
This is a collective operation.

GA_GET_DIAG

```
void GA_Get_diag(int g_a, int g_v)

g_a, g_v            - array handles                [input]
```

Inserts the diagonal elements of this matrix g_a into the vector g_v.
This is a collective operation.

GA_SCALE_ROWS

```
void GA_Scale_rows(int g_a, int g_v)

g_a, g_v            - array handles                [input]
```

Scales the rows of this matrix g_a using the vector g_v.
This is a collective operation.

GA_SCALE_COLS

```
void GA_Scale_cols(int g_a, int g_v)

g_a, g_v            - array handles                [input]
```

Scales the columns of this matrix g_a using the vector g_v.
This is a collective operation.

GA_NORM1

```
void GA_norm1(int g_a, double *nm)

g_a                 - array handle                  [input]
nm                  - matrix/vector 1-norm value    [output]
```

Computes the 1-norm of the matrix or vector g_a.
This is a collective operation.

GA_NORM_INFINITY

```
void GA_Norm_infinity(int g_a, double *nm)
```

```
g_a                  - array handle                      [input]
nm                   - matrix/vector infinity-norm value [output]
```

Computes the 1-norm of the matrix or vector g_a.
This is a collective operation.

GA_MEDIAN

```
void GA_Median(int g_a, int g_b, int g_c, int g_m)

g_a, g_b, g_c        - array handles                    [input]
g_m                  - array handle                     [output]
```

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).
This is a collective operation.

GA_MEDIAN_PATCH

```
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[])

g_a, g_b, g_c        - array handles                     [input]
g_m                  - array handle                      [output]
alo[], ahi[]         - g_a patch coordinates             [input]
blo[], bhi[]         - g_b patch coordinates             [input]
clo[], chi[]         - g_c patch coordinates             [input]
mlo[], mhi[]         - g_m patch coordinates             [output]
```

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).
This is a collective operation.

GA_STEP_MAX

```
void GA_Step_max(int g_a, int g_b, double *step)

g_a, g_b              - array handles where g_b is step direction [input]
step                  - maximum step size                         [output]
```

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 non-negative.
This is a collective operation.

GA_STEP_MAX2

```
void GA_Step_max2(int g_xx, int g_vv, int g_xxll, int g_xxuu, double * step2)

g_xx                 - array handle                 [input]
g_vv                 - step direction array handle  [input]
g_xxll               - lower bounds array handle    [input]
g_xxuu               - upper bounds array handle    [input]
step2                - maximum step size            [output]
```

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

GA_STEP_MAX_PATCH

```
void GA_Step_max_patch(int g_a, int alo[], int ahi[], int g_b, blo[], bhi[],
                       double *step)
```

```
g_a, g_b     - array handles where g_b is step direction [input]
step         - the maximum step                          [output]
```

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 non-negative.
This is a collective operation.

GA_STEP_MAX2_PATCH

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

```
g_xx                 - array handle                [input]
g_vv                 - step direction array handle [input]
g_xxll               - lower bounds array handle   [input]
g_xxuu               - upper bounds array handle   [input]
step2                - maximum step size           [output]
xxlo[], xxhi[]       - g_xx patch coordinates      [input]
vvlo[], vvhi[]       - g_vv patch coordinates      [input]
xxlllo[], xxllhi[]   - g_xxll patch coordinates    [input]
xxuulo[], xxuuhi[]   - g_xxuu patch coordinates    [output]
```

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

GA_PGROUP_GET_DEFAULT

```
int GA_Pgroup_get_default()
```

This function will return a handle to the default processor group ,which can then be used to create a global array using one of the NGA_create_*_config or GA_Set_pgroup calls.

This is a local operation.

GA_PGROUP_GET_MIRROR

```
int GA_Pgroup_get_mirror()
```

This function will return a handle to the mirrored processor group, which can then be used to create a global array using one of the NGA_create_*_config or GA_Set_pgroup calls.

This is a local operation.

GA_PGROUP_GET_WORLD

```
int GA_Pgroup_get_world()
```

This function will return a handle to the world processor group, which can then be used to create a global array using one of the NGA_create_*_config or GA_Set_pgroup calls.

This is a local operation.

GA_PGROUP_SYNC

```
void GA_Pgroup_sync(int p_handle)
```

```
p_handle                  processor group handle [input]
```

This operation executes a synchronization group across the processors in the processor group specified by p_handle. Nodes outside this group are unaffected.

This is a collective operation on the processor group specified by p_handle.

GA_PGROUP_BRDCST

```
void GA_Pgroup_brdcst(int p_handle, void* buf, int lenbuf, int root)
```

p_handle	processor group handle	[input]
buf	pointer to buffer containing data	[input/output]
lenbuf	length of data (in bytes)	[input]
root	processor sending message	[input]

Broadcast data from processor specified by root to all other processors in the processor group specified by p_handle. The length of the message in bytes is specified by lenbuf. The initial and broadcasted data can be found in the buffer specified by the pointer buf.

This is a collective operation on the processor group specified by p_handle.

GA_PGROUP_DGOP

```
void GA_Pgroup_dgop(int p_handle, double buf*, int n, char* op)
```

p_handle	processor group handle	[input]
buf	buffer containing data	[input/output]
n	number of elements in x	[input]
op	operation to be performed	[input]

buf[n] is a double precision array present on each processor in the processor group p_handle. The GA_Pgroup_dgop 'sums' all elements in buf[n] across all processors in the group specified by p_handle using the commutative operation specified by the character string op. The result is broadcast to all processor in p_handle. Allowed strings are "+", "*", "max", "min", "absmax", "absmin". The use of lowerecase for operators is necessary.

This is a collective operation on the processor group specifed by p_handle.

GA_PGROUP_IGOP

```
void GA_Pgroup_igop(int p_handle, double buf*, int n, char* op)
```

p_handle	processor group handle	[input]
buf	buffer containing data	[input/output]
n	number of elements in x	[input]
op	operation to be performed	[input]

buf[n] is an integer array present on each processor in the processor group p_handle. The GA_Pgroup_igop 'sums' all elements in buf[n] across all processors in the group specified by p_handle using the commutative operation specified by the character string op. The result is broadcast to all processor in p_handle. Allowed strings are "+", "*", "max", "min", "absmax", "absmin". The use of lowerecase for operators is necessary.

This is a collective operation on the processor group specifed by p_handle.

GA_PGROUP_LGOP

```
void GA_Pgroup_lgop(int p_handle, double buf*, int n, char* op)
```

p_handle	processor group handle	[input]
buf	buffer containing data	[input/output]
n	number of elements in x	[input]
op	operation to be performed	[input]

buf[n] is a long integer array present on each processor in the processor group p_handle. The GA_Pgroup_lgop 'sums' all elements in buf[n] across all processors in the group specified by p_handle using the commutative operation specified by the character string op. The result is broadcast to all processor in p_handle. Allowed strings are "+", "*", "max", "min", "absmax", "absmin". The use of lowerecase for operators is necessary.

This is a collective operation on the processor group specifed by p_handle.

GA_PGROUP_FGOP

```
void GA_Pgroup_lgop(int p_handle, double buf*, int n, char* op)
```

p_handle	processor group handle	[input]
buf	buffer containing data	[input/output]
n	number of elements in x	[input]
op	operation to be performed	[input]

buf[n] is a single precision array present on each processor in the processor group p_handle. The GA_Pgroup_fgop 'sums' all elements in buf[n] across all processors in the group specified by p_ handle using the commutative operation specified by the character string op. The result is broadcast to all processor in p_handle. Allowed strings are "+", "*", "max", "min", "absmax", "absmin". The use of lowerecase for operators is necessary.

This is a collective operation on the processor group specifed by p_handle.

GA_PGROUP_NNODES

```
int GA_Pgroup_nnodes(int p_handle)
```

p_handle	processor group handle	[input]

This function returns the number of processors contained in the group specified by p_handle.

This is a local local operation.

GA_PGROUP_NODEID

```
int GA_Pgroup_nodeid(int p_handle)
```

p_handle	processor group handle	[input]

This function returns the relative index of the processor in the processor group specified by p_ handle. This index will generally differ from the absolute processor index returned by GA_Nodeid if the processor group is not the world group.

This is a local operation.

GA_MERGE_MIRRORED

```
int GA_Merge_mirrored(int g_a)
```

g_a	array handles	[input]

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.

GA_IS_MIRRORED

```
int GA_Is_mirrored(int g_a)
```

g_a	array handle	[input]

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.

GA_MERGE_DISTR_PATCH

```
int NGA_Merge_distr_patch(int g_a, int alo[], int ahi[], int g_b, int blo[], int bhi[])
```

```
g_a, g_b                 - array handles         [input]
alo[], ahi[]             - g_a patch coordinates [input]
blo[], bhi[]             - g_b patch coordinates [input]
```

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 a collective operation.

GA_NBGET

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

```
g_a        - global array handle                                          [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section           [input]
hi[ndim]   - array of ending indices for global array section             [input]
buf        - pointer to the local buffer array where the data goes        [output]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
nbhandle   - pointer to the non-blocking request handle                   [input]
```

Non-blocking version of the blocking get operation. The get operation can be completed locally by making a call to the wait (e.g.NGA_NbWait) routine.

This is a non-blocking one-sided operation.

GA_NBPUT

```
void NGA_NbPut(int g_a, int lo[], int hi[], void* buf, int ld[], ga_nbhdl_t* nbhandle)
```

```
g_a        - global array handle                                          [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section           [input]
hi[ndim]   - array of ending indices for global array section             [input]
buf        - pointer to the local buffer array where the data is          [input]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
nbhandle   - pointer to the non-blocking request handle                   [input]
```

Non-blocking version of the blocking put operation. The put operation can be completed locally by making a call to the wait (e.g.NGA_NbWait) routine.

This is a non-blocking one-sided operation.

GA_NBACC

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

```
g_a        - global array handle                                          [input]
ndim       - number of dimensions of the global array
lo[ndim]   - array of starting indices for global array section           [input]
hi[ndim]   - array of ending indices for global array section             [input]
buf        - pointer to the local buffer array where the data is          [input]
ld[ndim-1] - array specifying leading dimensions/strides/extents for buffer array [input]
```

```
double/DoubleComplex/long *alpha -      scale factor                                [input]
nbhandle   - pointer to the non-blocking request handle                             [input]
```

Non-blocking version of the blocking accumulate operation. The accumulate operation can be completed locally by making a call to the wait (e.g.NGA_NbWait) routine.

This is a non-blocking one-sided operation.

GA_NBWAIT

```
void NGA_NbWait(ga_nbhdl_t* nbhandle)

nbhandle   - pointer to the non-blocking request handle                             [input]
```

This 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.

GA_WTIME

```
double GA_Wtime()
```

This function return a wall (or elapsed) time on the calling processor. Returns time in seconds representing elapsed wall-clock time since an arbitrary time in the past. Example:

```
double starttime, endtime;
starttime = GA_Wtime();
.... code snippet to be timed ....
endtime   = GA_Wtime();
printf("Time taken = %lf seconds\n", endtime-starttime);
```

This is a local operation.
This function is only available in release 4.1 or greater.

GA_SET_DEBUG

```
void GA_Set_debug(int dbg)

dbg        - value to set internal flag            [input]
```

This function sets an internal flag in the GA library to either true or false. The value of this flag can be recovered at any time using the GA_Get_debug function. The flag is set to false when the the GA library is initialized. This can be useful in a number of debugging situations, especially when examining the behavior of routines that are called in multiple locations in a code.

This is a local operation.

GA_GET_DEBUG

```
int GA_Get_debug()
```

This function returns the value of an internal flag in the GA library whose value can be set using the GA_Set_debug subroutine.

This is a local operation.

GA_PATCH_ENUM

```
void GA_Patch_enum(int g_a, int lo, int hi, int istart, int inc)

g_a              - array handle                    [input]
lo, hi           - patch coordinates               [input]
istart           - starting value of enumeration   [input]
inc              - increment value                 [input]
```

This subroutine enumerates the values of an array between elements lo and hi starting with the value istart and incrementing each subsequent value by inc. This operation is only applicable to 1-dimensional arrays. An example of its use is shown below:

```
GA_Patch_enum(g_a, 1, n, 7, 2);
g_a:  7  9 11 13 15 17 19 21 23 ...
```

This is a collective operation.

GA_SCAN_ADD

```
void GA_Scan_add(int g_src, int g_dest, int g_mask, int lo, int hi, int excl)

g_src        - handle for source array                      [input]
g_dest       - handle for destination array                 [output]
g_mask       - handle for integer array representing mask   [input]
lo, hi,      - low and high values of range on which operation
               is performed                                 [input]
excl         - value to signify if masked values are included in
               in add                                       [input]
```

This operation will add successive elements in a source vector g_src and put the results in a destination vector g_dest. The addition will restart based on the values of the integer mask vector g_mask. The scan is performed within the range specified by the integer values lo and hi. Note that this operation can only be applied to 1-dimensional arrays. The excl flag determines whether the sum starts with the value in the source vector corresponding to the location of a 1 in the mask vector (excl=0) or whether the first value is set equal to 0 (excl=1). Some examples of this operation are given below.

```
GA_Scan_add(g_src, g_dest, g_mask, 1, n, 0);
g_mask:   1  0  0  0  0  0  1  0  1  0  0  1  0  0  1  1  0
g_src:    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
g_dest:   1  3  6 10 16 21  7 15  9 19 30 12 25 39 15 16 33

GA_Scan_add(g_src, g_dest, g_mask, 1, n, 1);
g_mask:   1  0  0  0  0  0  1  0  1  0  0  1  0  0  1  1  0
g_src:    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
g_dest:   0  1  3  6 10 15  0  7  0  9 19  0 12 25  0  0 16
```

This is a collective operation.

GA_SCAN_COPY

```
void GA_Scan_copy(int g_src, int g_dest, int g_mask, int lo, int hi)

g_src        - handle for source array                      [input]
g_dest       - handle for destination array                 [output]
g_mask       - handle for integer array representing mask   [input]
lo, hi,      - low and high values of range on which operation
               is performed                                 [input]
```

This subroutine does a segmented scan-copy of values in the source array g_src into a destination array g_dest with segments defined by values in the integer mask array g_mask. The scan-copy operation is only applied to the range between the lo and hi indices. This operation is restriced to 1-dimensional arrays. The resulting destination array will consist of segments of consecutive elements with the same value. An example is shown below

```
GA_Scan_copy(g_src, g_dest, g_mask, 1, n);
g_mask:   1  0  0  0  0  0  1  0  1  0  0  1  0  0  1  1  0
```

```
g_src:    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
g_dest:   1  1  1  1  1  1  7  7  9  9  9 12 12 12 15 16 16
```

This is a collective operation.

GA_PACK

```
void GA_Pack(int g_src, int g_dest, int g_mask, int lo, int hi, int *icount)
```

```
g_src         - handle for source array                           [input]
g_dest        - handle for destination array                      [output]
g_mask        - handle for integer array representing mask        [input]
lo, hi,       - low and high values of range on which operation
                is performed
integer       - number of values in compressed array              [output]
```

The pack subroutine is designed to compress the values in the source vector g_src into a smaller destination array g_dest based on the values in an integer mask array g_mask. The values lo and hi denote the range of elements that should be compressed and icount is a variable that on output lists the number of values placed in the compressed array. This operation is the complement of the GA_Unpack operation. An example is shown below

```
GA_Pack(g_src, g_dest, g_mask, 1, n, &icount);
g_mask:   1  0  0  0  0  0  1  0  1  0  0  1  0  0  1  1  0
g_src:    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
g_dest:   1  7  9 12 15 16
icount:   6
```

This is a collective operation.

GA_UNPACK

```
void GA_Unpack(int g_src, int g_dest, int g_mask, int lo, int hi, int *icount)
```

```
g_src         - handle for source array                           [input]
g_dest        - handle for destination array                      [output]
g_mask        - handle for integer array representing mask        [input]
lo, hi,       - low and high values of range on which operation
                is performed
integer       - number of values in uncompressed array              [output]
```

The unpack subroutine is designed to expand the values in the source vector g_src into a larger destination array g_dest based on the values in an integer mask array g_mask. The values lo and hi denote the range of elements that should be compressed and icount is a variable that on output lists the number of values placed in the uncompressed array. This operation is the complement of the GA_Pack operation. An example is shown below

```
GA_Unpack(g_src, g_dest, g_mask, 1, n, &icount);
g_src:    1  7  9 12 15 16
g_mask:   1  0  0  0  0  0  1  0  1  0  0  1  0  0  1  1  0
g_dest:   1  0  0  0  0  0  7  0  9  0  0 12  0  0 15 16  0
icount:   6
```

This is a collective operation.

GA_LIST_NODEID
GA_MPI_COMMUNICATOR

obsolete functions as of release 3.0

Additional Explanations

Global arrays are distributed array objects supported in a message-passing program through the GA library calls. They can be created, destroyed and manipulated using a set of GA operations.

Attributes of GA operations

one-sided/independent - accesses shared/remote data without the remote process (data owner) cooperation -- unlike send/receive operations in MPI
collective - requires all processes to make the call, otherwise the code will hang
local - operation is local to each process and does not require communication
atomic - operation has mutual exclusion built in: concurrent accesses to the same data will be serialized
non-blocking - 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.

Language Interoperability

GA provides C and Fortran interfaces in the same mixed-language program to the same array objects. Because of the language interoperability, the C interface is influenced by some of the Fortran conventions. These are the major considerations:

- The set of supported datatypes is limited to double, Integer (defined as int on 32-bit and long on 64-bit platforms), and DoubleComplex. DoubleComplex is defined as a struct of two doubles holding real and imaginary parts of complex numbers.
- The underlying data layout uses Fortran convention (first array dimension changes first). This requires the GA C interface to swap order of dimensions and subscripts in function arguments passed through the C interface to be consistent with the C view of multidimensional array layout and indexing. This conversion makes GA multidimensional arrays look consistent with the C conventions for data layout of local multidimensional arrays at small extra cost (a fraction of a microsecond) associated with copying arguments. However, the conversion is avoided by building the package with USE_FAPI flag set which would change the array addressing/layout conventions to follow Fortran.

Memory Allocation an Usage

GA consumes memory for storing global arrays. The memory is either private to each process or shared. Local memory is allocated using MA (Memory Allocator library). Any temporary workspace GA is using comes from a statically allocated internal pool of memory or MA. This requires MA to be initilized before GA.

Higher-dimensional Array API

All releases of GA <= 2.4 supported only 2-dimensional arrays. Since release 3.0 arrays can have up to 7 dimensions. This limit can be increased at compile time. For backward compatibility reasons, the original explicitly 2-dimensional interfaces had to be preserved. Therefore, some GA operations have two APIs available. They are identified with "GA" or "NGA" prefix that correspond to either the old 2-dimensional or the new arbitrary(N)-dimensional version, respectively.

Note

This documentation describes the C API available in the version 3.0 (or higher) of GA.