The Global Arrays Toolkit
“Shared-Memory” Programming for “Distributed-Memory” Computers

Tim Stitt PhD
stitt@cscs.ch

Swiss National Supercomputing Centre
Manno, Switzerland

7th July 2008
Part I

Introduction
The Global Arrays (GA) Toolkit is an API for providing a portable “shared-memory” programming interface for “distributed-memory” computers.
Introduction
Shared-Memory Systems

The Global Arrays Toolkit
CSCS

Introduction
Global Arrays Toolkit - A Definition
Overview of Standard HPC Architectures
Global Arrays
The Global Arrays Toolkit

Advantages
1. Global view of shared data (global indexing)
2. Data mapping usually corresponds to original problem
3. More intuitive and (arguably) simpler programming paradigm

Disadvantages
1. Details of data locality is obscured
2. Programmer needs to avoid race conditions and synchronise updates to shared resources
Introduction
Distributed-Memory Systems

Advantages
1. Data locality is explicit
2. Highly scalable
3. Generally requires less expensive hardware

Disadvantages
1. More difficult programming paradigm
2. More complicated data access
3. Programmer needs to manage all communication and synchronisation
Introduction

Global Arrays - The Best of Both Worlds?

The Global Arrays Toolkit

Features

1. Distributed multidimensional arrays accessed through a shared-memory programming style
2. Single shared data structure (with global indexing)
3. Scalability of distributed-memory systems
4. Only useable for array data structures
Introduction
Global Arrays’ Model of Computation

Global Arrays Toolkit - A Definition
Overview of Standard HPC Architectures
Global Arrays
The Global Arrays Toolkit

Global Array
COPY TO LOCAL MEMORY
GET()

Global Array
COPY TO SHARED ARRAY
PUT()

Local Memory

COMPUTE
Introduction
Global Arrays Toolkit

1. Shared-memory model in context of distributed arrays
2. Much simpler than message-passing for many applications
3. Complete environment for parallel code development (including both task- and data-parallelism)
4. Compatible with MPI including packages such as PETSc
5. Data locality control similar to distributed-memory/message-passing model
6. Library-based; no special compiler required
7. Scalable
8. One-sided communication (i.e. no co-operative hand-shaking) for point-point communications
Some Facts

1. Developed by Jarek Nieplochal et al. at Pacific Northwest National Laboratory (PNNL) [1]

2. The GA Toolkit has been public-domain since 1994

3. Employed in several large codes: NWChem, GAMESS-UK and MOLPRO

4. Language interfaces include: Fortran, C, C++ and Python (approx. 200 routines[4])

5. Implements both blocking and non-blocking local/remote memory access
Introduction
Structure of Global Arrays Toolkit

Global Arrays Toolkit - A Definition
Overview of Standard HPC Architectures
Global Arrays
The Global Arrays Toolkit

Distributed Arrays Layer
(Memory Management, Index Translation)

Message Passing
Global Operations

ARMCI
One-sided communication
get(), put()
Introduction
Remote Data Access - Example

Message-Passing

if (PID != 0) then
  pack data in message
  send message to PE(0)
else
  copy local data to array
  do message=1,3
    receive message from ID(n)
    unpack message to array
  end do
end if

GA Toolkit

if (ID == 0) then
  call NGA_GET(ga,lo,hi,array,stride)
end if
The Global Arrays Toolkit

Introduction

Global Arrays Toolkit - A Definition

Overview of Standard HPC Architectures

Global Arrays

The Global Arrays Toolkit

Introduction

NGA_GET() Flowchart

(a) Issue multiple non-blocking one-sided get() calls

(b) Determine ownership and locality

(c) (d) Wait for all data transfers to complete
Latest Stable Version (4.2)

Homepage at http://www.emsl.pnl.gov/docs/global/

Platforms supported (32-bit and 64-bit)
- IBM SP, BlueGene
- Cray X1, XD1, XT3 and XT4
- Linux Clusters with Ethernet, Myrinet, Infiniband, or Quadrics
- Solaris
- Fujitsu
- Hitachi
- NEC
- HP
- Windows
The GA Toolkit requires a number of lower-level libraries for operation:

- MPI[3] (e.g. job startup, run-time execution and collective communications)
- ARMCI (primary communication layer)
- Memory Allocator (MA)
- Disk Resident Arrays (DRA)[7] optional
Introduction
When To Use GA - Rules of Thumb

Guidelines - When To Use GA

1. Applications with dynamic or irregular communication patterns
2. Calculations driven by dynamic load-balancing
3. Need one-sided access to shared data structures
4. Need high-level operations on distributed arrays and/or for out-of-core array-based algorithms (GA+DRA)
5. When message-passing coding becomes too complicated
6. Portability and performance is important
Introduction
When Not To GA Use - Rules of Thumb

Guidelines - When Not To Use GA

1. Require nearest neighbour communications with regular communication patterns
2. When synchronisation with cooperative point-point communication is required (e.g. Cholesky Factorisation)
3. When compiler optimisation and parallelisation is more effective
4. Parallel language support and compiler tools are sufficient
Part II

GA Programming Basics
GA Basics
Simplest GA Program Template

Fortran Template

```fortran
program GA_template
    use mpi
    implicit none

    ! Include GA Headers
    include "mafdecls.fh"
    include "global.fh"

    integer :: error

    ! Initialize Message-Passing
    call mpi_init(error)

    ! Initialize GA Library
    call ga_initialize()

    ... processing ...

    ! Terminate GA Library
    call ga_terminate()

    ! Terminate Message-Passing Lib
    call mpi_finalize(error)
end program GA_template
```

C Template

```c
#include <stdio.h>

// Include GA Headers
#include "mpi.h"
#include "ga.h"
#include "macdecls.h"

int main(int argc, char **argv) {

    // Initialize Message-Passing
    MPI_Init(&argc, &argv);

    // Initialize GA Library
    GA_Initialize();

    ... processing ...

    // Terminate GA Library
    GA_Terminate();

    // Terminate Message-Passing
    MPI_Finalize();

    return 0;
}
```
Important: The Message-Passing library (e.g. MPI) must be initialised before GA is initialised.

There are two interfaces to initialise Global Arrays:

**Interface 1**

**Fortran:** subroutine ga_initialize()

**C:** void GA_Initialize()

GA can consume as much memory as application needs to allocate global arrays

**Interface 2**

**Fortran:** subroutine ga_initialize_ltd(limit)

**C:** void GA_Initialize_ltd(size_t limit)

Aggregate GA memory is limited to \( \text{limit} \) bytes when allocating global arrays
The conventional way to terminate a GA program is to call the following function:

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

The programmer can also abort a running program (e.g. within a error-handling routine) by calling the function:

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

message  User Error Message
code     Termination Error Code
Within a parallel programming environment there are two important questions to ask:

**How many processes are working together?**

This can be answered in a GA environment by calling the following function:

- **Fortran:** `integer function ga.nnodes()`
- **C:** `int GA_Nnodes()`

**What is the ID of each process?**

This can be answered in a GA environment by calling the following function:

- **Fortran:** `integer function ga.nodeid()`
- **C:** `int GA_Nodeid()`
### GA Basics

#### Data Types

<table>
<thead>
<tr>
<th>MT_F_INT</th>
<th>Integer (4/8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT_F_REAL</td>
<td>Real</td>
</tr>
<tr>
<td>MT_F_DBL</td>
<td>Double Precision</td>
</tr>
<tr>
<td>MT_F_SCPL</td>
<td>Single Complex</td>
</tr>
<tr>
<td>MT_F_DCPL</td>
<td>Double Complex</td>
</tr>
</tbody>
</table>

**Table:** Fortran Data Types

<table>
<thead>
<tr>
<th>C_INT</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_LONG</td>
<td>long</td>
</tr>
<tr>
<td>C_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>C_DBL</td>
<td>double</td>
</tr>
<tr>
<td>C_SCPL</td>
<td>single complex</td>
</tr>
<tr>
<td>C_DCPL</td>
<td>double complex</td>
</tr>
</tbody>
</table>

**Table:** C Data Types
GA Basics
Compilation

Fortran 90 Codes

Compile with:

```bash
mpif90 -I/path_to_GA/include -preprocess_flag \$OPT_Flags -o foo foo.f90 -L/path_to_GA/lib/ -lglobal -lma -lsci -llinalg -larmci -ltcgmsg-mpi -lmpich -lm
```

C Codes

```bash
mpicc -I/path_to_GA/include -preprocess_flag \$OPT_Flags -o foo foo.c -L/path_to_GA/lib/ -lglobal -lma -lsci -llinalg -larmci -ltcgmsg-mpi -lmpich -lm -lm
```
Exercise 1 (15 minutes)

Overview

Modify the GA code template given in the slides to develop a parallel “Hello World” program.

Requirements

- Your code should display a “Hello World” message for each participating process (test with 1, 2 and 4 processes)
- Each process message should display its ID along with the total number of participating processes

Sample Output

<table>
<thead>
<tr>
<th>Message</th>
<th>ID</th>
<th>Total Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello World from process</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Hello World from process</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Hello World from process</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Hello World from process</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
Part III

Creating and Destroying Global Arrays
There are three (3) methods for creating global arrays:

1. The *original* interface supporting:
   - regular distributions
   - irregular distributions

2. Duplicating an existing global array

3. A *new* interface providing more explicit functionality
Creating Global Arrays

Original Interface - Regular Distributions

Definition

A *regular distribution* attempts to assign the same number of elements to each process. This allows for better *load-balancing* and overall *parallel efficiency*.

(a) Row Blocking
(b) Column Blocking
(c) 2D Blocking

Figure: Regular Array Distributions
Creating Global Arrays

Original Interface - Regular Distributions...continued

To create global arrays with regular distributions, call the following function:

**Fortran:**
```fortran
logical function nga_create(type, ndim, dims, name, chunk, ga)
```

**C:**
```c
int NGA_Create(int type, int ndim, int dims[], char *name, int chunk[])
```

- **type**: GA Data Type e.g. `MT_F_DBL`
- **ndim**: Number of Array Dimensions
- **dims**: Vector of Array Dimension Sizes
- **name**: Unique Character Identification String
- **chunk**: Minimum Blocking Size for each Dimension
- **ga**: Array Handle Returned for Future Reference
Creating a Global Array - Fortran Code Sample

```fortran
integer, dimension(2) :: chunk, dims
integer :: handle_A

! ... GA Initialization ...

! Set Global Array Dimensions
dims(1)=100000
 dims(2)=100000

! Use Default Blocking
chunk(1)=-1
chunk(2)=-1

if (.not. nga_create(MT_F_DBL, 2, dims, 'Array_A', chunk, handle_A)) then
    call ga_error("Unable to create Global Array for Array A’, handle_A")
end if

! ... GA Termination ...
```
Definition

In certain domains it can be beneficial to apply an irregular distribution, where the number of elements assigned per process is uneven.

Figure: Irregular Array Distributions
Creating Global Arrays

Original Interface - Irregular Distributions...continued

To create global arrays with irregular distributions, call the following function:

**Fortran:**
```fortran
logical function nga_create_irreg(type, ndim, dims, name, map, nblock, g_a)
```

**C:**
```c
int NGA_Create_irreg(int type, int ndim, int dims[], char *name, int nblock[], int map[])
```

- **type**: GA Data Type e.g. `MT_F_DBL`
- **ndim**: Number of Array Dimensions
- **dims**: Vector of Array Dimension Sizes
- **name**: Unique Character Identification String
- **map**: Starting Index for Each Block
- **nblock**: Number of Blocks Each Dimension is Divided Into
- **g_a**: Array Handle Returned for Future Reference
An Irregular Distribution Example

Consider the following irregular array distribution i.e. the distribution is non-uniform because processes $P1$ and $P4$ get 20 elements each and processes $P0$, $P2$, $P3$ and $P5$ receive only 10 elements each.

Fortran Code

```fortran
integer :: map(5), nblock(2), dims(2), A

! Set Dimension Blocks
nblock(1)=3; nblock(2)=2

! Set Dimension Sizes
dims(1)=8; dims(2)=10

! Set Starting Indices
map(1)=1; map(2)=3; map(3)=7; map(4)=1; map(5)=6

nga_create_irreg(MT_F_DBL, 2, dims, 'Array_A', &
                  map, nblock, A)
```
With an irregular distribution, the programmer specifies distribution points for every dimension using the map array argument.

- The GA library creates a distributed array that is a Cartesian product of distributions for each dimension.

\[
\text{nblock} = \{3, 2\}
\]

\[
\text{map} = \{1, 3, 7, 1, 6\}
\]

\[
\text{product} = \{(1,1),(1,6),(3,1),(3,6),(7,1),(7,6)\}
\]
Global arrays can be duplicated (i.e. inherit all the properties of an existing global array including distribution, type, dimensions etc.) with a call to the following function:

**Fortran:**
logical function ga_duplicate(g_a, g_b, name)

**C:**
int GA_Duplicate(int g_a, char *name)

- g_a: Existing Global Array
- g_b: New Duplicated Global Array
- name: Unique Character Identification String
Creating Global Arrays

New Interface

Due to the increasingly varied ways in which global arrays can be configured, a new set of flexible interfaces has been recently developed for creating global arrays.

- The new interface supports all the configurations that were accessible with the old interfaces
- It is anticipated that a new range of global array properties will only be supported via the new interface

The creation of global arrays, using the new interface, progresses with calls to the following functions:

**Step 1 - Mandatory**

Fortran: `integer function ga_create_handle()`  
C: `int GA_Create_handle()`

**Step 1:** Return a handle to the new global array
Creating Global Arrays

New Interface . . . continued

Step 2 - Mandatory

Fortran: subroutine ga_set_data(g_a, ndim, dims, type)
C: void GA_Set_data(int g_a, int ndim, int dims[], int type)

Step 2: Set the required properties of the global array

Step 3: Optional properties of the global array can now be set using the following collection of individual ga_set_XXX() routines.

Step 3 - Optional

Fortran: subroutine ga_set_array_name(g_a, name)
C: void GA_Set_array_name(int g_a, char *name)
Step 3 - Optional

Fortran: subroutine ga_set_chunk(g_a, chunk)
C: void GA_Set_chunk(int g_a, int chunk[])

Note: The default setting of chunk is $-1$ along all dimensions

Step 3 - Optional

Fortran: subroutine ga_set_irreg_distr(g_a, map, nblocks)
C: void GA_Set_irreg_distr(int g_a, int map[], int nblock[])
Step 4: After all the array properties have been set, memory for the global array is allocated by a call to the following function:

**Step 4 - Mandatory**

Fortran: `logical function ga_allocate(g_a)`

C: `int GA_Allocate(int g_a)`

After this (successful) call, the global array is ready for use.
Global arrays can be destroyed with a call to the following function:

**Fortran:**
```
logical function ga_destroy(g_a)
```

**C:**
```
void GA_Destroy(int g_a)
```

\[ g_a \quad \text{Global Array to be Destroyed} \]
Creating/Destroying Global Arrays
Programming Exercise

Exercise 2 (30 minutes)

Requirements

Modify your GA Template to create and destroy four (4) global arrays with the following requirements: (test your solution with 6 processes)

1. Create a 5000x5000 Integer Global Array using a column-striped regular distribution

2. Create the irregular distributed Global Array given in the slides
   - Look up the GA routine `ga_print_distribution()` in the Interface Documentation[4] and use it to verify the correct creation of the array

3. Duplicate the Global Arrays given in part (1) and part (2)

4. Create a fourth global array (with the same properties given in part (1)) using the new interface creation methods

5. Destroy all the created global arrays
Part IV

One-Sided Communications
One-Sided, Non-Collective Communications

The Global Arrays Toolkit provides the programmer with one-sided, non-collective communication operations for accessing data in global arrays without the cooperation of the process or processes that store the referenced data.

Benefits

1. The processes containing the referenced data are oblivious to other processes accessing and/or updating their data items

2. Since global array indices are still used to reference non-local data items, the calling process does not need to specify process IDs and remote address information
### Table: The Three Categories of One-Sided Operations in GA

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Blockwise Read/Write</td>
<td><code>ga_put()</code>, <code>ga_get()</code></td>
</tr>
<tr>
<td>Remote Atomic Update</td>
<td><code>ga_acc()</code>, <code>ga_read_inc()</code></td>
</tr>
<tr>
<td></td>
<td><code>ga_scatter_acc()</code></td>
</tr>
<tr>
<td>Remote Elementwise Read/Write</td>
<td><code>ga_scatter()</code>, <code>ga_gather()</code></td>
</tr>
</tbody>
</table>
To place data from a local buffer into a section of a global array, use the following function:

**Fortran:**
```fortran
subroutine nga_put(g_a, lo, hi, buf, ld)
```

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

- **g_a**  
  Global Array Handle

- **lo**  
  Array of Starting Patch Indices

- **hi**  
  Array of Ending Patch Indices

- **buf**  
  Local Buffer containing Data Values

- **ld**  
  Leading Dimensions for Buffer
One-Sided Communications
NGA_Put() - Local Communication

Local Buffer

Global Array

\[ l_0 = \{8, 1\} \]
\[ h_0 = \{15, 10\} \]
\[ l_d = \{8, 10\} \]

Figure: GA Put() One-Sided Operation
To place data from a section of a global array into a local buffer, use the following function:

**Fortran:**
```fortran
subroutine nga_get(g_a, lo, hi, buf, ld)
```

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

g_a  Global Array Handle
lo   Array of Starting Patch Indices
hi   Array of Ending Patch Indices
buf  Local Buffer to Receive Data Values
ld   Leading Dimensions for Buffer
One-Sided Communications
Atomic Accumulate

Accumulation

Frequently, data moved in a put operation has to be combined with the data at the target process, rather than replace it i.e. accumulation.

The Global Arrays Toolkit provides two operations accumulate() and read_inc() which allows a global array patch (array section) or element to be remotely updated, with the following benefits:

1. **Operations are atomic** i.e. the same patch of global array can be updated by multiple processes without loss of correctness or consistency

2. The processes owning the data are not involved in the atomic updates
One-Sided Communications
Atomic Accumulate . . . continued

The Global Arrays Toolkit
CSCS
One-Sided Communications
Introduction
Put
Get
Atomic Accumulate
Scatter/Gather

Global Array

\[ ga(i,j) = ga(i,j) + \alpha \cdot buf(k,l) \]

Figure: GA Atomic Accumulate Operation
To perform an atomic accumulate on a global array patch, call the following function:

**Fortran:** subroutine nga_acc(g_a, lo, hi, buf, ld, alpha)

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

- **g_a** — Global Array Handle
- **lo** — Array of Starting Patch Indices
- **hi** — Array of Ending Patch Indices
- **buf** — Local Buffer containing Data Values
- **ld** — Leading Dimensions for Buffer
- **alpha** — The Scaling Factor
To perform an atomic accumulate on a single global array element, call the following function:

**Fortran:**

```
integer function nga_read_inc(g_a, subscript, inc)
```

**C:**

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

- `g_a`: Global Array Handle
- `subscript`: Vector of Element Indices
- `inc`: Increment Value

**Notes**

1. The original global array element is returned
2. Only applies to Integer Global Arrays
3. Can be used to implement global counters for dynamic balancing
One-Sided Communications
NGA_Read_inc() ... continued

Figure: GA Atomic Read-Increment Operation
Definition

**Scatter** allows a set of values to be scattered (distributed) to non-contiguous positions in a global array.

**Figure:** Scattering Five Elements to a Global Array
To perform a scatter operation, call the following function:

**Fortran:** `subroutine nga_scatter(g_a, v, indices, n)`

**C:** `void NGA_Scatter(int g_a, void *v, int indices[], int n)`

- `g_a`: Global Array Handle
- `v` : Vector of Values
- `indices` : Array of Scatter Indices
- `n` : Number of Values
One-Sided Communications
NGA_Gather() - Local Communication

Definition

Gather allows a set of values to be gathered (collected) from non-contiguous global array positions to a local buffer.

To perform a gather operation, call the following function:

Fortran: subroutine nga_gather(g_a, v, indices, n)
C: void NGA_Gather(int g_a, void *v, int indices[], int n)

- g_a Global Array Handle
- v Local Vector of Values
- indices Array of Gather Indices
- n Number of Values
Part V

Utility Operations
To determine the ID of the process that stores a given **Global Array Index**, the following routine can be used:

**Fortran:**
```fortran
logical function nga_locate(g_a, subscript, owner)
```

**C:**
```c
int NGA_Locate(int g_a, int subscript[])
```

**Figure:** NGA_Locate() Operation

![Diagram of Global Array and Locality Information](image-url)

**Description:**
- **Global Array:**
  - PE 0
  - PE 1
  - PE 2
  - PE 3
- **Subscript:** `g_a{20,30}`
- **Owner:** `g_a{20,30} = 2`
- **Array containing Global Index:**
- **Process ID of Owner:**

---

Owner `g_a{20,30} = 2`
To determine the ID(s) of the process(es) that store a given Global Array Patch, the following routine can be used:

**Fortran:**
```
logical function nga_locate_region(g_a, lo, hi, map, proclist, np)
```

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

**Figure:** NGA_Locate_region()
Locality Information

Frequently a process will want to modify a patch of a global array that is stored locally. To provide direct access to this patch (and improve overall performance) the following procedure is required:

Procedure For Direct Access to Local Patch

1. **Determine the local patch of the Global Array**
   i.e. Which Part of the Global Array does the Process Own?
2. **Access the Local Data Patch**
3. **Operate on the Data Patch**
4. **Release Access to the Data Patch**
To determine the index range of a Global Array that a process owns, call the following routine:

**Fortran:**
```
subroutine nga_distribution(g_a, process, lo, hi)
```

**C:**
```
void NGA_Distribution(int g_a, int process, int lo[], int hi[])
```

**Figure:** GA Distribution Operation

process={1}, lo={lo1,lo2}, hi={hi1,hi2}
To obtain **read/write access** to the local patch data owned by the calling process, call the following routine:

**Fortran:**
```fortran
subroutine nga_access(g_a, lo, hi, index, ld)
```

**C:**
```c
void NGA_Access(int g_a, int lo[], int hi[], void *index, int ld[])
```

---

**NGA_Access() - Fortran Example**

```fortran
status=nga_create(MT_F_DBL, 2, dims,&
     'Array', chunk, g_a)
...
call nga_distribution(g_a, me, lo, hi)
call nga_access(g_a, lo, hi, index, ld)
call foo(dbl_mb(index), ld(1))
call nga_release(g_a,lo,hi)

subroutine foo(a, ldl)
    double precision a(ld1,*)
end subroutine
```

- `lo`  Starting Indices for Patch
- `hi`  Ending Indices for Patch
- `index`  Patch Starting Address
- `ld`  Leading Dimension of Patch
To release access to a Global Array patch (which was only used for reading), call the following routine:

**Fortran:**
```
subroutine nga_release(g_a, lo, hi)
```

**C:**
```
void NGA_Access(int g_a, int lo[], int hi[])
```

To release access to a Global Array patch (which was modified), call the following routine:

**Fortran:**
```
subroutine nga_release_update(g_a, lo, hi)
```

**C:**
```
void NGA_Access_update(int g_a, int lo[], int hi[])
```
Printing Global Arrays

The Global Arrays Toolkit provides routines to print:

1. the contents of a Global Array
2. the contents of a Global Array patch
3. the status of array operations
4. a summary of allocated arrays
To print an entire Global Array to standard output (with formatting) call the following routine:

**Fortran:** subroutine ga_print(g_a)

**C:** void GA_Print(int g_a)

To print a Global Array patch to standard output call the following routine:

**Fortran:** subroutine ga_print_patch(g_a, lo, hi, pretty)

**C:** void GA_Print_patch(int g_a, int lo[], int hi[], int pretty)

1. pretty=0 - dump all elements of patch without formatting
2. pretty=1 - display labelled array with formatted rows and columns
To print **global statistics** about array operations for the calling process, call the following routine:

**Fortran:**
```
subroutine ga_print_stats()
```

**C:**
```
void GA_Print_stats()
```

This routine will print:

1. the number of calls to the GA `create()`, `duplicate()`, `destroy()`, `get()`, `put()`, `scatter()`, `gather()` and `read_and_inc()` operations
2. the total amount of data moved in the GA primitive operations
3. the amount of data moved in the primitive GA operations to logically remote locations
4. the maximum memory consumption of Global Arrays (high-water mark)
To print distribution information \textit{(array mapping to processes)} for a Global Array, call the following routine:

\begin{itemize}
\item \textbf{Fortran}: \texttt{subroutine ga\_print\_distribution(g\_a)}
\item \textbf{C}: \texttt{void GA\_Print\_distribution(int g\_a)}
\end{itemize}

To display summary information on allocated arrays call the following routine:

\begin{itemize}
\item \textbf{Fortran}: \texttt{subroutine ga\_summarize(verbose)}
\item \textbf{C}: \texttt{void GA\_Summarize(int verbose)}
\end{itemize}

\textbf{1} \hspace{1cm} \texttt{verbose = 0 or 1}
The Global Arrays Toolkit provides other utility routines that are described further in the GA documentation. They include routines for:

1. Memory Availability
2. Cluster Topology and Details
3. Broadcast/Reduction
4. Array Dimension, Type and Name Inquiry
Part VI

Collective Communications
The Global Arrays Toolkit provides **collective operations** which can be applied to **whole** global arrays or **patches** of global arrays.

- Collective operations require that **all** processes initiate the collective call (although in general they only operate on their local array data)

GA collective operations span the following categories:

1. **Basic Array Operations**
2. **Linear Algebra Operations**
3. **Interfaces to 3rd Party Software Packages**
Since GA doesn’t specifically initialise newly created arrays, a global array can be set to 0 with the following routine:

**Fortran:** subroutine ga_zero(g_a)
**C:** void GA_Zero(int g_a)

If a global array is required to be initialised with a non-zero value, the following routine can be called:

**Fortran:** subroutine ga_fill(g_a, value)
**C:** void GA_Fill(int g_a, void *value)

**Note:** The type of value should match the type of the global array g_a.
To **scale** all the elements in a global array by the factor $s$, call the following routine:

**Fortran:**

```fortran
subroutine ga_scale(g_a, s)
```

**C:**

```c
void GA_Scale(int g_a, void *s)
```

**Note:** The type of $s$ should match the type of the global array $g\_a$. 
To copy the contents of one global array to another, call the following routine:

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

1. The global arrays `g_a` and `g_b` should be of the same type
2. The global arrays `g_a` and `g_b` should have the same number of elements and dimensions (but can be copied to an array with a different shape and/or distribution)
Collective Communications
Copying Arrays ... continued

---

**Figure:** Copying Global Arrays on 3x3 Process Grid

---

Array Copy - Fortran Example

```fortran
! Create Irregular Distribution
status = nga_create_irreg(MT_F_DBL, 2, dims, 'A', map, nbblocks, A)

! Create Regular Distribution
status = nga_create(MT_F_DBL, 2, dims, 'B', chunks, B)

... Initialise A ...

! Copy global array A to global array B
call ga_copy(A, B)
```
GA also provides analogous routines for initialising array patches (as opposed to whole arrays):

**Fortran:** subroutine nga_zero_patch(g_a, lo, hi)

**C:** void GA_Zero_patch(int g_a, int lo[], int hi[])

- **lo**  Starting Indices of Array Patch
- **hi**  Ending Indices of Array Patch

**Fortran:** subroutine nga_fill_patch(g_a, lo, hi, value)

**C:** void GA_Fill_patch(int g_a, int lo[], int hi[], void *value)
One of the most fundamental GA operations is `nga_copy_patch()`. This operation copies a patch in one global array to a patch in another global array.

**Fortran:**
```
subroutine nga_copy_patch(trans, g_a, lo1, hi1, g_b, lo2, hi2)
```

**C:**
```
void GA_Zero_patch(char trans, int g_a, int lo1[], int hi1[], int g_b, int lo2[], int hi2[])
```

**Notes:**
1. The source patch must be on a different global array than the destination patch
2. Array patches must be the same type
3. Array patches must have the same number of elements (but not necessarily the same shape)
Collective Communications
Copying Array Patches

Figure: Global Array Patch Copy with Reshape
Collective Communications
Copying Array Patches ... continued

**Figure:** Global Array Patch Copy with Reshape and Transpose
GA provides a range of routines for performing linear algebra operations on Global Arrays (routines are available for both whole array and patches).

To add two global arrays \( A \) and \( B \) (element-wise) with the result placed into array \( C \), call the following routine:

### Whole Arrays

**Fortran:**

```
subroutine ga_add(alpha, g_a, beta, g_b, g_c)
```

**C:**

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

\[
C = \alpha A + \beta B
\]
Collective Communications

Matrix Addition . . . continued

Patches

Fortran: subroutine ga_add_patch(alpha, g_a, lo1, hi1, beta, g_b, lo2, hi2, g_c, lo3, hi3)

C: void GA_Add(void *alpha, int g_a, int lo1[], int hi1[], void *beta, int g_b, int lo2[], int hi2[], int g_c, int lo3[], int hi3[])

Collective Communications
Introduction
Initialisation
Copying
Array Patches
Linear Algebra
3rd Party Interfaces
Synchronisation
Control
To perform matrix multiplication between two global arrays $A$ and $B$ (with the result placed in global array $C$), call the following operation:

Fortran: `subroutine nga_matmul_patch(transa, transb, alpha, beta, g_a, alo, ahi, g_b, blo, bhi, g_c, clo, chi)`

C: `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[])`

$$C = \alpha A \ast B + \beta C$$

Note:
1. $A$, $B$ and $C$ must be 2D Global Arrays

Note: The entries in $C$ must be of type `globval_t` and the entries in $A$ and $B$ must be of type `const globval_t`.
There are two other GA operations that can only be applied to 2D Global Arrays:

**Fortran:** `subroutine ga_symmetrize(g_a)`

**C:** `void GA_Symmetrize(int g_a)`

\[ A = \frac{1}{2} (A + A') \]

**Fortran:** `subroutine ga_transpose(g_a, g_b)`

**C:** `void GA_Transpose(int g_a, int g_b)`

\[ B = A' \]
Collective Communications
Further Operations

GA provides a wealth of whole array and patch array routines for many linear operations (please see [4] for more detailed information):

Further operations include:

1. **Dot Products**
2. **Elemental Operations**
   - Element Addition, Multiplication, Division, Reciprocals
   - Maximum/Minimum Elements
3. **Diagonal Operations**
   - Diagonal Shifts
   - Diagonal Initialisation/Updates
   - Diagonal Addition
4. **Row/Column Scaling**
5. **Norms**
6. **Medians**
ScaLAPACK

ScaLAPACK[5] is a well known software library for linear algebra computations on distributed-memory architectures. GA interfaces with this library to solve systems of linear equations and also to invert matrices.

PeIGS

The PeIGS library contains routines for solving standard and generalized real symmetric eigensystems. For more information about the availability of PeIGS with GA contact: fanngi@ornl.gov.

PETSc

GA can be intermixed with PETSc[6] routines (a library for the parallel scalable solution of scientific applications modelled by partial differential equations). Instructions for using PETSC with GA can be found at:

http://www.emsl.pnl.gov/docs/global/petsc.html
To solve the system of linear equations $AX = B$, call the following routine:

**Fortran:**
```fortran
integer function ga_solve(g_a, g_b)
```

**C:**
```c
int GA_Solve(int g_a, int g_b)
```

**Notes:**

1. In the first instance, a **Cholesky factorisation and Cholesky solve** will be performed
   - If a Cholesky factorisation is not successful, then a **LU** factorisation will be performed with a forward/backward substitution
2. On exit, $B$ will contain the solution $X$
To solve a system of linear equations $AX = B$ using the Cholesky factorisation of an $N \times N$ double precision symmetric positive definite matrix $A$, call the following routine:

**Fortran:**

```fortran
integer function ga_llt_solve(g_a, g_b)
```

**C:**

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

**Notes:**

1. On exit, $B$ will contain the solution $X$
To solve a system of linear equations $op(A)X = B$ (where $op(A) = A$ or $A'$) using a LU factorisation of a general real matrix $A$, call the following routine:

**Fortran:** subroutine ga_lu_solve(g_a, g_b)
**C:** int GA_Lu_solve(int g_a, int g_b)

**Notes:**

1. On exit, $B$ will contain the solution $X$ (possibly multiple RHS vectors)
To compute the inverse of a double precision matrix using the Cholesky factorisation of an \( N \times N \) double precision symmetric positive definite matrix \( A \), call the following routine:

Fortran:  
\[
\text{integer function ga_spd_inverse(g_a)}
\]

C:  
\[
\text{int GA_Spd_inverse(int g_a)}
\]

Notes:

- On exit, \( A \) will contain the inverse
Collective Communications
Improving Synchronisation Control of Collective Operations

Redundant Synchronisation

In some cases it may be possible (if you are careful) to remove **redundant implicit synchronisation** within collective GA operations (and improve performance).

1. Collective operations exploit implicit synchronisation to ensure local data is in a consistent state before it is accessed locally i.e. no outstanding communication operations

2. In many cases the **internal synchronisation points**, within back-back collective communications (or if the user calls an explicit synchronisation operation before the collective communication), can be removed:
To set the implicit synchronisation points for the next collective operation, call the following routine:

**Fortran:**
```
subroutine ga_mask_sync(prior_sync_mask, post_sync_mask)
```

**C:**
```
void GA_Mask_sync(int prior_sync_mask, int post_sync_mask)
```

- **prior_sync_mask** When false, disables synchronisation at start of next collective call
- **post_sync_mask** When false, disables synchronisation at end of next collective call
The Global Arrays Toolkit

CSCS

Collective Communications

Introduction
Initialisation
Copying
Array Patches
Linear Algebra
3rd Party Interfaces
Synchronisation Control

Collective Communications

Improving Synchronisation Control of Collective Operations . . . continued

Synchronisation Mask - Fortran Example

```fortran
! Mask Implicit Synchronisation
call ga_duplicate(g_a, g_b)
call ga_mask_sync(0,1)
call ga_zero(g_b)
```
Part VII

Inter-Process Synchronisation
Synchronisation

Introduction

Definition

Process *synchronisation* refers to the coordination of simultaneous processes to complete a task in order to get correct runtime order and avoid unexpected race conditions.

Classification of GA Synchronisation

The Global Arrays Toolkit provides three (3) types of synchronisation operations:

1. Lock with Mutex
2. Fence
3. Sync
**Background**

A *lock* and *mutex* are useful primitives in shared-memory programming. A mutex can be locked to exclusively protect access to a *critical section* of code e.g. to avoid the simultaneous use of a common resource, such as a *global variable*.

![Shared Memory Diagram](image.png)

**Figure**: Conflict with Shared Resource Update
Critical Section Algorithm

1. Create Mutexes
   ...
2. Lock on a Mutex
3. Perform Critical Section Operation
4. Unlock the Mutex
   ...
5. Destroy Mutexes

Notes

A mutex is a lock that can be obtained (set) by a process (if it is free), prior to a critical section of code. If a lock is set by a given process (and the process enters the critical section) no other processes can obtain the lock and therefore enter the critical section at the same time.
Synchronisation
Lock and Mutex \ldots continued

Step 1
PE x
PE y
Critical Section
PE z

Step 2
PE x
Critical Section
PE y
PE z

Step 3
Critical Section
PE y
PE z

Figure: Mutual Exclusion Using Locks
Synchronisation
GA_Create_mutexes() - Collective Operation

To create a mutex set in GA, call the following function:

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

number Number of Mutexes in Mutex Array

Note:
1. Only one set of mutexes can exist at one time
2. Mutexes are numbered $[0 \ldots number - 1]$
3. GA_Create_mutexes() is a collective operation
To destroy a mutex set in GA, call the following function:

**Fortran:**

```fortran
logical function ga_destroy_mutexes()
```

**C:**

```c
int GA_Destroy_mutexes()
```

Note:

1. Mutexes can be created and destroyed as many times as is needed
2. `GA_Destroy_mutexes()` is a collective operation
To **lock** a mutex object in GA, call the following function:

**Fortran:** subroutine ga_lock(mutex)

**C:** int GA_Lock(int mutex)

**mutex**  
Mutex Number

**Note:**

1. It is a fatal error for a process to lock a mutex object that has already been locked by this process
To **unlock** a mutex object in GA, call the following function:

**Fortran:**
```
subroutine ga_unlock(mutex)
```

**C:**
```
int GA_Unlock(int mutex)
```

**mutex**  
*Mutex Number*

**Note:**

1. It is a fatal error for a process to unlock a mutex object that has not been locked by this process.
Lock and Mutex Fortran Example

```fortran
! Create Mutex Set (with one mutex)
status = ga_create_mutexes(1)

! Check if Mutex Creation Successful
if (.not. status) then
    call ga_error("ga_create_mutexes failed",0)
end if

! Obtain Lock
call ga_lock(0)

! Now do something in Critical Section
call ga_put(g_a, ... )
...

! Release Lock
call ga_unlock(0)

! Destroy Mutexes
status = ga_destroy_mutexes(1)
if (.not. status) then
    call ga_error("ga_destroy_mutexes failed",0)
end if
```
Definition

A **fence** operation blocks the calling process until all the data transfers corresponding to the Global Array operations initiated by this process are complete.

**Figure:** GA Fence Operation
To initiate a fence in GA, call the following function:

**Fortran:** subroutine ga_init_fence()
**C:** void GA_Init_fence()

Note:

1. GA_Init_fence() initialises tracing of the completion status of global array data movement operations
A fence can be called in GA with the following function:

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

Note:

1. GA_Fence() blocks the calling process until all Global Array data transfers called after GA_Init_fence() are fully completed.
2. GA_Fence() must be called after a GA_Init_fence() (i.e. called in pairs)
3. GA_Init_fence()/GA_Fence() pairs can be nested
Synchronisation
Fence Examples

Fortran Example 1

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

*GA_Put()* can return before the data reaches its final destination. A *GA_Init_fence()*/*GA_Fence()* pair allows the process to wait until the data transfer is complete.

Fortran Example 2

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

The calling process will be blocked until the data movements of two *GA_Puts()* and one *GA_Scatter()* are completed.
Definition

The GA sync operation is a collective barrier operation which synchronises all the processes and ensures all Global Array operations are complete after the call.

Figure: Barrier Synchronisation
A GA `synch` operation can be called with the following function:

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

**Note:**
1. GA Sync operations should be inserted where necessary  
2. Increased synchronisation can result in reduced application performance
Part VIII

Processor Groups
Processor Groups

Definition

Processor Groups can be a useful technique for dividing the default domain (world) of processes into separate subgroups of processes.

Benefits for applying group management include:

1. Global Arrays created in a group are only distributed among the processes in the group
2. Collective operations on the subgroup are restricted to the processes in the group
3. A Synchronisation operation applied to a group will not affect the processes residing outside the group
4. Independent parallel operations can be applied to different subgroups concurrently
Processor Groups

**Figure:** Decomposition of World into Subgroups
Important Notes

1. When applying subgroups, it is a good idea to ensure that the default world is divided into a complete covering of non-overlapping subgroups.

2. Global Arrays are only created on the default group (usually the world group), and most global array operations are restricted to the default group:
   - Typically you change the default group prior to applying a Global Array operation (the global array operation will subsequently be applied to the new current default group).
   - Or use the GA routines that specifically accept subgroup array handles.
A new processor group can be created in GA with the following function:

**Fortran:**

```fortran
integer function ga_pgroup_create(proclist,size)
```

**C:**

```c
int GA_Pgroup_create(int *proclist, int size)
```

- `proclist` Vector of Processes
- `size` Number of Processes in Group

- This call must be executed by **all processes in the new subgroup**.
- The integer handle of the newly created process group will be returned on completion of the call.
A processor group can be applied to a Global Array using both original and new interfaces:

**Original Interface**

Fortran:  
```fortran
logical function nga_create_config(type, ndim, dims, name, chunk, p_handle, g_a)
```

C:  
```c
int NGA_Create_config(int type, int ndim, int dims[], char *name, int p_handle, int chunk[])
```

**New Interface**

Fortran:  
```
logical function ga_set_pgroup(g_a, p_handle)
```

C:  
```c
int NGA_Create_config(int g_a, int p_handle)
```
The default process group can be set with the following routine:

**Fortran:**  subroutine ga_pgroup_set_default(p_handle)

**C:**  void GA_Pgroup_set_default(int p_handle)

- This routine can set the default group to something other than the default *world group*
- This routine must be called by all processes within the process group represented by p_handle
- Once the default process group is set, all subsequent operations will be applied to the new default group.
To inquire about the number of nodes in a process group, call the following routine:

**Fortran:**  
integer function ga_pgroup_nnodes(p_handle)

**C:**  
int GA_Pgroup_nnodes(int p_handle)

To inquire about the local ID of the calling process within the group, call the following routine:

**Fortran:**  
integer function ga_pgroup_nodeid(p_handle)

**C:**  
int GA_Pgroup_nodeid(int p_handle)
To determine the handle for the current default process group, call the following routine:

**Fortran:**
```fortran
integer function ga_pgroup_get_default()
```

**C:**
```c
int GA_Pgroup_get_default()
```

To determine the handle for the current world process group, call the following routine:

**Fortran:**
```fortran
integer function ga_pgroup_get_world()
```

**C:**
```c
int GA_Pgroup_get_world()
```
Default Process Group Example

! Create Subgroup A
A = ga_pgroup_create(listA, nprocA)
call ga_pgroup_set_default(A)

! Perform some parallel work
call parallel_work()

! Reset Default Process Group to World Group
call ga_pgroup_set_default(ga_pgroup_get_world())

contains

subroutine parallel_work()

! Create Subgroup B
B = ga_pgroup_create(listB, nprocB)
call ga_pgroup_set_default(B)
call more_parallel_work

end subroutine
Part IX

Advanced Topics
**Definition**

*Ghost cells* are a common technique to reduce communication in parallel applications that exploit domain decomposition, where the data required by a given process to complete a calculation resides on a neighbouring process e.g. codes that employ finite-difference and finite-element methods.

**Figure:** Domain Decomposition - Data Exchange
Ghost cells need to be updated when they are modified on the neighbouring process. GA allows the programmer to perform this update with a single function.

**Figure:** Domain Decomposition With Ghost-Cells

PE 0 contains an extra column containing column 5 of $\Psi$. These ghost-cells reduce communication during calculation of $\Psi(4,4)$. 
Ghost cells can be automatically created for a global array (with regular distribution) using the following functions:

**Fortran:**
```
logical function nga_create_ghosts(type, ndim, dims, width, name, chunk, g_a)
```

**C:**
```
int NGA_Create_ghosts(int type, int ndim, int dims[], int width[], char *name, int chunk[])
```

**width**  
Vector of Ghost Cell Widths in each Dimension

```c
nga_create_ghosts(MT_F_DBL,2,dims,width,'Array_A',chunk,g_a)
```
Ghost cells can be automatically created for a global array (with irregular distribution) using the following functions:

**Fortran:**
```fortran
logical function nga_create_ghosts_irreg(type, ndim, dims, width, name, map, nblocks, ga)
```

**C:**
```c
int NGA_Create_ghosts_irreg(int type, int ndim, int dims[], int width[], char *name, int map[], int nblocks[])
```

**width**  Vector of Ghost Cell Widths in each Dimension
Ghost cells can be updated with values from their adjacent neighbours using the following functions:

**Fortran:** subroutine ga_update_ghosts(g_a)

**C:** void GA_Update_ghosts(int g_a)

**Note:**
1. The update operation assumes periodic (wrap-around) boundary conditions (see Periodic Interfaces)
Ghost Cells
Advanced Topic

Figure: Global Array With Uninitialised Ghost Cells

Figure: Global Array After Ghost Cell Update
GA also provides a routine to update ghost cells in **individual directions** (most useful when updates can be overlapped with computation).

**Fortran:**

```fortran
logical function nga_update_ghosts_dir(g_a, dimension, direction, corner)
```

**C:**

```c
int NGA_Update_ghosts_dir(int g_a, int dimension, int direction, int corner)
```

dimension    Coordinate direction is to be updated  
             (e.g. dimension=2 corresponds to y-axis in 2D or 3D system)

direction    Updated side is in positive or negative direction?
flag          Should corners on updated side be included?
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)

**Figure:** Equivalent Calls for 2D GA_Update_ghosts Call
The values of local ghost cells can be accessed using the following functions:

**Fortran:**
```
subroutine nga_access_ghosts(g_a, dims, index, ld)
```

**C:**
```
void NGA_Access_ghosts(int g_a, int dims[], void *index, int ld[])
```

dims: Array of Local Dimensions Including Ghost Cells  
index: Index Corresponding to the Local Global Array Patch  
ld: Dimensions of the Local Array Patch including Ghost Cells
Definition

One-sided Periodic Interfaces have been added to GA to support computational fluid dynamics problems on multi-dimensional grids.

1. They provide and index translation layer that allows put(), get() and accumulate() operations to extend beyond the boundaries of the global array.
2. The references outside the boundaries are wrapped around the global array like a torus.
### Periodic Interfaces

**Advanced Topic**

#### Figure: Horizontal Boundary Wrapping with Periodic Interface

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7</td>
<td>12</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>13</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9</td>
<td>14</td>
<td>19</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>21</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>2</td>
<td>7</td>
<td>12</td>
<td>17</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>23</td>
<td>3</td>
<td>8</td>
<td>13</td>
<td>18</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>24</td>
<td>4</td>
<td>9</td>
<td>14</td>
<td>19</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>25</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
### Periodic Interfaces

#### Advanced Topic

**Figure:** The Global Array Patch $[2:4, -1:3]$

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>17</td>
<td>22</td>
<td>2</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>23</td>
<td>3</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>24</td>
<td>4</td>
<td>9</td>
<td>14</td>
</tr>
</tbody>
</table>
To perform periodic operations, call the following functions:

**Fortran:**
```
subroutine nga_periodic_get(g_a, lo, hi, buf, ld)
```

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

**Fortran:**
```
subroutine nga_periodic_put(g_a, lo, hi, buf, ld)
```

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

**Fortran:**
```
subroutine nga_periodic_acc(g_a, lo, hi, buf, ld, alpha)
```

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

- **lo**: Array of Starting Patch Indices
- **hi**: Array of Ending Patch Indices
- **buf**: Local Buffer to Send/Receive Data Values
- **ld**: Leading Dimensions for Buffer
- **alpha**: The Scaling Factor
Non-blocking Operations
Advanced Topic

Definition

Improved performance can be obtained with **non-blocking communications**

- were the communication operation completes before the data buffer is ready to be overwritten

By returning from the communication operation after the data transfer has been initiated, **overlapping** of computation and communication can be performed.
Non-blocking Operations
Advanced Topic

Figure: Non-blocking Communication Example
Non-blocking Operation
Advanced Topic

Notes:

1. GA non-blocking communications (get/put/accumulate) are derived from the standard blocking interface by adding an extra handle request argument.

2. The wait function completes a non-blocking operation locally.

3. Waiting on a non-blocking put()/accumulate() ensures the data was injected into network and the user buffer is ready to be reused.

4. Waiting on a non-blocking get() ensures the data has arrived and the user buffer is ready to be used.

5. Unlike blocking communications, non-blocking communications are not ordered with respect to the destination. If ordering is required, a fence operation can be used to confirm remote completion.
Non-blocking operations can be called with the following routines:

**Fortran Routines**

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

nbhandle  Handle for Non-Blocking Request
Non-blocking Operations
Advanced Topic

C Routines

void NGA_NbPut(int g_a, int lo[], int hi[], void buf[], int ld[], ga_nbhdl_t* nbhandle)
void NGA_NbGet(int g_a, int lo[], int hi[], void buf[], int ld[], ga_nbhdl_t* nbhandle)
void NGA_NbAcc(int g_a, int lo[], int hi[], void buf[], int ld[], void *alpha, ga_nbhdl_t* nbhandle)
subroutine nga_NbWait(ga_nbhdl_t* nbhandle)

nbhandle Handle for Non-Blocking Request
Non-blocking Operations
Advanced Topic

Non-Blocking GA Fortran Example

```fortran
double precision buf1(nmax,nmax)
double precision buf2(nmax,nmax)

! Evaluate lo1, hi1
call nga_nbaget(g_a,lo1,hi1,buf1,ld1,nb1)
ncount=1
do while (...)
   if (mod(ncount,2) .eq. 1) then
      ! Evaluate lo2, hi2
      call nga_nbaget(g_a,lo2,hi2,buf2,ld2,nb2)
      call nga_nbwait(nb1)
      ! Do work with data in buf1
   else
      ! Evaluate lo1, hi1
      call nga_nbaget(g_a,lo1,hi1,buf1,ld1,nb1)
      call nga_nbwait(nb2)
      ! Do work with data in buf2
   endif
   ncount=ncount+1
end do
```
Message-passing requires cooperation on both sides
Once the message is initiated on PE $y$ the sending processor can continue with computation. PE $x$ is never involved in the communication.

One-sided communications can be implemented with shared-memory, threads, network hardware, Remote Direct Memory Access (RDMA) and vendor-specific mechanisms.

Common one-sided communication libraries: ARMCI, SHMEM and MPI-2.
GA Interface

call ga_lu_solve(gA,gB)

instead of...

call pdgetrf(n,m,locaA,p,q,dA,ind,info)
call pdgetrs(trans,n,mb,locA,p,q,dA,dB,info)
A general, portable, efficient one-sided communication interface

ARMCI offers an extensive set of Remote Memory Access (RMA) communication functionality utilising network interfaces on clusters and supercomputers[2]:

1. data transfer operations optimised for contiguous and noncontiguous (strided, scatter/gather, I/O vector) data transfers
2. atomic operations
3. memory management and synchronisation
4. locks

GA’s primary communication interfaces combine the ARMCI interfaces with a global-array index translation.
Performance Notes:

1. On cluster connects, ARMCI achieves bandwidth close to the underlying network protocols.
   - Similarly, latency is comparable if the native platform protocol supports an equivalent remote memory operation (e.g. `elan_get()` on Quadrics).

2. For systems that do not support a native remote `get()` the latency can include the cost of interrupt processing that is used by ARMCI to implement the `get()` operation.

GA can support the programmer in controlling data distribution when creating global arrays:

1. You can allow GA to determine the array distribution
2. You can specify the decomposition of one array dimension and allow GA to determine the others
3. You can specify the block size for all dimensions
4. You can specify an irregular distribution as a Cartesian product of irregular distributions for each dimension

GA maintains locality information for all global arrays which can be accessed via query functions to find:

- the array data portion held by a given processor
- which process owns a particular array element
- block lists owned by each process for a given global array section
Blocking Store Operations

There are two types of completion for store operations (local and remote):

1. The blocking store operation completes after the operation is completed locally i.e. the user buffer containing the source data can be overwritten.
2. The blocking store operation completes remotely after either:
   - a fence operation
   - a barrier synchronisation
Ordered Blocking

1. Blocking loads/store operations are ordered **only** if they target overlapping sections of a global array
2. Blocking loads/stores that target different array sections complete arbitrarily

Non-Blocking Operations

1. Non-blocking load/store operations complete in arbitrary order. Wait/Test operations can be used to order completion of these operations, if required.
Local Memory Allocation

The Global Arrays Toolkit requires the Memory Allocator (MA) library, which is a collection of routines for performing dynamic memory allocation for C, Fortran and mixed-language applications. GA uses MA to provide all its dynamically allocated local memory (global memory is allocated via ARMCI using operating system shared memory operations).

- MA provides both stack and heap memory management operations.
- MA provides memory availability and utilisation information and statistics.
- MA supports both C and Fortran data types.
- MA offers both debugging and verification mechanisms (memory boundary guards).
Supplementary Information
Disk Resident Arrays (DRAs)

- Data transferred between disk and global memory using simple read/write commands.
- I/O operations have nonblocking interface to allow computation overlapping.
- Whole or sections of global arrays can be transferred (using global indexing).

```
dra_write(g_a, d_a, request)
```
Reshaping and transpose operations allowed during the transfer.

Disk arrays can be accessed by arbitrary number of processors.

Distribution on the disk is optimised for large data transfer.

Hints provided by the user allow improved performance for specific I/O patterns.
Disk-Resident Arrays automatically decomposed into multiple files
References I


[Jack Dongarra et al., 2008] MPI: A Message-Passing Interface Standard v1.3
References II

[J. Nieplocha et al.]  
Global Arrays Toolkit Interface  
http://www.emsl.pnl.gov/docs/global/userinterface.html

[Blackford, L. et al.]  
ScaLAPACK Users’ Guide  
Society for Industrial and Applied Mathematics, 1997

[Satish Balay et al.]  
PETSc Web page  

[J. Nieplocha et al.]  
Disks Resident Arrays  
http://www.emsl.pnl.gov/docs/parsoft/dra/disk.arrays.html