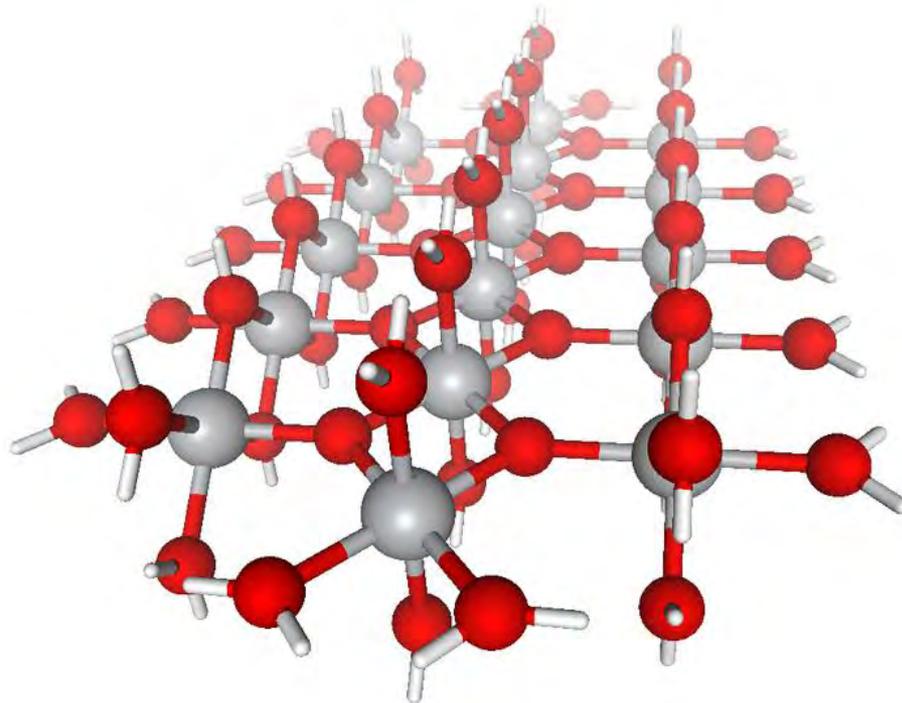


# Overview of the Global Arrays Parallel Software Development Toolkit

Bruce Palmer, Manoj Kumar Krishnan, Sriram Krishnamoorthy,  
Ahbinav Vishnu, Jeff Daily, Daniel Chavarria, Patrick Nichols

# Outline

- ▶ Programming Model
- ▶ Basic Functions in GA
- ▶ Advanced Functionality
- ▶ Summary

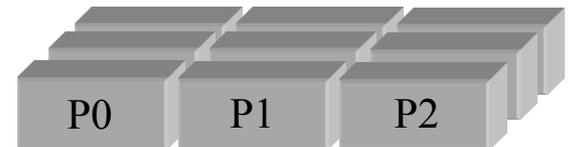
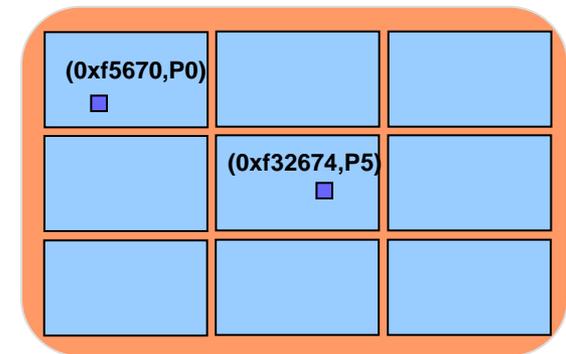


# Distributed Data vs Shared Memory

## 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)

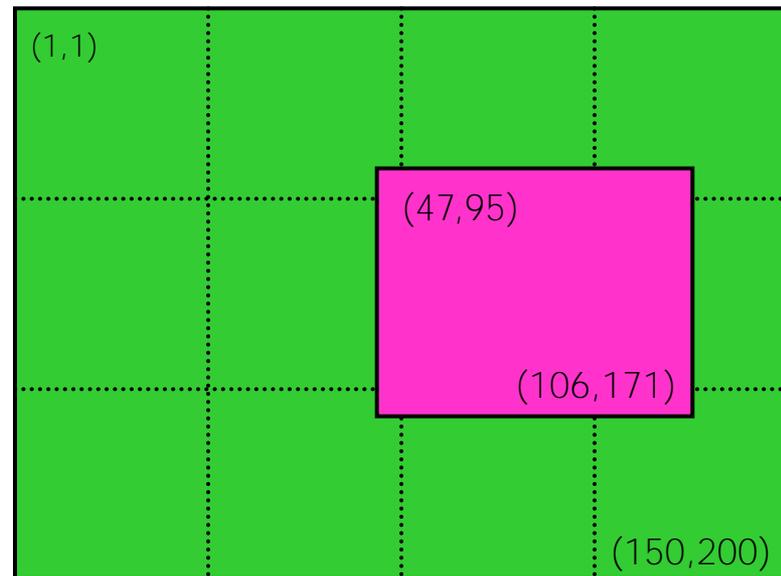


# Distributed Data vs Shared Memory (Cont).

## Shared Memory:

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

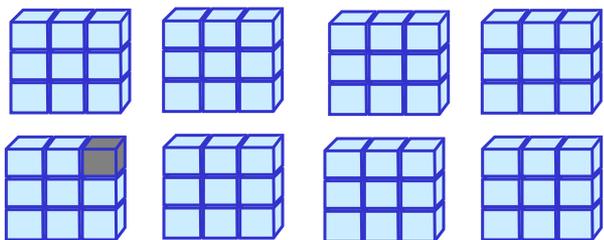
Data is mapped out in a natural manner (usually corresponding to the original problem) and access is easy. Information on data locality is obscured and leads to loss of performance.



# Global Arrays

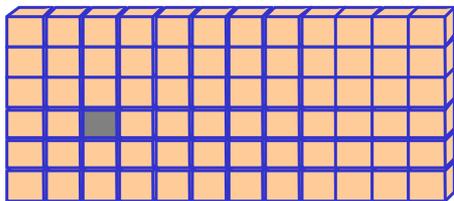
Distributed dense arrays that can be accessed through a shared memory-like style

Physically distributed data



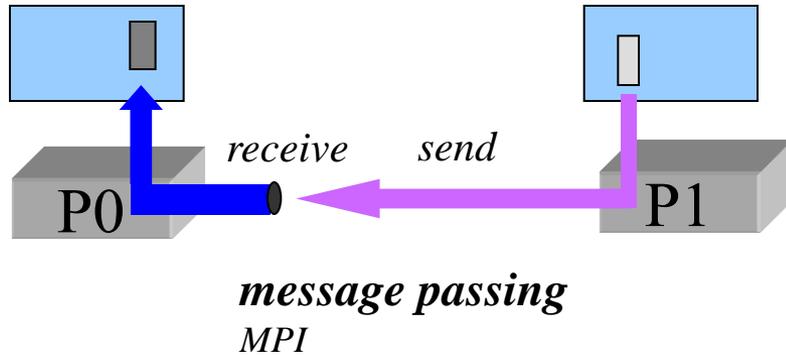
single, shared data structure/  
global indexing

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



Global Address Space

# One-sided Communication

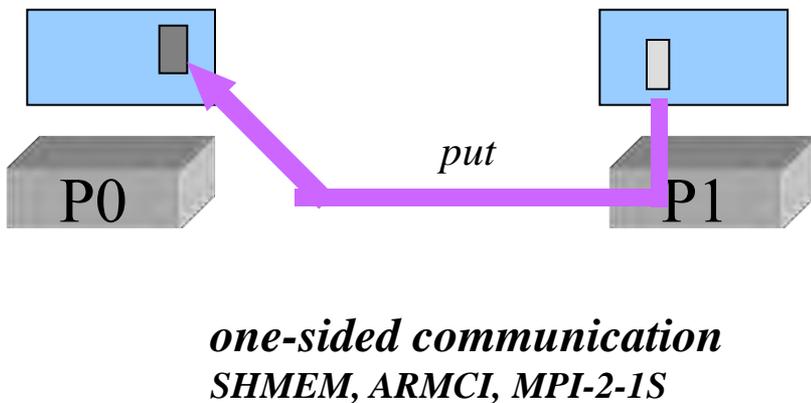


## Message Passing:

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.



# Global Arrays Approach to Parallel Programming

- ▶ Uses a one-sided communication model
- ▶ One-sided communication supports the creation of a Partitioned Global Address Space (PGAS) programming model
  - Allows developers to access data using a global index instead of supplying index transformations that transform between the original problem space and the location of individual blocks of data described in the (processor,local index) space
  - Simplifies programming enormously in many cases

# One-sided 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
- Data consistency must be explicitly managed

# Global Arrays

- ▶ Shared memory 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

# GA Core Capabilities

- ▶ Distributed array library
  - dense arrays 1-7 dimensions
  - four data types: *integer*, *real*, *double precision*, *double complex*
  - global rather than per-task view of data structures
  - user control over data distribution: regular and irregular
- ▶ Collective and shared-memory style operations
  - `ga_sync`, `ga_scale`, etc
  - `ga_put`, `ga_get`, `ga_acc`
  - nonblocking `ga_put`, `ga_get`, `ga_acc`
- ▶ Interfaces to third party parallel numerical libraries
  - PeiGS, Scalapack, SUMMA, Tao
    - example: to solve a linear system using LU factorization

```
call ga_lu_solve(g_a, g_b)
```

instead of

```
call pdgetrf(n,m, locA, p, q, dA, ind, info)
call pdgetrs(trans, n, mb, locA, p, q, dA,dB,info)
```

# Interoperability and Interfaces

- ▶ Language interfaces to Fortran, C, C++, Python
- ▶ Interoperability with MPI and MPI libraries
  - e.g., PETSC, CUMULVS
- ▶ Explicit interfaces to other systems that expand functionality of GA
  - ScaLAPACK-scalable linear algebra software
  - Peigs-parallel eigensolvers
  - TAO-advanced optimization package

# Structure of GA

Application programming language interface

Fortran

C

C++

Python

Global Arrays and MPI are completely interoperable. Code can contain calls to both libraries.

distributed arrays layer  
*memory management, index translation*

MPI  
*Global operations*

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

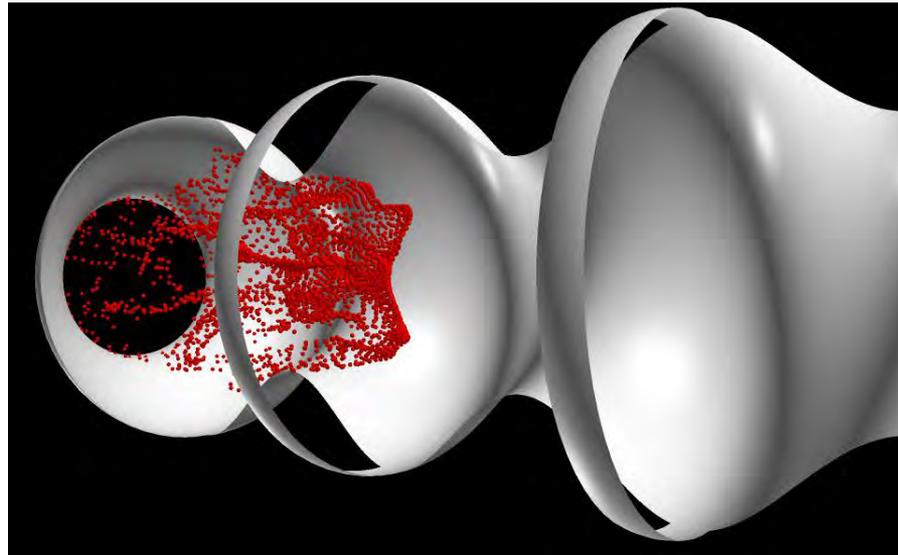
system specific interfaces  
*LAPI, GM/Myrinet, threads, VIA,...*

# 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

# Outline

- ▶ Programming Model
- ▶ Basic Functions in GA
- ▶ Advanced Functionality
- ▶ Summary



# Basic GA Operations

- ▶ GA programming model is very simple.
- ▶ Most of a parallel program can be written with these basic calls
  - **GA\_Initialize, GA\_Terminate**
  - **GA\_Nnodes, GA\_Nodeid**
  - **GA\_Create, GA\_Destroy**
  - **GA\_Put, GA\_Get**
  - **GA\_Sync**
  - **GA\_Distribution**

```
subroutine ga_initialize()
subroutine ga_terminate()

integer function ga_nnodes()
integer function ga_nodeid()

logical function nga_create(type,dim,dims,name,chunk,g_a)
logical function ga_destroy(g_a)

subroutine nga_put(g_a, lo, hi, buf, ld)
subroutine nga_get(g_a, lo, hi, buf, ld)

subroutine ga_sync()

subroutine nga_distribution(g_a, iproc, lo, hi)
subroutine nga_access(g_a, lo, hi, index, ld)
```

# GA Initialization/Termination

- ▶ Initialize GA after initializing message-passing library and terminate GA before terminating message-passing library
  - GA\_Initialize()
  - GA\_Initialize\_ltd(size)
  - GA\_Terminate()

```
program main
#include "mafdecls.h"
#include "global.fh"
integer ierr

c
call mpi_init(ierr)
call ga_initialize()

c
write(6,*) 'Hello world'

c
call ga_terminate()
call mpi_finilize()
end
```

# Parallel Environment - Process Information

## ► Parallel Environment:

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

```
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'
```

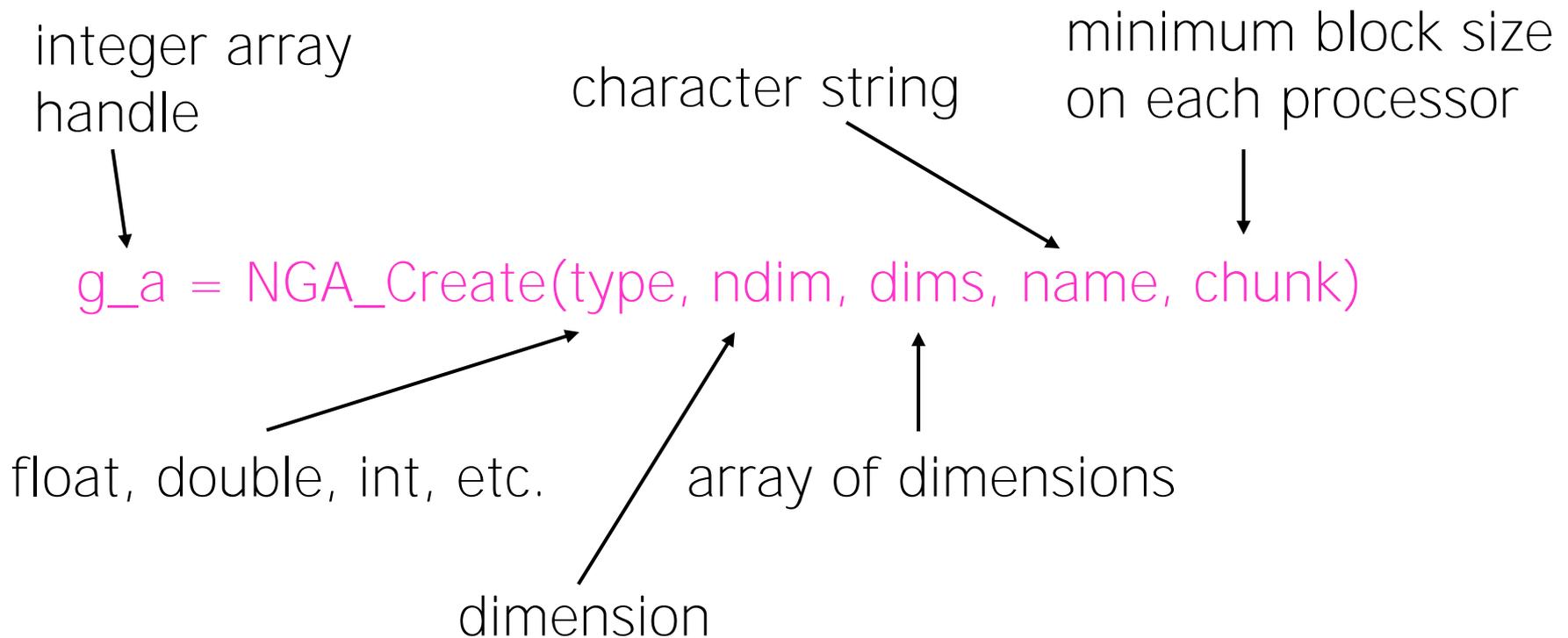
```
call ga_terminate()
call mpi_finalize()
end
```

```
$ 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
```



# Creating Global Arrays



# New Interface for Creating Global Arrays

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

```
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, nblock)  
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)
```

# Remote Data Access in GA

## Message Passing:

identify size and location of data blocks

loop over processors:

if (me = P<sub>N</sub>) then

pack data in local message buffer

send block of data to message buffer on P<sub>0</sub>

else if (me = P<sub>0</sub>) then

receive block of data from P<sub>N</sub> in message buffer

unpack data from message buffer to local buffer

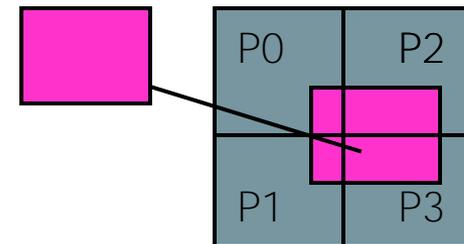
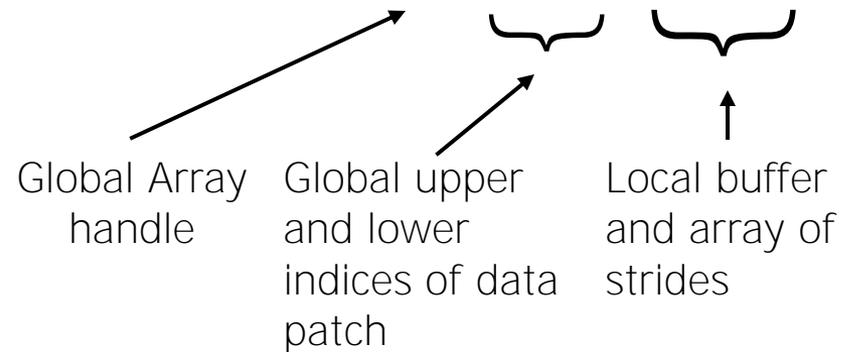
endif

end loop

copy local data on P<sub>0</sub> to local buffer

## Global Arrays:

**NGA\_Get(g\_a, lo, hi, buffer, Id);**



# Data Locality

What data does a processor own?

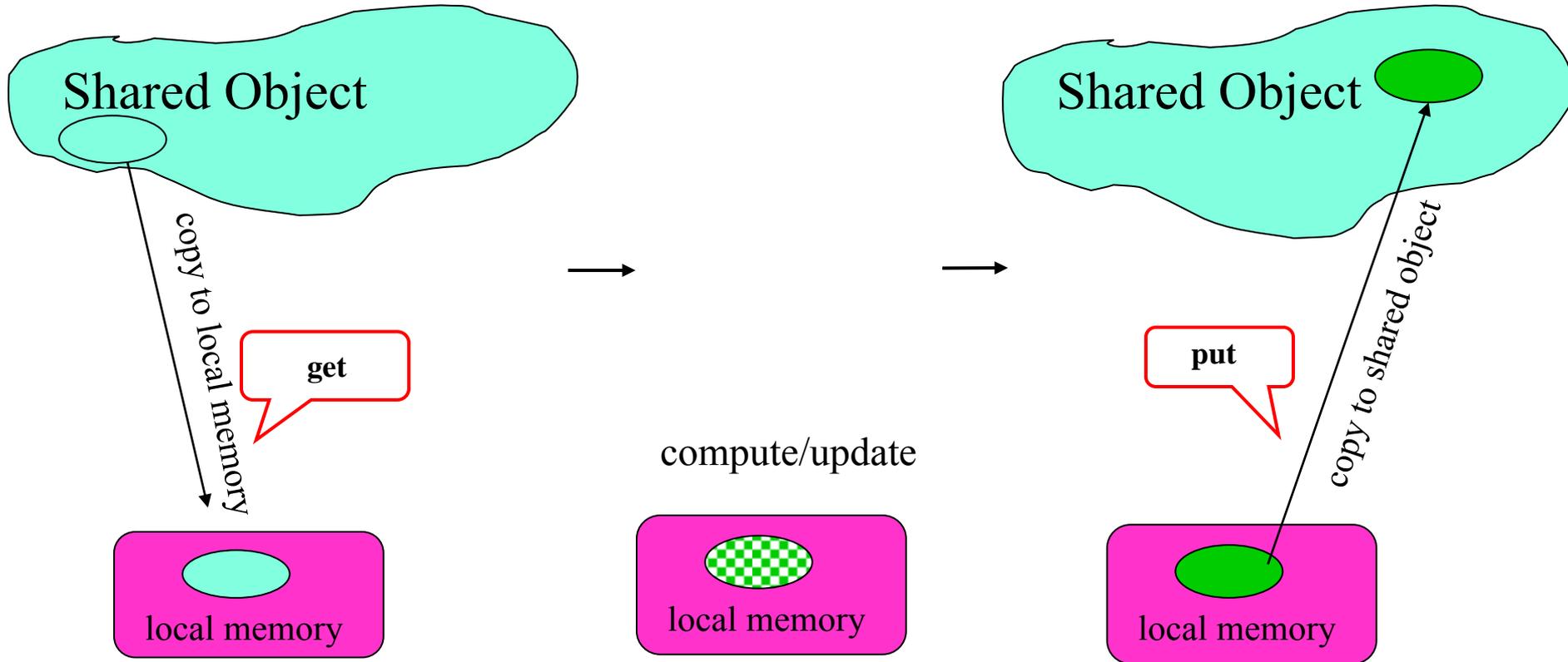
```
NGA_Distribution(g_a, iproc, lo, hi);
```

Where is the data?

```
NGA_Access(g_a, lo, hi, ptr, ld);
```

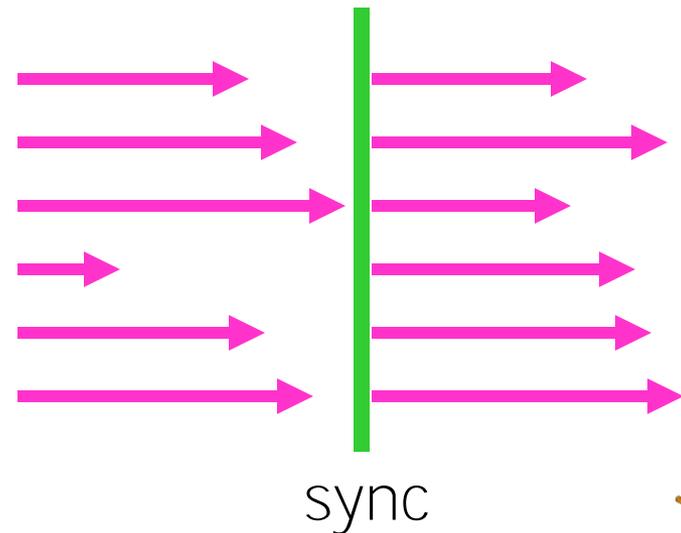
Use this information to organize calculation so that maximum use is made of locally held data

# Global Array Model of Computations

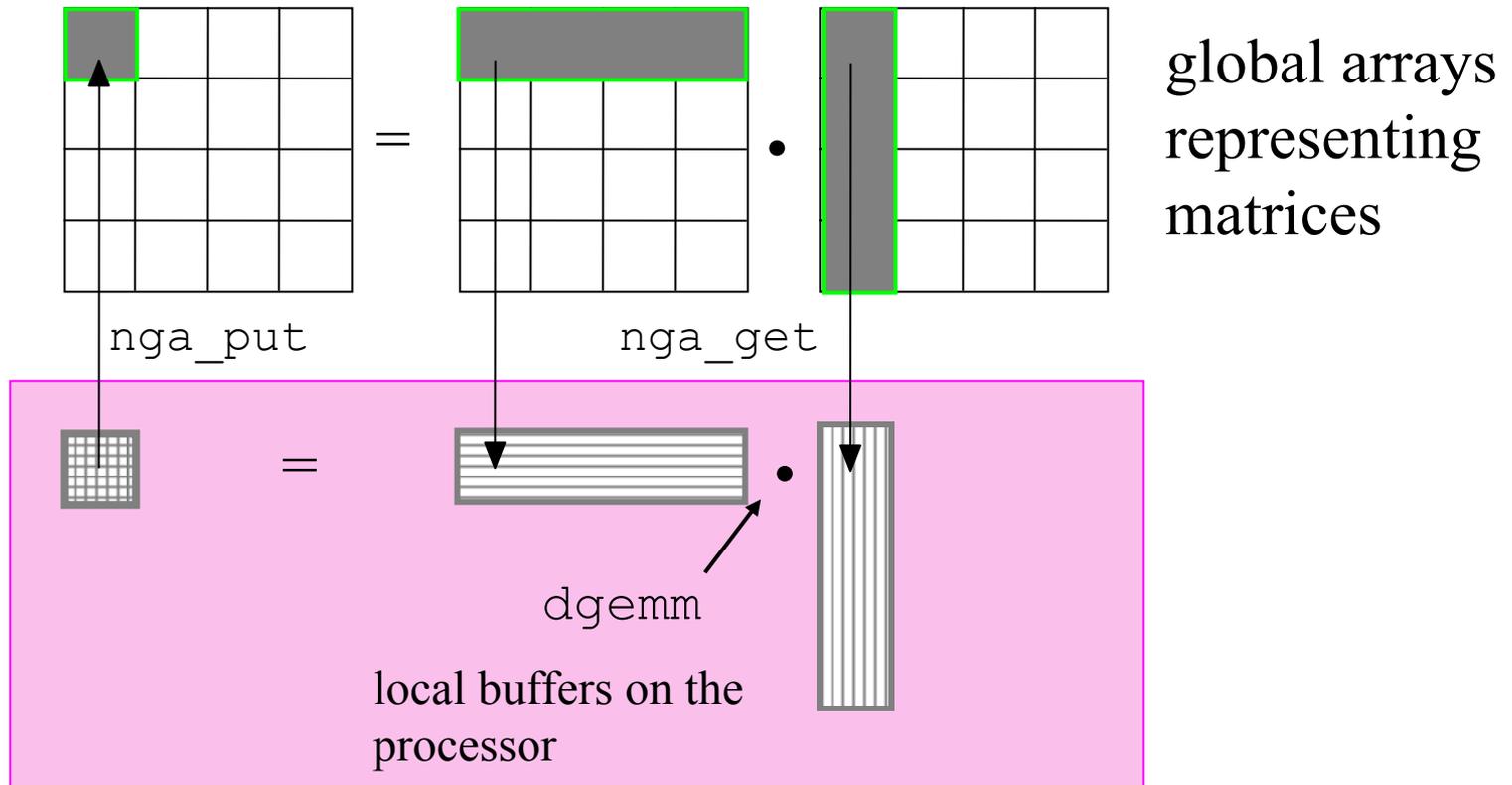


# Sync

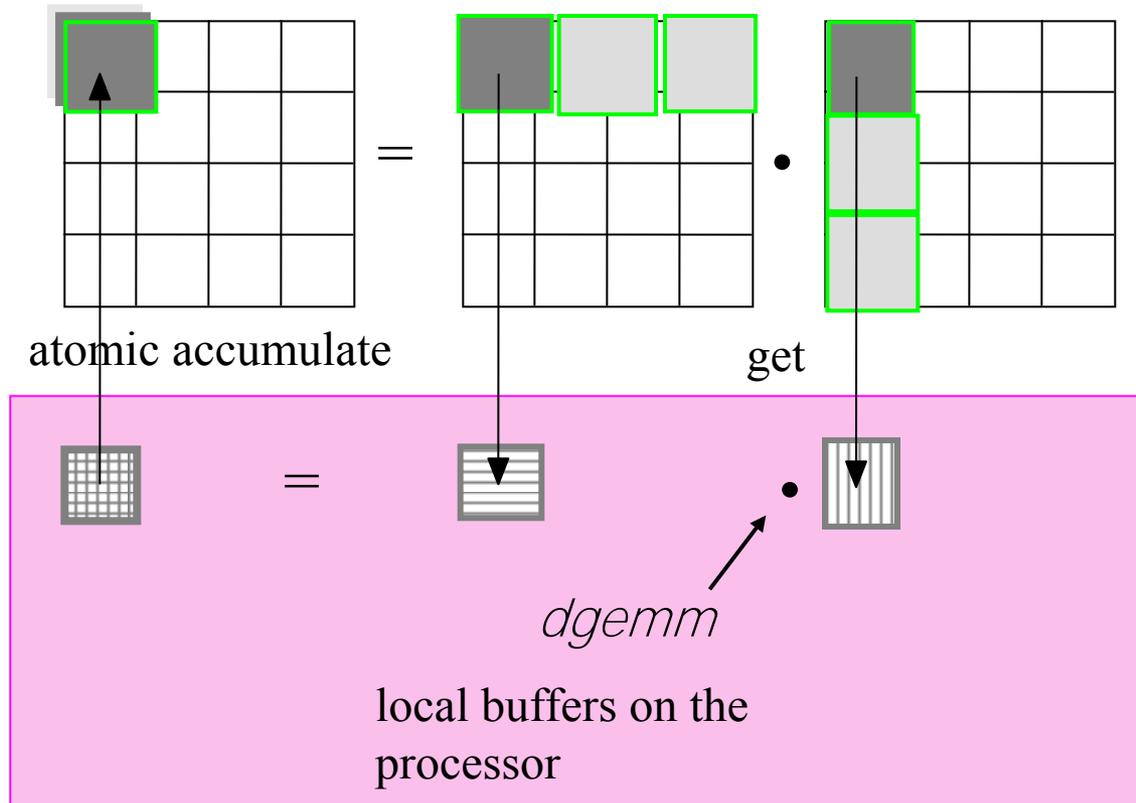
- ▶ GA\_Sync is a collective operation
- ▶ It acts as a barrier, which synchronizes all the processes and also ensures that all outstanding communication operations are completed



# Example: Matrix Multiply



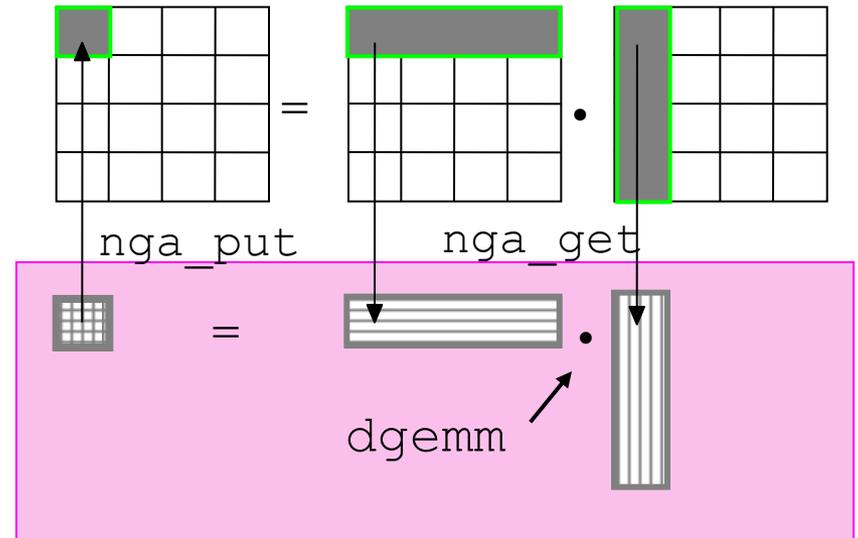
# Matrix Multiply (a better version)



**more scalable!**  
(less memory,  
higher parallelism)

# 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);
```



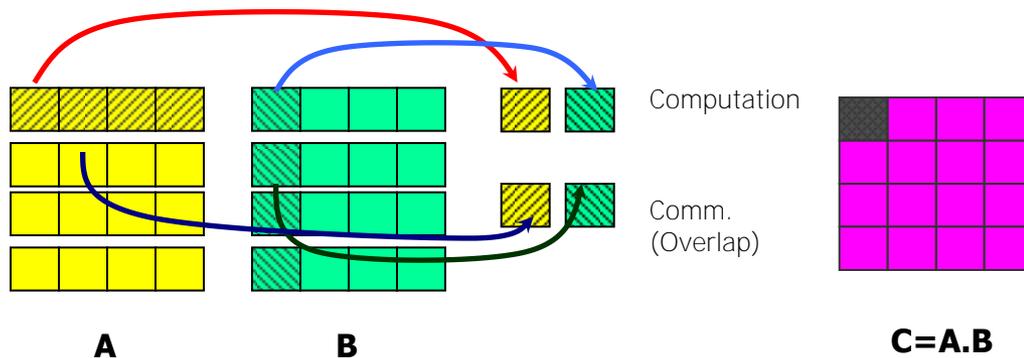
# Non-Blocking Communication

- ▶ Allows overlapping of data transfers and computations
  - Technique for latency hiding
- ▶ Nonblocking operations initiate a communication call and then return control to the application immediately
- ▶ operation completed locally by making a call to the *wait* routine

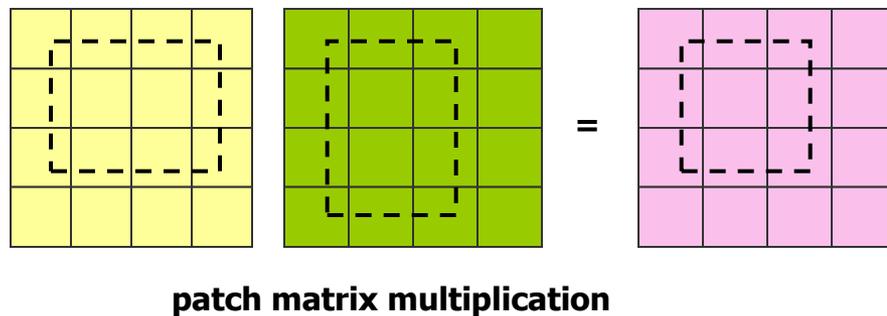
```
NGA_Nbget(g_a, lo, hi, buf, ld, nbhandle)
```

```
NGA_Nbwait(nbhandle)
```

# SUMMA Matrix Multiplication



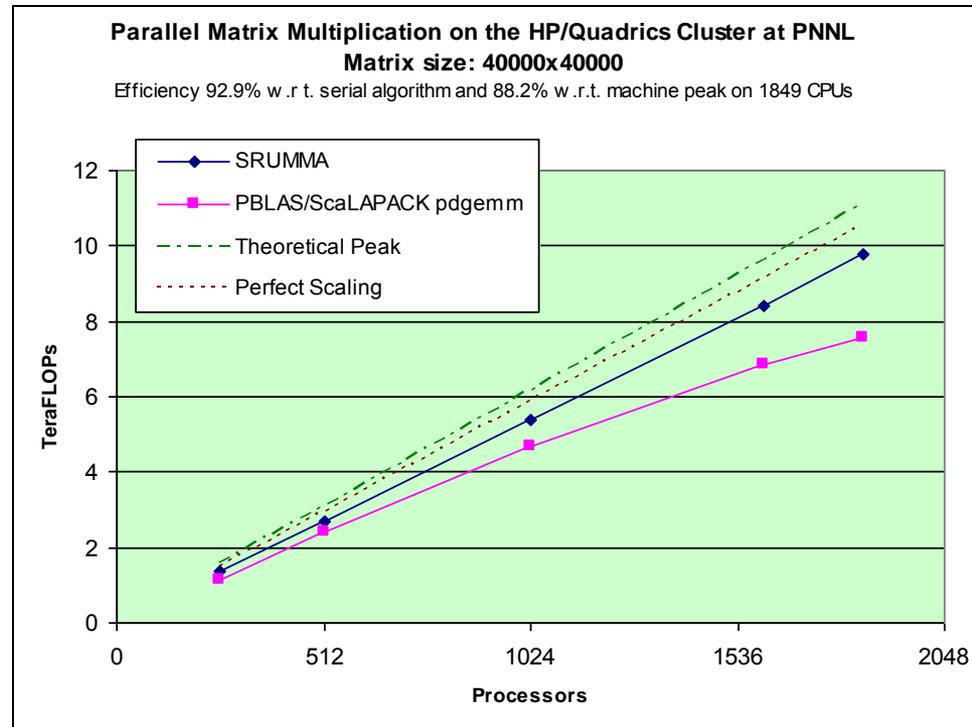
Issue NB Get A and B blocks  
**do** (until last chunk)  
 issue NB Get to the next blocks  
 wait for previous issued call  
 compute  $A*B$  (sequential dgemm)  
 NB atomic accumulate into "C"  
 matrix  
**done**



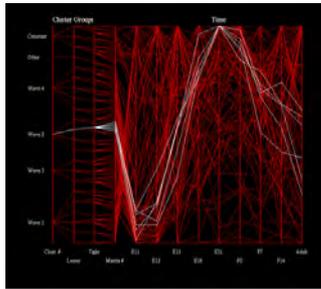
**Advantages:**

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

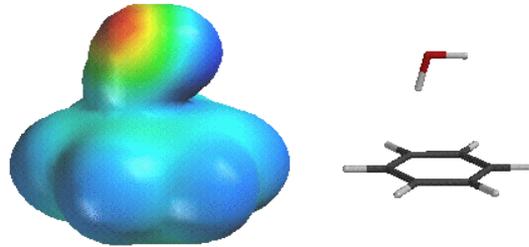
# SUMMA Matrix Multiplication: Improvement over PBLAS/ScaLAPACK



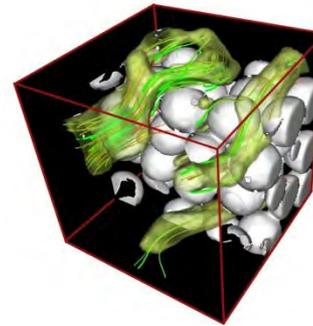
# Application Areas



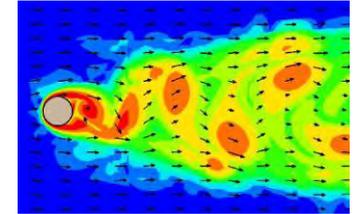
bioinformatics



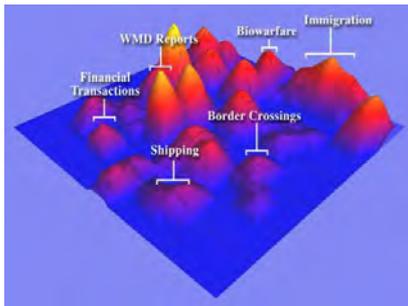
electronic structure chemistry  
GA is the standard programming model



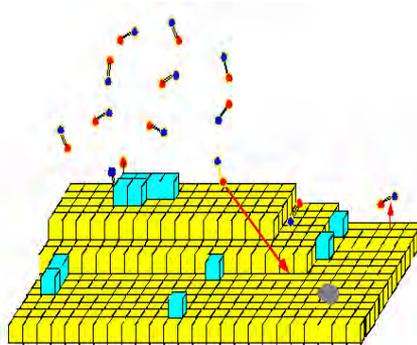
smoothed particle hydrodynamics



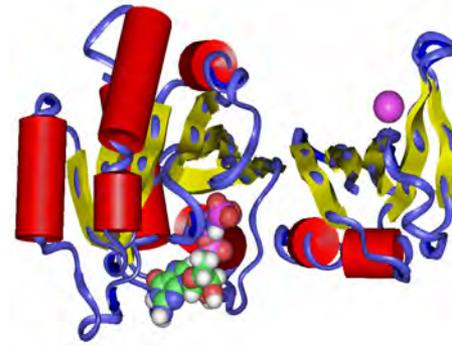
fluid dynamics



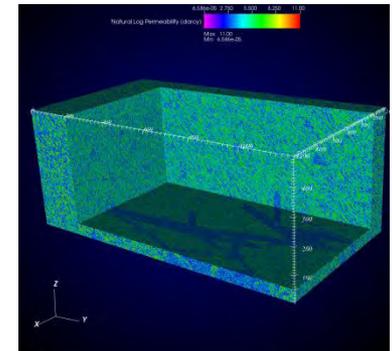
visual analytics



material sciences



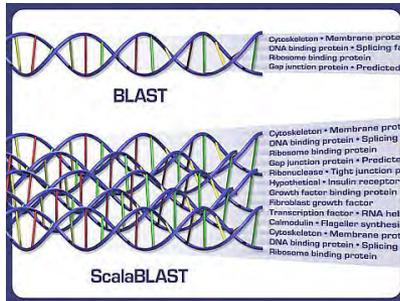
molecular dynamics



hydrology

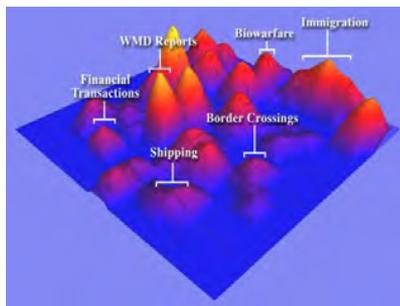
Others: financial security forecasting, astrophysics, climate analysis

# Recent Applications



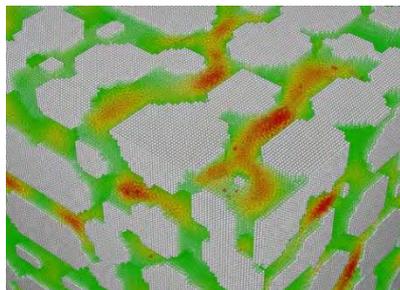
## ScalaBLAST

C. Oehmen and J. Nieplocha. ScalaBLAST: "A scalable implementation of BLAST for high performance data-intensive bioinformatics analysis." IEEE Trans. Parallel Distributed Systems, Vol. 17, No. 8, 2006



## Parallel Inspire

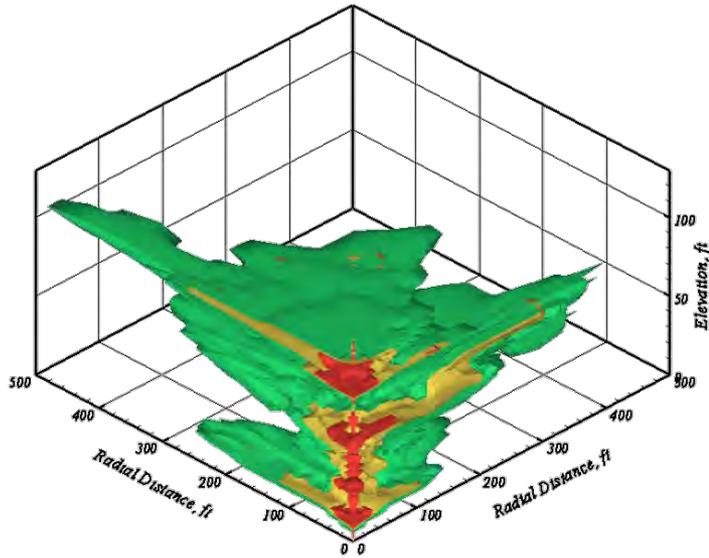
Krishnan M, SJ Bohn, WE Cowley, VL Crow, and J Nieplocha, "Scalable Visual Analytics of Massive Textual Datasets", Proc. IEEE International Parallel and Distributed Processing Symposium, 2007.



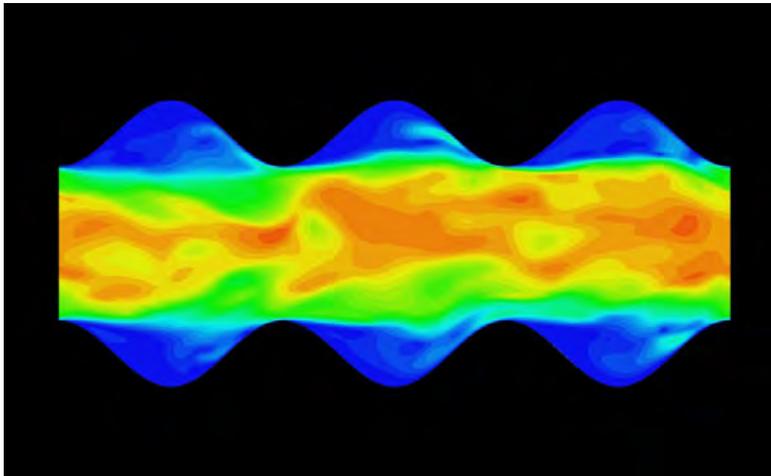
## Smooth Particle Hydrodynamics

B. Palmer, V. Gurumoorthi, A. Tartakovsky, T. Scheibe, A Component-Based Framework for Smoothed Particle Hydrodynamics Simulations of Reactive Fluid Flow in Porous Media", Int. J. High Perf. Comput. App., Vol 24, 2010

# Recent Applications



Subsurface Transport Over Multiple Phases: STOMP

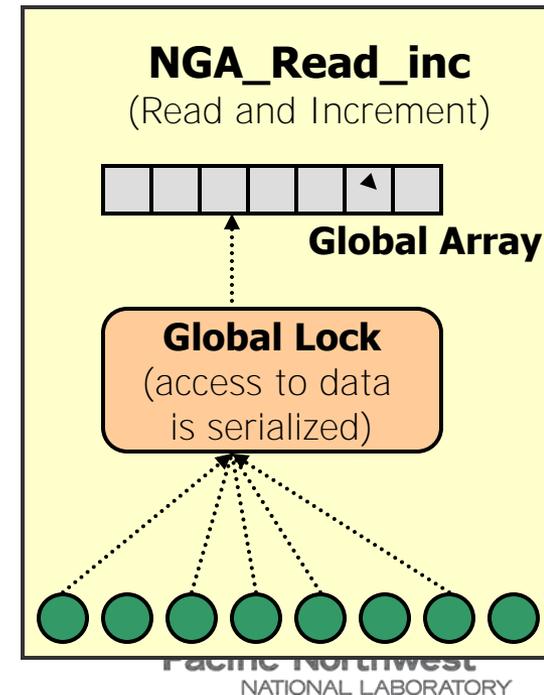


Transient Energy Transport  
Hydrodynamics Simulator: TETHYS

# Read and Increment

- ▶ **nga\_read\_inc**: remotely updates a particular element in an integer global array and returns the original value:
  - Applies to integer arrays only
  - Can be used as a global counter for dynamic load balancing

```
c Create task counter
  call 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 ...
```

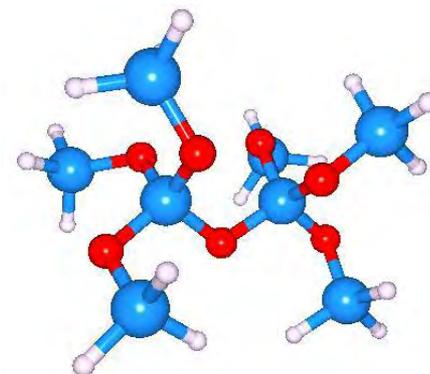


# Hartree-Fock SCF

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

$$F_{\mu\nu} C_{k\nu} = \epsilon D_{\mu\nu} C_{k\nu}$$

$$D_{\mu\nu} = \sum_k C_{\mu k} C_{\nu k}$$

$$F_{\mu\nu} = h_{\mu\nu} + \frac{1}{2} \sum_{\omega\lambda} [2(\mu\nu | \omega\lambda) - (\mu\omega | \nu\lambda)] D_{\omega\lambda}$$

# 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

467



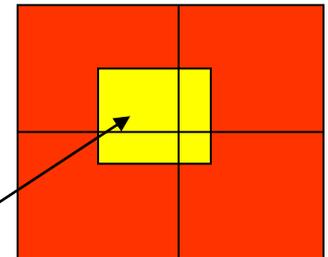
Read and  
increment  
counter

```
do i
do j
do k
do l
  F(i, j) = ..
```

Evaluate  
block

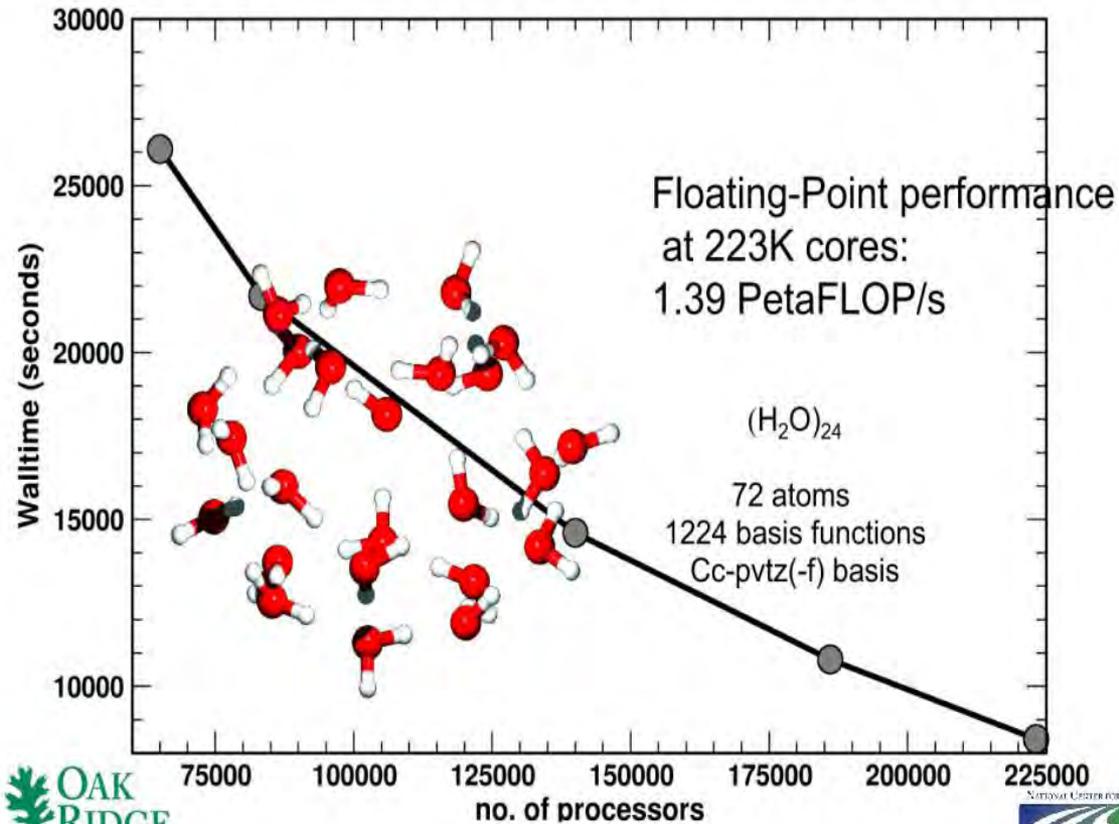


Accumulate  
results



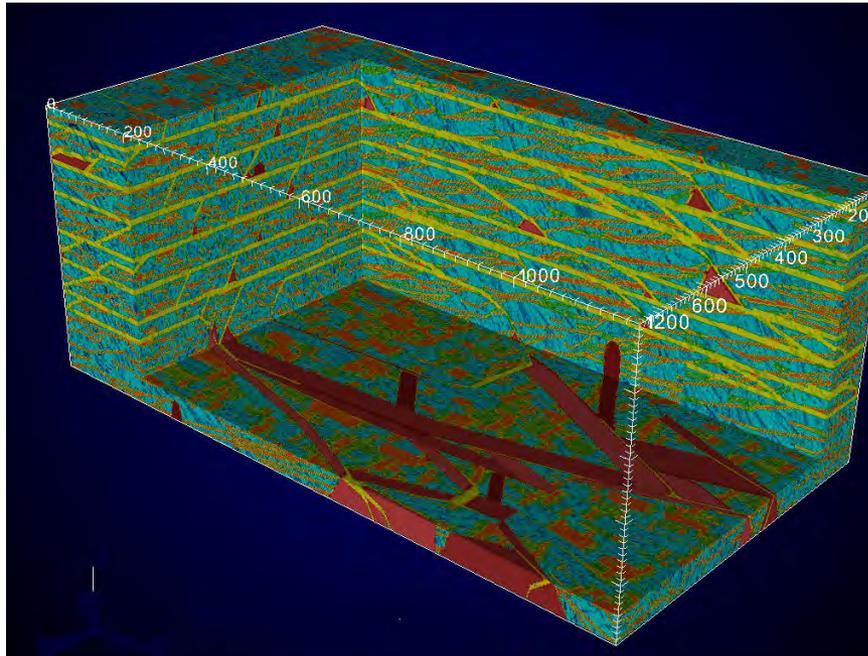
# Gorden Bell finalist at SC09 - GA Crosses the Petaflop Barrier

- ▶ GA-based parallel implementation of coupled cluster calculation performed at **1.39 petaflops using over 223,000 processes** on ORNL's Jaguar petaflop system
  - Apra et. al., "*Liquid water: obtaining the right answer for the right reasons*", SC 2009.
- ▶ Global Arrays is one of two programming models that have achieved this level of performance



# Outline

- ▶ Programming Model
- ▶ Basic Functions in GA
- ▶ Advanced Functionality
- ▶ Summary



# Global Array Processor Groups

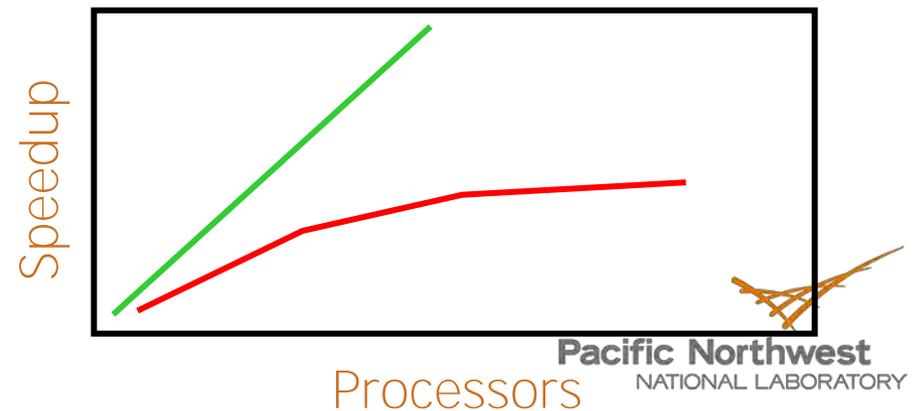
Many parallel applications can potentially make use of groups. These include

- Numerical evaluation of gradients
- Monte Carlo sampling over initial conditions or uncertain parameter sets
- Free energy perturbation calculations (chemistry)
- Nudged elastic band calculations (chemistry and materials science)
- Sparse matrix-vector operations (NAS CG benchmark)
- Data layout for partially structured grids
- Multi-physics applications

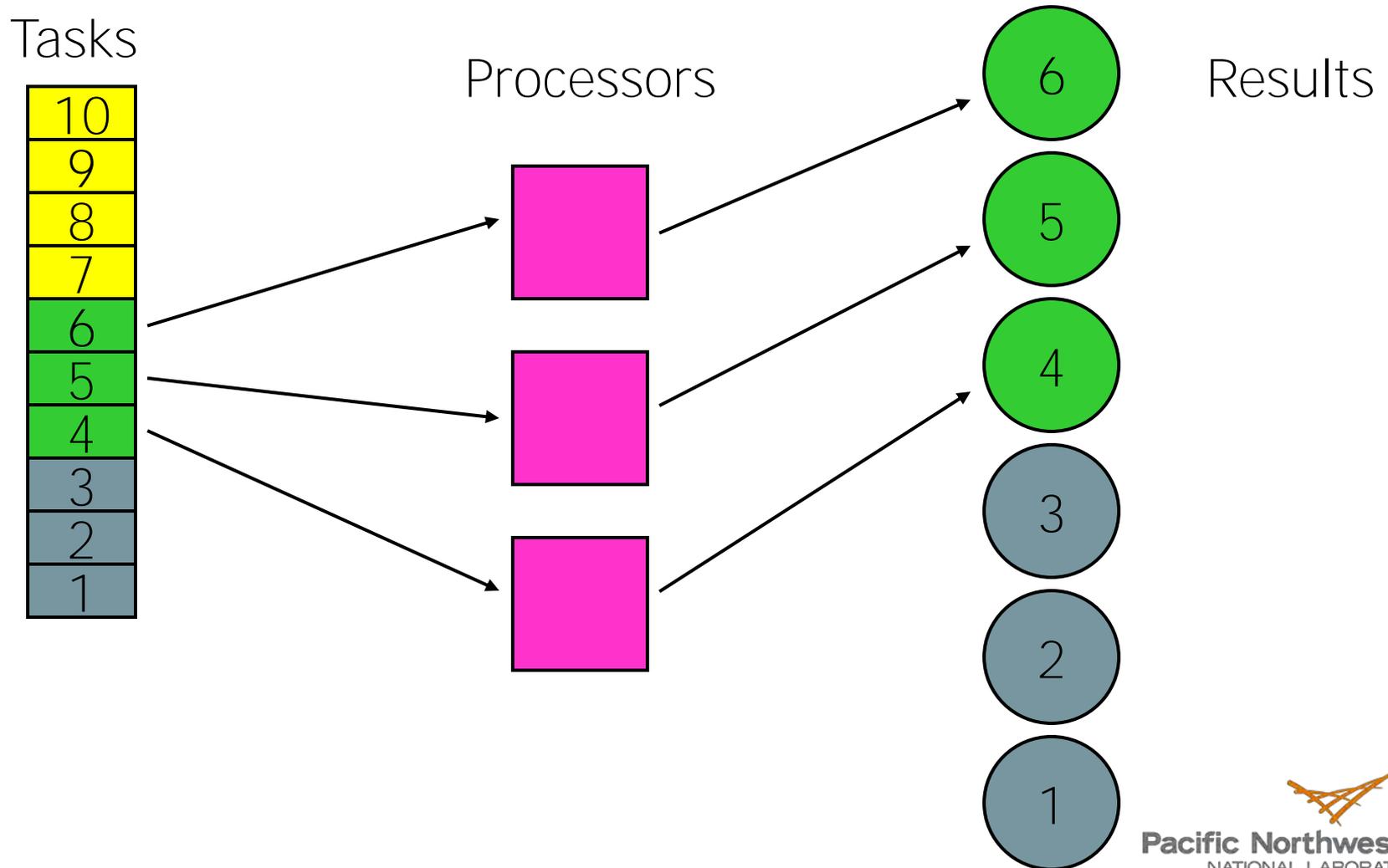
# Global Array Processor Groups

If the individual calculations are small enough then each processor can be used to execute one of the tasks (embarrassingly parallel algorithms).

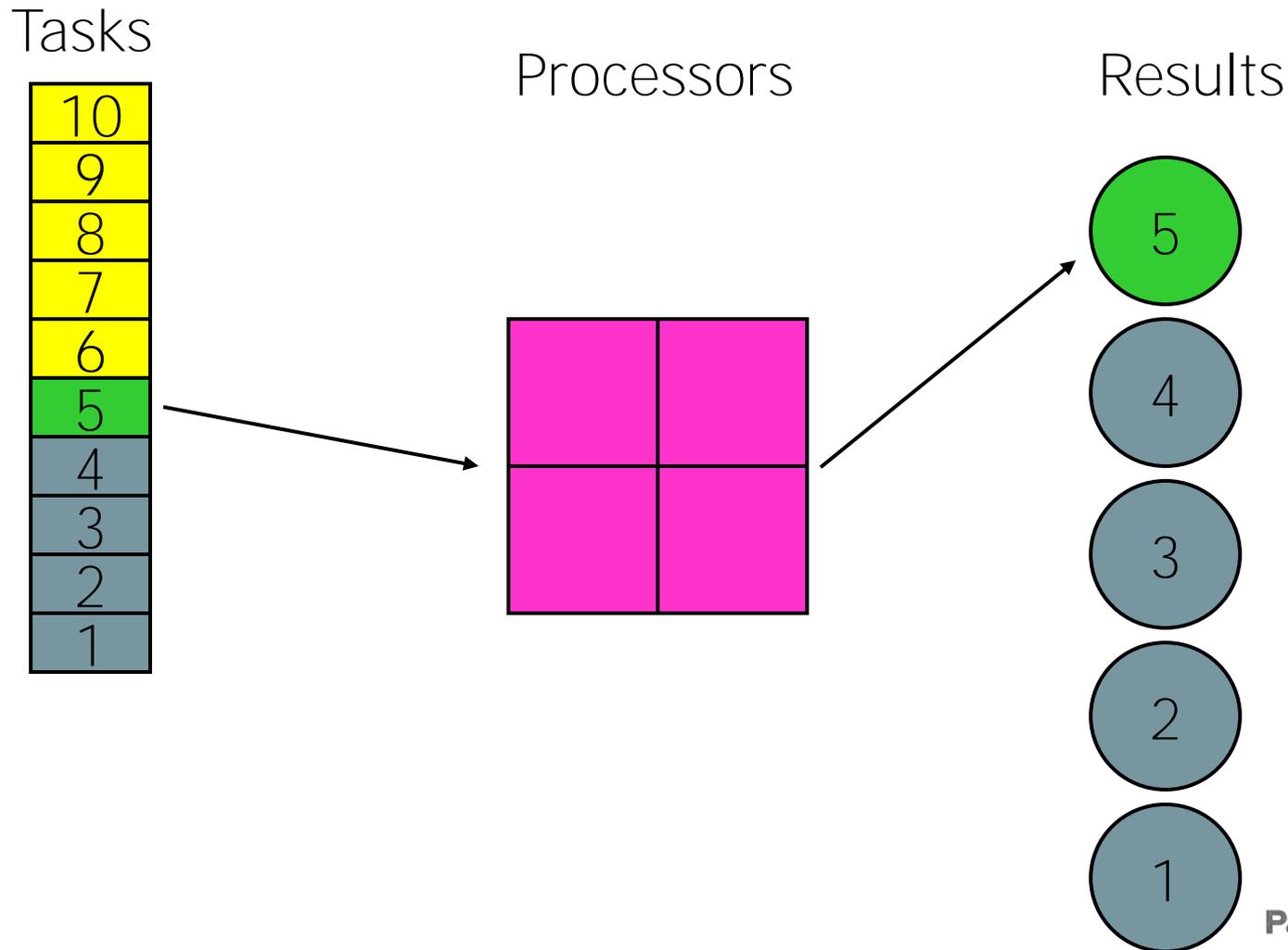
If the individual tasks are large enough that they must be distributed amongst several processors then the only option (usually) is to run each task sequentially on multiple processors. This limits the total number of processors that can be applied to the problem since parallel efficiency degrades as the number of processors increases.



# Multiple Tasks with Groups



# Multiple Tasks with Groups

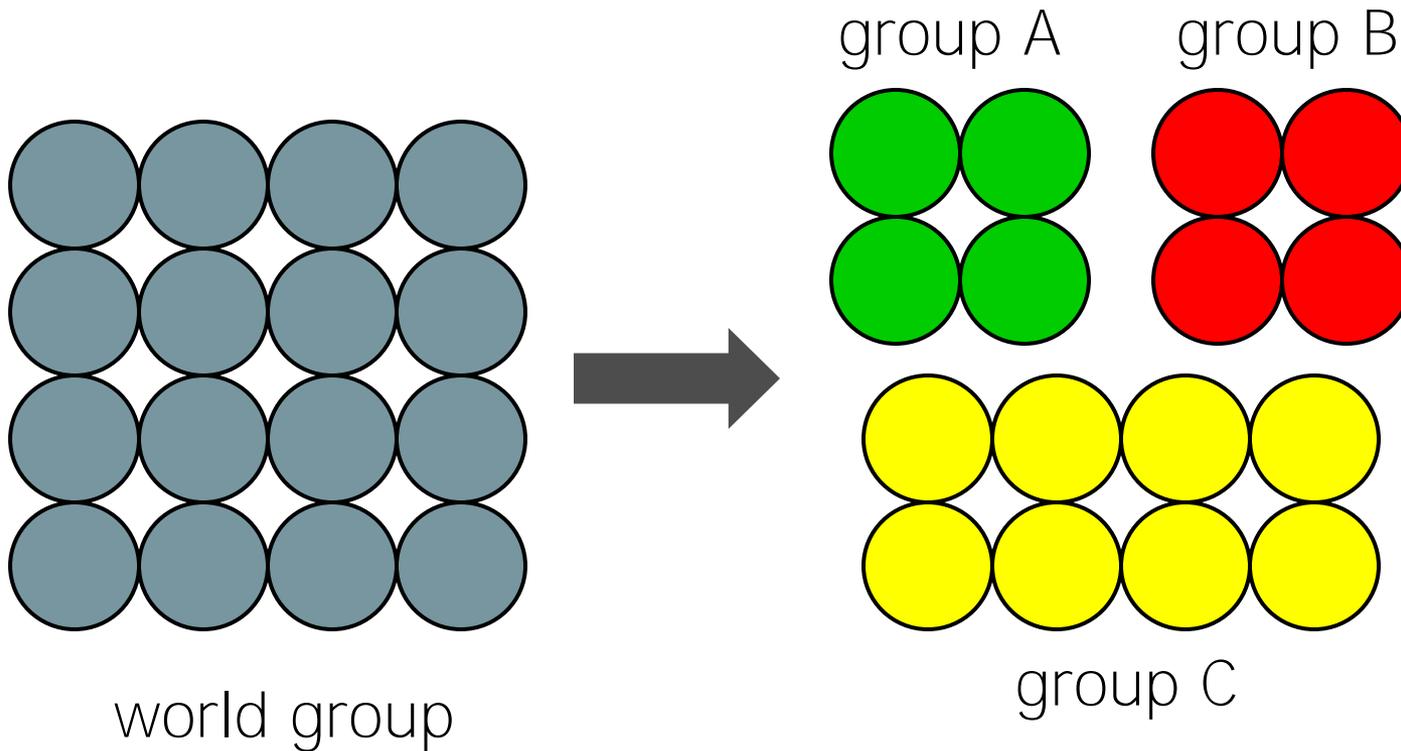


# Global Array Processor Groups

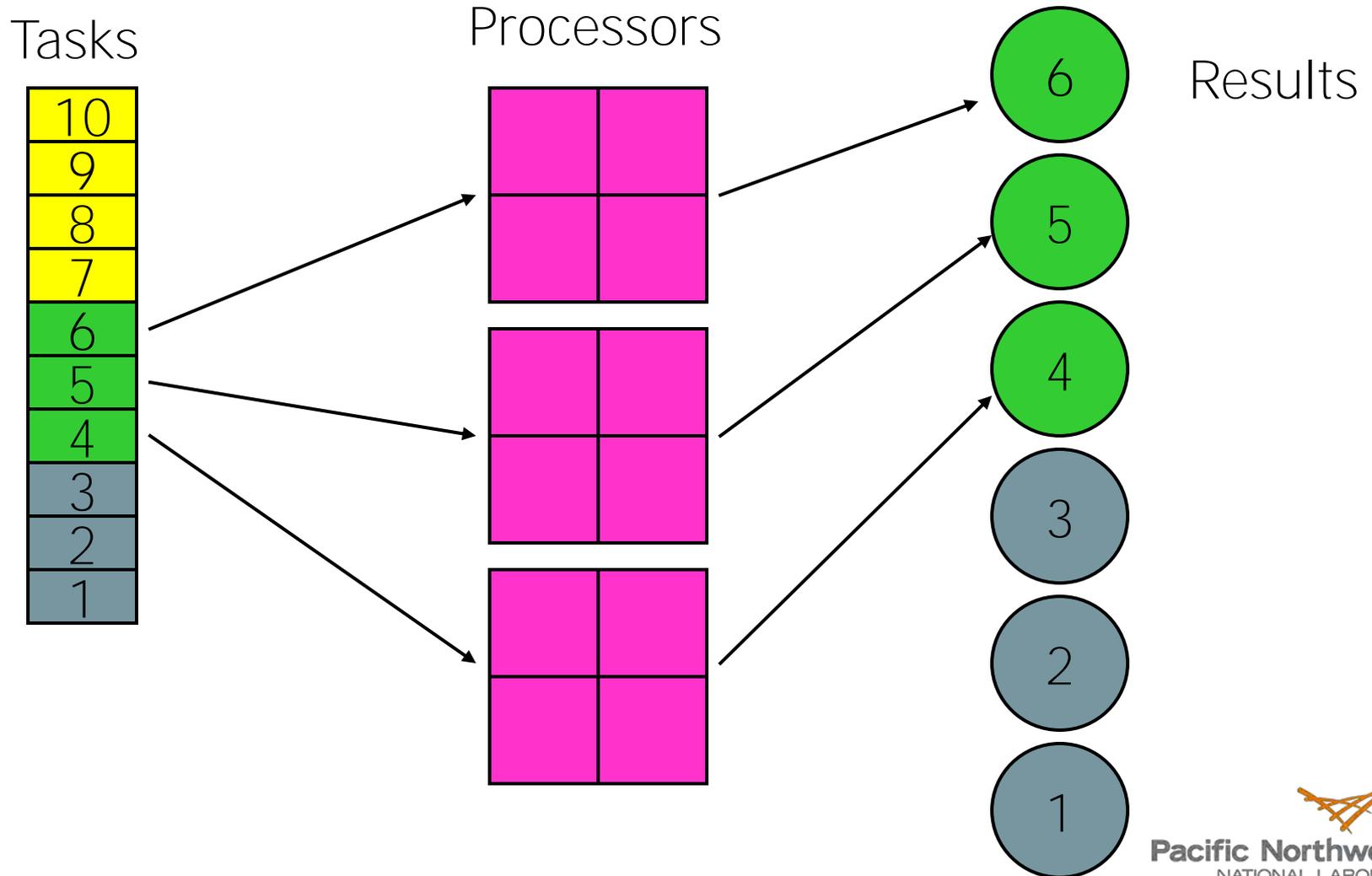
Alternatively the collection of processors can be decomposed into processor groups. These processor groups can be used to execute parallel algorithms *independently* of one another. This requires

- global operations that are restricted in scope to a particular group instead of over the entire domain of processors (world group)
- distributed data structures that are restricted to a particular group

# Processor Groups (Schematic)



# Multiple Tasks with Groups



# Creating Processor Groups

**integer function ga\_pgroup\_create(list, count)**

Returns a handle to a group of processors. The total number of processors is count, the individual processor IDs are located in the array list.

**subroutine ga\_pgroup\_set\_default(p\_grp)**

Set the default processor to p\_grp. All arrays created after this point are created on the default processor group, all global operations are restricted to the default processor group unless explicit directives are used. Initial value of the default processor group is the world group.

# Explicit Operations on Groups

## Explicit Global Operations on Groups

```
ga_pgroup_sync(p_grp)
ga_pgroup_brdcst(p_grp, type, buf, lenbuf, root)
ga_pgroup_igop(p_grp, type, buf, lenbuf, op)
ga_pgroup_dgop(p_grp, type, buf, lenbuf, op)
```

## Query Operations on Groups

```
ga_pgroup_nnodes(p_grp)
ga_pgroup_nodeid(p_grp)
```

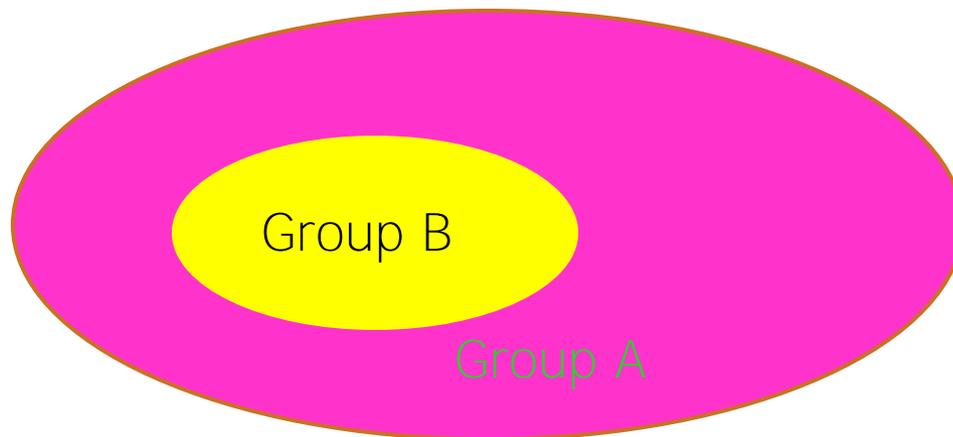
## Access Functions

```
integer function ga_pgroup_get_default()
integer function ga_pgroup_get_world()
```

# Communication between Groups

Copy and copy\_patch operations are supported for global arrays that are created on different groups. One of the groups must be completely contained in the other (nested).

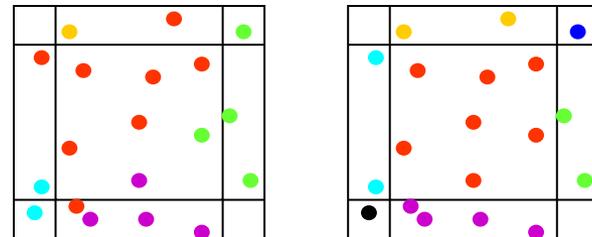
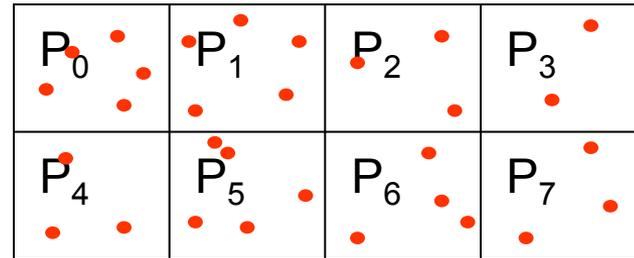
The copy or copy\_patch operation must be executed by all processors on the nested group (group B in illustration)



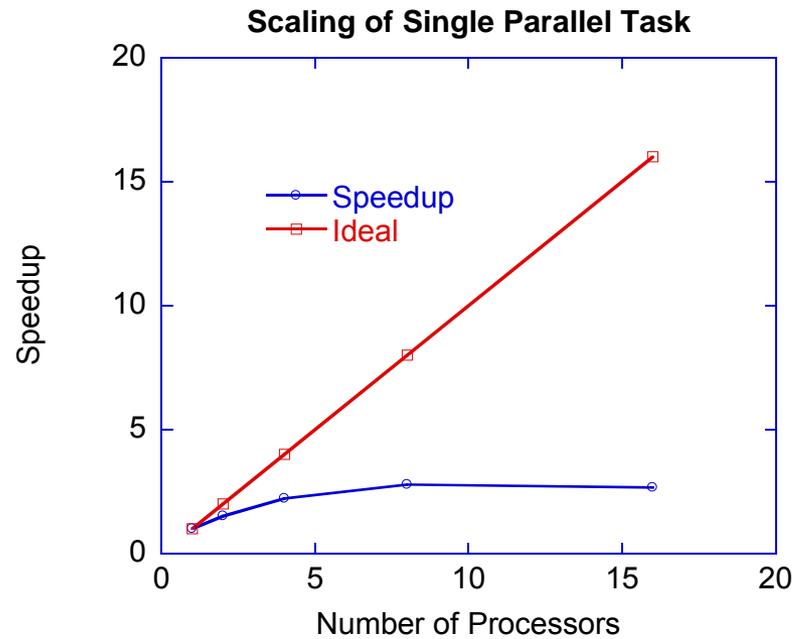
# MD Example

Spatial Decomposition Algorithm:

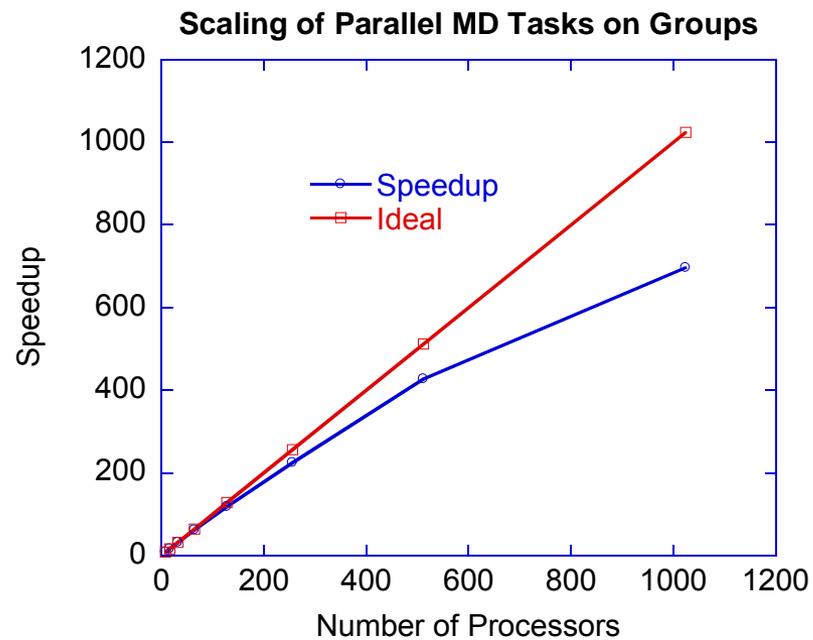
- Partition particles among processors
- Update coordinates at every step
- Update partitioning after fixed number of steps



# MD Parallel Scaling



# MD Performance on Groups



# Sparse Data Manipulation

## ▶ ga\_scan\_add

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

## ▶ ga\_scan\_copy

```
g_mask:  1  0  0  0  0  1  0  1  0  0  1  0  0  0  1  1  0
g_src:   5  8  7  3  2  6  9  7  3  4  8  2  3  6  9 10  7
g_dest:  5  5  5  5  5  6  6  7  7  7  8  8  8  8  9 10 10
```

# Sparse Data Manipulation

## ▶ ga\_pack

```
g_mask:  1  0  0  0  0  1  0  1  0  0  1  0  0  0  1  0  0
g_src:   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
g_dest:  1  6  8 11 15
```

## ▶ ga\_unpack

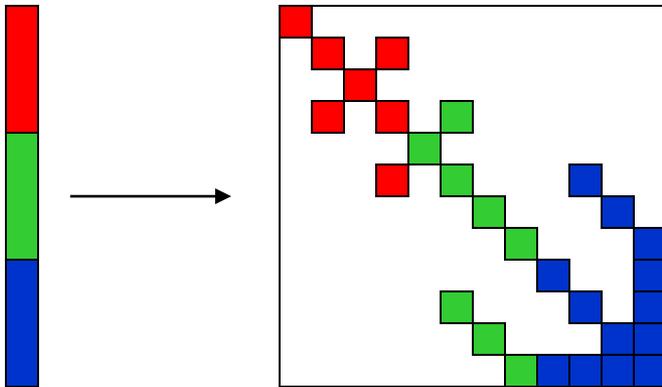
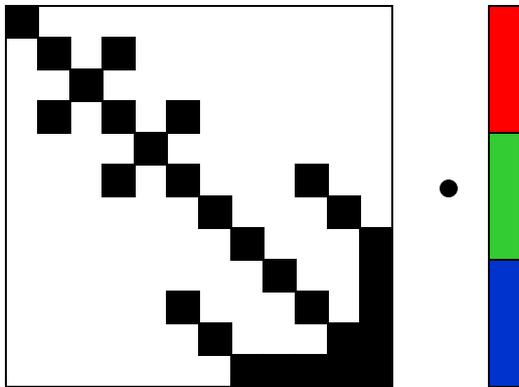
```
g_mask:  1  0  0  0  0  1  0  1  0  0  1  0  0  0  1  0  0
g_src:   1  6  8 11 15
g_dest:  1  0  0  0  0  6  0  8  0  0 11  0  0  0 15 0  0
```

# Compressed Sparse Row Matrix

$$\begin{bmatrix} 0 & 0 & 1 & 3 & 0 \\ 2 & 0 & 0 & 0 & 5 \\ 0 & 7 & 0 & 9 & 0 \\ 3 & 0 & 4 & 0 & 5 \\ 0 & 2 & 0 & 0 & 6 \end{bmatrix}$$

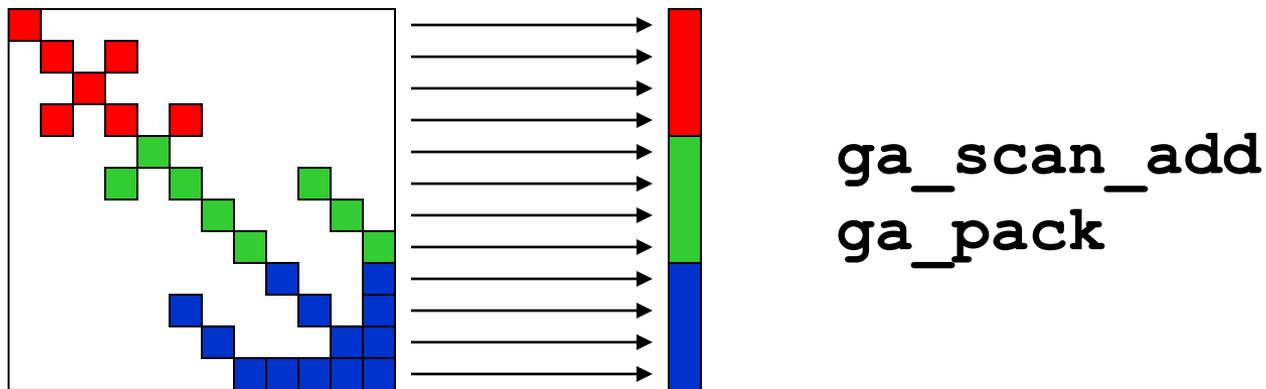
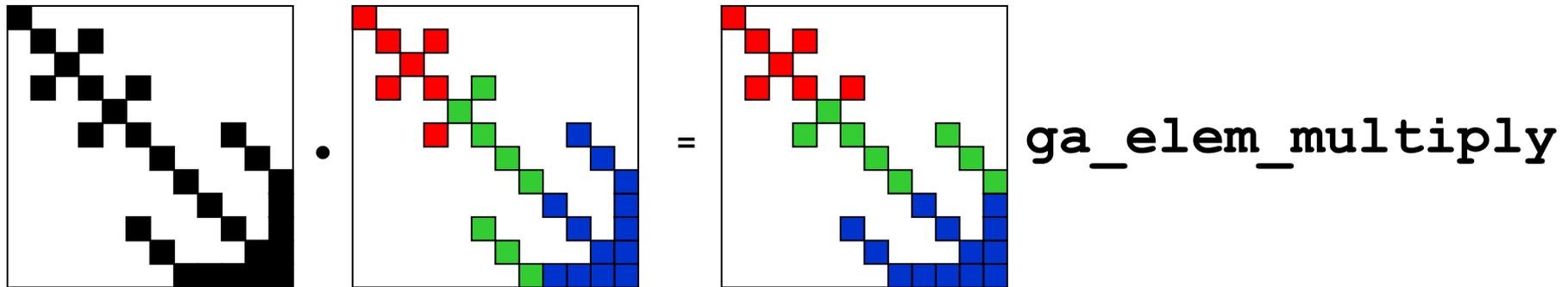
VALUES:	<b>1</b>	<b>3</b>	<b>2</b>	<b>5</b>	<b>7</b>	<b>9</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>2</b>	<b>6</b>
J-INDEX:	3	4	1	5	2	4	1	3	5	2	5
I-INDEX:	1	3	5	7	10	12					

# Sparse Matrix-Vector Multiply



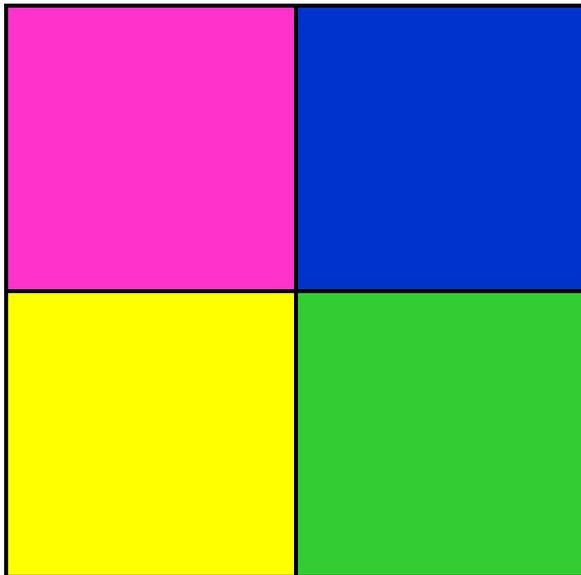
`nga_access`  
`nga_gather`

# Sparse Matrix-Vector Multiply

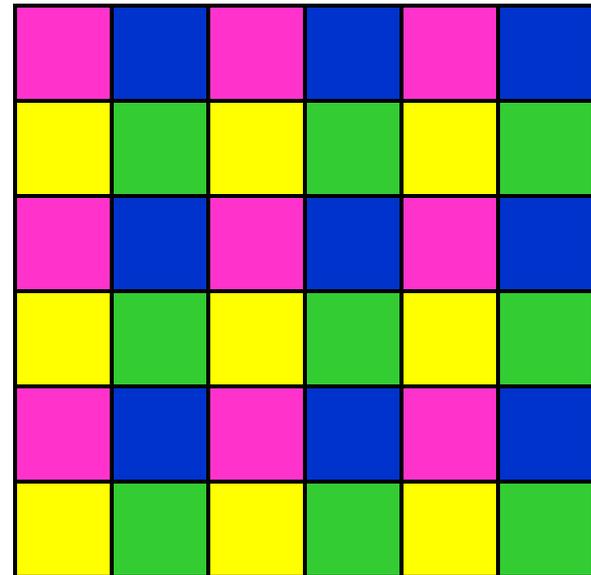


# Block-cyclic Data Distributions

Normal Data Distribution



Block Cyclic Data Distribution



# Block-cyclic Data Distributions

Simple Distribution

0	6	12	18	24	30
1	7	13	19	25	31
2	8	14	20	26	32
3	9	15	21	27	33
4	10	16	22	28	34
5	11	17	23	29	35

Scalapack Distribution

	0	1	0	1	0	1
0	0,0	0,1				
1	1,0	1,1				
0						
1						
0						
1						

# Block-cyclic Data Distributions

- ▶ 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

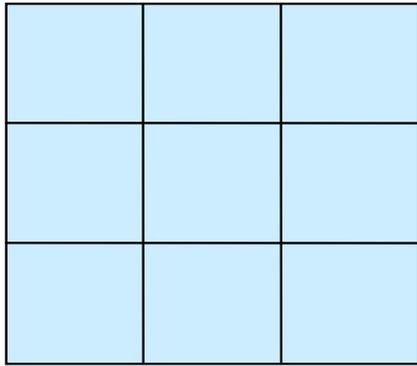
# Creating Block-cyclic Arrays

```
integer function ga_create_handle()  
subroutine ga_set_data(g_a, ndim, dims, type)  
subroutine ga_set_array_name(g_a, name)  
subroutine ga_set_block_cyclic(g_a, b_dims)  
subroutine ga_set_block_cyclic_proc_grid(g_a,  
                                          dims, proc_grid)  
subroutine ga_allocate(g_a)
```

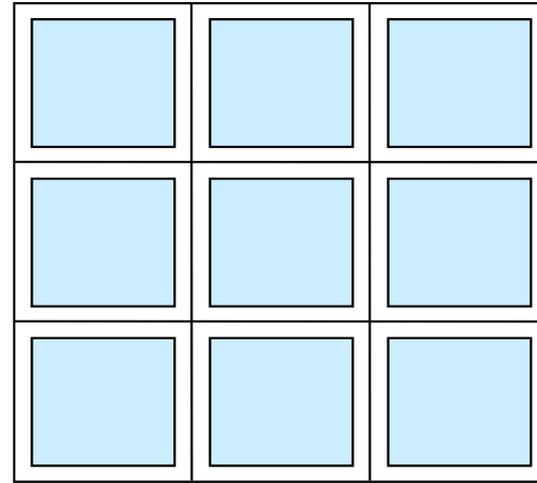
# Block-Cyclic Methods

```
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)
```

# Ghost Cells



normal global array



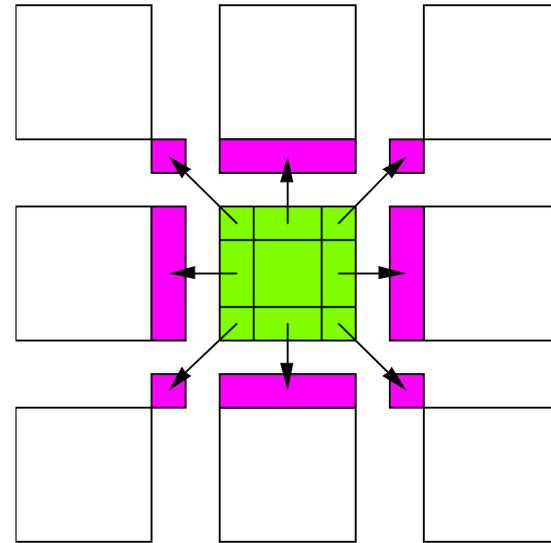
global array with ghost cells

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

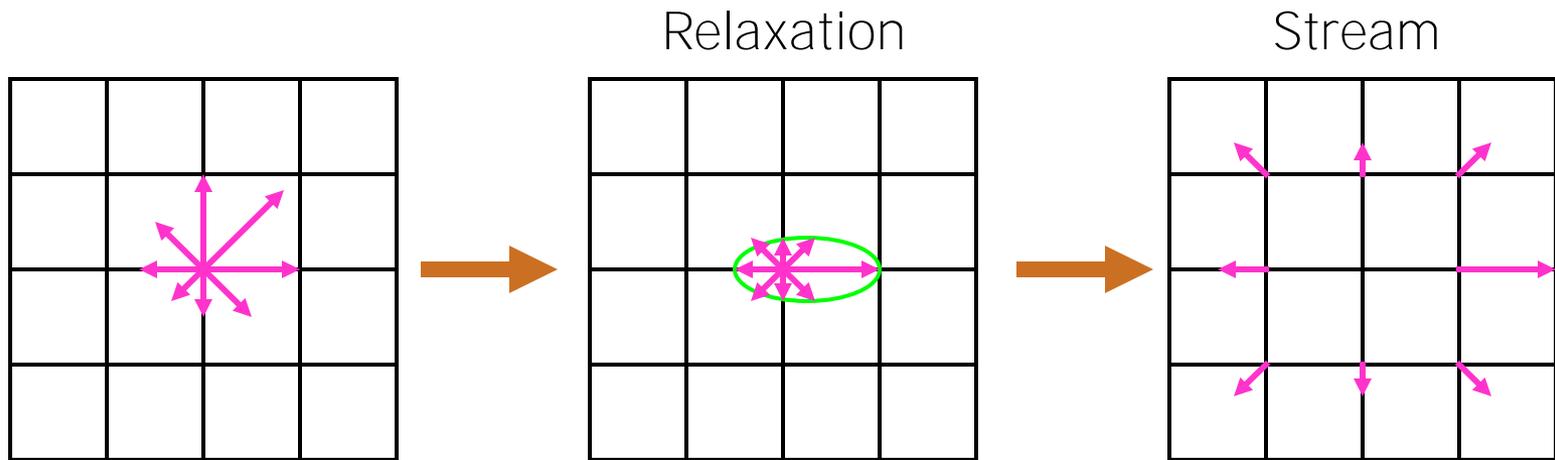
# Ghost Cell Update

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.

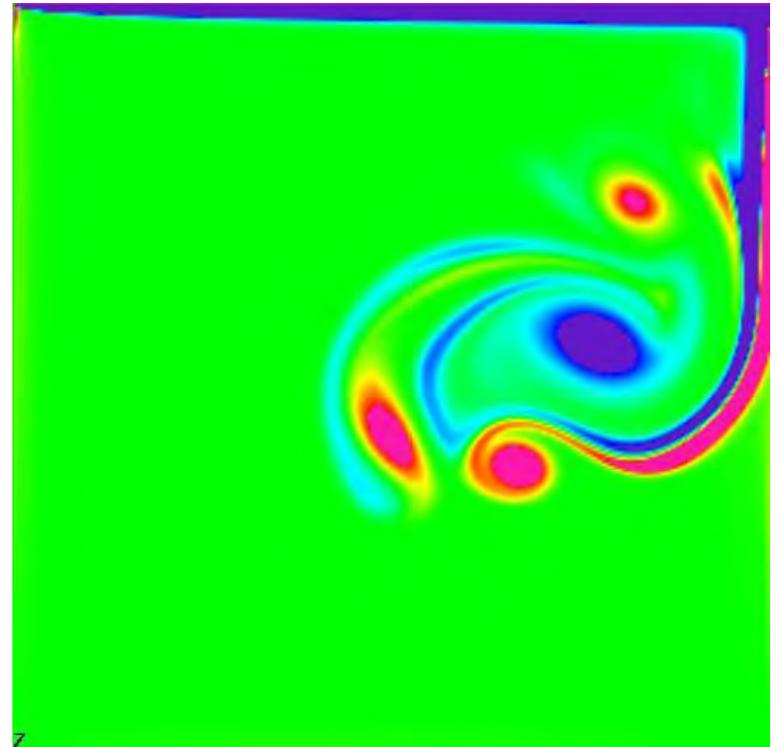
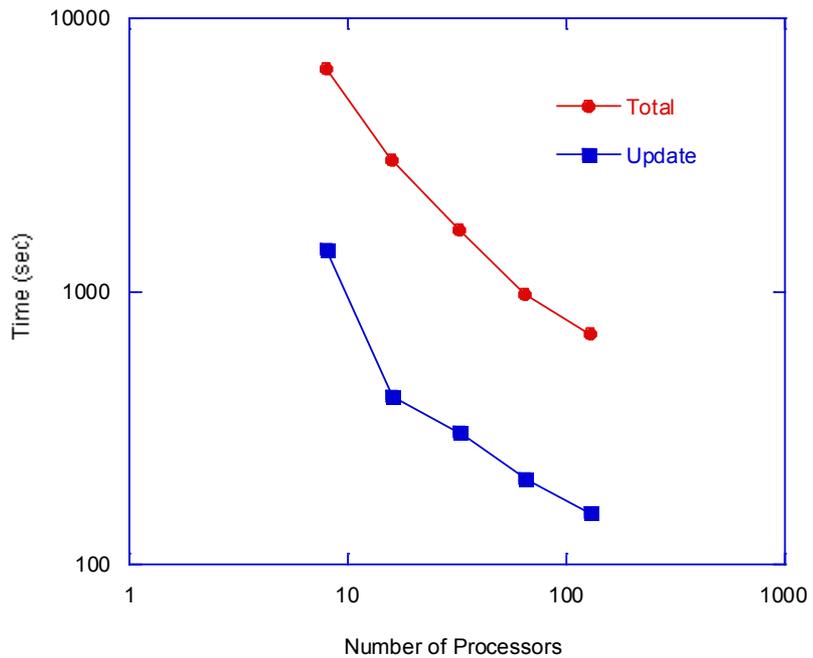


# Lattice Boltzmann Simulation

$$f_i(\mathbf{r} + \mathbf{e}_i, t + \Delta t) = f_i(\mathbf{r}, t) - \frac{1}{\tau} (f_i(\mathbf{r}, t) - f_i^{eq}(\mathbf{r}, t))$$

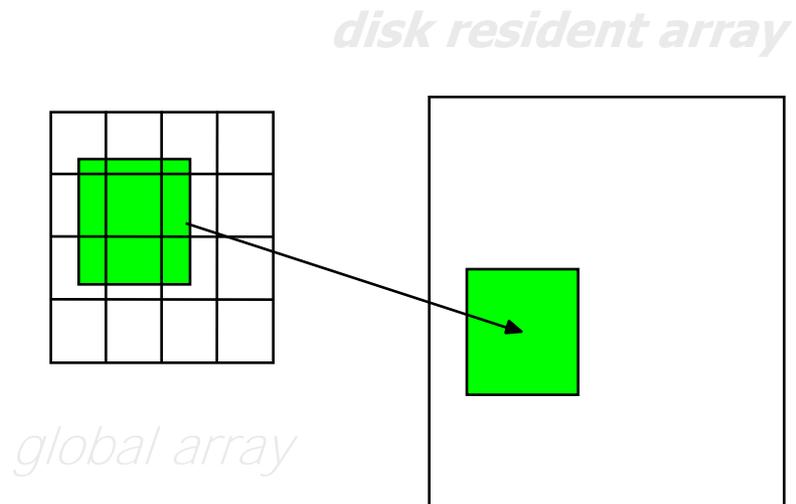


# Lattice Boltzmann Performance



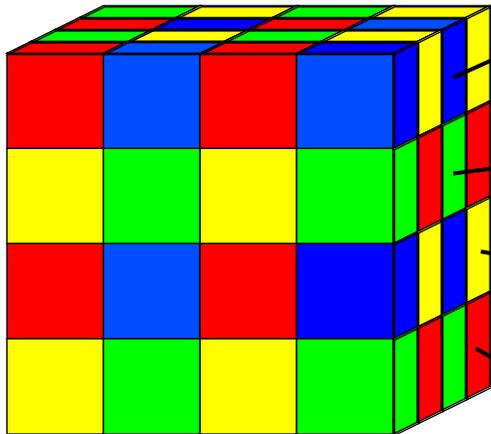
# Disk Resident Arrays

- ▶ Extend GA model to disk
  - system similar to Panda (U. Illinois) but higher level APIs
- ▶ Provide easy transfer of data between N-dim arrays stored on disk and distributed arrays stored in memory
- ▶ Use when
  - Arrays too big to store in core
  - checkpoint/restart
  - out-of-core solvers

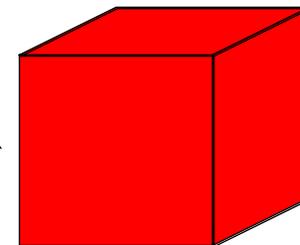
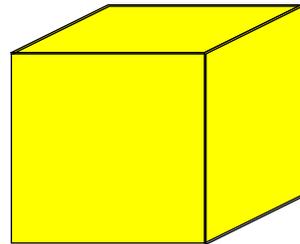
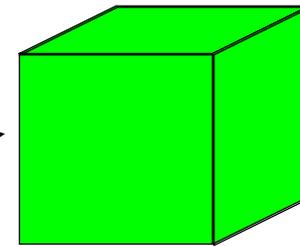
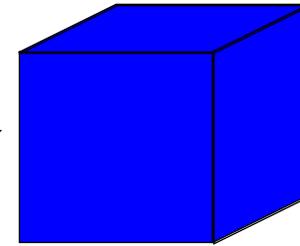


# High Bandwidth Read/Write

Disk Resident Array



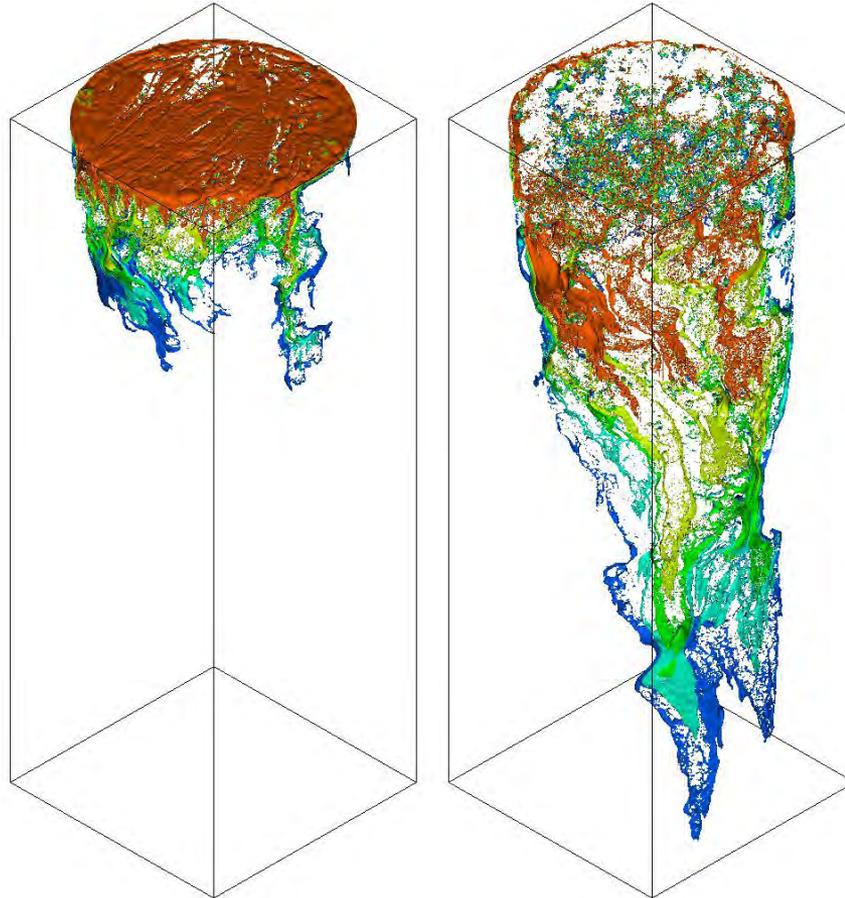
Disk Resident Arrays  
automatically  
decomposed into  
multiple files



Disks

# Outline

- ▶ Programming Model
- ▶ Basic Functions in GA
- ▶ Advanced Functionality
- ▶ Summary



# Related Programming Tools

## ▶ Co-Array Fortran

- Distributed Arrays
- One-Sided Communication
- No Global View of Data

## ▶ UPC

- Model Similar to GA but only applicable to C programs
- Global Shared Pointers could be used to implement GA functionality
  - C does not really support multi-dimensional arrays

## ▶ High level functionality in GA is missing from these systems

# Ongoing/Future Work

- ▶ Scalability to 100k+ processes
- ▶ Support for multithreaded execution
- ▶ Fault Tolerance
- ▶ Data Decomposition and Load balancing
- ▶ Support for Hybrid Platforms
- ▶ Performance tools for GA/ARMCI
- ▶ Task Scheduling and Futures
- ▶ Global Pointer Arrays
- ▶ Energy Optimization

# Current Release

## ▶ 5.0.2

- Autoconf/automake/libtool build
- Restricted arrays
- ARMCI enhancements
  - On-demand connection management for InfiniBand
  - Improved scalability for fence

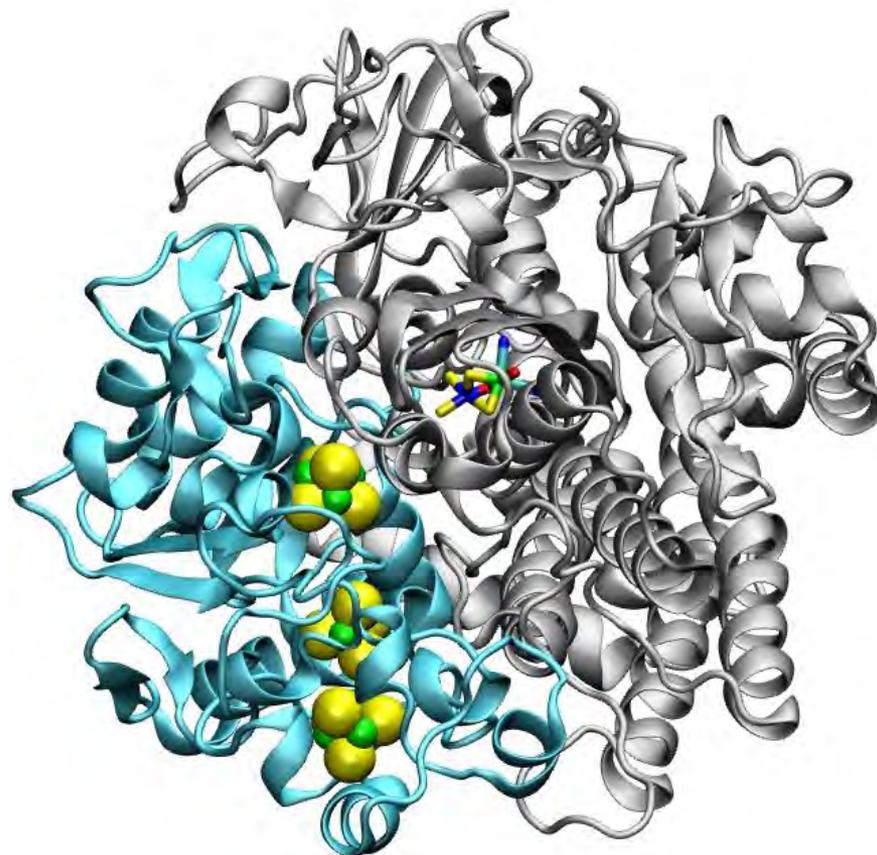
# Summary

- ▶ The idea has proven very successful
  - efficient on a wide range of architectures
    - core operations tuned for high performance
  - library substantially extended but all original (1994) APIs preserved
  - increasing number of application areas
- ▶ Supported and portable tool that works in real applications

# Source Code and More Information

- ▶ Version 5.0.2 available
- ▶ Homepage at <http://www.emsl.pnl.gov/docs/global/>
- ▶ Platforms (32 and 64 bit)
  - BlueGene/L, BlueGene/P
  - Cray XT, XE
  - Linux Cluster with Ethernet, Infiniband
  - HP
  - Altix
  - Fujitsu
  - Windows

# Supplementary Material



# Useful GA Functions (Fortran)

```
subroutine ga_initialize()
subroutine ga_terminate()

integer function ga_nnodes()
integer function ga_nodeid()

logical function nga_create(type,dim,dims,name,chunk,g_a)
    integer type (MT_F_INT, MT_F_DBL, etc.)
    integer dim
    integer dims(dim)
    character*(*) name
    integer chunk(dim)
    integer g_a
logical function ga_duplicate(g_a,g_b,name)
    integer g_a
    integer g_b
    character*(*) name
logical function ga_destroy(g_a)
    integer g_a

subroutine ga_sync()
```

# Useful GA Functions (Fortran)

```
subroutine nga_distribution(g_a, node_id, lo, hi)
  integer g_a
  integer node_id
  integer lo(dim)
  integer hi(dim)
subroutine nga_put(g_a, lo, hi, buf, ld)
  integer g_a
  integer lo(dim)
  integer hi(dim)
  fortran array buf
  integer ld(dim-1)
subroutine nga_get(g_a, lo, hi, buf, ld)
  integer g_a
  integer lo(dim)
  integer hi(dim)
  fortran array buf
  integer ld(dim-1)
```

# Useful GA Functions (C)

```
void GA_Initialize()
void GA_Terminate()

int GA_Nnodes()
int GA_Nodeid()

int NGA_Create(type, dim, dims, name, chunk) Returns GA handle g_a
    int type (C_INT, C_DBL, etc.)
    int dim
    int dims[dim]
    char* name
    int chunk[dim]
int GA_Duplicate(g_a, name) Returns GA handle g_b
    int g_a
    char* name
void GA_Destroy(g_a)
    int g_a

void GA_Sync()
```

# Useful GA Functions (C)

```
void NGA_Distribution(g_a, node_id, lo, hi)
    int g_a
    int node_id
    int lo[dim]
    int hi[dim]
void NGA_Put(g_a, lo, hi, buf, ld)
    int g_a
    int lo[dim]
    int hi[dim]
    void* buf
    int ld[dim-1]
void NGA_Get(g_a, lo, hi, buf, ld)
    int g_a
    int lo[dim]
    int hi[dim]
    void* buf
    int ld[dim-1]
```

# Problems 1 and 2

- ▶ Chose Fortran or C version of problems (whichever language you prefer)
  - Xxxx.c or xxx.F
- ▶ Search file for comments marked with ###
- ▶ Using the text as hints, replace the comments with subroutines or functions from the GA library to create a working code
- ▶ Compile and run

# Problem 1 (1D Transpose)

- ▶ Transpose a distributed 1D vector containing  $N$  elements in the order  $1, 2, \dots, N$  into a distributed vector containing  $N$  elements in the order  $N, N-1, \dots, 2, 1$
- ▶ Fortran version of this problem is in the file `transp1D.F.tutorial`
- ▶ C version is in `transp1D.c.tutorial`
- ▶ Working versions of these codes are in `transp1D.F` and `transp1D.c`

## Problem 2 (Matrix Multiplication)

- ▶ A simple matrix multiply algorithm that initializes two large matrices as GAs. It then multiplies a block of columns by a block or rows from the GAs locally on each processor and copies the result into a third global array
- ▶ Fortran version of this problem is in the file `matrix.F.tutorial`
- ▶ C version is in `matrix.c.tutorial`
- ▶ Working versions are in `matrix.F` and `matrix.c`

# Problem 3

- ▶ Both the codes in problems 1 & 2 initialize the data by initializing a local array on processor 0 with all the data and then copying it to a distributed global array. For real problems it is usually undesirable to have all the data located on one processor at any point in the calculation. Can you modify these codes (problem 1 and 2) so that each processor only initializes the data owned by that processor?
- ▶ **1D transpose (Problem 1)**
  - Modify code so that each processor only initializes the local array `a()` with the data owned by that processor and then copy that data to the global array `g_a`
  - Hint: Use `nga_distribution` and `nga_put`
  - You will also need to modify the result checking part of the code as well so that it also only uses smaller portions of the total GA
  - Hint: copy locally held part of result GA into local array `b` and corresponding part of original vector into local array `a` and compare (use arrays `lo`, `hi`, `lo2`, `hi2` to get this data).
- ▶ **Matrix Multiply (Problem 2)**
  - Modify code so that each processor only initializes the local arrays `a` and `b` with the data held locally by that processor. Then copy that data to the global arrays `g_a` and `g_b`.
  - Hint: Use `nga_distribution` and `nga_put`