Parallel Programming Using the Global Arrays Toolkit: Now and Into the Future

Bruce Palmer, Manojkumar Krishnan, Abhinav Vishnu, Sriram Krishnamoorthy, Jeff Daily, Daniel Chavarria, Patrick Nichols

Pacific Northwest National Laboratory
Global Arrays

- Developed over 20 years
- Under active development and focusing on preparing for future exascale platforms
- Available across platforms from PCs to leadership machines
- Easy access to distributed data on multiprocessor machines
  - High programmer productivity
- Library available from: http://www.emsl.pnl.gov/docs/global
Outline of the Tutorial

- Introduction to Global Arrays programming model
- Basic GA commands
- Advanced features of the GA Toolkit
- Current and future developments in GA
Distributed Data vs Shared Memory

Shared Memory:

Data is in a globally accessible address space, any processor can access data by specifying its location using a global index.

Data is mapped out in a natural manner (usually corresponding to the original problem) and access is easy. Information on data locality is obscured and leads to loss of performance.
Distributed vs Shared Data View

**Distributed Data:**

Data is explicitly associated with each processor, accessing data requires specifying the location of the data on the processor and the processor itself.

Data locality is explicit but data access is complicated. Distributed computing is typically implemented with message passing (e.g. MPI)

![Diagram showing processors and data locations](image)

- (0xf5670,P0)
- (0xf32674,P5)
Global Arrays

Distributed dense arrays that can be accessed through a shared memory-like style

Physically distributed data

single, shared data structure/
global indexing

e.g., access $A(4,3)$ rather than $buf(7)$ on task 2

Global Address Space
Creating Global Arrays

\[
g_a = \text{NGA\textunderscore Create}(\text{type, ndim, dims, name, chunk})
\]

- integer array handle
- character string
- minimum block size on each processor
- float, double, int, etc.
- array of dimensions
- dimension
One-sided Communication

Message Passing:
Message requires cooperation on both sides. The processor sending the message (P1) and the processor receiving the message (P0) must both participate.

One-sided Communication:
Once message is initiated on sending processor (P1) the sending processor can continue computation. Receiving processor (P0) is not involved. Data is copied directly from switch into memory on P0.
Remote Data Access in GA vs MPI

Message Passing:

identify size and location of data blocks

loop over processors:
   if (me = P_N) then
      pack data in local message buffer
      send block of data to message buffer on P0
   else if (me = P0) then
      receive block of data from P_N in message buffer
      unpack data from message buffer to local buffer
   endif
end loop

copy local data on P0 to local buffer

Global Arrays:

NGA_Get(g_a, lo, hi, buffer, ld);

Global Array handle
Global upper and lower indices of data patch
Local buffer and array of strides
Onesided vs Message Passing

► Message-passing
  ■ Communication patterns are regular or at least predictable
  ■ Algorithms have a high degree of synchronization
  ■ Data consistency is straightforward

► One-sided
  ■ Communication is irregular
    ● Load balancing
  ■ Algorithms are asynchronous
    ● But also can be used for synchronous algorithms
  ■ Data consistency must be explicitly managed
Global Array Model of Computations

Shared Object

get

local memory

cmpute/update

Shared Object

put
copy to shared object

local memory

copy to local memory

local memory
Global Arrays vs. Other Models

Advantages:

- Inter-operates with MPI
  - Use more convenient global-shared view for multi-dimensional arrays, but can use MPI model wherever needed
- Data-locality and granularity control is explicit with GA’s get-compute-put model, unlike the non-transparent communication overheads with other models (except MPI)
- Library-based approach: does not rely upon smart compiler optimizations to achieve high performance

Disadvantage:

- Data consistency must be explicitly managed
Global Arrays (cont.)

- Shared data model in context of distributed dense arrays
- **Much** simpler than message-passing for many applications
- Complete environment for parallel code development
- Compatible with MPI
- Data locality control similar to distributed memory/message passing model
- Extensible
- Scalable
Data Locality in GA

What data does a processor own?

\[ \text{NGA\_Distribution}(g\_a, \text{iproc}, \text{lo}, \text{hi}); \]

Where is the data?

\[ \text{NGA\_Access}(g\_a, \text{lo}, \text{hi}, \text{ptr}, \text{ld}) \]

Use this information to organize calculation so that maximum use is made of locally held data
Example: Matrix Multiply

\[ \text{local buffers on the processor} \]

\[ \text{global arrays representing matrices} \]

\[ \text{nga\_put} \]

\[ \text{nga\_get} \]

\[ \text{dgemm} \]
Matrix Multiply (a better version)

more scalable!
(less memory, higher parallelism)

atomic accumulate

gemmm
local buffers on the processor
Application Areas

- electronic structure chemistry
- bioinformatics
- visual analytics
- fluid dynamics
- smoothed particle hydrodynamics
- material sciences
- molecular dynamics
- hydrology

Others: financial security forecasting, astrophysics, biology, climate analysis
Recent Applications

ScalaBLAST


Parallel Inspire


Smooth Particle Hydrodynamics

Outline of the Tutorial

- Introduction to Global Arrays programming model
- Basic GA commands
- Advanced features of the GA Toolkit
- Current and future developments in GA
Global Arrays and MPI are completely interoperable. Code can contain calls to both libraries.

Application programming language interface

Fortran  C  C++  Python

distributed arrays layer
memory management, index translation

MPI
Global operations

ARMCI
portable 1-sided communication
put, get, locks, etc

system specific interfaces
LAPI, GM/Myrinet, threads, VIA,..
Writing GA Programs

- GA requires the following functionalities from a message passing library (MPI/TCGMSG):
  - initialization and termination of processes
  - Broadcast, Barrier
  - a function to abort the running parallel job in case of an error

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

- GA is compatible with MPI
Source Code and More Information

- Version 5.0.2 available
- Homepage at http://www.emsl.pnl.gov/docs/global/
- Platforms
  - IBM SP, BlueGene
  - Cray XT, XE6 (Gemini)
  - Linux Cluster with Ethernet, Infiniband
  - Solaris
  - Fujitsu
  - Hitachi
  - NEC
  - HP
  - Windows
Documentation on Writing, Building and Running GA programs

- For detailed information
  - GA Webpage
    - GA papers, APIs, user manual, etc.
    - (Google: Global Arrays)
    - http://www.emsl.pnl.gov/docs/global/
  - GA User Manual
    - http://www.emsl.pnl.gov/docs/global/user.html
  - GA API Documentation
    - GA Webpage => User Interface
  - GA Support/Help
    - hpctools@pnl.gov or hpctools@emsl.pnl.gov
  - 2 mailing lists: GA User Forum, and GA Announce
Installing GA

- GA 5.0 established autotools (configure && make && make install) for building
  - No environment variables are required
    - Traditional configure env vars CC, CFLAGS, CPPFLAGS, LIBS, etc
  - Specify the underlying network communication protocol
    - Only required on clusters with a high performance network
      - e.g. If the underlying network is Infiniband using OpenIB protocol
      - configure --with-openib
  - GA requires MPI for basic start-up and process management
    - You can either use MPI or TCGMSG wrapper to MPI
      - MPI is the default: configure
      - TCGMSG-MPI wrapper: configure --with-mpi --with-tcgmsg
      - TCGMSG: configure --with-tcgmsg

- Various “make” targets
  - “make” to build GA libraries
  - “make install” to install libraries
  - “make checkprogs” to build tests and examples
  - “make check MPIEXEC='mpiexec -np 4'” to run test suite

- VPATH builds: one source tree, many build trees i.e. configurations
Compiling and Linking GA Programs

Your Makefile: Please refer to the CFLAGS, FFLAGS, CPPFLAGS, LDFLAGS and LIBS variables, which will be printed if you “make flags”.

```bash
# ==============================================================
# Suggested compiler/linker options are as follows.
# GA libraries are installed in /Users/d3n000/ga/ga-5-0/bld_openmpi_shared/lib
# GA headers are installed in /Users/d3n000/ga/ga-5-0/bld_openmpi_shared/include
#
CPPFLAGS="-I/Users/d3n000/ga/ga-5-0/bld_openmpi_shared/include"
#
LDFLAGS="-L/Users/d3n000/ga/ga-5-0/bld_openmpi_shared/lib"
#
# For Fortran Programs:
FFLAGS="-fdefault-integer-8"
LIBS="-lga -framework Accelerate"
#
# For C Programs:
CFLAGS=""
LIBS="-lga -framework Accelerate -L/usr/local/lib/gcc/x86_64-apple-darwin10/4.6.0
-L/usr/local/lib/gcc/x86_64-apple-darwin10/4.6.0/..//../ -lgfortran"
# ==============================================================

You can use these variables in your Makefile:
For example: gcc $(CPPLAGS) $(LDLAGS) -o ga_test ga_test.c $(LIBS)
Writing GA Programs

GA Definitions and Data types

- C programs include files: ga.h, macdecls.h
- Fortran programs should include the files: mafdecls.fh, global.fh.

```c
#include <stdio.h>
#include "mpi.h"
#include "ga.h"
#include "macdecls.h"

int main( int argc, char **argv ) {
    /* Parallel program */
}
```
Running GA Programs

- Example: Running a test program “ga_test” on 2 processes for GA built using the MPI runtime
- mpirun -np 2 ga_test
- Running a GA program is same as MPI
11 Basic GA Operations

- GA programming model is very simple.
- Most of a parallel program can be written with these basic calls
  - `GA_Init`, `GA_Terminate`
  - `GA_Nnodes`, `GA_Nodeid`
  - `GA_Create`, `GA_Destroy`
  - `GA_Put`, `GA_Get`
  - `GA_Sync`
There are two functions to initialize GA:

- **Fortran**
  - subroutine ga_initialize()
  - subroutine ga_initialize_ltd(limit)

- **C**
  - void GA_Initialize()
  - void GA_Initialize_ltd(size_t limit)

- **Python**
  - import ga, then ga.set_memory_limit(limit)

To terminate a GA program:

- **Fortran** subroutine ga_terminate()
- **C** void GA_Terminate()
- **Python** N/A
Parallel Environment - Process Information

Parallel Environment:
- how many processes are working together (size)
- what their IDs are (ranges from 0 to size-1)

To return the process ID of the current process:
- **Fortran** integer function ga_nodeid()
- **C** int GA_Nodeid()
- **Python** nodeid = ga.nodeid()

To determine the number of computing processes:
- **Fortran** integer function ga_nnodes()
- **C** int GA_Nnodes()
- **Python** nnodes = ga.nnodes()
Parallel Environment - Process Information (EXAMPLE)

```fortran
program main
#include "mafdecls.h"
#include "global.fh"
integer ierr,me,nproc

call mpi_init(ierr)
call ga_initialize()

me = ga_nodeid()
size = ga_nnodes()
write(6,*) 'Hello world: My rank is ' + me + ' out of ' + & size + ' processes/nodes'

call ga_terminate()
call mpi_finalize()
end
```

$ mpirun --np 4 helloworld
Hello world: My rank is 0 out of 4 processes/nodes
Hello world: My rank is 2 out of 4 processes/nodes
Hello world: My rank is 3 out of 4 processes/nodes
Hello world: My rank is 1 out of 4 processes/nodes

$ mpirun --np 4 python helloworld.py
Hello world: My rank is 0 out of 4 processes/nodes
Hello world: My rank is 2 out of 4 processes/nodes
Hello world: My rank is 3 out of 4 processes/nodes
Hello world: My rank is 1 out of 4 processes/nodes
GA Data Types

- **C Data types**
  - C_INT - int
  - C_LONG - long
  - C_FLOAT - float
  - C_DBL - double
  - C_SCPL - single complex
  - C_DCPL - double complex

- **Fortran Data types**
  - MT_F_INT - integer (4/8 bytes)
  - MT_F_REAL - real
  - MT_F_DBL - double precision
  - MT_F_SCPL - single complex
  - MT_F_DCPL - double complex
Creating/Destroying Arrays

To create an array with a regular distribution:

- **Fortran** logical function nga_create(type, ndim, dims, name, chunk, g_a)
- **C** int NGA_Create(int type, int ndim, int dims[], char *name, int chunk[])
- **Python**

```python
g_a = ga.create(type, dims, name='', chunk=None, int pgroup=-1)
```

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>a unique character string</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>GA data type</td>
<td></td>
</tr>
<tr>
<td>dims()</td>
<td>array dimensions</td>
<td></td>
</tr>
<tr>
<td>chunk()</td>
<td>minimum size that dimensions should be chunked</td>
<td></td>
</tr>
<tr>
<td>g_a</td>
<td>array handle for future references</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
dims(1) = 5000
 dims(2) = 5000
 chunk(1) = -1 !Use defaults
 chunk(2) = -1
if (.not. nga_create(MT_F_DBL,2,dims,'Array_A',chunk,g_a))
+    call ga_error("Could not create global array A",g_a)
```
Creating/Destroying Arrays (cont.)

► To create an array with an irregular distribution:

- **Fortran** logical function nga_create_irreg (type, ndim, dims, array_name, map, nblock, g_a)
- **C** int NGA_Create_irreg(int type, int ndim, int dims[], char* array_name, nblock[], map[])
- **Python**

```python
    g_a = ga.create_irreg(int gtype, dims, block, map, name="", pgroup=-1)
```

- **character**(*) name - a unique character string [input]
- **integer** type - GA datatype [input]
- **integer** dims - array dimensions [input]
- **integer** nblock(*) - no. of blocks each dimension is divided into [input]
- **integer** map(*) - starting index for each block [input]
- **integer** g_a - integer handle for future references [output]
Creating/Destroying Arrays (cont.)

Example of irregular distribution:
- The distribution is specified as a Cartesian product of distributions for each dimension. The array indices start at 1.
  - The figure demonstrates distribution of a 2-dimensional array 8x10 on 6 (or more) processors. block[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.

```bash
block(1) = 3
block(2) = 2
map(1) = 1
map(2) = 3
map(3) = 7
map(4) = 1
map(5) = 6
if (.not. nga_create_irreg(MT_F_DBL,2,dims, &
  'Array_A',map,block,g_a)) &
call ga_error("Could not create array A",g_a)
```
Creating/Destroying Arrays (cont.)

▶ To *duplicate* an array:
  - **Fortran** logical function `ga_duplicate(g_a, g_b, name)`
  - **C** `int GA_Duplicate(int g_a, char *name)`
  - **Python** `ga.duplicate(g_a, name)`

▶ Global arrays can be *destroyed* by calling the function:
  - **Fortran** subroutine `ga_destroy(g_a)`
  - **C** `void GA_Destroy(int g_a)`
  - **Python** `ga.destroy(g_a)`

```plaintext
call nga_create(MT_F_INT,dim,dims, +    'array_a',chunk,g_a)
call ga_duplicate(g_a,g_b,'array_b')
call ga_destroy(g_a)
```

- `g_a, g_b;`: integer
- `name`: character(*)
- `g_a`: a character string [input]
- `g_a`: Integer handle for reference array [input]
- `g_b`: Integer handle for new array [output]
Put/Get

**Put** copies data from a local array to a global array section:

- **Fortran** subroutine nga_put(g_a, lo, hi, buf, ld)
- **C** void NGA_Put(int g_a, int lo[], int hi[], void *buf, int ld[])
- **Python** ga.put(g_a, buf, lo=None, hi=None)

**Get** copies data from a global array section to a local array:

- **Fortran** subroutine nga_get(g_a, lo, hi, buf, ld)
- **C** void NGA_Get(int g_a, int lo[], int hi[], void *buf, int ld[])
- **Python** buffer = ga.get(g_a, lo, hi, numpy.ndarray buffer=None)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>g_a global array handle</td>
<td>[input]</td>
</tr>
<tr>
<td>integer</td>
<td>lo(), hi() limits on data block to be moved</td>
<td>[input]</td>
</tr>
<tr>
<td>Double precision/complex/integer</td>
<td>buf local buffer</td>
<td>[output]</td>
</tr>
<tr>
<td>integer</td>
<td>ld() array of strides for local buffer</td>
<td>[input]</td>
</tr>
</tbody>
</table>

Proudly Operated by Battelle Since 1965
Put/Get (cont.)

Example of *put* operation:

- transfer data from a local buffer (10 x10 array) to (7:15,1:8) section of a 2-dimensional 15 x10 global array into \( lo=\{7,1\} \), \( hi=\{15,8\} \), \( ld=\{10\} \)

```fortran
double precision buf(10,10)
:   :   :
call nga_put(g_a,lo,hi,buf,ld)
```

![Diagram showing data transfer from local to global array]
Atomic Accumulate

Accumulate combines the data from the local array with data in the global array section:

- **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)

- **Python**
  ga.acc(g_a, buffer, lo=None, hi=None, alpha=None)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>g_a array handle</td>
<td>[input]</td>
</tr>
<tr>
<td>integer</td>
<td>lo(), hi() limits on data block to be moved</td>
<td>[input]</td>
</tr>
<tr>
<td>double</td>
<td>precision/complex buf local buffer</td>
<td>[input]</td>
</tr>
<tr>
<td>integer</td>
<td>ld() array of strides for local buffer</td>
<td>[input]</td>
</tr>
<tr>
<td>double</td>
<td>precision/complex alpha arbitrary scale factor</td>
<td>[input]</td>
</tr>
</tbody>
</table>
Atomic Accumulate (cont)

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

- **Sync** is a collective operation
- It acts as a barrier, which synchronizes all the processes and ensures that all the Global Array operations are complete at the call
- The functions are:
  - **Fortran** subroutine `ga_sync()`
  - **C** `void GA_Sync()`
  - **Python** `ga.sync()`

**GA_sync is the main mechanism in GA for guaranteeing data consistency**
Global Operations

- **Fortran**
  
  subroutine ga_brdcst(type, buf, lenbuf, root)
  subroutine ga_igop(type, x, n, op)
  subroutine ga_dgop(type, x, n, op)

- **C**
  
  void GA_Brcdst(void *buf, int lenbuf, int root)
  void GA_Igop(long x[], int n, char *op)
  void GA_Dgop(double x[], int n, char *op)

- **Python**
  
  buffer = ga.brdcst(buffer, root)
  buffer = ga.gop(x, op)
Global Array Model of Computations

Shared Object

- copy to local memory
- get

local memory

compute/update

local memory

Shared Object

- copy to shared object
- put

local memory
## Locality Information

- Discover array elements held by each processor
  - **Fortran**  `nga_distribution(g_a, proc, lo, hi)`
  - **C**        `void NGA_Distribution(int g_a, int proc, int *lo, int *hi)`
  - **Python**   `lo, hi = ga.distribution(g_a, proc=-1)`

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>integer</td>
<td>array handle</td>
<td>[input]</td>
</tr>
<tr>
<td>g_a</td>
<td>integer</td>
<td>processor ID</td>
<td>[input]</td>
</tr>
<tr>
<td>proc</td>
<td>integer</td>
<td>lower index</td>
<td>[output]</td>
</tr>
<tr>
<td>lo(ndim)</td>
<td>integer</td>
<td>upper index</td>
<td>[output]</td>
</tr>
<tr>
<td>hi(ndim)</td>
<td>integer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```fortran
do iproc = 1, nproc
   write(6,*) 'Printing g_a info for processor', iproc
   call nga_distribution(g_a, iproc, lo, hi)
   do j = 1, ndim
      write(6,*) j, lo(j), hi(j)
   end do
endo do
```
Example: Matrix Multiply

/* Determine which block of data is locally owned. Note that
the same block is locally owned for all GAs. */
NGA_Distribution(g_c, me, lo, hi);
/* Get the blocks from g_a and g_b needed to compute this block in
g_c and copy them into the local buffers a and b. */
lo2[0] = lo[0]; lo2[1] = 0; hi2[0] = hi[0]; hi2[1] = dims[0]-1;
NGA_Get(g_a, lo2, hi2, a, ld);
lo3[0] = 0; lo3[1] = lo[1]; hi3[0] = dims[1]-1; hi3[1] = hi[1];
NGA_Get(g_b, lo3, hi3, b, ld);
/* Do local matrix multiplication and store the result in local
buffer c. Start by evaluating the transpose of b. */
for(i=0; i < hi3[0]-lo3[0]+1; i++)
    for(j=0; j < hi3[1]-lo3[1]+1; j++)
        btrns[j][i] = b[i][j];
/* Multiply a and b to get c */
for(i=0; i < hi[0] - lo[0] + 1; i++) {
    for(j=0; j < hi[1] - lo[1] + 1; j++) {
        c[i][j] = 0.0;
        for(k=0; k<dims[0]; k++)
            c[i][j] = c[i][j] + a[i][k]*btrns[j][k];
    }
}
/* Copy c back to g_c */
NGA_Put(g_c, lo, hi, c, ld);
New Interface for Creating Arrays

► Developed to handle the proliferating number of properties that can be assigned to Global Arrays

Fortran

integer function ga_create_handle()

subroutine ga_set_data(g_a, dim, dims, type)

subroutine ga_set_array_name(g_a, name)

subroutine ga_set_chunk(g_a, chunk)

subroutine ga_set_irreg_distr(g_a, map, nblobk)

subroutine ga_set_ghosts(g_a, width)

subroutine ga_set_block_cyclic(g_a, dims)

subroutine ga_set_block_cyclic_proc_grid(g_a, dims, proc_grid)

logical function ga_allocate(g_a)
New Interface for Creating Arrays

C

int GA_Create_handle()
void GA_Set_data(int g_a, int dim, int *dims,
    int type)
void GA_Set_array_name(int g_a, char* name)
void GA_Set_chunk(int g_a, int *chunk)
void GA_Set_irreg_distr(int g_a, int *map,
    int *nblock)
void GA_Set_ghosts(int g_a, int *width)
void GA_Set_block_cyclic(int g_a, int *dims)
void GA_Set_block_cyclic_proc_grid(int g_a, int
    *dims,
    int *proc_grid)
int GA_Allocate(int g_a)
New Interface for Creating Arrays

**Python**

```python
handle = ga.create_handle()
ga.set_data(g_a, dims, type)
ga.set_array_name(g_a, name)
ga.set_chunk(g_a, chunk)
ga.set_irreg_distr(g_a, map, nblock)
ga.set_ghosts(g_a, width)
ga.set_block_cyclic(g_a, dims)
ga.set_block_cyclic_proc_grid(g_a, dims, proc_grid)
bool ga.allocate(int g_a)
```
integer ndim, dims(2), chunk(2)
integer g_a, g_b
logical status

ndim = 2
dims(1) = 5000
dims(2) = 5000
chunk(1) = 100
chunk(2) = 100

Create global array A using old interface

status = nga_create(MT_F_DBL, ndim, dims, chunk, 'array_A', g_a)

Create global array B using new interface

C

C

C

C

g_b = ga_create_handle()
call ga_set_data(g_b, ndim, dims, MT_F_DBL)
call ga_set_chunk(g_b, chunk)
call ga_set_name(g_b, 'array_B')
call ga_allocate(g_b)
Basic Array Operations

**Whole Arrays:**

- To set all the elements in the array to zero:
  - **Fortran** subroutine `ga_zero(g_a)`
  - **C** `void GA_Zero(int g_a)`
  - **Python** `ga.zero(g_a)`

- To assign a single value to all the elements in array:
  - **Fortran** subroutine `ga_fill(g_a, val)`
  - **C** `void GA_Fill(int g_a, void *val)`
  - **Python** `ga.fill(g_a, val)`

- To scale all the elements in the array by factor `val`:
  - **Fortran** subroutine `ga_scale(g_a, val)`
  - **C** `void GA_Scale(int g_a, void *val)`
  - **Python** `ga.scale(g_a, val)`
Basic Array Operations (cont.)

Whole Arrays:
- To copy data between two arrays:
  - **Fortran** subroutine ga_copy(g_a, g_b)
  - **C** void GA_Copy(int g_a, int g_b)
  - **Python** ga.copy(g_a, g_b)
- Arrays must be same size and dimension
- Distribution may be different

Global Arrays g_a and g_b distributed on a 3x3 process grid

call ga_create(MT_F_INT, ndim, dims, 'array_A', chunk_a, g_a)
call nga_create(MT_F_INT, ndim, dims, 'array_B', chunk_b, g_b)

... Initialize g_a ....
call ga_copy(g_a, g_b)
Basic Array Patch Operations

- Patch Operations:
  - The copy patch operation:
    - **Fortran**
      ```fortran
      subroutine nga_copy_patch(trans, g_a, alo, ahi, g_b, blo, bhi)
      
      subroutine nga_copy_patch(trans, g_a, alo, ahi, g_b, blo, bhi)
      ```
    - **C**
      ```c
      void NGA_Copy_patch(char trans, int g_a, int alo[], int ahi[], int g_b, int blo[], int bhi[])
      ```
    - **Python**
      ```python
ga.copy(g_a, g_b, alo=None, ahi=None, blo=None, bhi=None, bint trans=False)
      ```
  - Number of elements must match
    
    | 0 | 1 | 2 |
    |---|---|---|
    | 3 | 4 | 5 |
    | 6 | 7 | 8 |

    | 0 | 1 | 2 |
    |---|---|---|
    | 3 | 4 | 5 |
    | 6 | 7 | 8 |

    Copy
Basic Array Patch Operations (cont.)

- Patches (Cont):
  - To set only the region defined by $lo$ and $hi$ to zero:
    - **Fortran** subroutine nga_zero_patch(g_a, lo, hi)
    - **C** void NGA_Zero_patch(int g_a, int lo[], int hi[])
    - **Python** ga.zero(g_a, lo=None, hi=None)
  - To assign a single value to all the elements in a patch:
    - **Fortran** subroutine nga_fill_patch(g_a, lo, hi, val)
    - **C** void NGA_Fill_patch(int g_a, int lo[], int hi[], void *val)
    - **Python** ga.fill(g_a, value, lo=None, hi=None)
Basic Array Patch Operations (cont.)

Patches (Cont):

- To scale the patch defined by lo and hi by the factor val:
  - **Fortran** subroutine nga_scale_patch(g_a, lo, hi, val)
  - **C** void NGA_Scale_patch(int g_a, int lo[], int hi[], void *val)
  - **Python** ga.scale(g_a, value, lo=None, hi=None)

- The copy patch operation:
  - **Fortran** subroutine nga_copy_patch(trans, g_a, alo, ahi, g_b, blo, bhi)
  - **C** void NGA_Copy_patch(char trans, int g_a, int alo[],
  int ahi[], int g_b, int blo[], int bhi[])
  - **Python** ga.copy(g_a, g_b, alo=None, ahi=None, blo=None, bhi=None, bint trans=False)
Outline of the Tutorial

- Introduction to Global Arrays programming model
- Basic GA commands
- Advanced features of the GA Toolkit
- Current and future developments in GA
**Scatter/Gather**

- **Scatter** puts array elements into a global array:
  - **Fortran** subroutine nga_scatter(g_a, v, subscript_array, n)
  - **C** void NGA_Scatter(int g_a, void *v, int *subscript_array[], int n)
  - **Python** ga.scatter(g_a, values, subsarray)

- **Gather** gets the array elements from a global array into a local array:
  - **Fortran** subroutine nga_gather(g_a, v, subscript_array, n)
  - **C** void NGA_Gather(int g_a, void *v, int *subscript_array[], int n)
  - **Python** values = ga.gather(g_a, subsarray, numpy.ndarray(values=None))

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>g_a array handle</td>
<td>[input]</td>
</tr>
<tr>
<td>double precision</td>
<td>v(n) array of values</td>
<td>[input/output]</td>
</tr>
<tr>
<td>integer</td>
<td>n number of values</td>
<td>[input]</td>
</tr>
<tr>
<td>integer</td>
<td>subscript_array location of values in global array</td>
<td>[input]</td>
</tr>
</tbody>
</table>

*Proudly Operated by Battelle Since 1965*
Example of *scatter* operation:

- Scatter the 5 elements into a 10x10 global array
  - Element 1 \( v[0] = 5 \)
    - \( \text{subsArray}[0][0] = 2 \)
    - \( \text{subsArray}[0][1] = 3 \)
  - Element 2 \( v[1] = 3 \)
    - \( \text{subsArray}[1][0] = 3 \)
    - \( \text{subsArray}[1][1] = 4 \)
  - Element 3 \( v[2] = 8 \)
    - \( \text{subsArray}[2][0] = 8 \)
    - \( \text{subsArray}[2][1] = 5 \)
  - Element 4 \( v[3] = 7 \)
    - \( \text{subsArray}[3][0] = 3 \)
    - \( \text{subsArray}[3][1] = 7 \)
  - Element 5 \( v[4] = 2 \)
    - \( \text{subsArray}[4][0] = 6 \)
    - \( \text{subsArray}[4][1] = 3 \)

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

```fortran
integer subscript(ndim,nlen) :
call nga_scatter(g_a,v,subscript,nlen)
```
Read and Increment

- *Read inc* remotely updates a particular element in an integer global array and returns the original value:
  - **Fortran**  integer function nga_read_inc(g_a, subscript, inc)
  - **C**  long NGA_Read_inc(int g_a, int subscript[], long inc)
  - **Python**  val = ga.read_inc(g_a, subscript, inc=1)
- Applies to integer arrays only
- Can be used as a global counter for dynamic load balancing

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>g_a</td>
</tr>
<tr>
<td>integer</td>
<td>subscript(ndim), inc</td>
</tr>
</tbody>
</table>
c Create task counter
    status = nga_create(MT_F_INT, one, one, chunk, g_counter)
call ga_zero(g_counter)
:
itask = nga_read_inc(g_counter, one, one)

... Translate itask into task ...

Every integer value is read once and only once by some processor
Hartree-Fock SCF

Obtain variational solutions to the electronic Schrödinger equation

\[ H\Psi = E\Psi \]

within the approximation of a single Slater determinant.

Assuming the one electron orbitals are expanded as

\[ \phi_i(\mathbf{r}) = \sum_{\mu} C_{i\mu} \chi_{\mu}(\mathbf{r}) \]

the calculation reduces to the self-consistent eigenvalue problem

\[ F_{\mu\nu} C_{k\nu} = \varepsilon D_{\mu\nu} C_{k\nu} \]

\[ D_{\mu\nu} = \sum_{k} C_{\mu k} C_{\nu k} \]

\[ F_{\mu\nu} = h_{\mu\nu} + \frac{1}{2} \sum_{\omega\lambda} \left[ 2(\mu \nu | \omega \lambda) - (\mu \omega | \nu \lambda) \right] D_{\omega \lambda} \]
Parallelizing the Fock Matrix

The bulk of the work involves computing the 4-index elements \((\mu \nu | \omega \lambda)\). This is done by decomposing the quadruple loop into evenly sized blocks and assigning blocks to each processor using a global counter. After each processor completes a block it increments the counter to get the next block.

\[
\begin{align*}
\text{do } & i \\
\text{do } & j \\
\text{do } & k \\
\text{do } & l \\
F(i,j) = & \ldots \\
\end{align*}
\]
Gorden Bell finalist at SC09 - GA Crosses the Petaflop Barrier

- GA-based parallel implementation of coupled cluster calculation performed at **1.39 petaflops using over 223,000 processes** on ORNL's Jaguar petaflop system


- Global Arrays is one of two programming models that have achieved this level of performance.
Direct Access to Local Data

- Global Arrays support abstraction of a distributed array object
- Object is represented by an integer handle
- A process can access its portion of the data in the global array
- To do this, the following steps need to be taken:
  - Find the distribution of an array, i.e. which part of the data the calling process owns
  - Access the data
  - Operate on the data: read/write
  - Release the access to the data
Access

- To provide direct access to local data in the specified patch of the array owned by the calling process:
  - **Fortran** subroutine nga_access(g_a, lo, hi, index, ld)
  - **C** void NGA_Access(int g_a, int lo[], int hi[],
  void *ptr, int ld[])
  - **Python** ndarray = ga.access(g_a, lo=None, hi=None)
- Processes can access the local position of the global array
  - Process “0” can access the specified patch of its local position of the array
  - Avoids memory copy
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 do_subroutine_task(dbl_mb(index),ld(1))
call nga_release(g_a,lo,hi)
subroutine do_subroutine_task(a,ld1)
double precision a(ld1,*)

Access: gives a pointer to this local patch
The non-blocking APIs are derived from the blocking interface by adding a handle argument that identifies an instance of the non-blocking request.

**Fortran**
- 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)

**C**
- 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)
- int NGA_NbWait(ga_nbhdl_t* nbhandle)

**Python**
- handle = ga.nbput(g_a, buffer, lo=None, hi=None)
- buffer, handle = ga.nbget(g_a, lo=None, hi=None, numpy.ndarray buffer=None)
- handle = ga.nbacc(g_a, buffer, lo=None, hi=None, alpha=None)
- ga.nbwait(handle)
Non-Blocking Operations

de double precision buf1(nmax,nmax)
  double precision buf2(nmax,nmax)
  :  
call nga_nbget(g_a,lo1,hi1,buf1,ld1,nb1)
ncount = 1
do while(......)
  if (mod(ncount,2).eq.1) then
    ... Evaluate lo2, hi2
    call nga_nbget(g_a,lo2,hi2,buf2,nb2)
call nga_wait(nb1)
    ... Do work using data in buf1
  else
    ... Evaluate lo1, hi1
    call nga_nbget(g_a,lo1,hi1,buf1,nb1)
call nga_wait(nb2)
    ... Do work using data in buf2
endif
ncount = ncount + 1
end do
SRUMMA Matrix Multiplication

\[ C = A \cdot B \]

**Computation**

**Comm. (Overlap)**

**Issue NB Get A and B blocks**

do (until last chunk)

issue NB Get to the next blocks

wait for previous issued call compute A*B (sequential dgemm)

NB atomic accumulate into “C” matrix

done

**Advantages:**
- Minimum memory
- Highly parallel
- Overlaps computation and communication
  - latency hiding
- exploits data locality
- patch matrix multiplication (easy to use)
- dynamic load balancing

http://hpc.pnl.gov/projects/srumma/

Pacific Northwest National Laboratory

Proudly Operated by Battelle Since 1965
SRUMMA Matrix Multiplication: Improvement over PBLAS/ScaLAPACK

Parallel Matrix Multiplication on the HP/Quadrics Cluster at PNNL
Matrix size: 40000x40000
Efficiency 92.9% w.r.t. serial algorithm and 88.2% w.r.t. machine peak on 1849 CPUs

- **SRUMMA**
- **PBLAS/ScaLAPACK pdgemm**
- **Theoretical Peak**
- **Perfect Scaling**

---

Pacific Northwest
NATIONAL LABORATORY

_Proudly Operated by Battelle Since 1965_
Cluster Information

Example:
- 2 nodes with 4 processors each. Say, there are 7 processes created.
  - `ga_cluster_nnodes` returns 2
  - `ga_cluster_nodeid` returns 0 or 1
  - `ga_cluster_nprocs(inode)` returns 4 or 3
  - `ga_cluster_procid(inode,iproc)` returns a processor ID
Cluster Information (cont.)

To return the total number of nodes that the program is running on:

- **Fortran** integer function ga_cluster_nnodes()
- **C** int GA_Cluster_nnodes()
- **Python** nnodes = ga.cluster_nnodes()

To return the node ID of the process:

- **Fortran** integer function ga_cluster_nodeid()
- **C** int GA_Cluster_nodeid()
- **Python** nodeid = ga.cluster_nodeid()
To return the number of processors available on node inode:
- **Fortran** `integer function ga_cluster_nprocs(inode)`
- **C** `int GA_Cluster_nprocs(int inode)`
- **Python** `nprocs = ga.cluster_nprocs(inode)`

To return the processor ID associated with node inode and the local processor ID `iproc`:
- **Fortran** `integer function ga_cluster_procid(inode, iproc)`
- **C** `int GA_Cluster_procid(int inode, int iproc)`
- **Python** `procid = ga.cluster_procid(inode, iproc)`

0(0) 1(1) 4(0) 5(1)
2(2) 3(3) 6(2) 7(3)
Accessing Processor Memory

Node

SMP Memory

R_8  R_9  R_{10}  R_{11}

P_8  P_9  P_{10}  P_{11}

ga_access
Processor Groups

- To create a new processor group:
  - **Fortran**  integer function ga_pgroup_create(list, size)
  - **C**  int GA_Pgroup_create(int *list, int size)
  - **Python**  pgroup = ga.pgroup_create(list)

- To assign a processor groups:
  - **Fortran**  logical function nga_create_config(type, ndim, dims, name, chunk, p_handle, g_a)
  - **C**  int NGA_Create_config(int type, int ndim, int dims[], char *name, int p_handle, int chunk[])
  - **Python**  g_a = ga.create(type, dims, name, chunk, pgroup=-1)

- **g_a** - global array handle [input]
- **p_handle** - processor group handle [output]
- **list(size)** - list of processor IDs in group [input]
- **size** - number of processors in group [input]
Processor Groups

world group

group A  group B

group C
Processor Groups (cont.)

- To set the default processor group
  - **Fortran** subroutine ga_pgroup_set_default(p_handle)
  - **C** void GA_Pgroup_set_default(int p_handle)
  - **Python** ga.pgroup_set_default(p_handle)

- To access information about the processor group:
  - **Fortran**
    - integer function ga_pgroup_nnodes(p_handle)
    - integer function ga_pgroup_nodeid(p_handle)
  - **C**
    - int GA_Pgroup_nnodes(int p_handle)
    - int GA_Pgroup_nodeid(int p_handle)
  - **Python**
    - nnodes = ga.pgroup_nnodes(p_handle)
    - nodeid = ga.pgroup_nodeid(p_handle)

integer p_handle - processor group handle [input]
To determine the handle for a standard group at any point in the program:

- **Fortran**
  - integer function ga_pgroup_get_default()
  - integer function ga_pgroup_get_mirror()
  - integer function ga_pgroup_get_world()

- **C**
  - int GA_Pgroup_get_default()
  - int GA_Pgroup_get_mirror()
  - int GA_Pgroup_get_world()

- **Python**
  - p_handle = ga.pgroup_get_default()
  - p_handle = ga.pgroup_get_mirror()
  - p_handle = ga.pgroup_get_world()
Default Processor Group

```c
create subgroup p_a

p_a = ga_pgroup_create(list, nproc)
call ga_pgroup_set_default(p_a)
call parallel_task()
call ga_pgroup_set_default(ga_pgroup_get_world())
```

```subroutine parallel_task()
p_b = ga_pgroup_create(new_list, new_nproc)
call ga_pgroup_set_default(p_b)
call parallel_subtask()
```
MD Application on Groups

Scaling of Single Parallel Task

Scaling of Parallel MD Tasks on Groups
Creating Arrays with Ghost Cells

To create arrays with ghost cells:

- For arrays with regular distribution:
  - **Fortran**
    
    ```
    logical function nga_create_ghosts(type, dims, width, array_name, chunk, g_a)
    ```
  - **C**
    
    ```
    int int NGA_Create_ghosts(int type, int ndim, int dims[], int width[], char *array_name, int chunk[])
    ```
  - **Python**
    
    ```
    g_a = ga.create_ghosts(type, dims, width, name="", chunk=None, pgroup=-1)
    ```

- For arrays with irregular distribution:
  - **n-d Fortran**
    
    ```
    logical function nga_create_ghosts_irreg(type, dims, width, array_name, map, block, g_a)
    ```
  - **C**
    
    ```
    int NGA_Create_ghosts_irreg(int type, int ndim, int dims[], int width[], char *array_name, int map[], int block[])
    ```
  - **Python**
    
    ```
    g_a = ga.create_ghosts_irreg(type, dims, width, block, map, name="", pgroup=-1)
    ```

integer width(ndim) - array of ghost cell widths [input]
Ghost Cells

Operations:

- **NGA_Create_ghosts** - creates array with ghosts cells
- **GA_Update_ghosts** - updates with data from adjacent processors
- **NGA_Access_ghosts** - provides access to “local” ghost cell elements
- **NGA_Nbget_ghost_dir** - nonblocking call to update ghosts cells
Ghost Cell Update

Automatically update ghost cells with appropriate data from neighboring processors. A multiprotocol implementation has been used to optimize the update operation to match platform characteristics.
Periodic Interfaces

- Periodic interfaces to the one-sided operations have been added to Global Arrays in version 3.1 to support computational fluid dynamics problems on multidimensional grids.

- They provide an index translation layer that allows users to request blocks using put, get, and accumulate operations that possibly extend beyond the boundaries of a global array.

- The references that are outside of the boundaries are wrapped around inside the global array.

- Current version of GA supports three periodic operations:
  - periodic get
  - periodic put
  - periodic acc
Periodic Interfaces

\[
\begin{align*}
\text{ndim} &= 2 \\
\text{dims}(1) &= 10 \\
\text{dims}(2) &= 10 \\
\text{lo}(1) &= 6 \\
\text{lo}(2) &= 6 \\
\text{hi}(1) &= 11 \\
\text{hi}(2) &= 11 \\
\text{call nga_periodic_get(g_a,lo,hi,buf,ld)}
\end{align*}
\]
Periodic Get/Put/Accumulate

- **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[])
- **Python** ndarray = ga.periodic_get(g_a, lo=None, hi=None, buffer=None)

- **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[])
- **Python** ga.periodic_put(g_a, buffer, lo=None, hi=None)

- **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)
- **Python** ga.periodic_acc(g_a, buffer, lo=None, hi=None, alpha=None)
Lock and Mutex

- *Lock* works together with *mutex*.
- Simple synchronization mechanism to protect a critical section
- To enter a critical section, typically, one needs to:
  - Create mutexes
  - Lock on a mutex
  - Do the exclusive operation in the critical section
  - Unlock the mutex
  - Destroy mutexes
- The *create mutex* functions are:
  - **Fortran** logical function `ga_create_mutexes(number)`
  - **C** `int GA_Create_mutexes(int number)`
  - **Python** `bool ga.create_mutexes(number)`
Lock and Mutex (cont.)

Lock

Unlock
Lock and Mutex (cont.)

The *destroy mutex* functions are:

- **Fortran**  
  logical function ga_destroy_mutexes()

- **C**  
  int GA_Destroy_mutexes()

- **Python**  
  bool ga.destroy_mutexes()

The *lock* and *unlock* functions are:

- **Fortran**  
  subroutine ga_lock(int mutex)
  subroutine ga_unlock(int mutex)

- **C**  
  void GA_lock(int mutex)
  void GA_unlock(int mutex)

- **Python**  
  ga.lock(mutex)
  ga.unlock(mutex)

integer mutex [input] ! mutex id
Fence

- *Fence* blocks the calling process until all the data transfers corresponding to the Global Array operations initiated by this process complete.

- For example, since `ga_put` might return before the data reaches final destination, `ga_init_fence` and `ga_fence` allow process to wait until the data transfer is fully completed.
  - `ga_init_fence();`
  - `ga_put(g_a, ...);`
  - `ga_fence();`

- The *initialize fence* functions are:
  - **Fortran** subroutine `ga_init_fence()`
  - **C** `void GA_Init_fence()`
  - **Python** `ga.init_fence()`

- The *fence* functions are:
  - **Fortran** subroutine `ga_fence()`
  - **C** `void GA_Fence()`
  - **Python** `ga.fence()`
Synchronization Control in Collective Operations

- To eliminate redundant synchronization points:
  - **Fortran** subroutine `ga_mask_sync(prior_sync_mask, post_sync_mask)`
  - **C** `void GA_Mask_sync(int prior_sync_mask, int post_sync_mask)`
  - **Python** `ga.mask_sync(prior_sync_mask, post_sync_mask)`

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

```c
duplicate
status = ga_duplicate(g_a, g_b)
call ga_mask(0,1)
call ga_zero(g_b)
```

```cpp
duplicate
call ga_duplicate(g_a, g_b)
call ga_mask(0,1)
call ga_zero(g_b)
```

```py
duplicate
gb_status = ga_duplicate(g_a, g_b)
gb_call ga_mask(0,1)
gb_call ga_zero(g_b)
```

```java
duplicate
gb_status = ga_duplicate(g_a, g_b)
gb_call ga_mask(0,1)
gb_call ga_zero(g_b)
```

Proudly Operated by Battelle Since 1965
Linear Algebra

To add two 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)

- **Python**
  ga.add(g_a, g_b, g_c, alpha=None, beta=None, 
  alo=None, ahi=None, blo=None, bhi=None, 
  clo=None, chi=None)

To multiply arrays:

- **Fortran**
  subroutine ga_dgemm(transa, transb, m, n, k, alpha, g_a, g_b, beta, g_c)

- **C**
  void GA_Dgemm(char ta, char tb, int m, int n, int k, double alpha, int g_a, int g_b, double beta, int g_c)

- **Python**
  def gemm(bool ta, bool tb, m, n, k, alpha, g_a, g_b, beta, g_c)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha, beta</td>
<td>double precision</td>
<td>- scale factor</td>
<td>[input]</td>
</tr>
<tr>
<td>g_a, g_b, g_c</td>
<td>complex/integer/integer</td>
<td>- array handles</td>
<td>[input]</td>
</tr>
<tr>
<td>transa, transb</td>
<td>character*1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m, n, k</td>
<td>integer</td>
<td></td>
<td>[input]</td>
</tr>
</tbody>
</table>

*1: Character string.
To compute the element-wise dot product of two arrays:

- Three separate functions for data types
  - Integer
    - Fortran: `ga_idot(g_a, g_b)`
    - C: `GA_Idot(int g_a, int g_b)`
  - Double precision
    - Fortran: `ga_ddot(g_a, g_b)`
    - C: `GA_Ddot(int g_a, int g_b)`
  - Double complex
    - Fortran: `ga_zdot(g_a, g_b)`
    - C: `GA_Zdot(int g_a, int g_b)`

- Python has only one function: `ga_dot(g_a, g_b)`

```
integer         g_a, g_b           [input]
integer         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)
```
Linear Algebra (cont.)

► To symmetrize a matrix:
  ▪ **Fortran**  subroutine ga_symmetrize(g_a)
  ▪ **C**  void GA_Symmetrize(int g_a)
  ▪ **Python**  ga.symmetrize(g_a)

► To transpose a matrix:
  ▪ **Fortran**  subroutine ga_transpose(g_a, g_b)
  ▪ **C**  void GA_Transpose(int g_a, int g_b)
  ▪ **Python**  ga.transpose(g_a, g_b)
Linear Algebra on Patches

To add element-wise two patches and save the results into another patch:

- **Fortran** subroutine nga_add_patch(alpha, g_a, alo, ahi, beta, g_b, blo, bhi, g_c, clo, chi)
- **C** void NGA_Add_patch(void *alpha, int g_a[], int ahi[], void *beta, int g_b[], int bhi[], int g_c[], int clo[], int chi[])
- **Python** ga.add(g_a, g_b, g_c, alpha=None, beta=None, alo=None, ahi=None, blo=None, bhi=None, clo=None, chi=None)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>g_a, g_b, g_c</td>
<td>[input]</td>
</tr>
<tr>
<td>dbl prec/comp/int</td>
<td>alpha, beta</td>
<td>scale factors [input]</td>
</tr>
<tr>
<td>integer</td>
<td>ailo, aihi, ajlo, ajhi</td>
<td>g_a patch coord [input]</td>
</tr>
<tr>
<td>integer</td>
<td>bilo, bihi, bjlo, bjhi</td>
<td>g_b patch coord [input]</td>
</tr>
<tr>
<td>integer</td>
<td>cilo, cihi, cjlo, cjhi</td>
<td>g_c patch coord [input]</td>
</tr>
</tbody>
</table>
Linear Algebra on Patches (cont.)

To perform matrix multiplication:

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

<table>
<thead>
<tr>
<th>Type</th>
<th>g_a, ailo, aihi, ajlo, ajhi</th>
<th>g_b, bilo, bihi, bjlo, bjhi</th>
<th>g_c, cilo, cihi, cjlo, cjhi</th>
<th>Patch Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>patch of g_a</td>
<td>patch of g_b</td>
<td>patch of g_c</td>
<td>[input]</td>
</tr>
<tr>
<td>integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>character*1</td>
<td>transa, transb</td>
<td></td>
<td></td>
<td>transpose flags</td>
</tr>
</tbody>
</table>
Linear Algebra on Patches (cont.)

To compute the element-wise dot product of two arrays:

- Three separate functions for data types
  - Integer
    - **Fortran**  `nga_idot_patch(g_a, ta, alo, ahi, g_b, tb, blo, bhi)`
    - **C**  `NGA_Idot_patch(int g_a, char* ta, int alo[], int ahi[], int g_b, char* tb, int blo[], int bhi[])`
  - Double precision
    - **Fortran**  `nga_ddot_patch(g_a, ta, alo, ahi, g_b, tb, blo, bhi)`
    - **C**  `NGA_Ddot_patch(int g_a, char* ta, int alo[], int ahi[], int g_b, char* tb, int blo[], int bhi[])`
  - Double complex
    - **Fortran**  `nga_zdot_patch(g_a, ta, alo, ahi, g_b, tb, blo, bhi)`
    - **C**  `NGA_Zdot_patch(int g_a, char* ta, int alo[], int ahi[], int g_b, char* tb, int blo[], int bhi[])`
  - Python has only one function:  `ga.dot(g_a, g_b, alo=None, ahi=None, blo=None, bhi=None, bint ta=False, bint tb=False)`

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>GA_Idot</td>
<td>(int g_a, int g_b)</td>
</tr>
<tr>
<td>long</td>
<td>GA_Ldot</td>
<td>(int g_a, int g_b)</td>
</tr>
<tr>
<td>float</td>
<td>GA_Fdot</td>
<td>(int g_a, int g_b)</td>
</tr>
<tr>
<td>double</td>
<td>GA_Ddot</td>
<td>(int g_a, int g_b)</td>
</tr>
<tr>
<td>DoubleComplex</td>
<td>GA_Zdot</td>
<td>(int g_a, int g_b)</td>
</tr>
</tbody>
</table>
Block-Cyclic Data Distributions

Normal Data Distribution

Block-Cyclic Data Distribution
Block-Cyclic Data (cont.)

<table>
<thead>
<tr>
<th>Simple Distribution</th>
<th>Scalapack Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0       6    12   18   24   30</td>
<td>0       0,0   0,1</td>
</tr>
<tr>
<td>1       7    13   19   25   31</td>
<td>1       1,0   1,1</td>
</tr>
<tr>
<td>2       8    14   20   26   32</td>
<td>0       0,1   1</td>
</tr>
<tr>
<td>3       9    15   21   27   33</td>
<td>1       1,1   0</td>
</tr>
<tr>
<td>4       10   16   22   28   34</td>
<td>0       1</td>
</tr>
<tr>
<td>5       11   17   23   29   35</td>
<td>0       1</td>
</tr>
</tbody>
</table>

Proudly Operated by Battelle Since 1965
Block-Cyclic Data (cont.)

- Most operations work exactly the same, data distribution is transparent to the user.
- Some operations (matrix multiplication, non-blocking put, get) not implemented.
- Additional operations added to provide access to data associated with particular sub-blocks.
- You need to use the new interface for creating Global Arrays to get create block-cyclic data distributions.
Creating Block-Cyclic Arrays

Must use new API for creating Global Arrays

- **Fortran**
  subroutine ga_set_block_cyclic(g_a, dims)
  subroutine ga_set_block_cyclic_proc_grid(g_a, dims, proc_grid)

- **C**
  void GA_Set_block_cyclic(int g_a, int dims[])
  void GA_Set_block_cyclic_proc_grid(g_a, dims[], proc_grid[])

- **Python**
  ga.set_block_cyclic(g_a, dims)
  ga.set_block_cyclic_proc_grid(g_a, block, proc_grid)

integer dims[] - dimensions of blocks
integer proc_grid[] - dimensions of processor grid (note that product of all proc_grid dimensions)
Block-Cyclic Methods

Methods for accessing data of individual blocks

- **Fortran**
  - subroutine `ga_get_block_info(g_a, num_blocks, block_dims)`
  - integer function `ga_total_blocks(g_a)`
  - subroutine `nga_access_block_segment(g_a, iproc, index, length)`
  - subroutine `nga_access_block(g_a, idx, index, ld)`
  - subroutine `nga_access_block_grid(g_a, subscript, index, ld)`

- **C**
  - `void GA_Get_block_info(g_a, num_blocks[], block_dims[])`
  - `int GA_Total_blocks(int g_a)`
  - `void NGA_Access_block_segment(int g_a, int iproc, void *ptr, int *length)`
  - `void NGA_Access_block(int g_a, int idx, void *ptr, int *ld[])`
  - `void NGA_Access_block_grid(int g_a, int subscript[], void *ptr, int *ld[])`

- **Python**
  - `num_blocks, block_dims = ga.get_block_info(g_a)`
  - `blocks = ga.total_blocks(g_a)`
  - `ndarray = ga.access_block_segment(g_a, iproc)`
  - `ndarray = ga.access_block(g_a, idx)`
  - `ndarray = ga.access_block_grid(g_a, subscript)`

- **Parameters**
  - `length` - total size of blocks held on processor
  - `idx` - index of block in array (for simple block-cyclic distribution)
  - `subscript[]` - location of block in block grid (for Scalapack distribution)
Interfaces to Third Party Software Packages

- **Scalapack**
  - Solve a system of linear equations
  - Compute the inverse of a double precision matrix

- **TAO**
  - General optimization problems

- **Interoperability with Others**
  - PETSc
  - CUMULVS
To determine the process ID that owns the element defined by the array subscripts:

- **Fortran**  
  logical function nga_locate(g_a, subscript, owner)

- **C**  
  int NGA_Locate(int g_a, int subscript[])

- **Python**  
  proc = ga.locate(g_a, subscript)

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

integer g_a  
Integer subscript(ndim)  
integer owner  
array handle  
external subscript  
process id  
[input]  
[input]  
[output]  

owner=5
Data Mapping Information (cont.)

To return a list of process IDs that own the patch:

- **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[])
- **Python** map,procs = ga.locate_region(g_a, lo, hi)

<table>
<thead>
<tr>
<th>Integer</th>
<th>np</th>
<th>- number of processors that own a portion of block</th>
<th>[output]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>g_a</td>
<td>- global array handle</td>
<td>[input]</td>
</tr>
<tr>
<td>Integer</td>
<td>ndim</td>
<td>- number of dimensions of the global array</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>lo(ndim)</td>
<td>- array of starting indices for array section</td>
<td>[input]</td>
</tr>
<tr>
<td>Integer</td>
<td>hi(ndim)</td>
<td>- array of ending indices for array section</td>
<td>[input]</td>
</tr>
<tr>
<td>Integer</td>
<td>map(2<em>ndim,</em>)</td>
<td>- array with mapping information</td>
<td>[output]</td>
</tr>
<tr>
<td>Integer</td>
<td>procs(np)</td>
<td>- list of processes that own a part of array section</td>
<td>[output]</td>
</tr>
</tbody>
</table>

procs = {0,1,2,4,5,6}
map = {lo_{01}, lo_{02}, hi_{01}, hi_{02},
lo_{11}, lo_{12}, hi_{11}, hi_{12},
lo_{21}, lo_{22}, hi_{21}, hi_{22},
lo_{41}, lo_{42}, hi_{41}, hi_{42},
lo_{51}, lo_{52}, hi_{51}, hi_{52},
lo_{61}, lo_{62}, hi_{61}, hi_{62}}
Outline of the Tutorial

- Introduction to Global Arrays programming model
- Basic GA commands
- Advanced features of the GA Toolkit
- Current and future developments in GA
Profiling Capability

- Weak bindings for ARMCI and GA API
  - Enable custom user wrappers to intercept these calls
- ARMCI/GA support in TAU
  - On par with support for MPI
  - Available in current stable TAU release
- Performance patterns for ARMCI in SCALASCA
  - Analysis of traces from ARMCI/GA programs
  - Available in an upcoming SCALASCA release
- Consistent naming convention (NGA_)
Restricted Arrays

Create arrays in which only a few processors have data or arrays in which data is distributed to processors in a non-standard way.

```
ga_set_restricted(g_a, list, nproc)
```
Restricted Arrays

4 nodes, 16 processors

Standard data distribution

User-specified distribution
TASCEL-Dynamic Load Balancing

Express computation as collection of tasks
- Tasks operate on data stored in PGAS (Global Arrays)
- Executed in collective task parallel phases

TASCEL runtime system manages task execution
- Load balancing, locality optimization, etc.

Extends Global Arrays’ execution model
Global Pointer Arrays

Create arrays where each array element can be an arbitrary data object
- May be more limited in Fortran where each array object might need to be restricted to an arbitrarily sized array of some type

Access blocks of array elements or single elements and copy them into local buffers using standard put/get syntax

Potential Applications
- Block sparse matrix
- Embedded refined grids
- Recursive data structures
Global Pointer Arrays (cont.)

Pointer Array

Pointer

Array

Data
Global Pointer Arrays (cont.)
Fault Tolerance

Domain Science

Application

Data Redundancy/Fault Recovery Layer

Global Arrays

Non-MPI TCGMSG

Fault Resilient ARMCI

Fault Tolerant Management Infrastructure

Fault Tolerant Barrier

Non-MPI message passing

Network
Fault Tolerance (cont.)

- Exploration of multiple data redundancy models for fault tolerance
- Recent demonstrations of fault tolerance with
  - Global Arrays and ARMCI
- Design and implementation of CCSD(T) using this methodology
  - Ongoing Demonstrations at PNNL booth
- Future ongoing developments for leading platforms
  - Cray and IBM based systems
Exascale Challenges

- Node architecture will change significantly
  - Multiple memory and program spaces
    - Develop GA support for Hybrid Platforms
  - Small amounts of memory per core forces the use of non-SPMD programming/execution models
    - Thread safety - support for multithreaded execution
  - There’s not enough memory (or memory bandwidth) to fully replicate data in private process spaces
    - Distributing GA metadata within nodes
  - Greater portability challenges
    - Refactoring ARMCI
Exascale Challenges

- Much shorter mean time between failures
  - Fault tolerant GA and ARMCI
- Likely traditional SPMD execution will not be feasible
- Programming models with *intrinsic* parallelism will be needed
  - MPI & GA in their current incarnations only have *external* parallelism
- Data consistency will be more of a challenge at extreme scales
Scalability – GA Metadata is a key component

- GA currently allocates metadata for each global array in a replicated manner on each process
- OK for now on petascale systems with $O(10^5)$ processes
  - $200,000 \times 8$ bytes = 1.5 MB per global array instance
  - Not that many global arrays in a typical application

---

**Diagram:**

- **P0**
  - Local global array portion owned by P0

- **P1**
  - Local global array portion owned by P1

Pointers to other processes global array portions

n entries on each process
Scalability – Proposed Metadata Overhead Reduction

- Share metadata between processes on the same shared memory domain (today’s “node”)
- Reduce metadata storage by the number of processes per shared memory domain

Shared Memory Domain

Local global array portion owned by P0

Pointers to global array portions

Local global array portion owned by P1
Summary

- **Global Arrays** supports a global address space
  - Easy mapping between distributed data and original problem formulation

- **One-sided communication**
  - No need to coordinate between sender and receiver
  - Random access patterns are easily programmed
    - Load balancing

- **High Performance**
  - Demonstrated scalability to 200K+ cores and greater than 1 Petaflop performance

- **High programmer productivity**
  - Global address space and one-sided communication eliminate many programming overheads
Thanks

- The Steering Committee of IPDPS
  - In particular Alan Sussman (Program Chair)
- DOE Office of Advanced Scientific and Computing Research
- PNNL Extreme Scale Computing Initiative
Discussion