

Overview of the Global Arrays Parallel Software Development Toolkit

Bruce Palmer

**Manoj Kumar Krishnan, Sriram Krishnamoorthy,
Ahbinav Vishnu, Patrick Nichols, Jeff Daily**

Pacific Northwest National Laboratory

Overview



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

Distributed Data vs Shared Memory

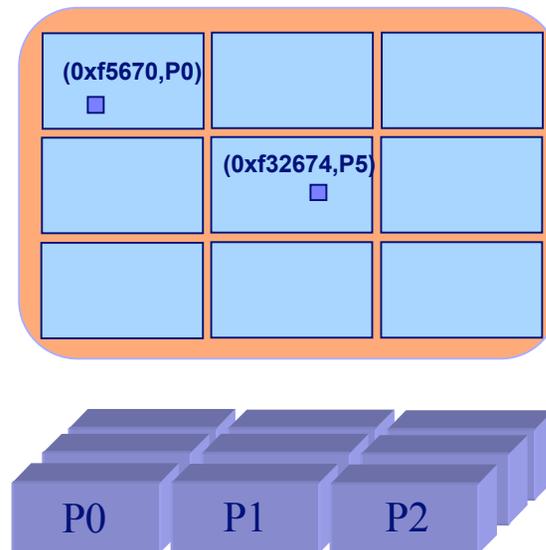


Distributed Data:

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)



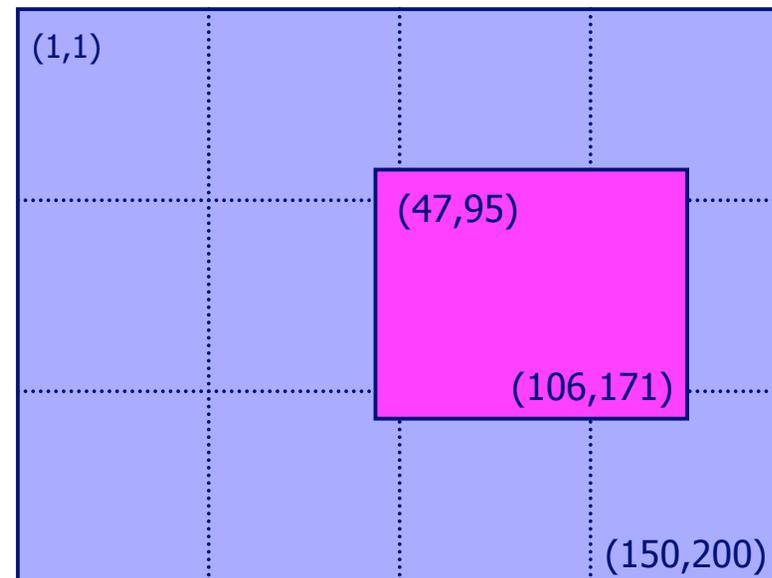
Distributed Data vs Shared Memory (Cont).



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.

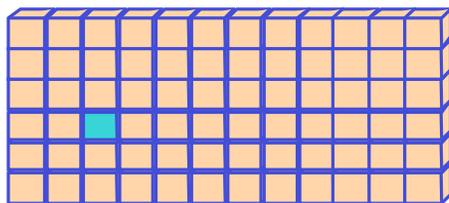
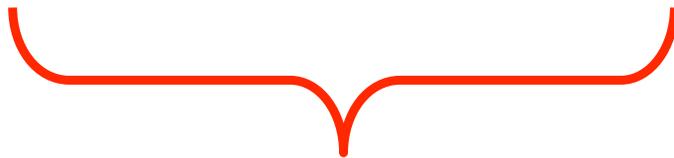
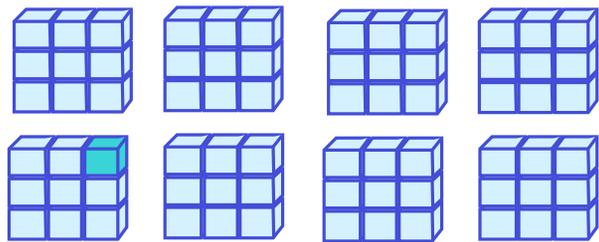




Global Arrays

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

Physically distributed data



Global Address Space

single, shared data structure/
global indexing

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

Global Arrays (cont.)



- 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

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

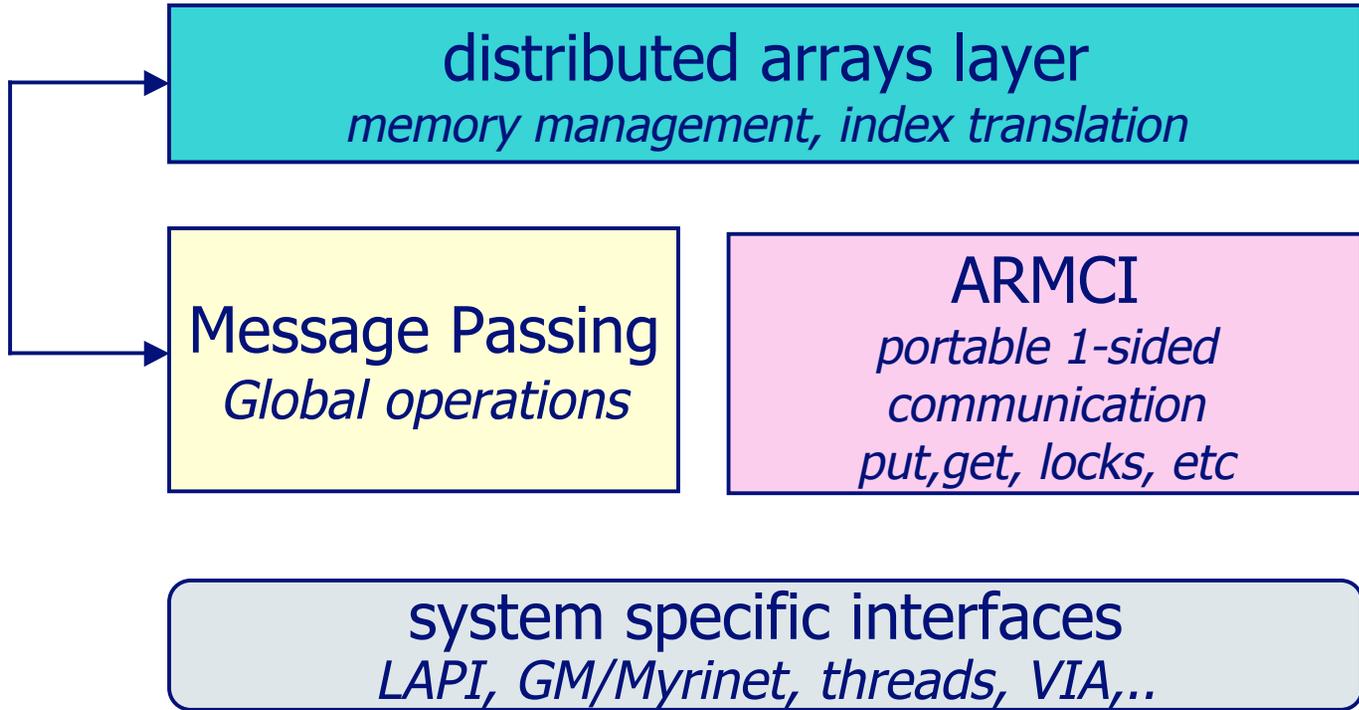


F90 Java

Application programming language interface

Fortran 77 C C++ Python Babel

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



Linking the GA library



Also, to test your GA programs, suggested compiler/linker options are as follows.

```
GA libraries are built in /home/d3g293/ga-4-0-5/lib/LINUX64
INCLUDES = -I/home/d3g293/ga-4-0-5/include
```

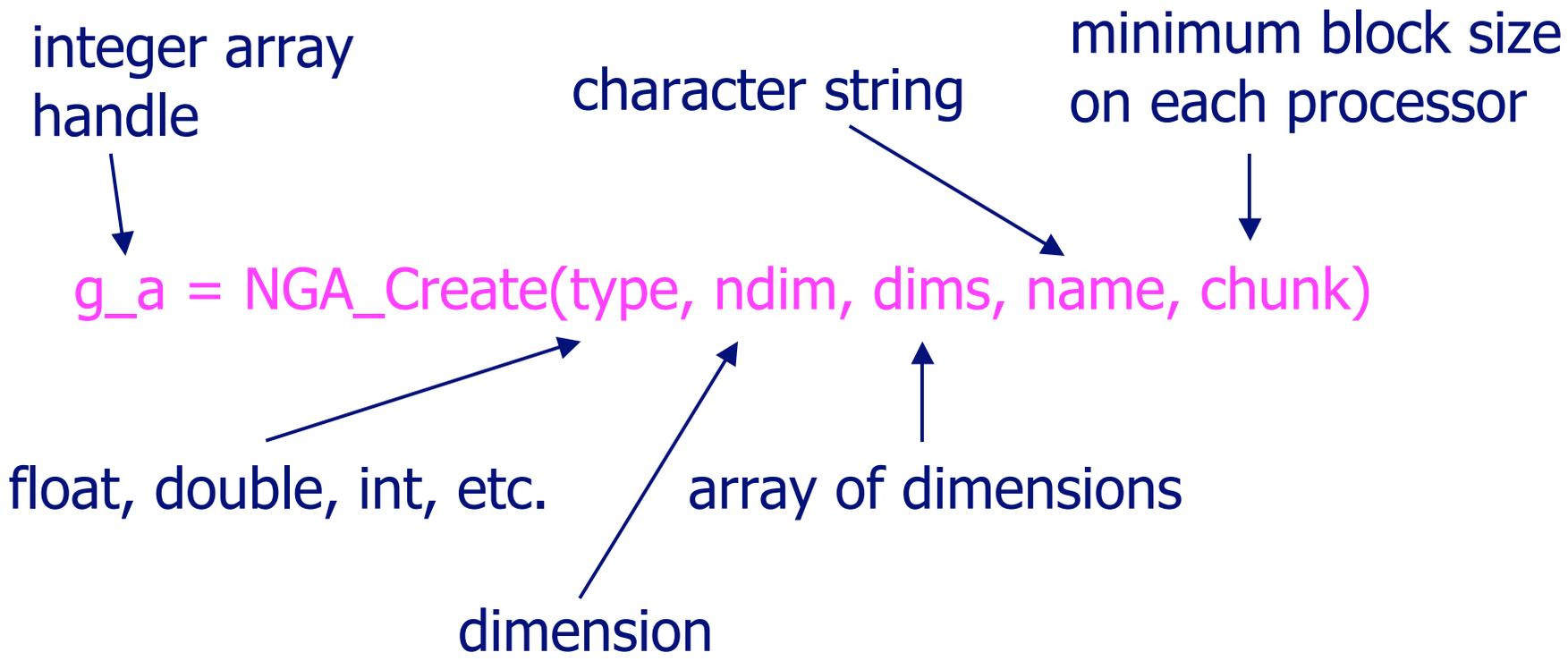
For Fortran Programs:

```
FLAGS = -g -Vaxlib -cm -w90 -w95 -align -cm -w90 -w95
-align -i8
LIBS = -L/home/d3g293/ga-4-0-5/lib/LINUX64 -lglobal -lma
-llinalg -larmci -L/home/software/ia64/mpich-1.2.7-gcc/lib
-ltcgmsg-mpi -lmpich -lm
```

For C Programs:

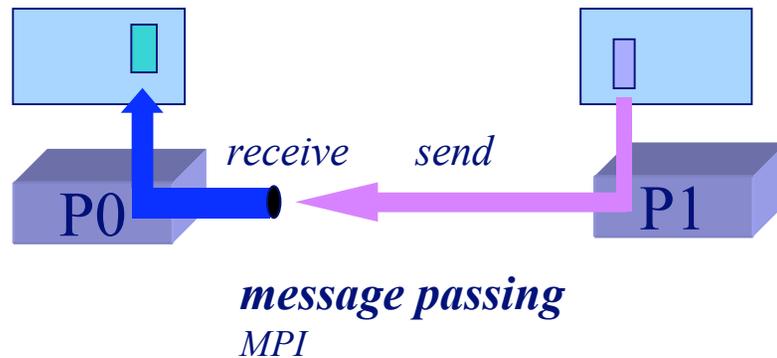
```
FLAGS = -g -nofor_main -cm -w90 -w95 -align -cm -w90 -w95
-align -i8
LIBS = -L/home/d3g293/ga-4-0-5/lib/LINUX64 -lglobal -lma
-llinalg -larmci -L/home/software/ia64/mpich-1.2.7-gcc/lib
-ltcgmsg-mpi -lmpich -lm -lm
```

Creating Global Arrays



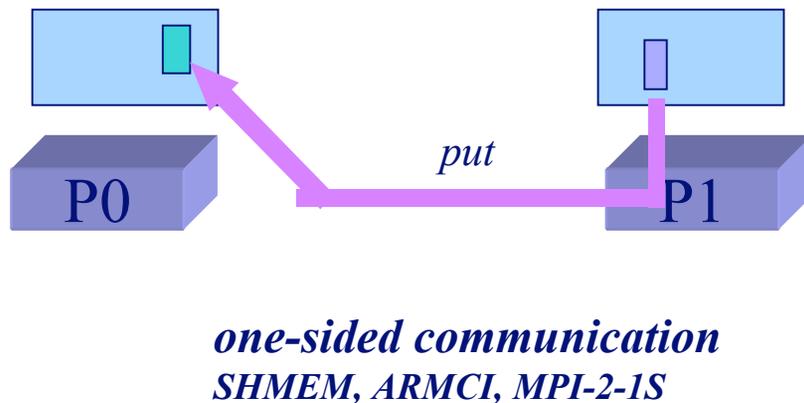


One-sided Communication



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



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
```

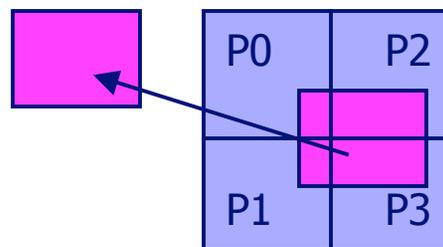
end loop

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



Data Locality



What data does a processor own?

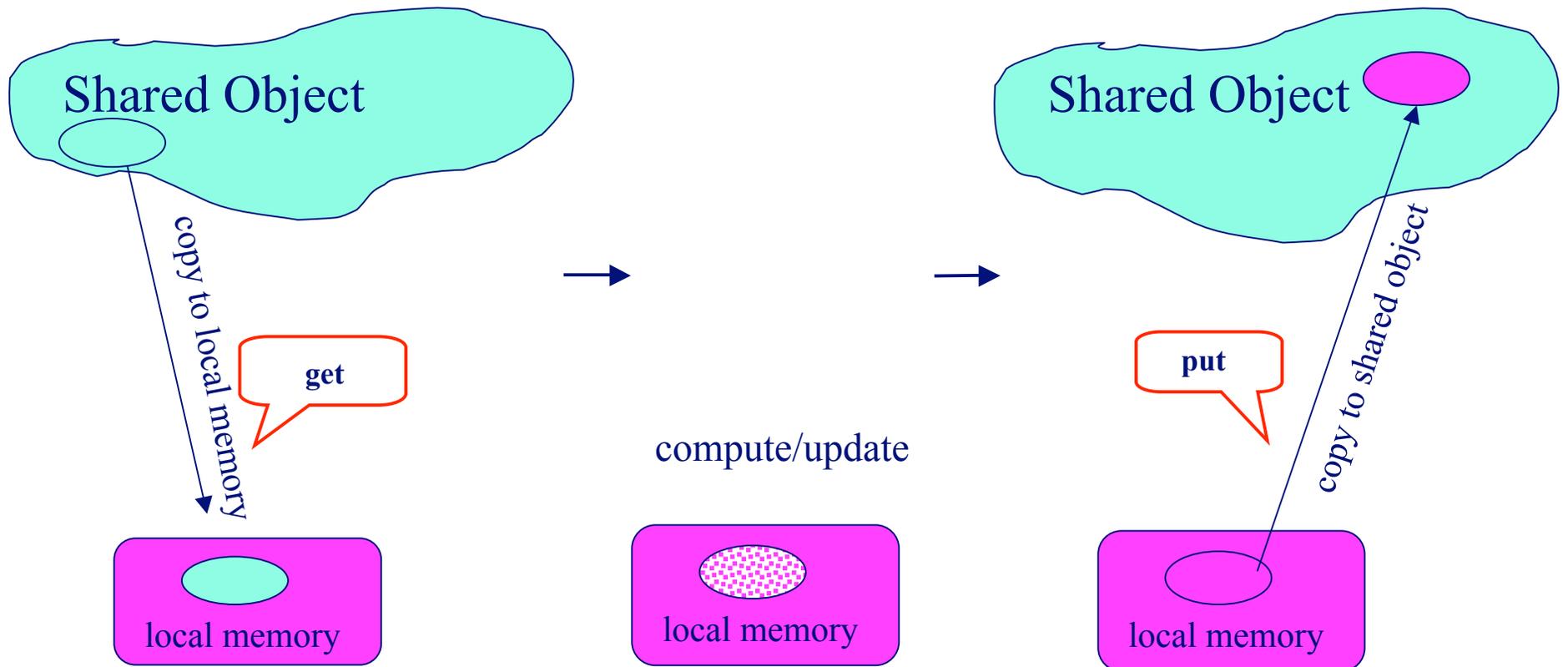
```
NGA_Distribution(g_a, iproc, lo, hi);
```

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

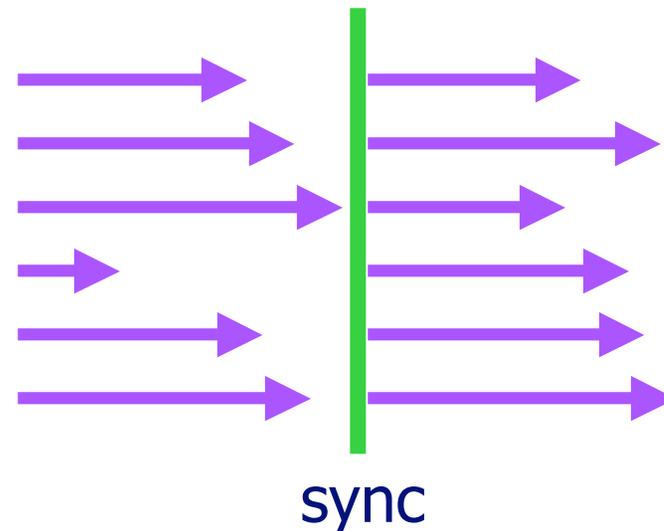
Global Array Model of Computations



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





Basic GA Operations

- GA programming model is very simple.
- Most of the parallel programs 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**

```
subroutine ga_initialize()
subroutine ga_terminate()

integer function ga_nnodes()
integer function ga_nodeid()

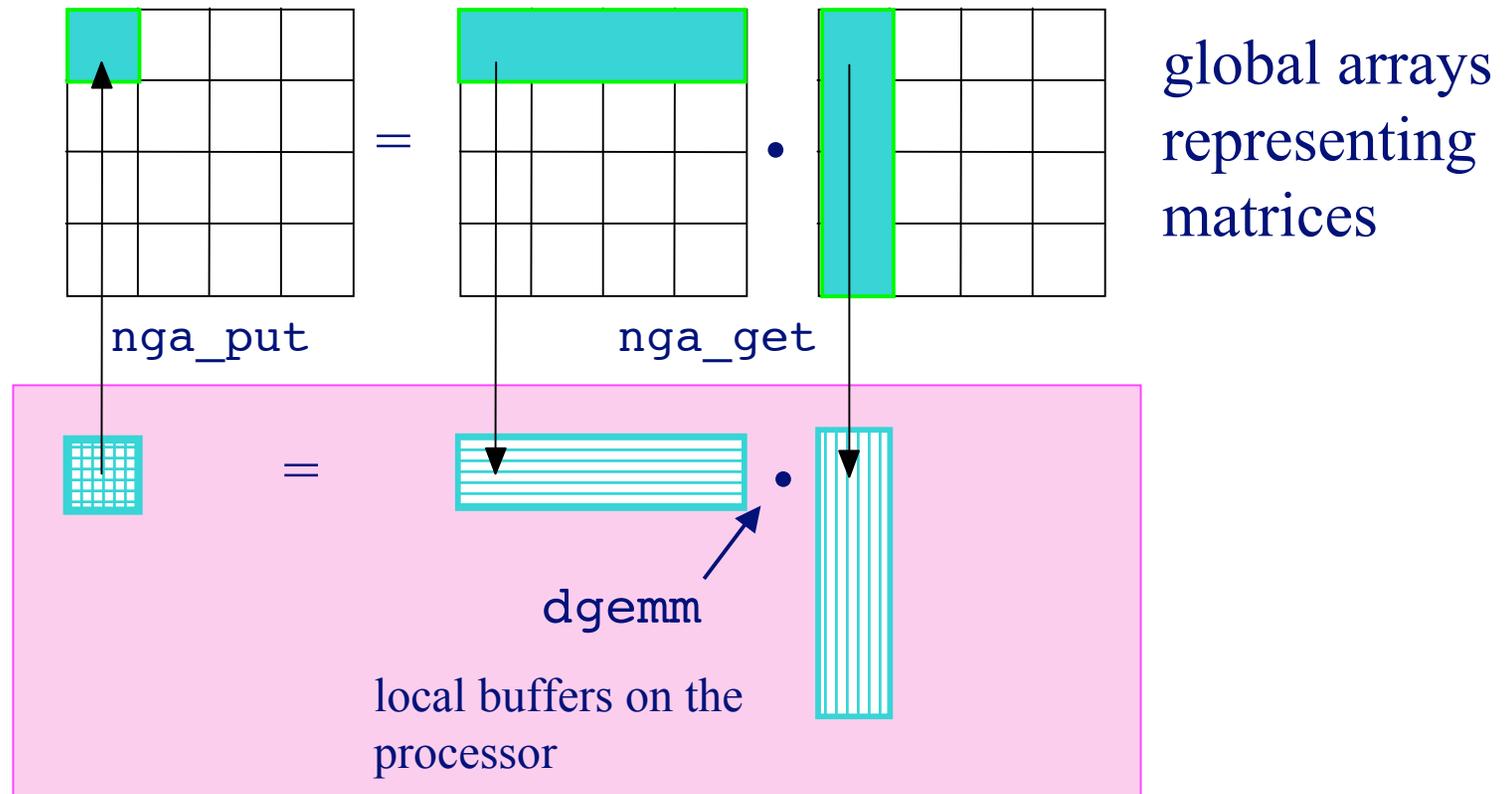
logical function nga_create(type,dim,dims,name,chunk,g_a)
logical function ga_destroy(g_a)

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

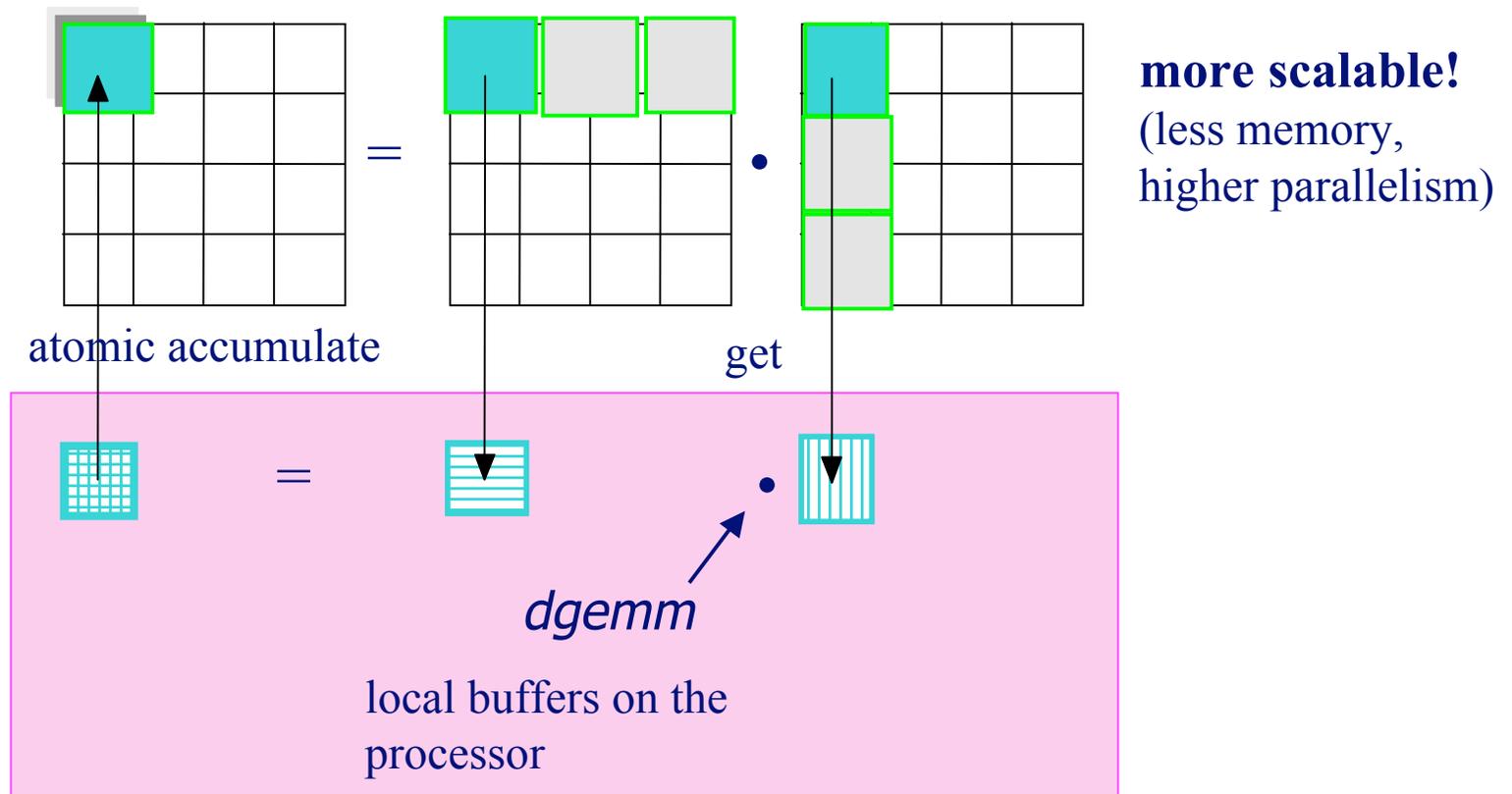
subroutine ga_sync()
```



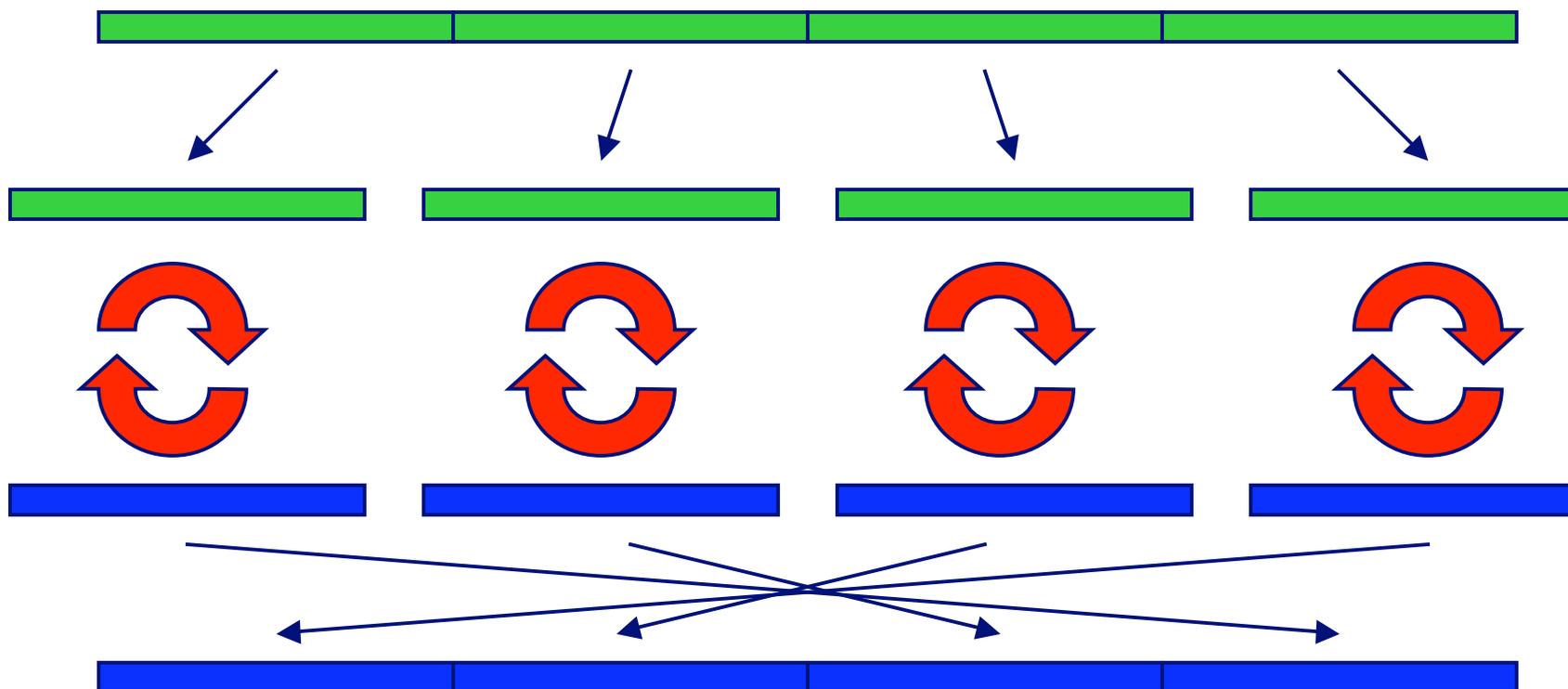
Example: Matrix Multiply



Matrix Multiply (a better version)



Example: 1-D Transpose



Example: 1-D Transpose (cont.)



```
#define    NDIM          1
#define    TOTALELEMS   197
#define    MAXPROC      128
    program main
    implicit none
#include "mafdecls.fh"
#include "global.fh"

    integer dims(3), chunk(3), nprocs, me, i, lo(3), hi(3), lo1(3)
    integer hil(3), lo2(3), hi2(3), ld(3), nelem
    integer g_a, g_b, a(MAXPROC*TOTALELEMS), b(MAXPROC*TOTALELEMS)
    integer heap, stack, ichk, ierr
    logical status
    heap = 300000
    stack = 300000
```

Example: 1-D Transpose (cont.)



```
c   initialize communication library
call mpi_init(ierr)
c   initialize ga library
call ga_initialize()
me = ga_nodeid()
nprocs = ga_nnodes()
dims(1) = nprocs*TOTALELEMS + nprocs/2  ! Unequal data distribution
ld(1) = MAXPROC*TOTALELEMS
chunk(1) = TOTALELEMS  ! Minimum amount of data on each processor
status = ma_init(MT_F_DBL, stack/nprocs, heap/nprocs)

c   create a global array
status = nga_create(MT_F_INT, NDIM, dims, "array A", chunk, g_a)
status = ga_duplicate(g_a, g_b, "array B")

c   initialize data in GA
do i=1, dims(1)
    a(i) = i
end do
lo1(1) = 1
hi1(1) = dims(1)
if (me.eq.0) call nga_put(g_a,lo1,hi1,a,ld)
call ga_sync() ! Make sure data is distributed before continuing
```

Example: 1-D Transpose (cont.)



```
c      invert data locally
call nga_distribution(g_a, me, lo, hi)
call nga_get(g_a, lo, hi, a, ld) ! Use locality
nelem = hi(1)-lo(1)+1
do i = 1, nelem
  b(i) = a(nelem - i + 1)
end do

c      invert data globally
lo2(1) = dims(1) - hi(1) + 1
hi2(1) = dims(1) - lo(1) + 1
call nga_put(g_b, lo2, hi2, b, ld)
call ga_sync() ! Make sure inversion is complete
```

Example: 1-D Transpose (cont.)



```
c      check inversion
      call nga_get(g_a,lo1,hi1,a,ld)
      call nga_get(g_b,lo1,hi1,b,ld)
      ichk = 0
      do i= 1, dims(1)
         if (a(i).ne.b(dims(1)-i+1).and.me.eq.0) then
            write(6,*) "Mismatch at ",i
            ichk = ichk + 1
         endif
      end do
      if (ichk.eq.0.and.me.eq.0) write(6,*) "Transpose OK"

      status = ga_destroy(g_a) ! Deallocate memory for arrays
      status = ga_destroy(g_b)
      call ga_terminate()
      call mpi_finalize(ierr)
      stop
      end
```

Non-Blocking Communication



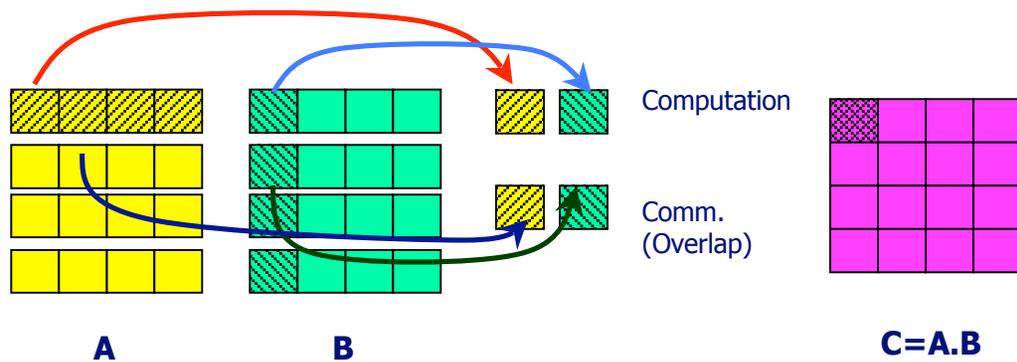
- Allows overlapping of data transfers and computations
 - Technique for latency hiding
- Nonblocking operations initiate a communication call and then return control to the application immediately
- operation completed locally by making a call to the *wait* routine

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

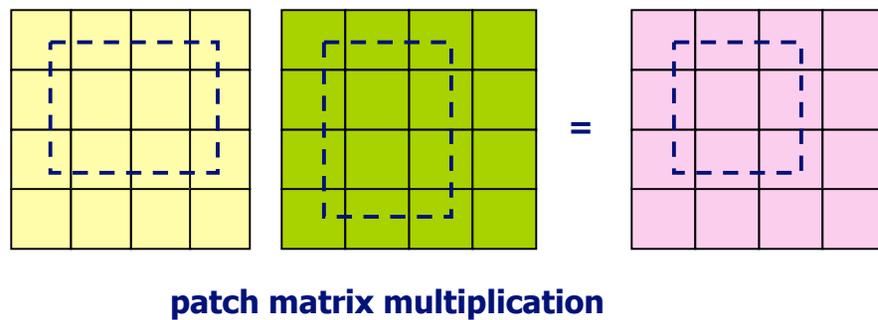
```
NGA_Nbwait(nbhandle)
```



SUMMA Matrix Multiplication



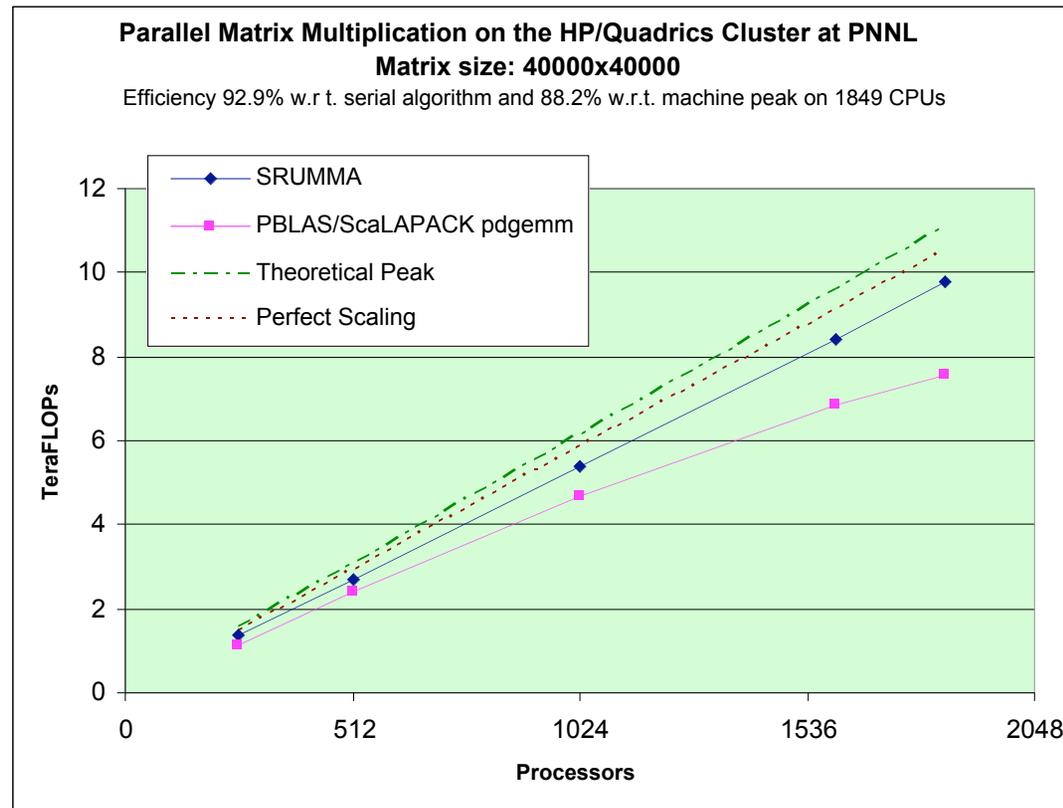
Issue NB Get A and B blocks
do (until last chunk)
 issue NB Get to the next blocks
 wait for previous issued call
 compute $A*B$ (sequential dgemm)
 NB atomic accumulate into "C"
 matrix
done



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

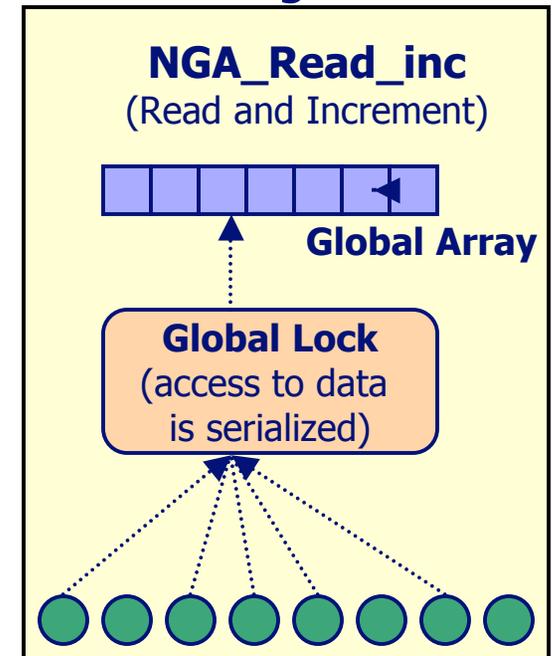




Read and Increment

- `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 ...
```



Global Array Processor Groups



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

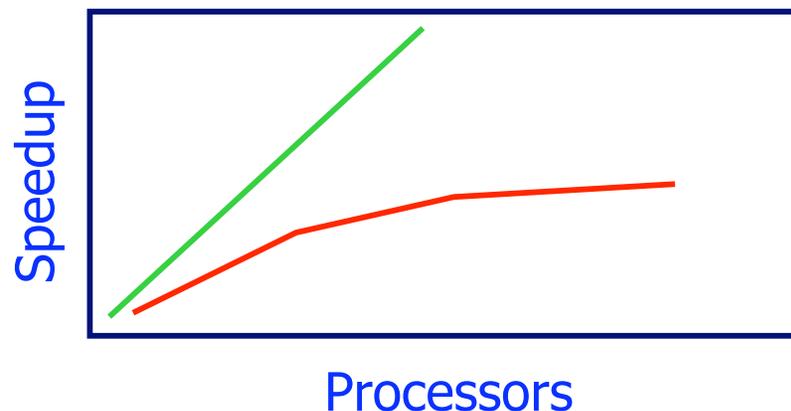
- Numerical evaluation of gradients
- Monte Carlo sampling over initial conditions or uncertain parameter sets
- Free energy perturbation calculations (chemistry)
- Nudged elastic band calculations (chemistry and materials science)
- Sparse matrix-vector operations (NAS CG benchmark)
- Data layout for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

Global Array Processor Groups

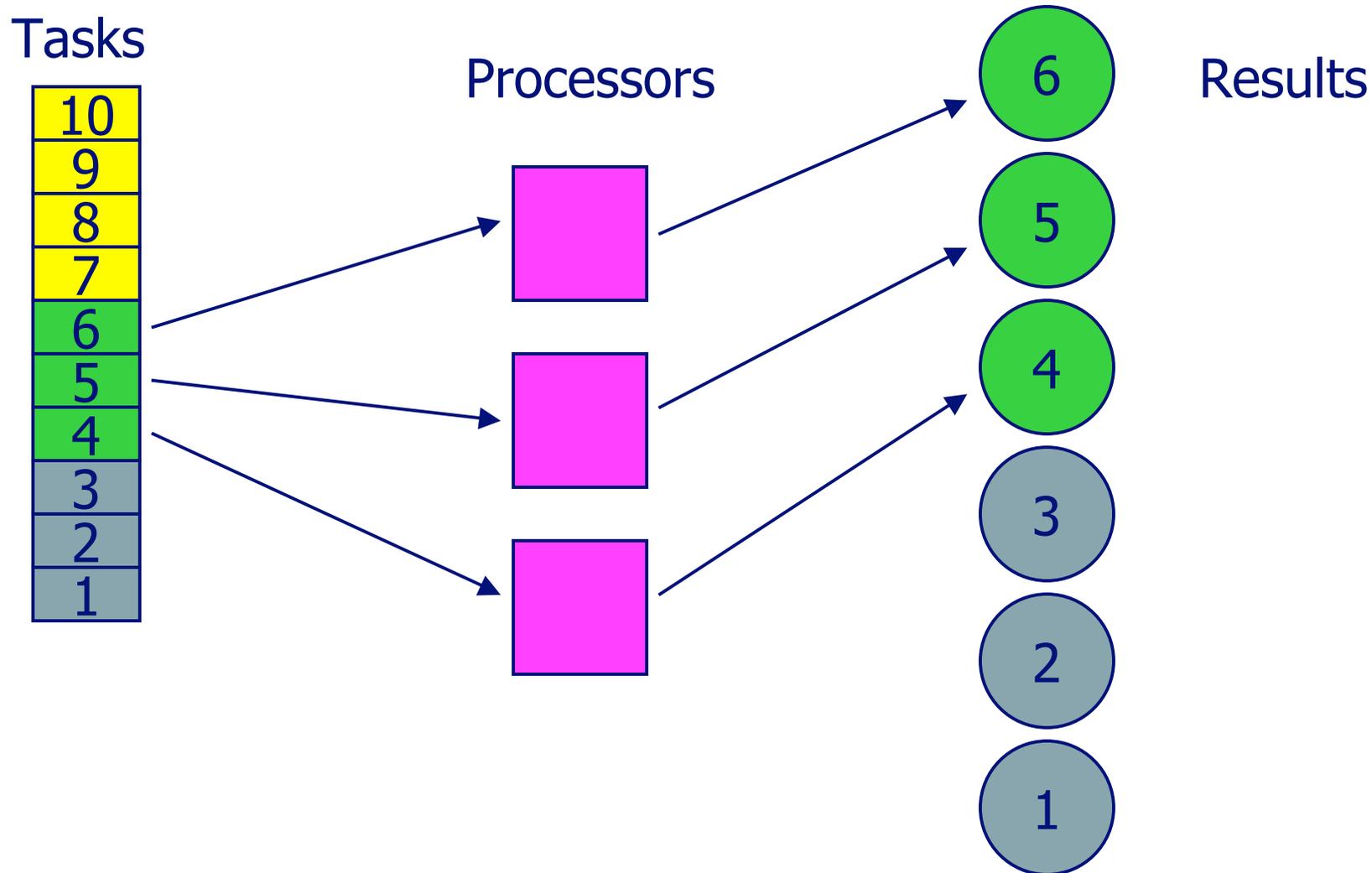


If the individual calculations are small enough then each processor can be used to execute one of the tasks (embarrassingly parallel algorithms).

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



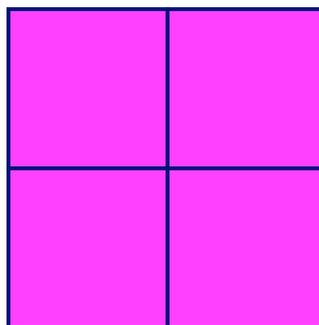
Multiple Tasks with Groups



Tasks

10
9
8
7
6
5
4
3
2
1

Processors



Results



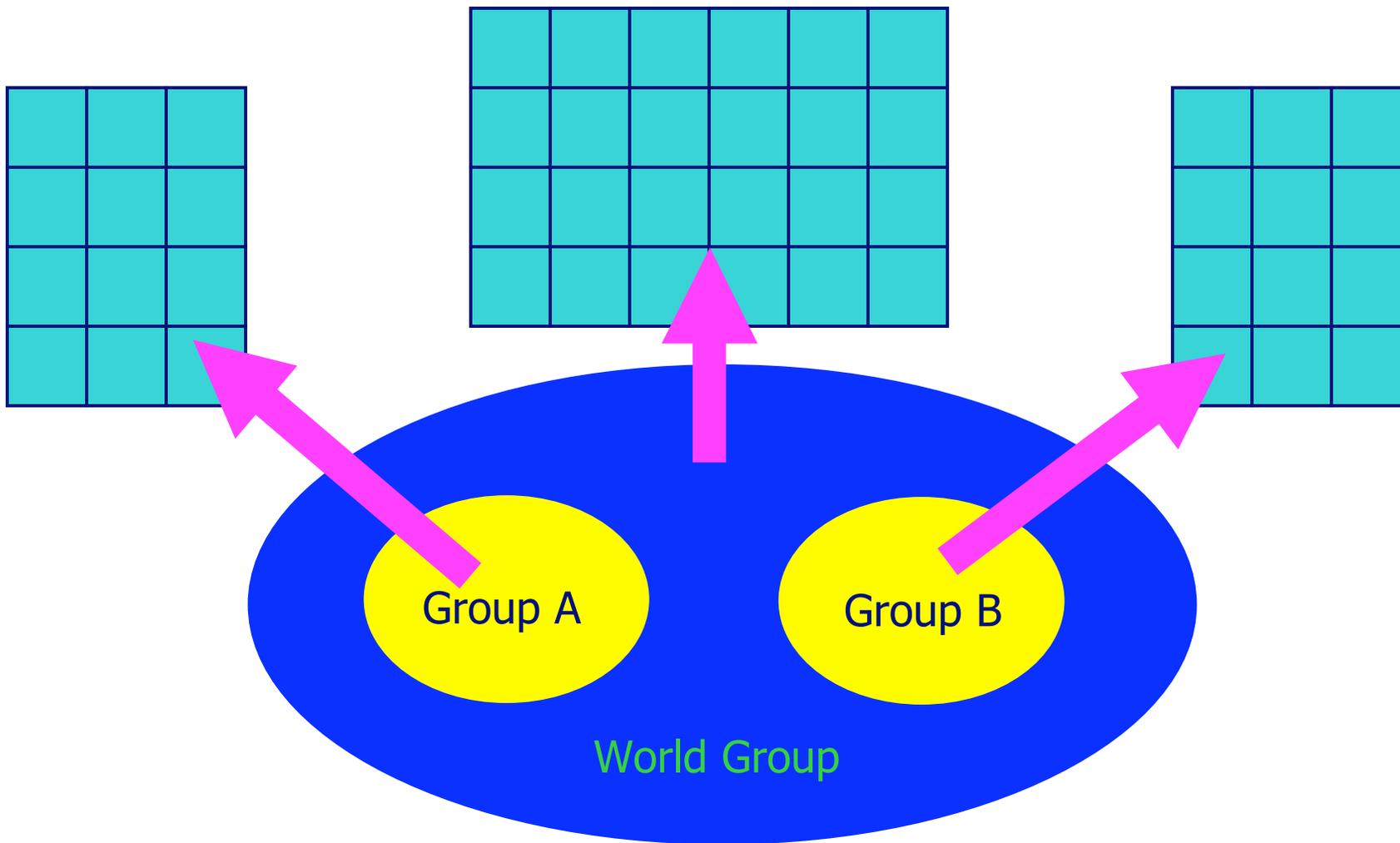
Global Array Processor Groups



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

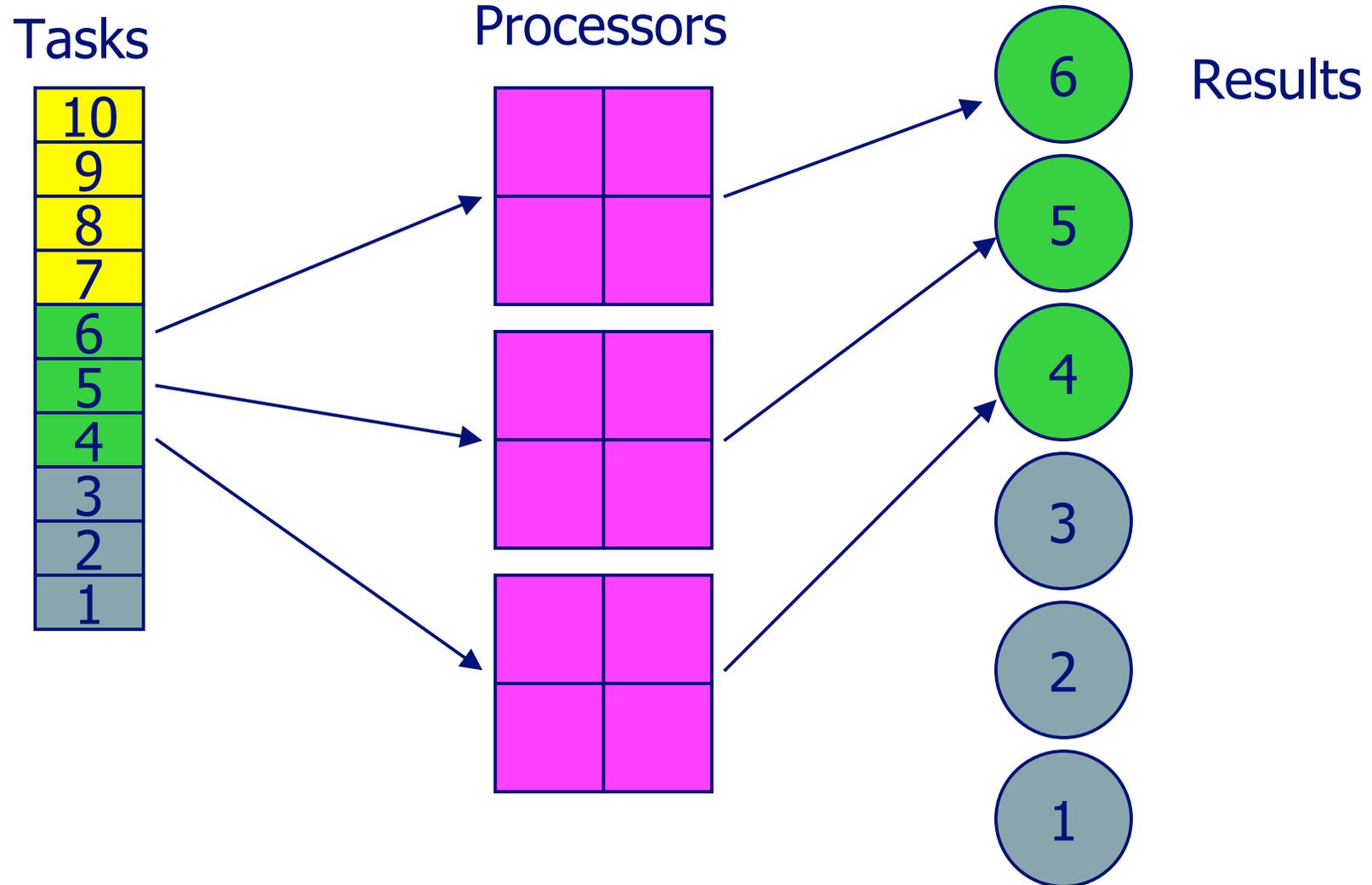
- global operations that are restricted in scope to a particular group instead of over the entire domain of processors (world group)
- distributed data structures that are restricted to a particular group

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

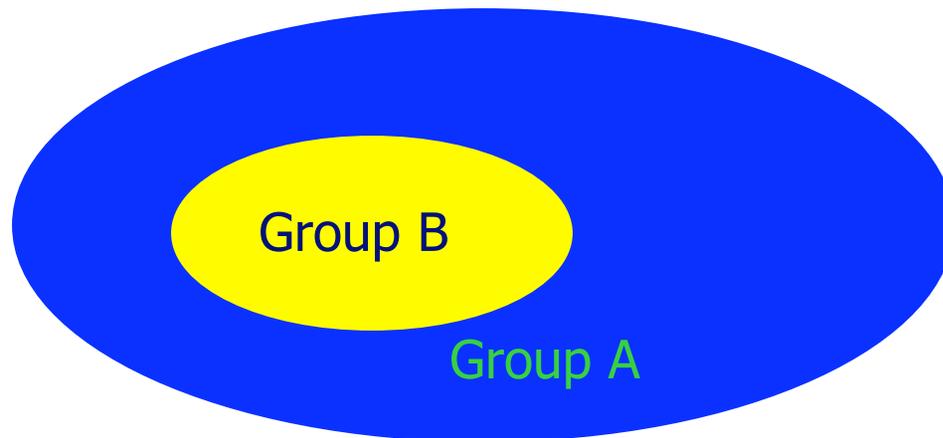
```
integer function ga_pgroup_get_default()
integer function ga_pgroup_get_world()
```

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)

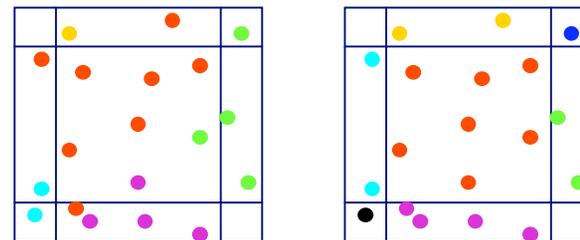
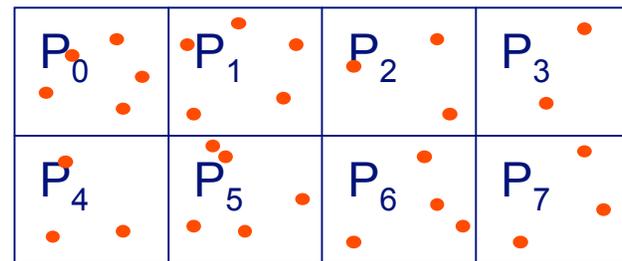


MD Example

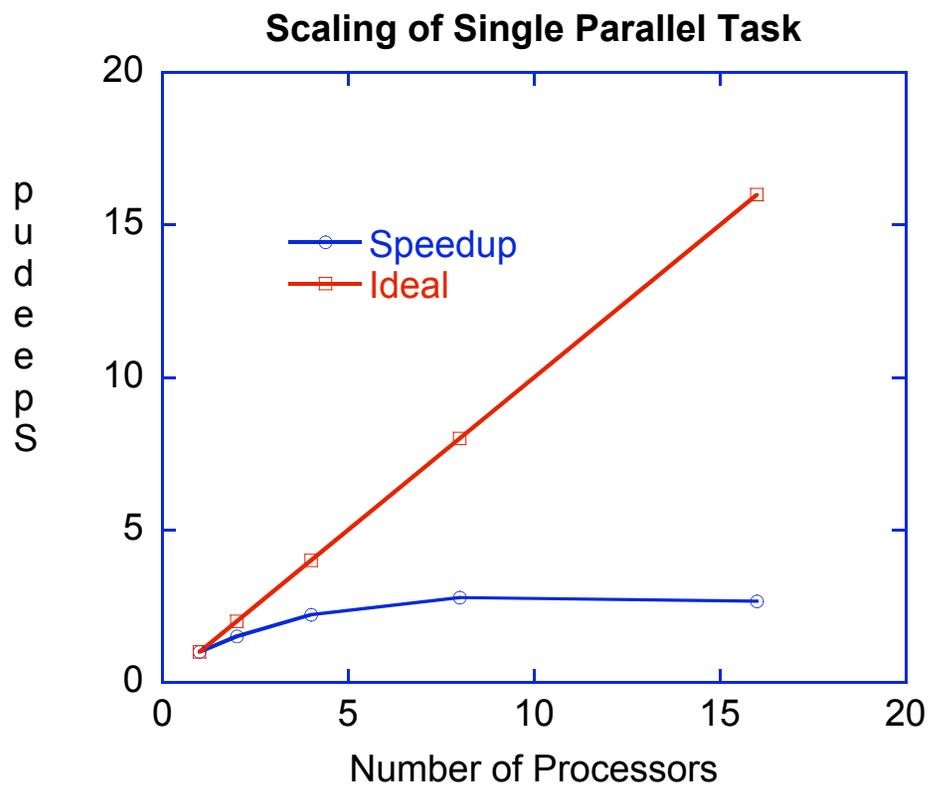


Spatial Decomposition Algorithm:

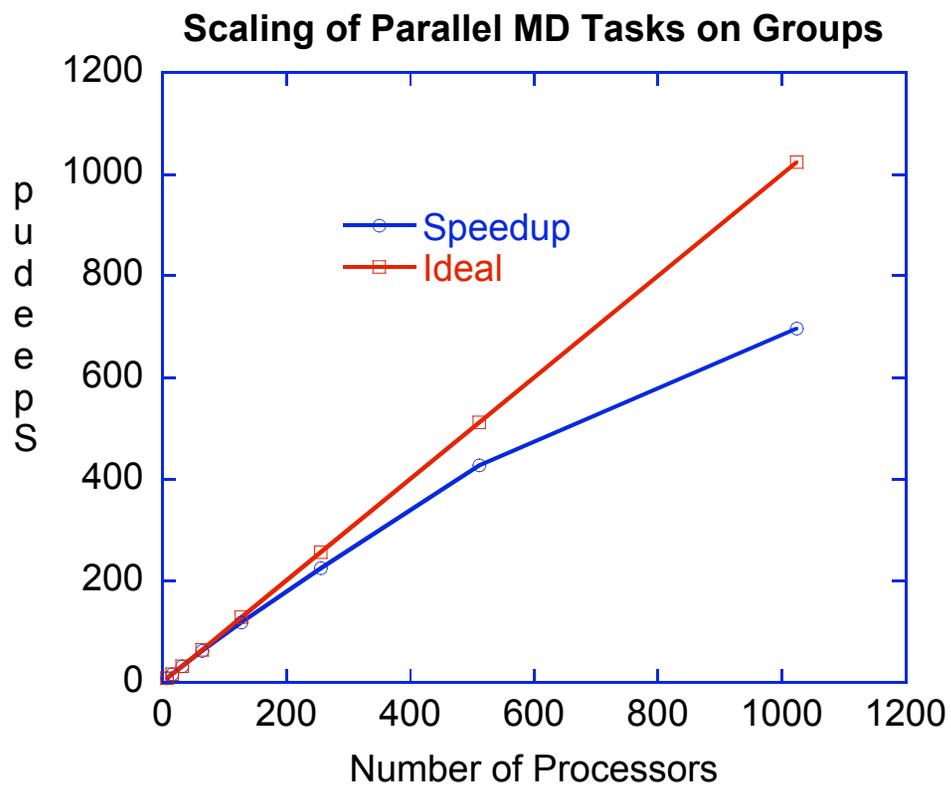
- Partition particles among processors
- Update coordinates at every step
- Update partitioning after fixed number of steps



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
```



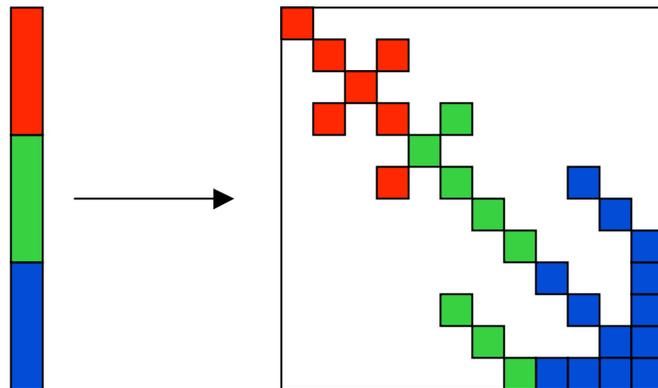
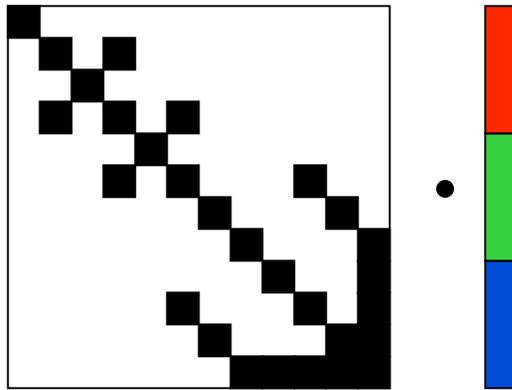
Compressed Sparse Row Matrix

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

VALUES: **1** **3** **2** **5** **7** **9** **3** **4** **5** **2** **6**
J-INDEX: 3 4 1 5 2 4 1 3 5 2 5
I-INDEX: 1 3 5 7 10 12

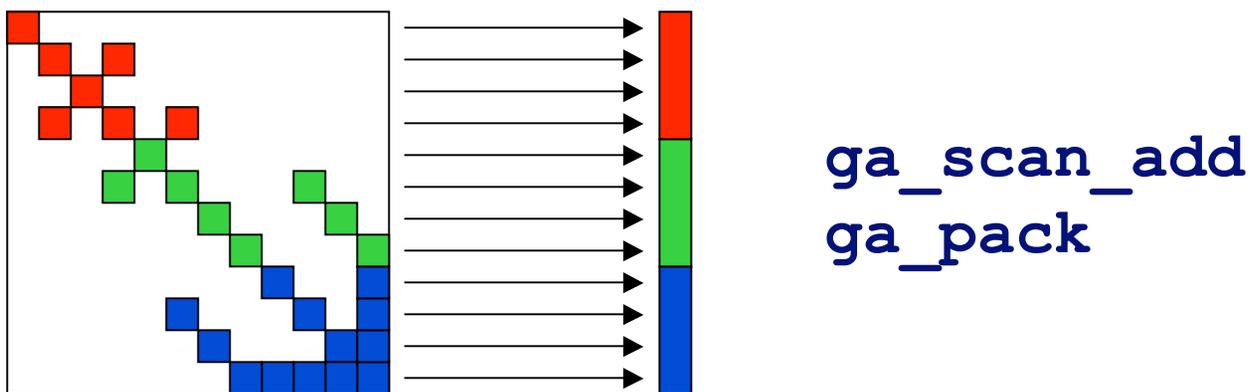
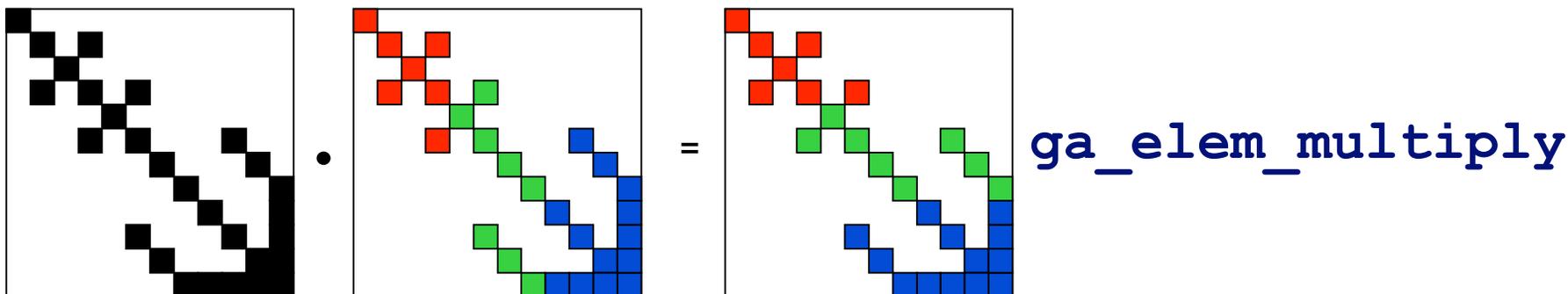


Sparse Matrix-Vector Multiply



`nga_access`
`nga_gather`

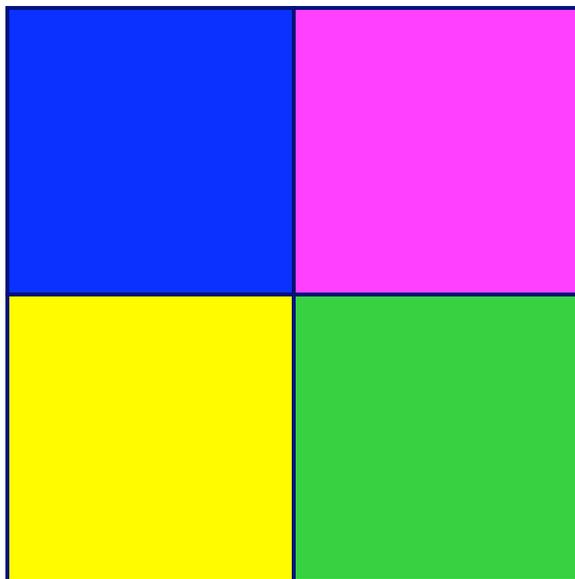
Sparse Matrix-Vector Multiply



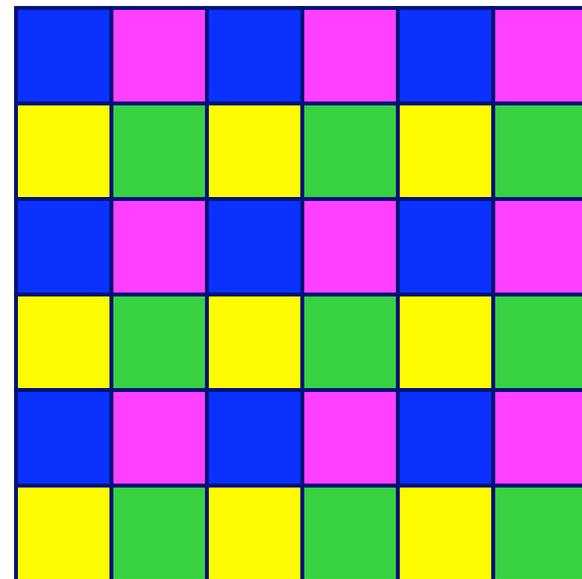
Block-cyclic Data Distributions



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)
```

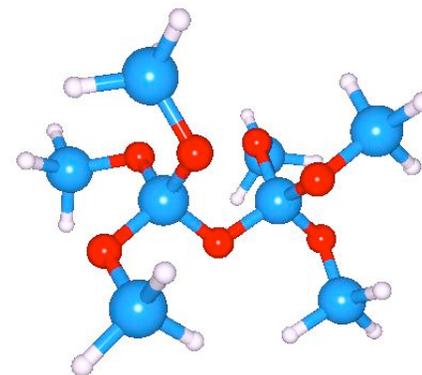

Hartree-Fock SCF



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

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

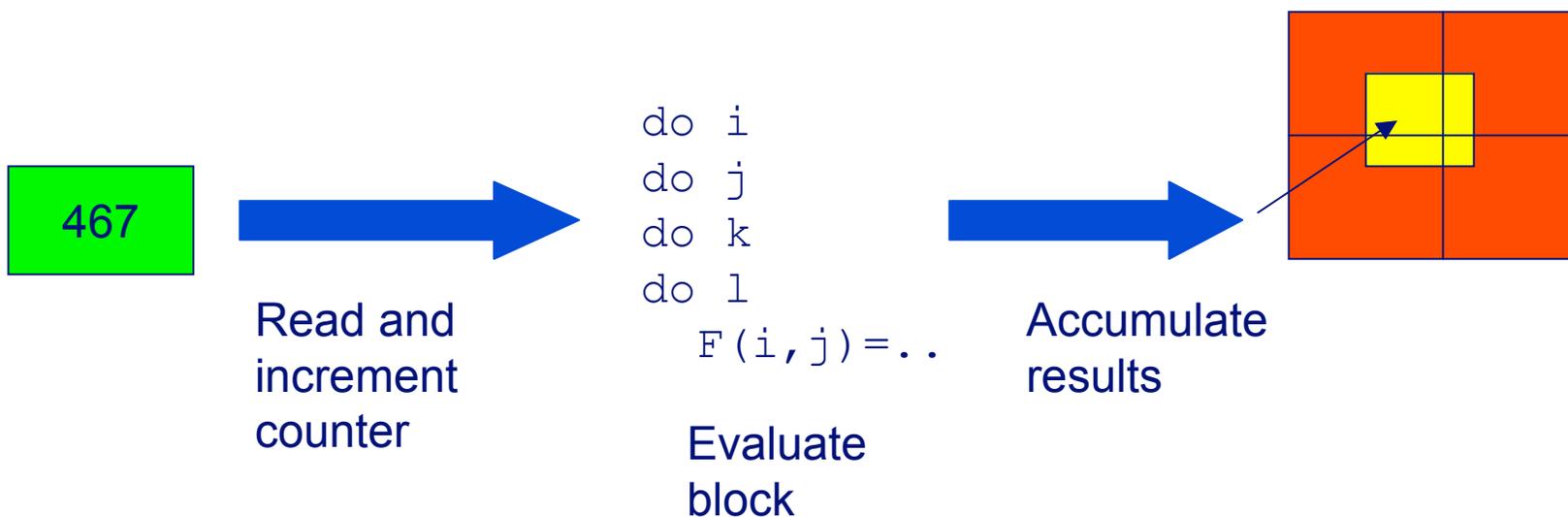
$$D_{\mu\nu} = \sum_k C_{\mu k} C_{\nu k}$$

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

Parallelizing the Fock Matrix



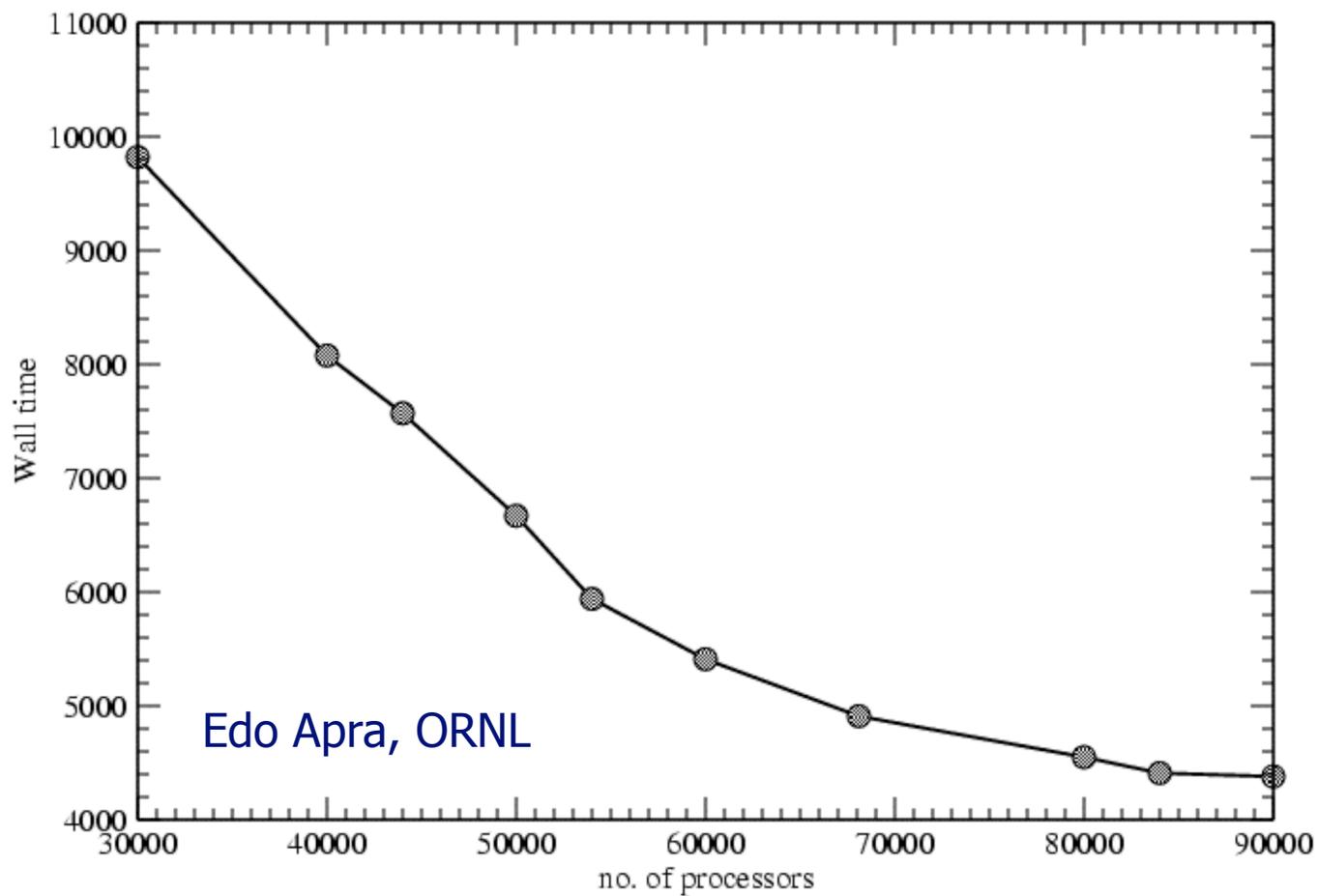
The bulk of the work involves computing the 4-index elements ($__|__$). This is done by decomposing the quadruple loop into evenly sized blocks and assigning blocks to each processor using a global counter. After each processor completes a block it increments the counter to get the next block



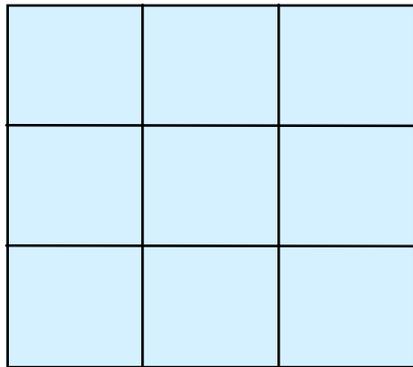
NWChem Scaling



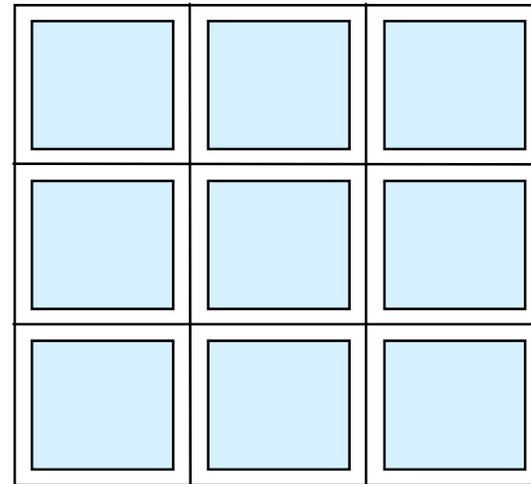
NWChem CCSD module



Ghost Cells



normal global array



global array with ghost cells

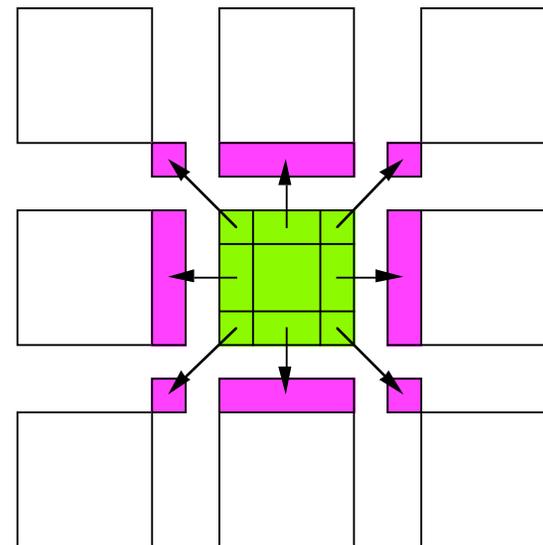
Operations:

- | | |
|---------------------|--|
| 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



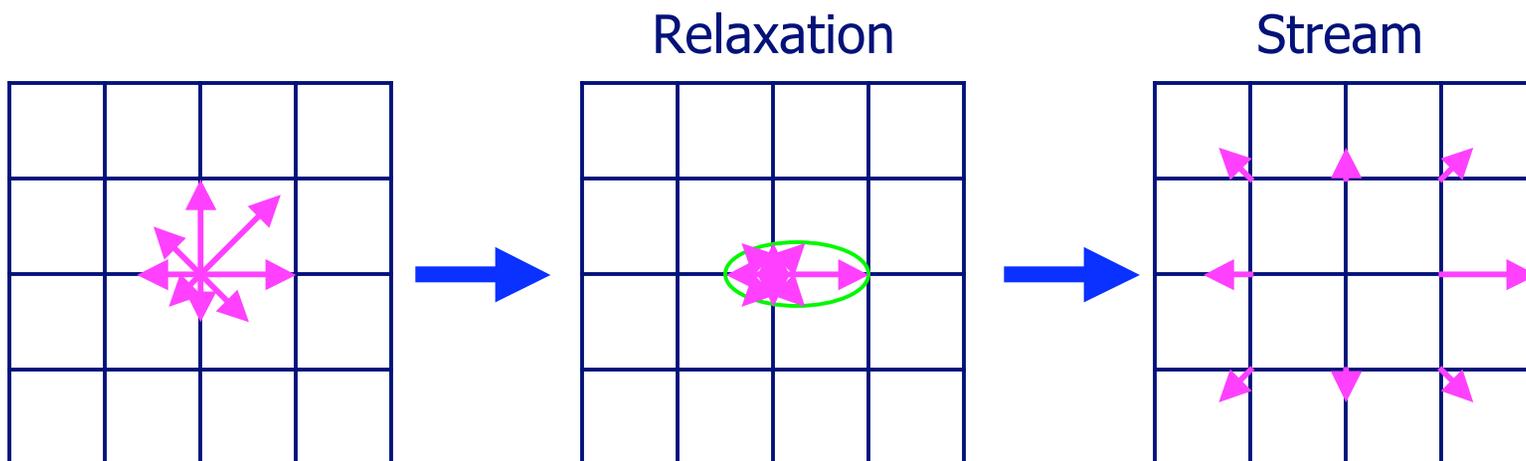
Automatically update ghost cells with appropriate data from neighboring processors. A multiprotocol implementation has been used to optimize the update operation to match platform characteristics.



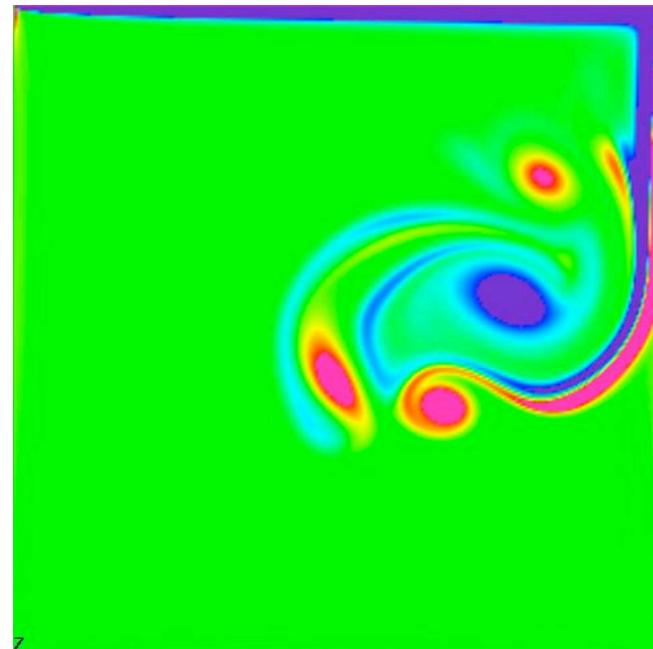
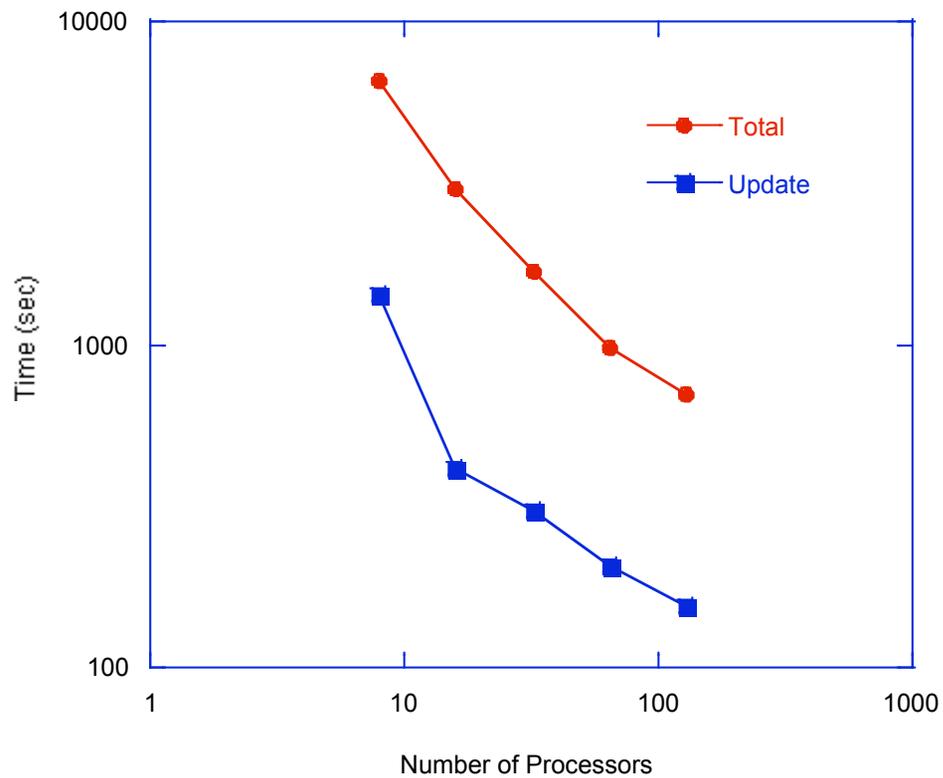
Lattice Boltzmann Simulation



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



Lattice Boltzmann Performance



ScalaBLAST

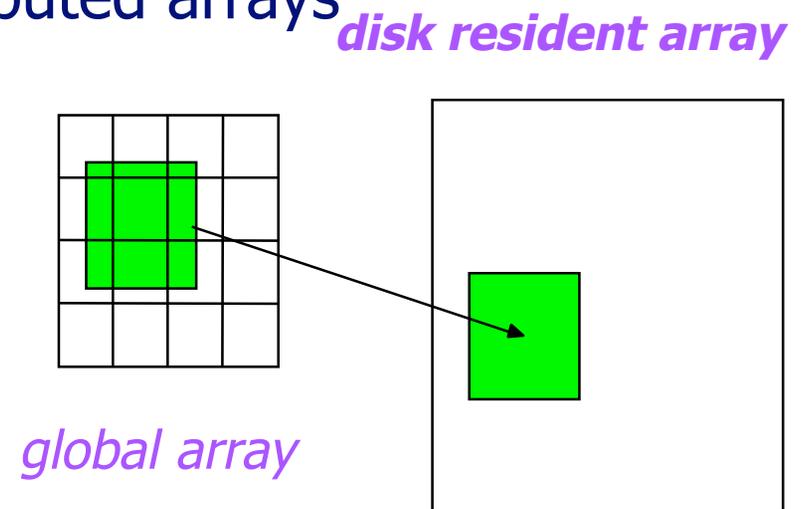


- ScalaBLAST is for doing high-throughput BLAST calculations in a cluster or supercomputer.
- ScalaBLAST divides the collection of queries over available processors
 - Proportional speedup on a few processors or on thousands
 - Efficient on commodity clusters or on high-end machines
- Deals with constantly growing database size by distributing one copy of database across processors using a single Global Array



Disk Resident Arrays

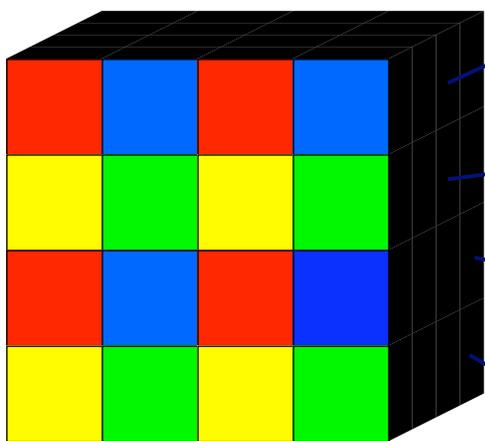
- Extend GA model to disk
 - system similar to Panda (U. Illinois) but higher level APIs
- Provide easy transfer of data between N-dim arrays stored on disk and distributed arrays stored in memory
- Use when
 - Arrays too big to store in core
 - checkpoint/restart
 - out-of-core solvers



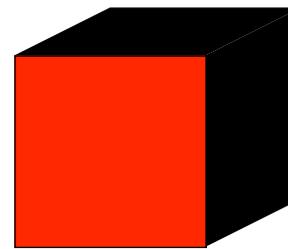
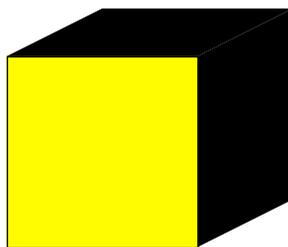
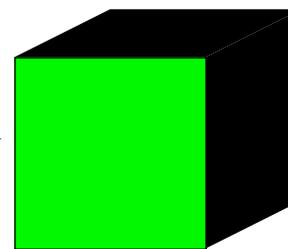
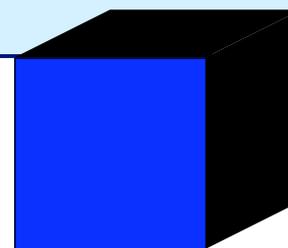
High Bandwidth Read/Write



Disk Resident Array



Disk Resident Arrays automatically decomposed into multiple files



Disks

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
- Autoconfig Builds
- Fault Tolerance
- Data Decomposition and Load balancing
- Support for Hybrid Platforms
- Performance tools for GA/ARMCI

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 4.2 available
- Homepage at <http://www.emsl.pnl.gov/docs/global/>
- Platforms (32 and 64 bit)
 - IBM SP, BlueGene/L, BlueGene/P
 - Cray X1, XD1, XT3, XT4
 - Linux Cluster with Ethernet, Myrinet, Infiniband, or Quadrics
 - HP
 - SGI Altix
 - Solaris
 - Fujitsu
 - Windows

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()
```

Use 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`