High Performance Computing in Python using NumPy and the Global Arrays Toolkit

Jeff Daily[1]

P. Saddayappan[2], Bruce Palmer[1], Manojkumar Krishnan[1],
Sriram Krishnamoorthy[1], Abhinav Vishnu[1], Daniel Chavarría[1],
Patrick Nichols[1]

[1]Pacific Northwest National Laboratory
[2]Ohio State University

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by Battelle *Since 1965*

Outline of the Tutorial

- Parallel Programming Models
  - Performance vs. Abstraction vs. Generality
  - Distributed Data vs. Shared Memory
  - One-sided communication vs. Message Passing
- Overview of the Global Arrays Programming Model
- Intermediate GA Programming Concepts and Samples
- Advanced GA Programming Concepts and Samples
- Global Arrays in NumPy (GAiN)

Parallel Programming Models

- Single Threaded
  - Data Parallel, e.g. HPF
- Multiple Processes
  - Partitioned-Local Data Access
    - MPI
  - Uniform-Global-Shared Data Access
    - OpenMP
  - Partitioned-Global-Shared Data Access
    - Co-Array Fortran
  - Uniform-Global-Shared + Partitioned Data Access
    - UPC, Global Arrays, X10

Parallel Programming Models in Python

- Single Threaded
  - Data Parallel, e.g. HPF
- Multiple Processes
  - Partitioned-Local Data Access
    - MPI (mpi4py)
  - Uniform-Global-Shared Data Access
    - OpenMP (within a C extension – no direct Cython support yet)
  - Partitioned-Global-Shared Data Access
    - Co-Array Fortran
  - Uniform-Global-Shared + Partitioned Data Access
    - UPC, Global Arrays (as of 5.0.x), X10
- Others: PyZMQ, IPython, PiCloud, and more

High Performance Fortran

- Single-threaded view of computation
- Data parallelism and parallel loops
- User-specified data distributions for arrays
- Compiler transforms HPF program to SPMD program
  - Communication optimization critical to performance
- Programmer may not be conscious of communication implications of parallel program

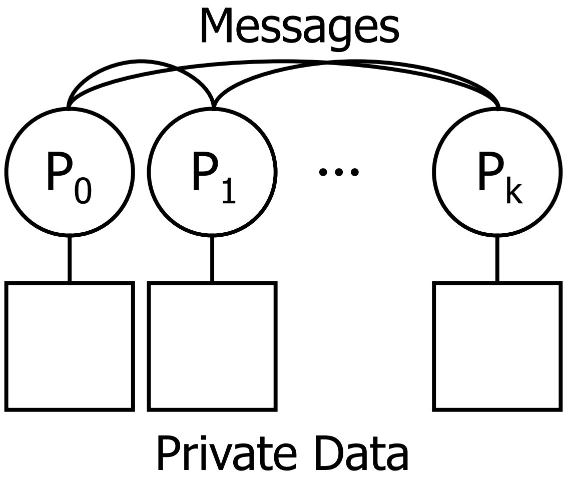
```
HPF$ Independent
DO I = 1,N
HPF$ Independent
    DO J = 1,N
         A(I,J) = B(J,I)
    END
END
```

```
HPF$ Independent
DO I = 1,N
HPF$ Independent
   DO J = 1,N
      A(I,J) = B(I,J)
   END
END
```

```
s=s+1
A(1:100) = B(0:99)+B(2:101)
HPF$ Independent
Do I = 1,100
    A(I) = B(I-1)+B(I+1)
End Do
```

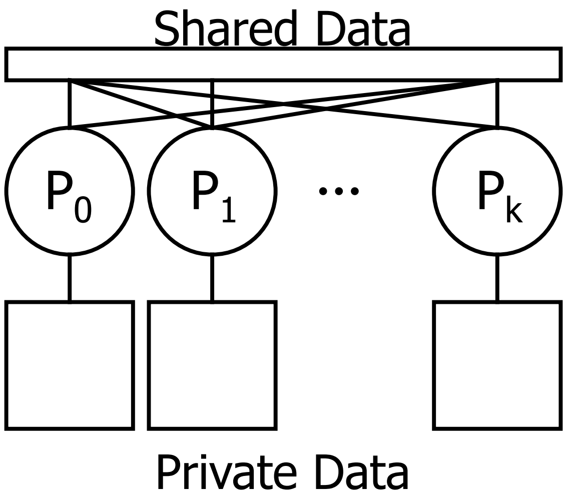
Message Passing Interface

- Most widely used parallel programming model today
- Bindings for Fortran, C, C++, MATLAB
- P parallel processes, each with local data
  - MPI-1: Send/receive messages for inter-process communication
  - MPI-2: One-sided get/put data access from/to local data at remote process
- Explicit control of all inter-processor communication
  - Advantage: Programmer is conscious of communication overheads and attempts to minimize it
  - Drawback: Program development/debugging is tedious due to the partitioned-local view of the data



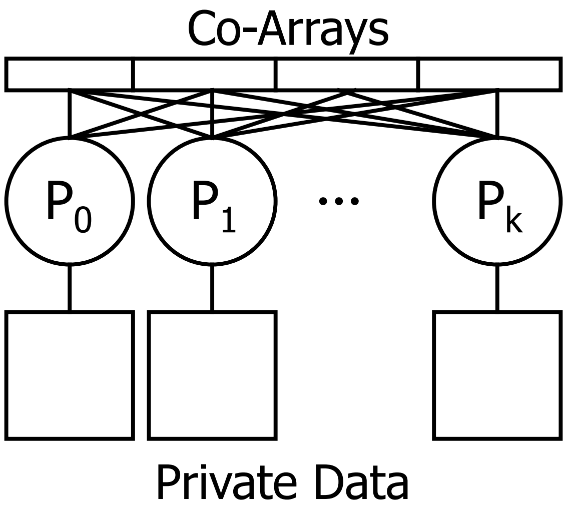
OpenMP

- Uniform-Global view of shared data
- Available for Fortran, C, C++
- Work-sharing constructs (parallel loops and sections) and global-shared data view ease program development
- Disadvantage: Data locality issues obscured by programming model



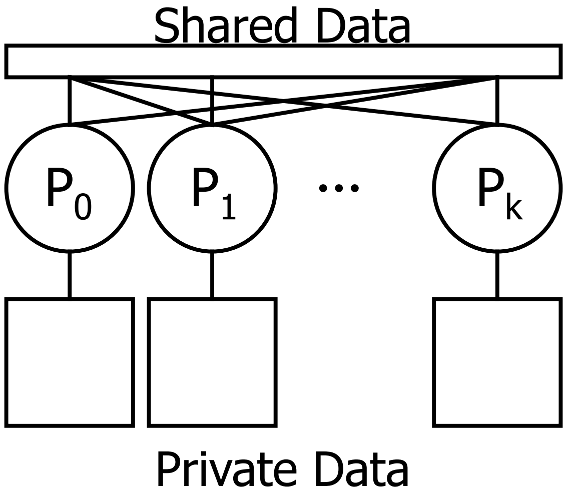
Co-Array Fortran

- Partitioned, but global-shared data view
- SPMD programming model with local and shared variables
- Shared variables have additional co-array dimension(s), mapped to process space; each process can directly access array elements in the space of other processes
  - A(I,J) = A(I,J)[me-1] + A(I,J)[me+1]
- Compiler optimization of communication critical to performance, but all non-local access is explicit



Unified Parallel C (UPC)

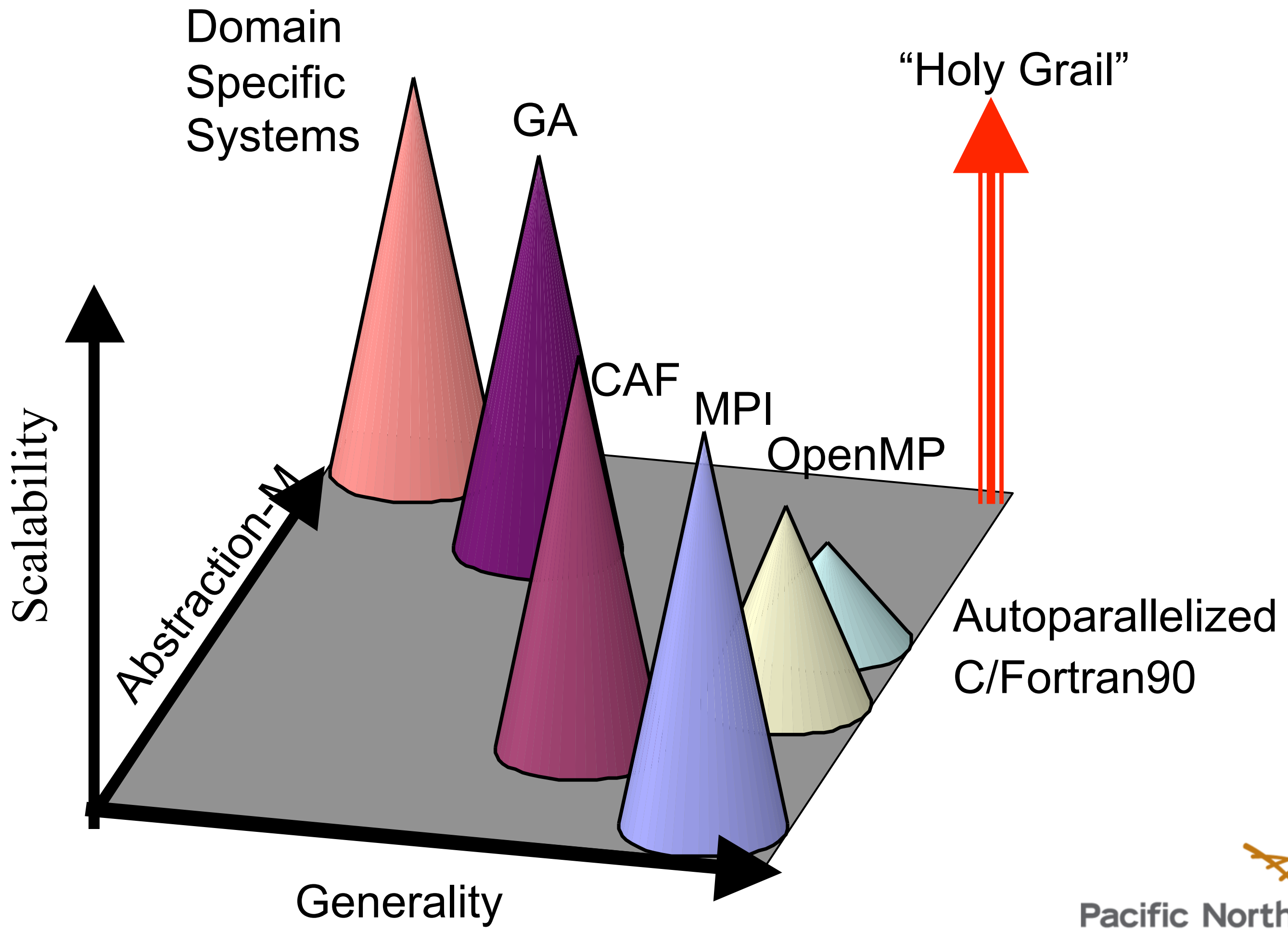
- SPMD programming model with global shared view for arrays as well as pointer-based data structures
- Compiler optimizations critical for controlling inter-processor communication overhead
  - Very challenging problem since local vs. remote access is not explicit in syntax (unlike Co-Array Fortran)
  - Linearization of multidimensional arrays makes compiler optimization of communication very difficult
- Performance study with NAS benchmarks (PPoPP 2005, Mellor-Crummey et. al.) compared CAF and UPC
  - Co-Array Fortran had significantly better scalability
  - Linearization of multi-dimensional arrays in UPC was a significant source of overhead



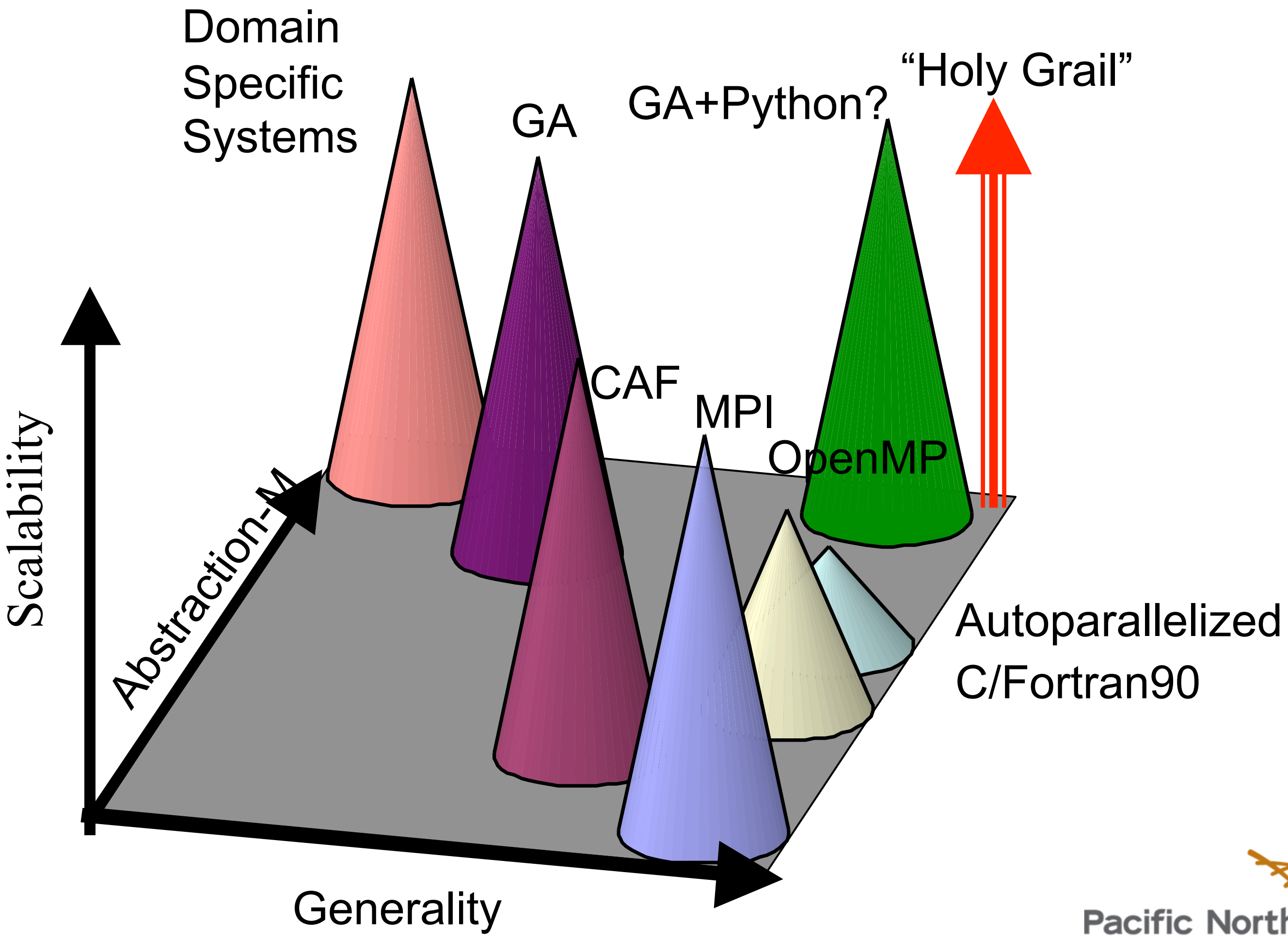
Global Arrays vs. Other Models

- 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:
  - Only useable for array data structures

Performance vs. Abstraction and Generality

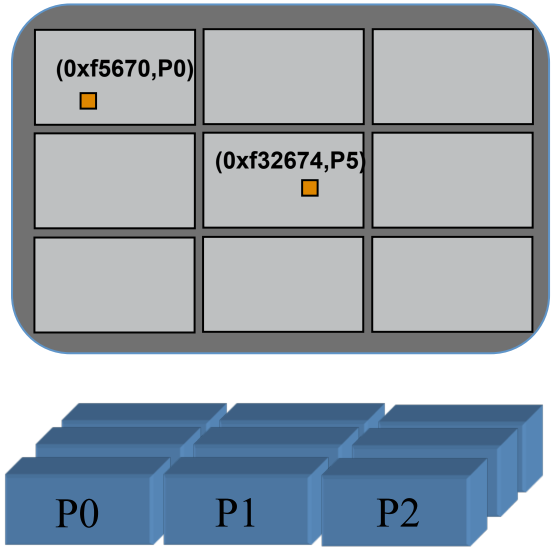


Performance vs. Abstraction and Generality



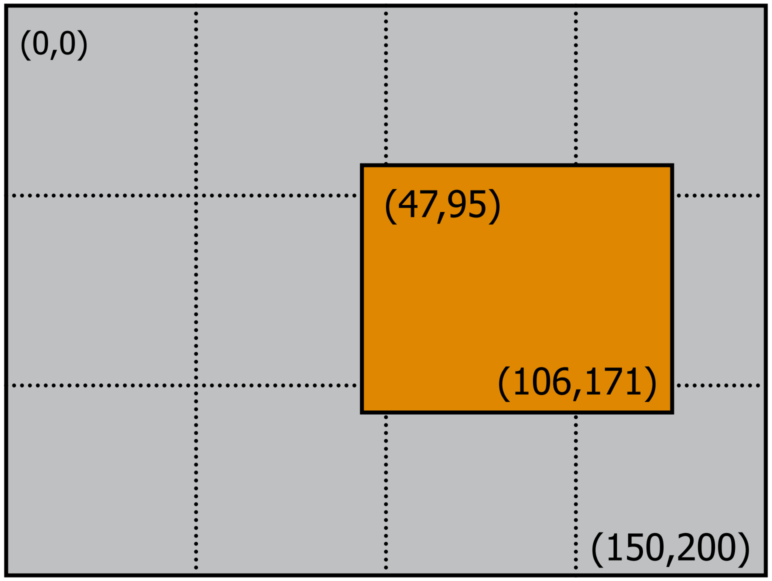
Distributed Data vs Shared Memory

- 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)
  - To copy element from P5 to P0 using MPI
    - P0 posts comm.recv(obj, 5)
    - P5 posts comm.send(buf[27], 5)

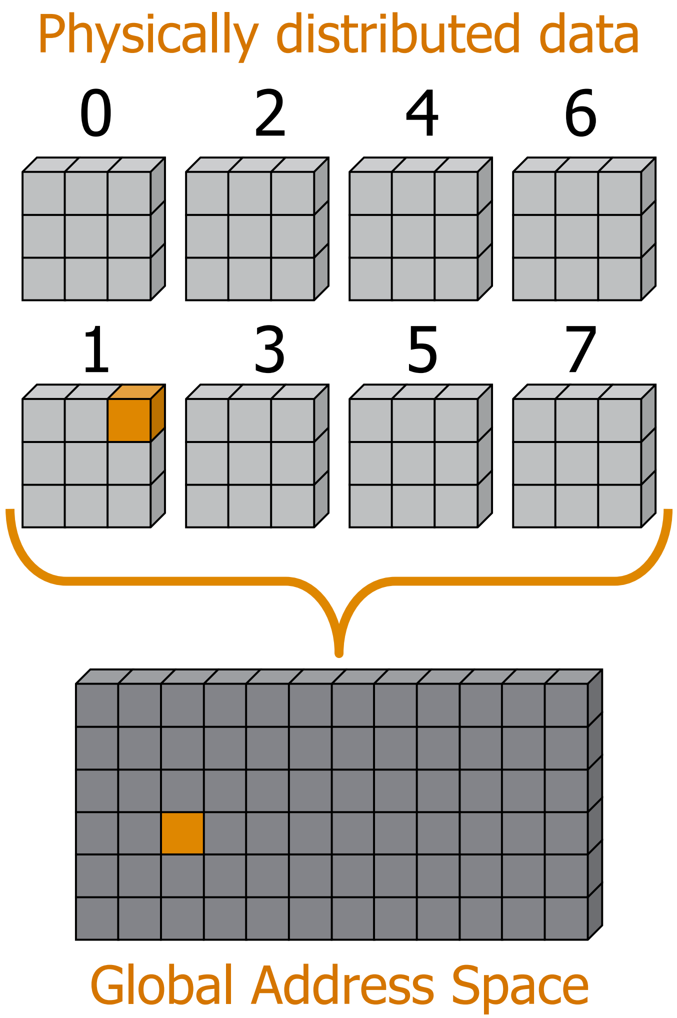


Distributed Data vs Shared Memory (cont.)

- 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.

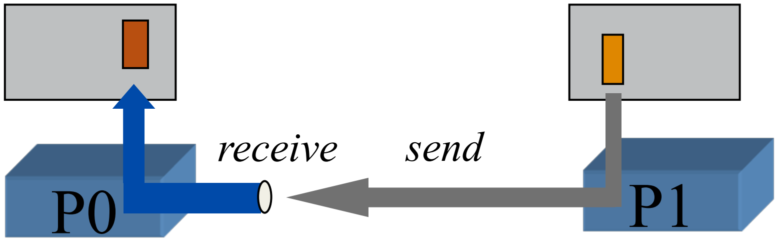


Global Arrays

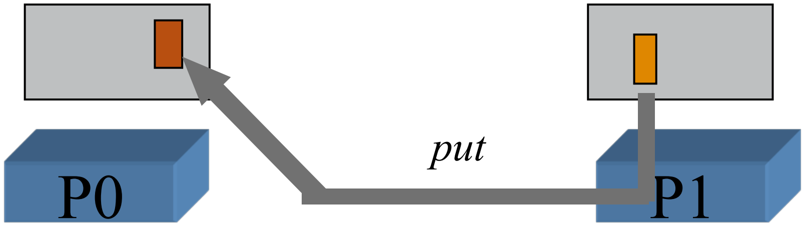


- Distributed dense arrays that can be accessed through a shared memory-like style
- single, shared data structure/ global indexing
  - e.g., `ga.get(a, (3,2))` rather than buf[6] on process 1

One-sided Communication



message passing
MPI



one-sided communication
SHMEM, ARMCI, MPI-2-1S

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
```

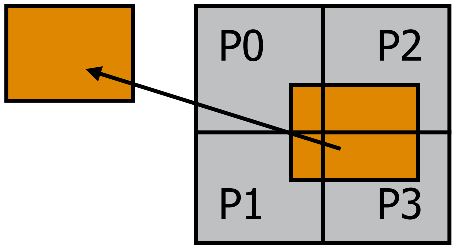
Global Arrays:

buf=ga.get(g_a, lo=None, hi=None, buffer=None)

Global Array handle

Global upper and lower indices of data patch

Local ndarray buffer



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

Outline of the Tutorial

- Parallel Programming Models
- Overview of the Global Arrays Programming Model
  - **Downloading, Building GA using `configure && make`**
  - 10 Basic GA Commands
  - GA Models for Computation
- Intermediate GA Programming Concepts and Samples
- Advanced GA Programming Concepts and Samples
- Global Arrays in NumPy (GAiN)

Source Code and More Information

- Version 5.0.3 available, trunk to become 5.1
- Homepage at http://www.emsl.pnl.gov/docs/global/
- Platforms
  - IBM SP, BlueGene
  - Cray XT, XE6 (Gemini)
  - Linux Cluster with Ethernet, Myrinet, Infiniband, or Quadrics
  - Solaris
  - Fujitsu
  - Hitachi
  - NEC
  - HP
  - Windows

Writing and Running GA programs

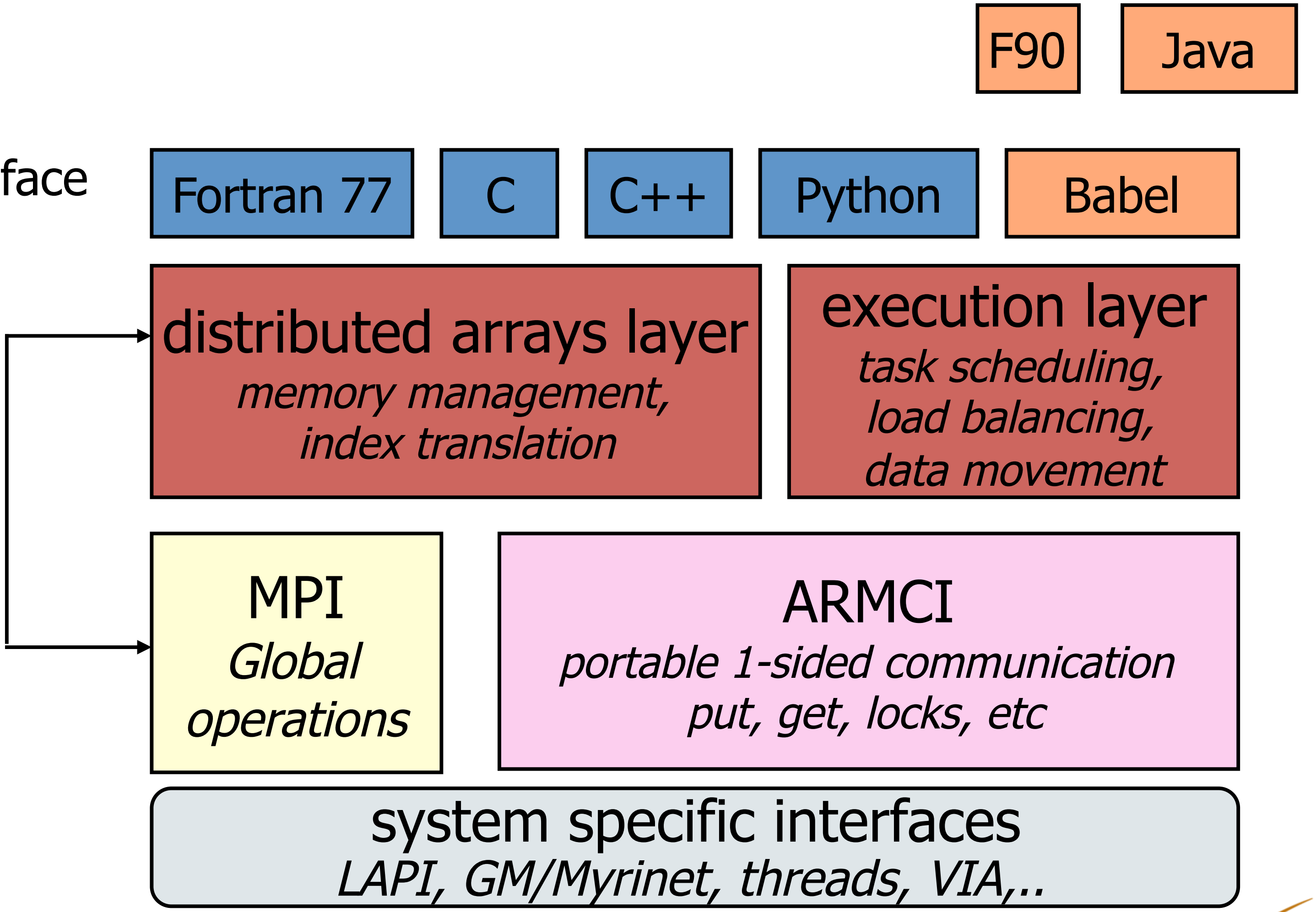
- Topics to cover so that we can all start programming!
  - Installing GA
  - Writing GA programs
  - Running GA programs

Writing and Running GA programs (cont.)

- GA Webpage
  - http://www.emsl.pnl.gov/docs/global/
  - GA papers, APIs, user manual, etc.
  - Google: Global Arrays
- GA API Documentation
  - GA Webpage, click on “User Interface”
  - http://www.emsl.pnl.gov/docs/global/userinterface.html
- GA Support/Help/Announcements
  - hpctools@googlegroups.com

Structure of GA

Application programming language interface



F90

Java

Fortran 77

C

C++

Python

Babel

distributed arrays layer

memory management, index translation

execution layer

task scheduling, load balancing, data movement

MPI

Global operations

ARMCI

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

system specific interfaces

LAPI, GM/Myrinet, threads, VIA,..

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

Installing GA

- GA 5.0 uses autotools (`configure && make && make install`) for building
  - 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. Infiniband: `configure --with-openib`
    - Best guess: `configure --enable-autodetect`
  - GA requires MPI for basic start-up and process management
    - MPI is the default, searches for MPI compilers e.g. mpicc, mpif90
- Various `make` targets
  - `make` to build GA libraries
  - `make install` to install libraries
  - `make checkprogs` to build C/Fortran tests and examples
  - `make check MPIEXEC="mpiexec -np 4"` to run test suite
- VPATH builds: one source tree, many build trees i.e. configurations

```
tar -xzf ga-5-0-3.tgz; cd ga-5-0-3
mkdir bld; cd bld; ../configure; make
```

Installing GA for Python

- GA requires MPI for basic start-up and process management
  - MPI is the default: `configure`
  - MPI compilers are searched for by default e.g. `mpicc`
- Need to enable shared libraries: `--enable-shared`
- Build it: `make && make python`
  - Installs GA libs/headers, runs setup.py build and install
- Python bindings always built from top-level source tree

Outline of the Tutorial

- Parallel Programming Models
- Overview of the Global Arrays Programming Model
  - Downloading, Building GA using `configure && make`
  - **10 Basic GA Commands**
  - GA Models for Computation
- Intermediate GA Programming Concepts and Samples
- Advanced GA Programming Concepts and Samples
- Global Arrays in NumPy (GAiN)

GA Basic Operations

- GA programming model is very simple
- Most 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.acc(…)`
  - `ga.sync()`
- We cover these and more in the next slides

GA Initialization/Termination

- For Python, there is only `import ga`
- To set maximum limit for GA memory, use `ga.set_memory_limit(limit)`
- For Python, GA termination happens during `atexit()`

Where to Find the Tutorial Code

- From the top level GA source directory
  - ./python/tutorial
- Don’t look at the answers!
  - e.g. matrix.answer.py instead of matrix.py
- Some programs serve as a sample, some as a problem
  - hello.py, hello2.py already work
  - matrix.py, transpose.py require fixing by you

Running First GA Program – Hello World

- Requires MPI
  - Needs a process manager
  - Also certain collective operations
- import ga
  - C’s `GA_Initialize()` called
  - C’s `GA_Terminate()` registered with `atexit()`
- Single Program, Multiple Data

```
$ mpiexec -np 4 python hello.py
Hello World!
Hello World!
Hello World!
Hello World!
```

```
# file: hello.py
import mpi4py.MPI # initialize Message Passing Interface
import ga # initialize Global Arrays
print “Hello World!”
```

To Run:

```
mpiexec -np 4 python tutorial/hello.py
```

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:
  - `nodeid = ga.nodeid()`
- To determine the number of computing processes:
  - `nnodes = ga.nnodes()`

Hello World with Process Information

```
$ mpiexec -np 4 python hello2.py
hello from 0 out of 4
hello from 2 out of 4
hello from 3 out of 4
hello from 1 out of 4
```

```
# file: hello.py
import mpi4py.MPI # initialize Message Passing Interface
import ga # initialize Global Arrays
print “Hello from %s of %s” % (ga.nodeid(),ga.nnodes())
```

To Run:

```
mpiexec -np 4 python tutorial/hello2.py
```

GA Data Types

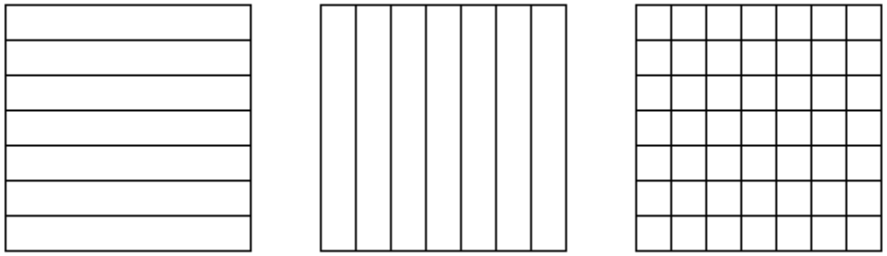
- C/Python Data types
  - `C_INT` - int
  - `C_LONG` - long
  - `C_LONGLONG` - long long
  - `C_FLOAT` - float
  - `C_DBL` - double
  - `C_SCPL` - single complex
  - `C_DCPL` - double complex
- Fortran Data types (don’t use these for Python)
  - `F_INT` - integer (4/8 bytes)
  - `F_REAL` - real
  - `F_DBL` - double precision
  - `F_SCPL` - single complex
  - `F_DCPL` - double complex

Creating Arrays

To *create* an array with a regular distribution:

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

string	name	- a unique character string	[input]
integer	type	- GA data type	[input]
integer	dims()	- array dimensions	[input]
integer	chunk()	- minimum size that dimensions should be chunked into	[input]
integer	g_a	- array handle for future references	[output]



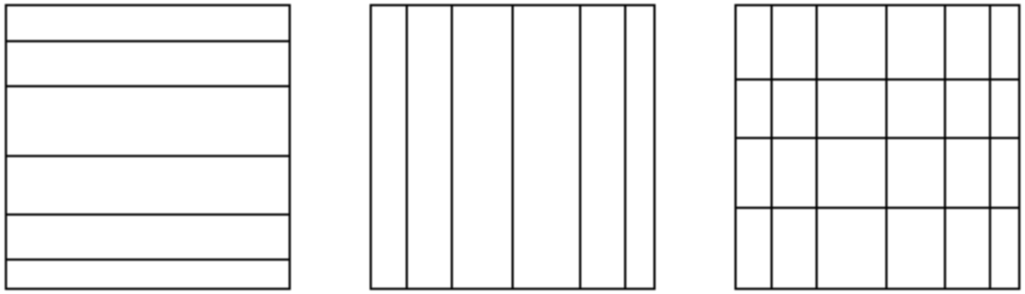
```
g_a = ga.create(ga.C_DBL, [5000,5000], “Array_A”)
if not g_a:a
    ga.error(“Could not create global array A”, g_a)
```

Creating Arrays with Irregular Distributions

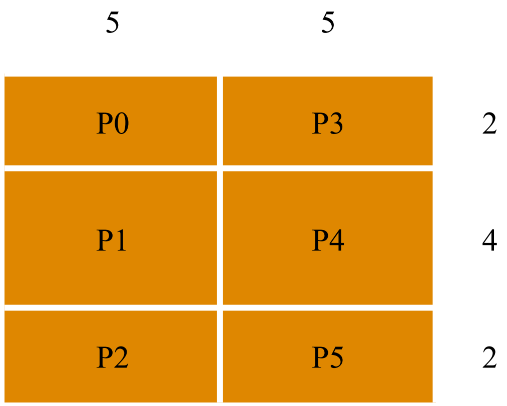
To *create* an array with an irregular distribution:

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

string	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]



Irregular Distributions Explained



- Example of irregular distribution:
  - The distribution is specified as a Cartesian product of distributions for each dimension. The array indices start at 0.
    - The figure demonstrates distribution of an 8x10 array on 6 (or more) processors
      - *block*=[3,2]
      - map = [0,2,6,0,5]; len(map) = 5
    - The distribution is nonuniform because, P1 and P4 get 20 elements each and processors P0,P2,P3, and P5 only 10 elements each.

```
block = [3,2]
map = [0,2,6,0,5]
g_a = ga.create_irreg(ga.C_DBL, [8,10], “Array A”, block, map)
if not g_a:
    ga.error(“Could not create global array A”,g_a)
```

Duplicating and Destroying Arrays

To *duplicate* an array:

```
g_b = ga.duplicate(g_a, name=“”)
```

Creates a new array by applying all properties of given array to the new array.

Global arrays can be *destroyed* by calling the function:

```
ga.destroy(g_a)
```

```
g_a = ga_create(ga.C_INT, [200,300])
g_b = ga_duplicate(g_a)
ga.destroy(g_a)
```

Put/Get

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

```
ga.put(g_a, buffer, lo=None, hi=None)
```

integer	g_a	global array handle	[input]
integer	lo(),hi()	limits on data block to be moved	[input]
double/complex/integer	buf	local buffer	[input]

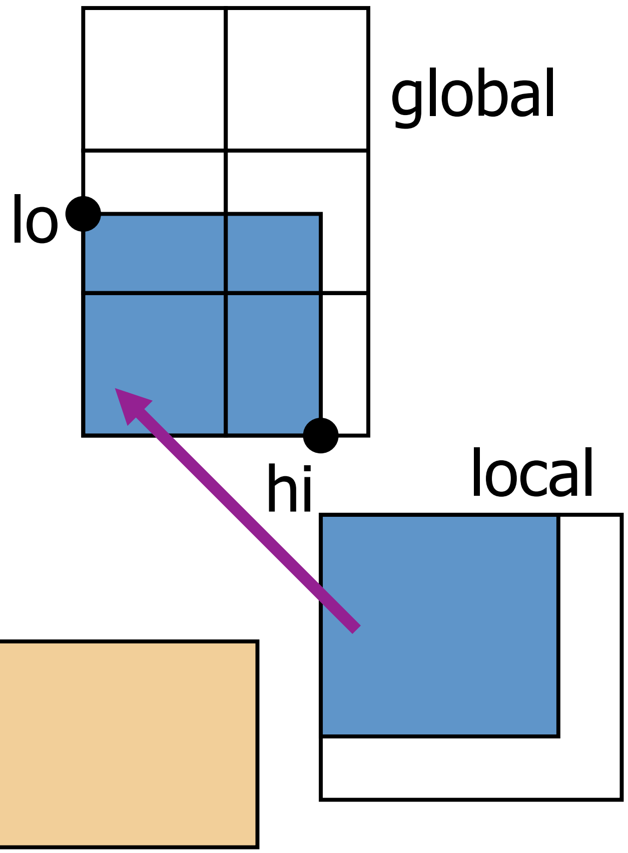
Get copies data from a global array section to a local array:

```
buffer = ga.get(g_a, lo=None, hi=None, buffer=None)
```

integer	g_a	global array handle	[input]
integer	lo(),hi()	limits on data block to be moved	[input]
double/complex/integer	buf	local buffer	[output]

Put/Get (cont.)

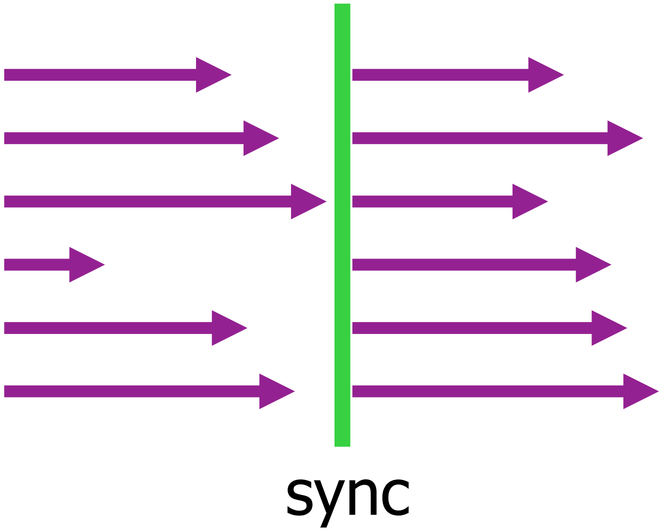
- Example of *put* operation:
  - local buffer must be either 1D contiguous or same shape as lo/hi patch
  - Here: local array sliced to 9x9 patch, put to 18x12 global array



```
buf = numpy.arange(15*15).reshape(15,15)
ga.put(g_a, buf[:9,:9], (9,0), (18,9))
```

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
- `ga.sync()`



Locality Information

Discover array elements held by each processor

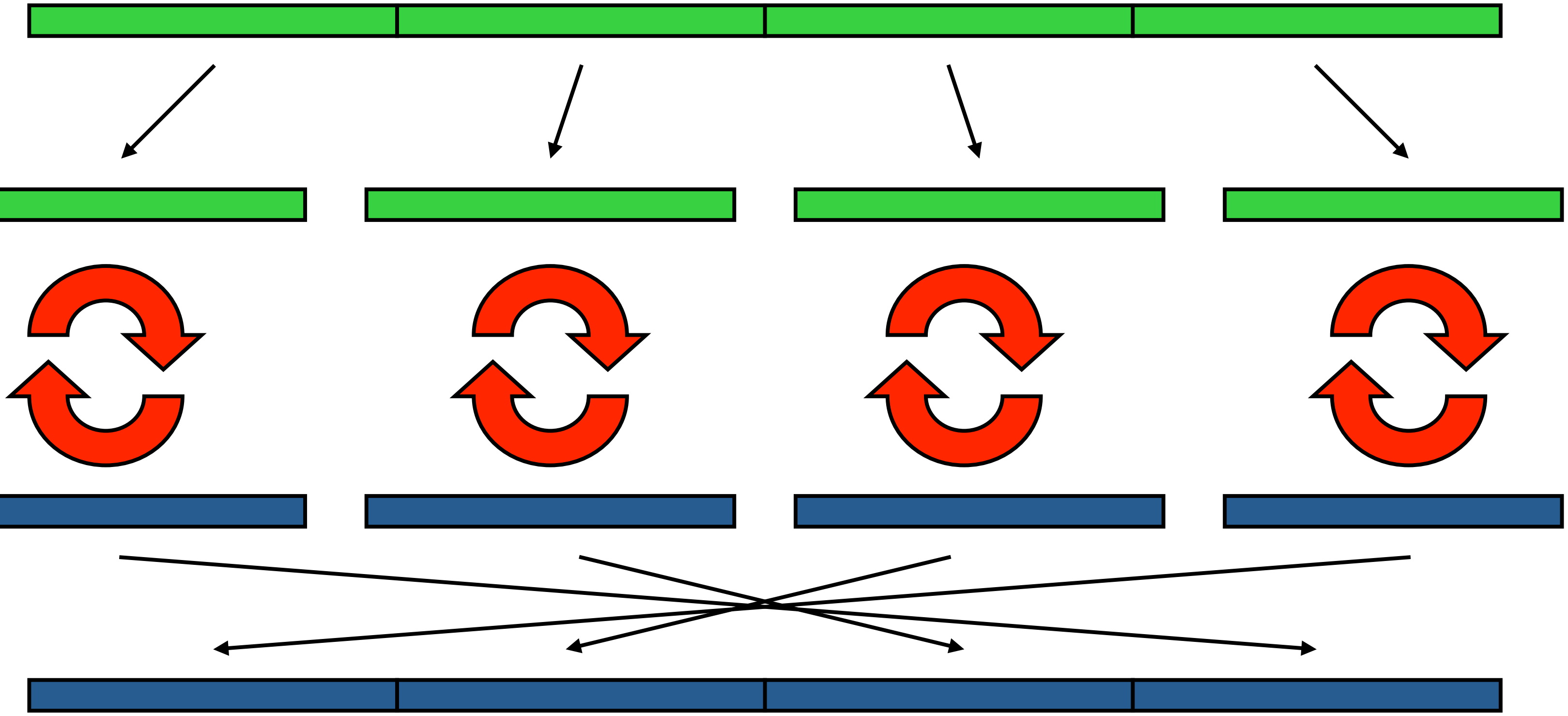
```
lo,hi = ga.distribution(g_a, proc=-1)
```

integer	g_a	array handle	[input]
integer	proc	processor ID	[input]
integer	lo(ndim)	lower index	[output]
integer	hi(ndim)	upper index	[output]

Follows Python half-open convention – lo is inclusive, hi is exclusive

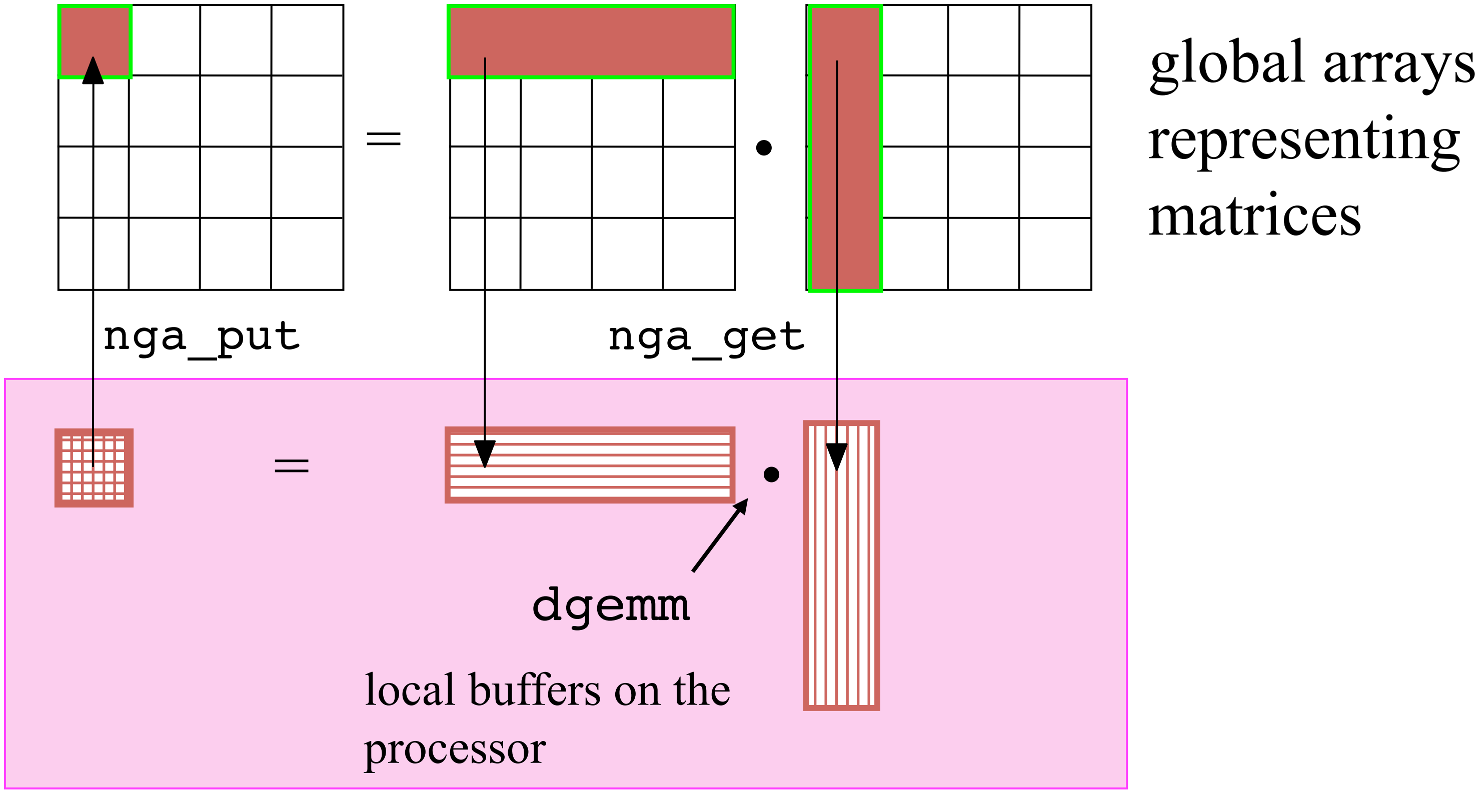
```
def print_distribution(g_a):
    for i in range(ga.nnodes()):
        print "Printing g_a info for processor", i
        lo,hi = ga.distribution(g_a, i)
        print "%s lo=%s hi=%s" % (i,lo,hi)
```

Example: 1-D Transpose (transp1D.py)



You now know enough for your first *real* application!

Example: Matrix Multiply (matrix.py)

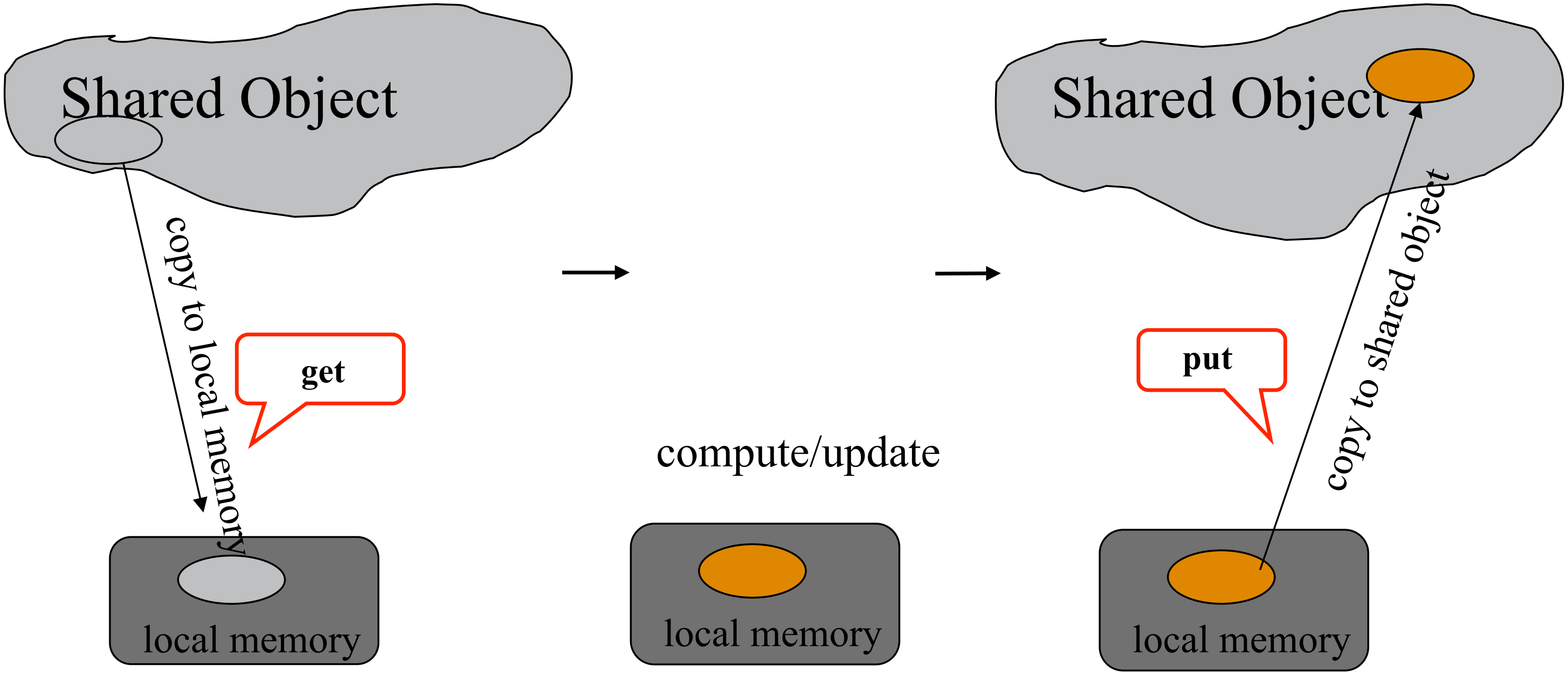

=
•
global arrays representing matrices
nga_put
nga_get
=
•
dgemm
local buffers on the processor

You now know enough for your second *real* application!

Outline of the Tutorial

- Parallel Programming Models
- Overview of the Global Arrays Programming Model
  - Downloading and Building GA using `configure && make`
  - 10 Basic GA Commands
  - **GA Models for Computation**
- Intermediate GA Programming Concepts and Samples
- Advanced GA Programming Concepts and Samples
- Global Arrays in NumPy (GAiN)

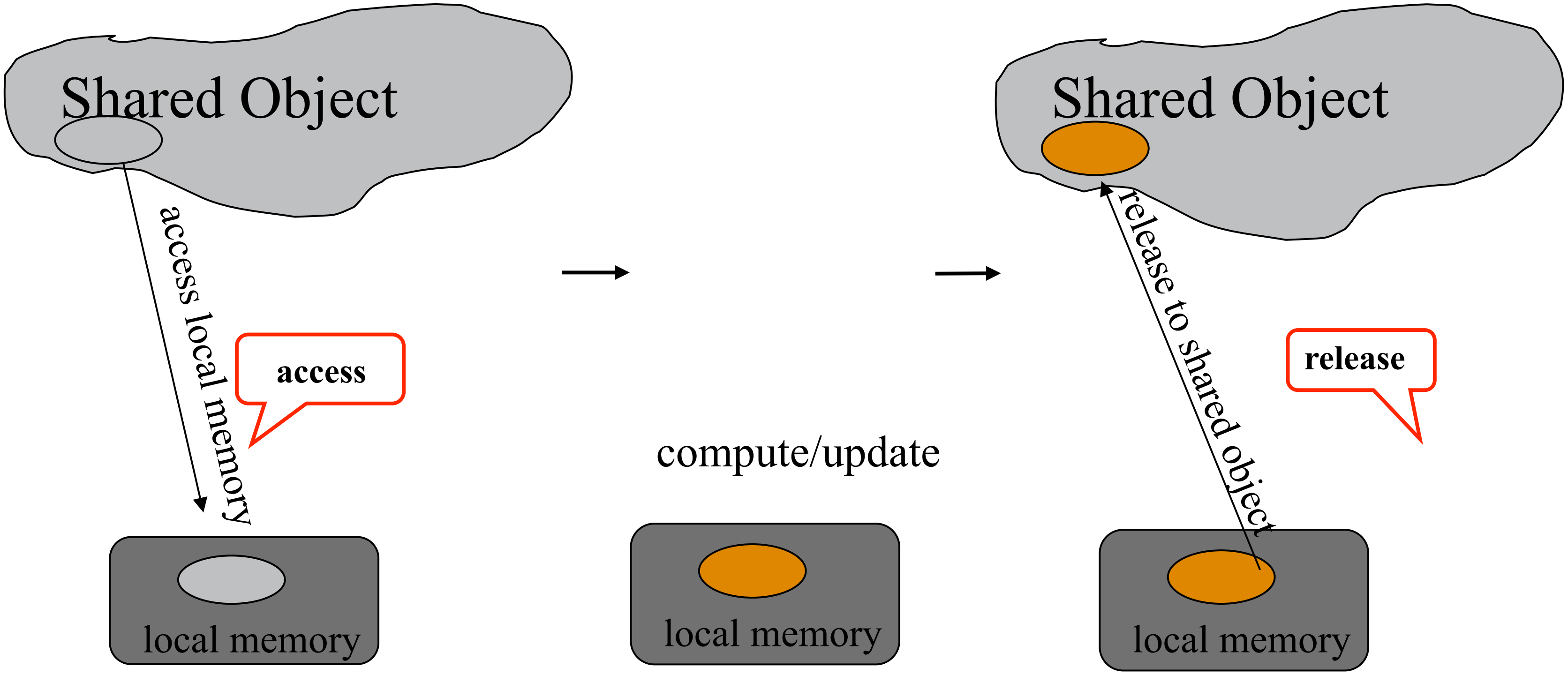
GA Model of Computations: Get/Put



- Shared memory view for distributed dense arrays
- Get-Local/Compute/Put-Global model of computation
- MPI-Compatible
- Data locality and granularity control similar to message passing model

Pacific Northwest NATIONAL LABORATORY

GA Model of Computations: Access/Release



- Access-Local/Compute/Release-Global model of computation
- No communication!
- Be aware that other processes may be trying to get/put the same data

Pacific Northwest NATIONAL LABORATORY

Data Locality in GA

What data does a processor own?

```
lo,hi = ga.distribution(g_a, iproc=-1)
```

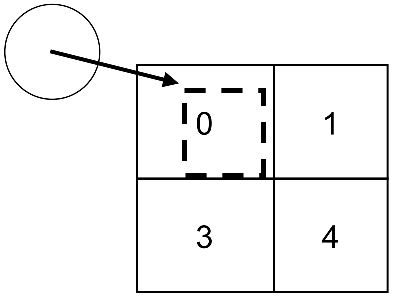
Where is the data?

```
data = ga.access(g_a, lo=None, hi=None, proc=-1)
```

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

Data Locality in GA (cont.)

- 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

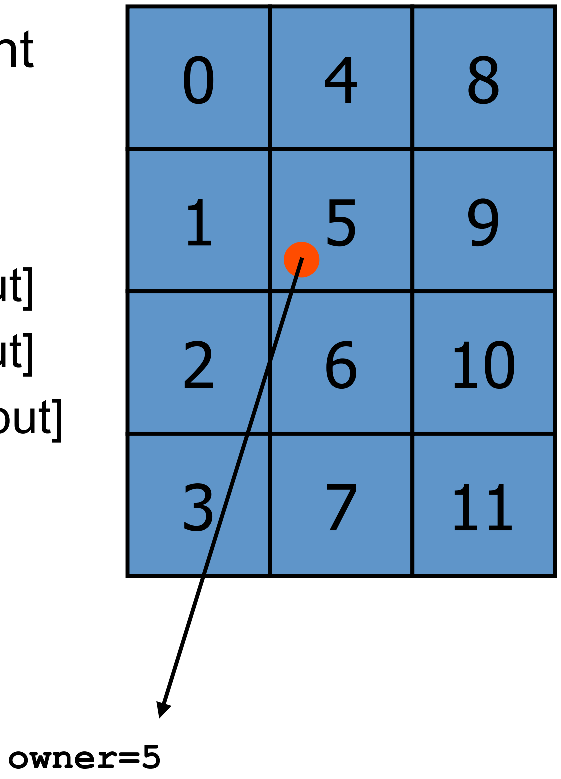


Locality Information

- To determine the process ID that owns the element defined by the array subscripts:

```
proc = ga.locate(g_a, subscript)
```

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

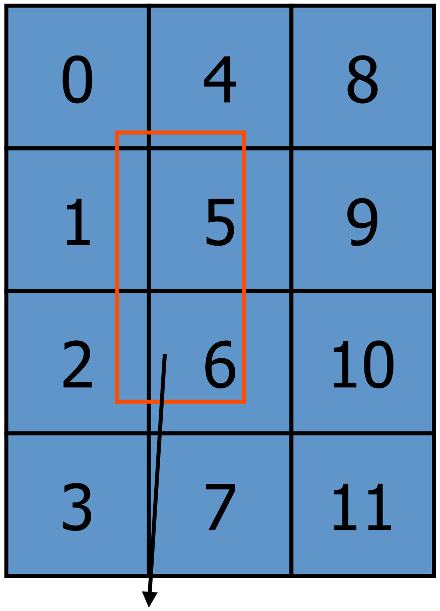


Locality Information (cont.)

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

```
map,procs = ga.locate_region(g_a, lo, hi)
```

integer	np	- number of processors that own a portion of block	[output]
integer	g_a	- global array handle	[input]
integer	ndim	- number of dimensions of the global array	
integer	lo(ndim)	- array of starting indices for array section	[input]
integer	hi(ndim)	- array of ending indices for array section	[input]
integer	map(2*ndim,*)	- array with mapping information	[output]
integer	procs(np)	- list of processes that own a part of array section	[output]



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}}

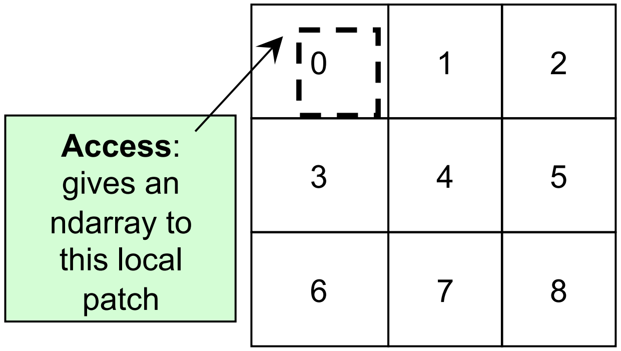
Access and Release

To provide direct access to local data in the specified patch of the array owned by the calling process:

```
buffer = ga.access(g_a, lo=None, hi=None, proc=-1)
```

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
- Defaults to entire local array
- ***Returns None if no local data***



Access: gives an ndarray to this local patch

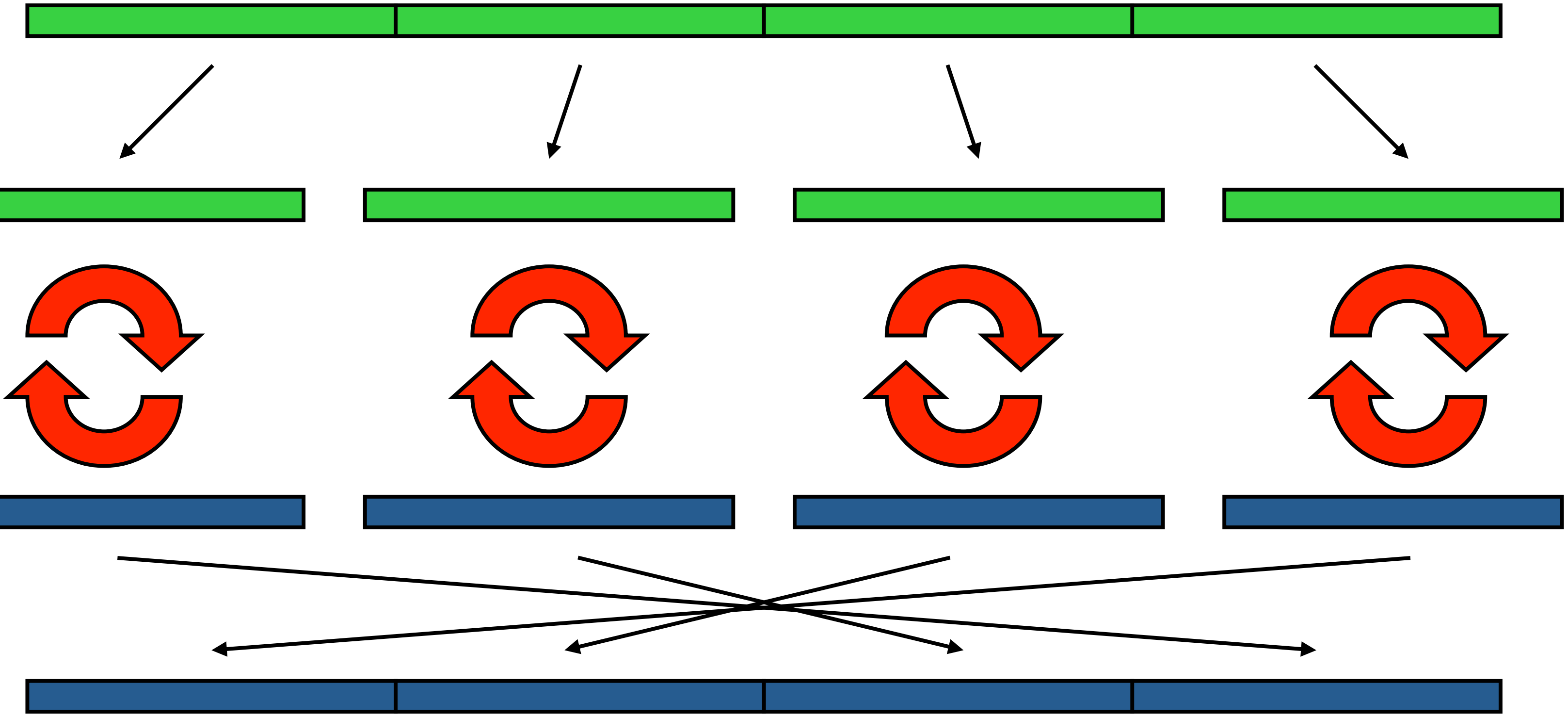
If not modified:

```
ga.release(g_a, lo=None, hi=None)
```

If modified:

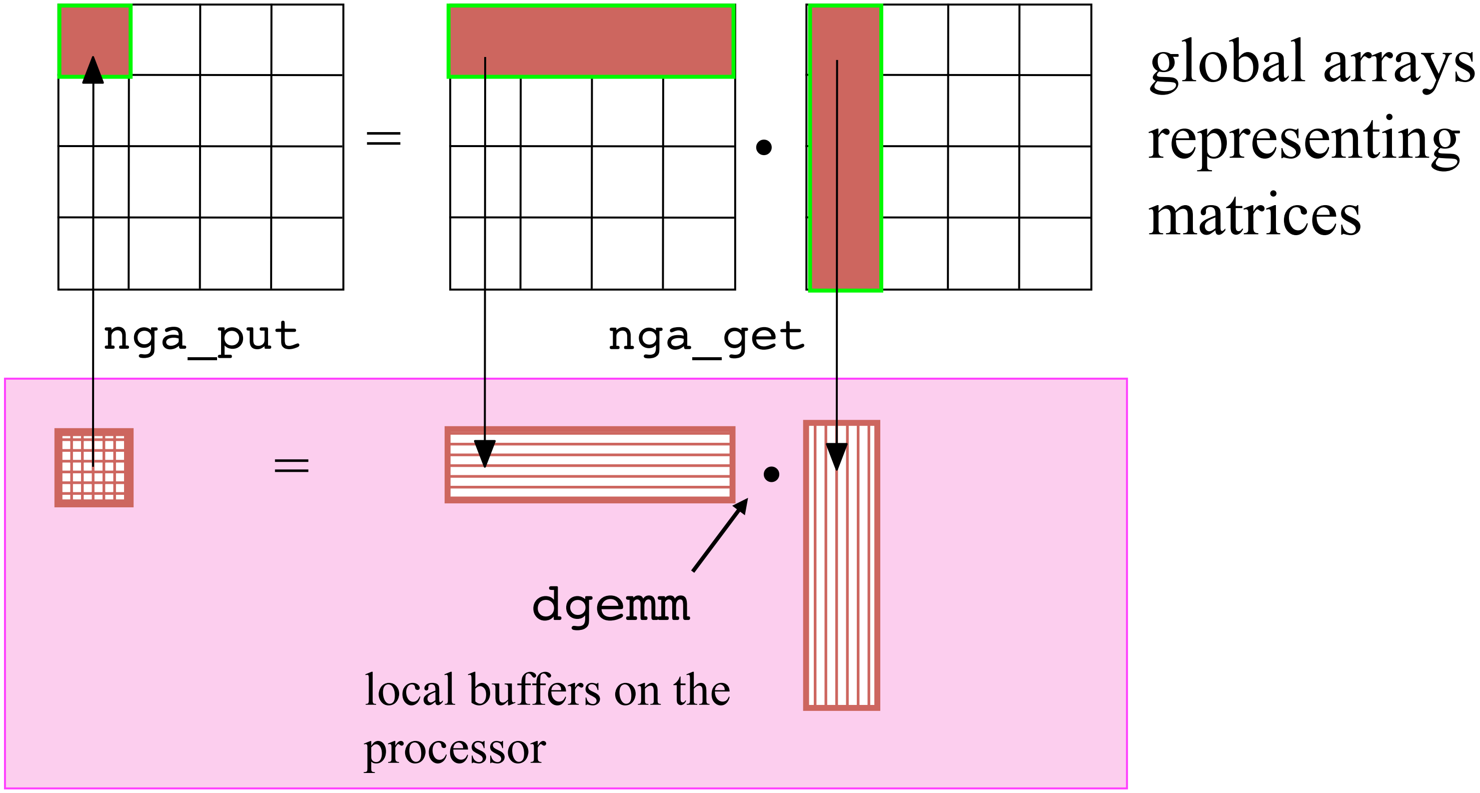
```
ga.release_update(g_a, lo=None, hi=None)
```

Example: 1-D Transpose (transp1D.py)



Can you do this again but use ga.access() somewhere?

Example: Matrix Multiply (matrix.py)


=
•
global arrays representing matrices
nga_put
nga_get
=
•
dgemm
local buffers on the processor

Can you do this again but use ga.access() somewhere?

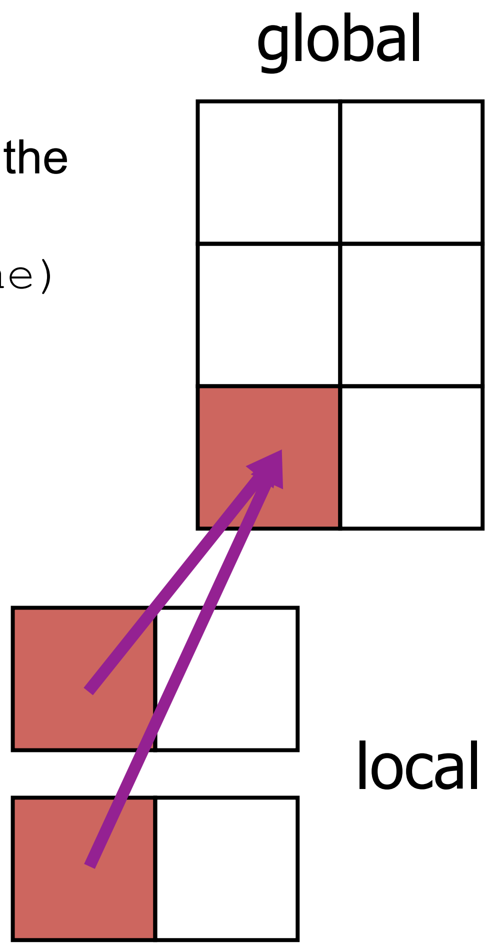
Atomic Accumulate

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

```
ga.acc(g_a, buffer, lo=None, hi=None, alpha=None)
```

integer	g_a array handle		[input]
integer	lo(), hi() limits on data block to be moved		[input]
double/complex/int	buffer	local buffer	[input]
double/complex/int	alpha	arbitrary scale factor	[input]

$$g\_a(i,j) = g\_a(i,j)+alpha*buf(k,l)$$



Global Operations

```
buffer = ga.brdcst(buffer, root)
```

Sends vector from root process to all other processes.

```
buffer = ga.gop(x, op)
```

Combines buffers from all processes using “op”.

Op can be “+”, “*”, “max”, “min”, “absmax”, “absmin”

Alternatively:

```
ga.gop_add(…), ga.gop_multiply(…), ga.gop_max(…),
ga.gop_min(…), ga.gop_absmax(…), ga.gop_absmin(…)
```

Basic Array Operations

- Whole Arrays or Array Patches:
  - To set all the elements in the array to zero:
    - `ga.zero(g_a, lo=None, hi=None)`
  - To assign a single value to all the elements in array:
    - `ga.fill(g_a, val, lo=None, hi=None)`
  - To scale all the elements in the array by factor *val*:
    - `ga.scale(g_a, val, lo=None, hi=None)`

Pacific Northwest
NATIONAL LABORATORY

Example: Calculating PI (pi.py)

You know enough of the API to try the next example!

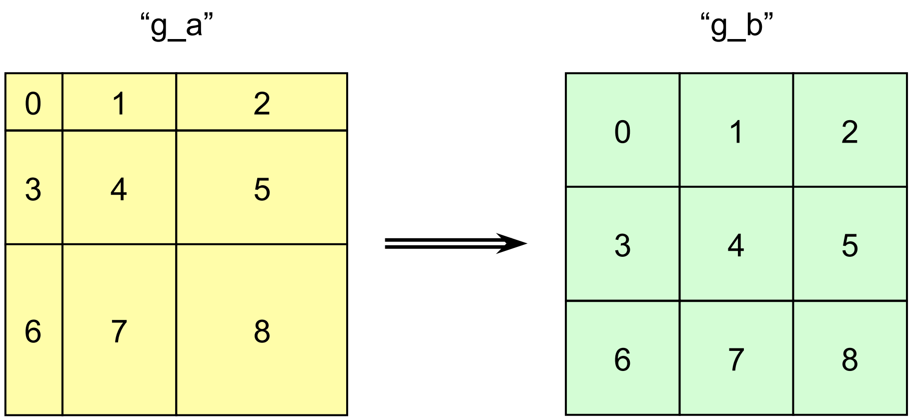
Outline of the Tutorial

- Parallel Programming Models
- Overview of the Global Arrays Programming Model
- Intermediate GA Programming Concepts and Samples
- Advanced GA Programming Concepts and Samples
- Global Arrays in NumPy (GAiN)

Copy

- Whole Arrays:
  - **To copy data between two arrays:**
    - `ga.copy(g_a, g_b)`
  - Arrays must be same size and dimension
  - Distribution may be different
  - See “copy.py” for sample

```
g_a = ga.create(ga.C_INT, [4,25],
          chunk=[4,-1])
g_b = ga.create(ga.C_INT, [4,25],
          chunk=[-1,25])
# fill GA’s with values
ga.copy(g_a, g_b)
```

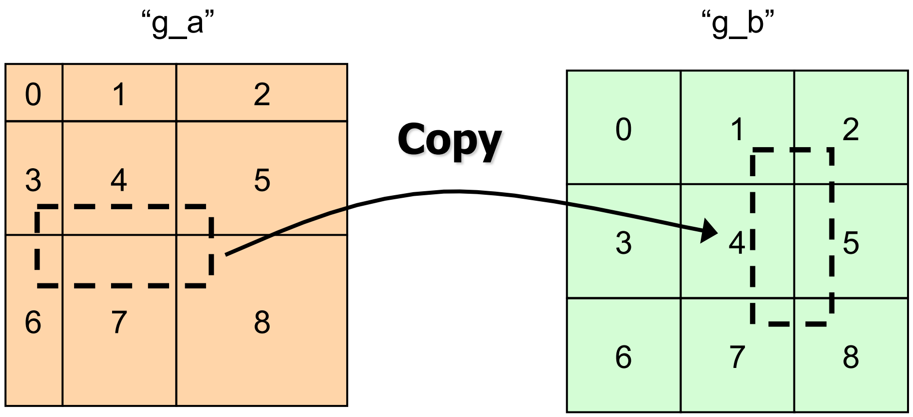


Global Arrays g_a and g_b distributed on a 3x3 process grid

Copy Patches

- Patch Operations:
  - The copy patch operation:

    -
      ```
      ga.copy(g_a, g_b,
                   alo=None, ahi=None,
                   blo=None, bhi=None, trans=False)
      ```
  - Number of elements must match



Scatter/Gather

- *Scatter* puts array elements into a global array:
  - `ga.scatter(g_a, values, subsarray)`
- *Scatter accumulate* puts array elements into a global array:
  - `ga.scatter_acc(g_a, values, subsarray, alpha=None)`
- *Gather* gets the array elements from a global array into a local array:
  - `values = ga.gather(g_a, subsarray, values=None)`

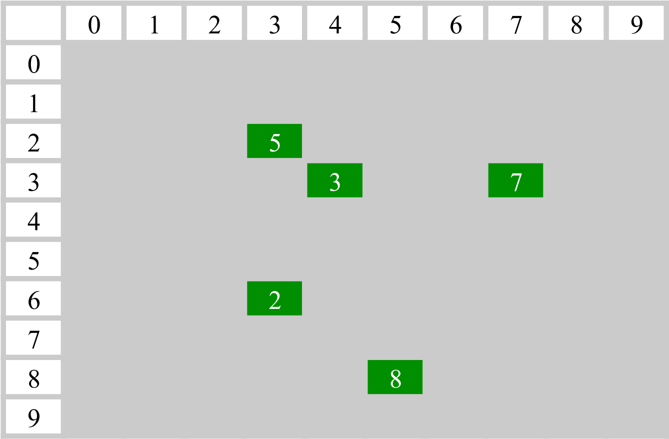
integer	g_a	array handle	[input]
double/comple/int	values	array of values	[input/output]
integer	n	number of values	[input]
integer	subsarray	coordinates within global array	[input]

“values” is a 1D vector

“subsarray” can be either 2D of shape=(N,ndim) or flattened 1D version thereof

Scatter/Gather (cont.)

- Example of *scatter* operation:
  - Scatter the 5 elements into a 10x10 global array
    - Element 1 v[0] = 5 subsArray[0][0] = 2
subsArray[0][1] = 3
    - Element 2 v[1] = 3 subsArray[1][0] = 3
subsArray[1][1] = 4
    - Element 3 v[2] = 8 subsArray[2][0] = 8
subsArray[2][1] = 5
    - Element 4 v[3] = 7 subsArray[3][0] = 3
subsArray[3][1] = 7
    - Element 5 v[4] = 2 subsArray[4][0] = 6
subsArray[4][1] = 3
  - After the *scatter* operation, the five elements would be scattered into the global array as shown in the figure.



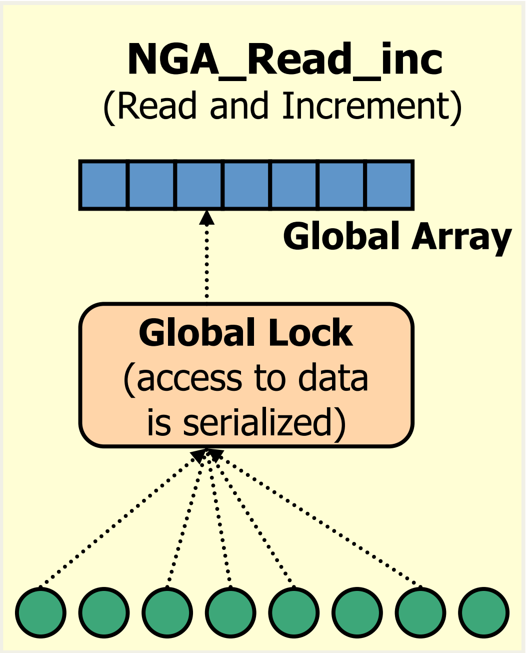
	0	1	2	3	4	5	6	7	8	9
0										
1										
2				5						
3					3			7		
4										
5										
6				2						
7										
8						8				
9										

Read and Increment

- *Read_inc* remotely updates a particular element in an integer global array and returns the original value:
  - `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

integer	g_a	[input]
integer	subscript(ndim), inc	[input]

```
# Create task counter
g_counter = ga.create(ga.C_INT, [1])
ga.zero(g_counter)
:
itask = ga.read_inc(g_counter, [0])
# ... Translate itask into task …
```



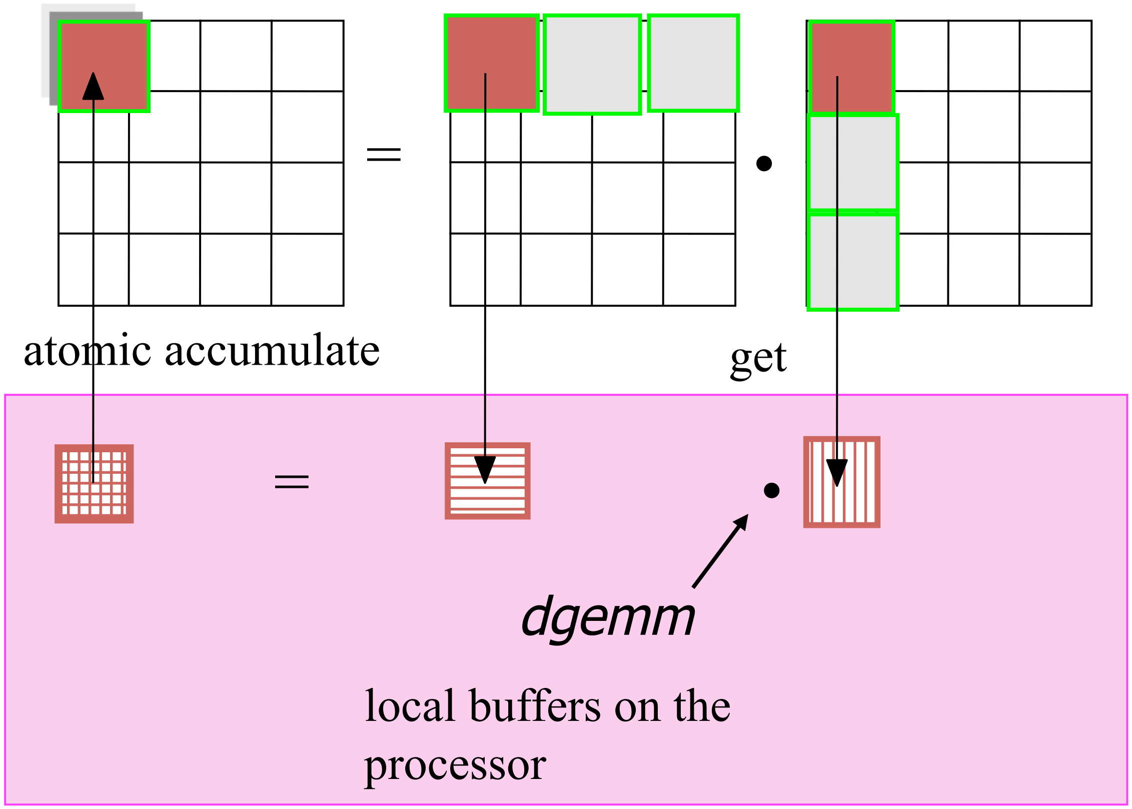
Outline of the Tutorial

- Parallel Programming Models
- Overview of the Global Arrays Programming Model
- Intermediate GA Programming Concepts and Samples
- Advanced GA Programming Concepts and Samples
- Global Arrays in NumPy (GAiN)

Non-blocking Operations

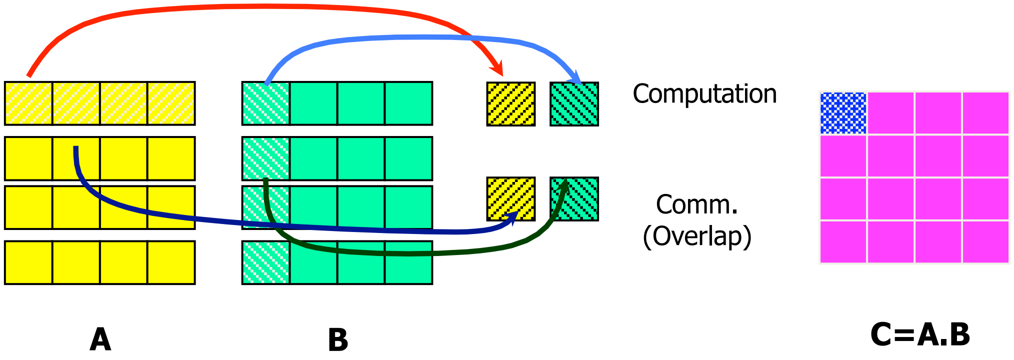
- The non-blocking APIs are derived from the blocking interface by adding a handle argument that identifies an instance of the non-blocking request.
  - `handle = ga.nbput(g_a, buffer, lo=None, hi=None)`
  - `buffer,handle = ga.nbget(g_a, lo=None, hi=None, numpy.ndarray buffer=None)`
  - `handle = ga.nbacc(g_a, buffer, lo=None, hi=None, alpha=None)`
  - `ga.nbwait(handle)`

Matrix Multiply (a better version)

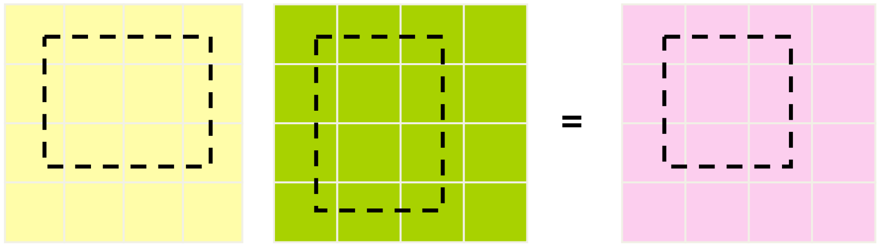


more scalable!
(less memory,
higher parallelism)

SRUMMA 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
```

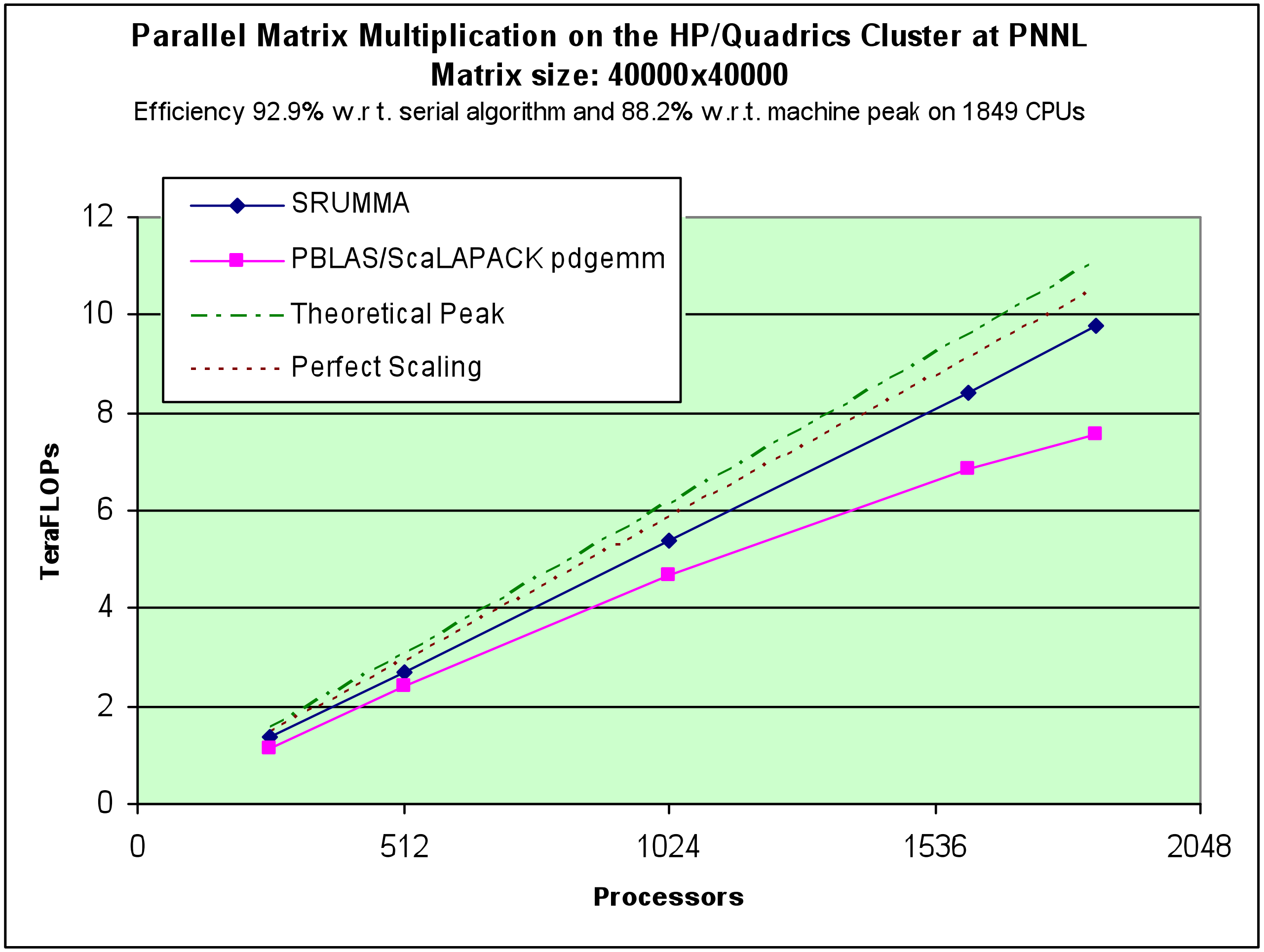


patch matrix multiplication

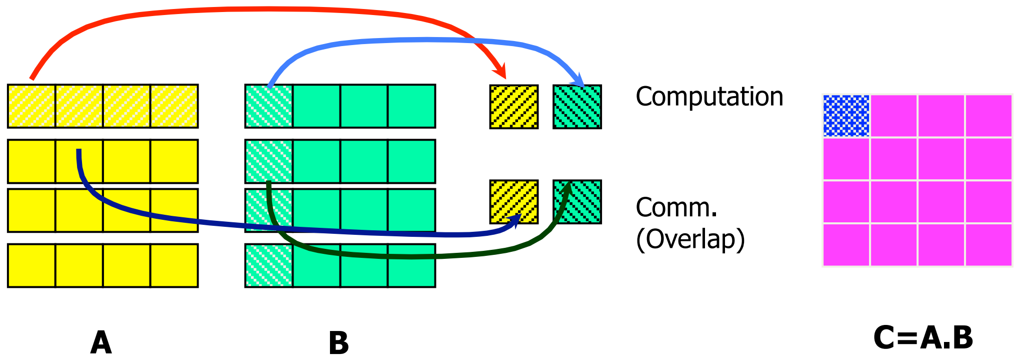
Advantages:

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

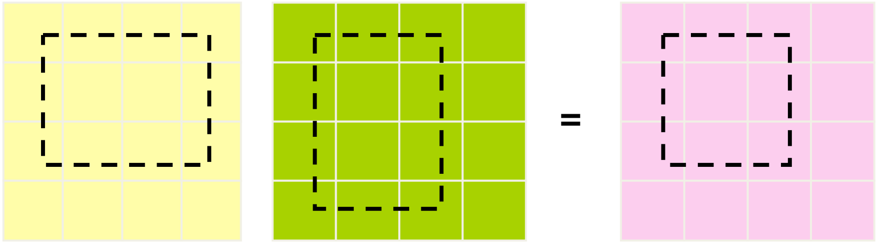
SRUMMA Matrix Multiplication: Improvement over PBLAS/ScaLAPACK



Example: SRUMMA 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
```



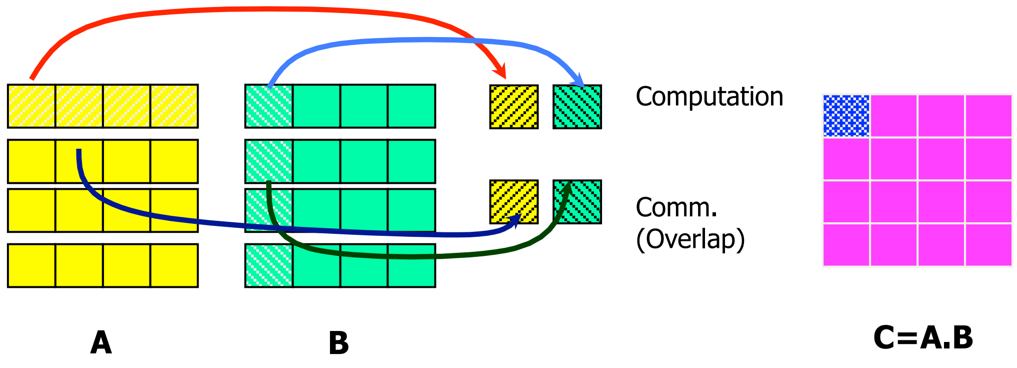
patch matrix multiplication

Advantages:

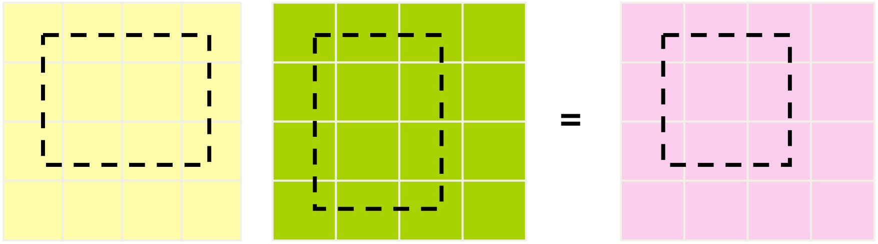
- Minimum memory
- Highly parallel
- Overlaps computation and communication
    - latency hiding
- exploits data locality
- patch matrix multiplication (easy to use)
- dynamic load balancing

Alright, give the next example a try: `srumma.py`

Example: SRUMMA Using `ga.read_inc()`



```
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
```



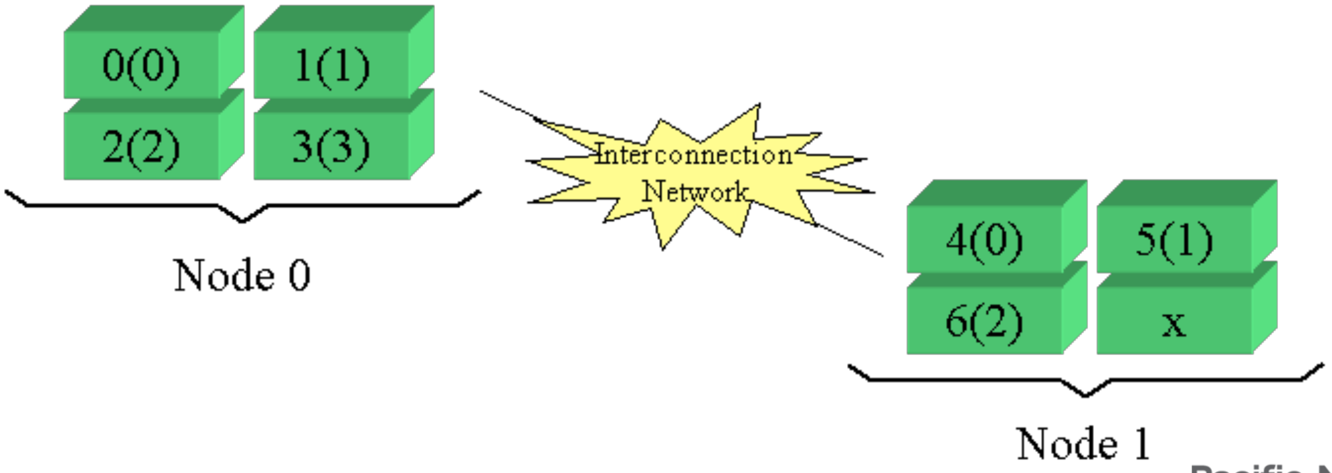
patch matrix multiplication

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

Can you modify `srumma.py` to use `ga.read_inc()`?

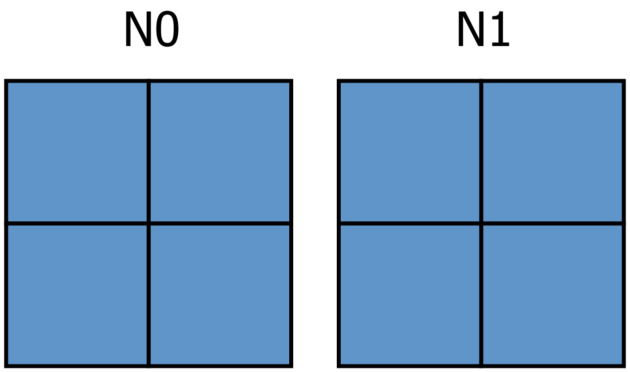
Cluster Information

- *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



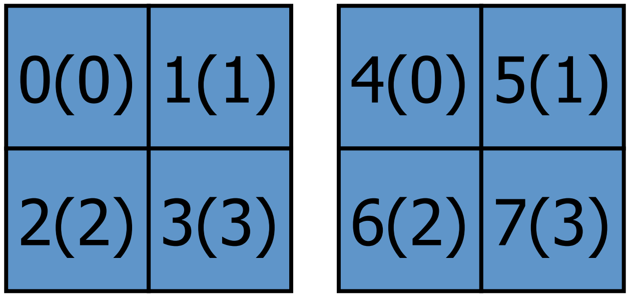
Cluster Information (cont.)

- To return the total number of nodes that the program is running on:
  - `nnodes = ga.cluster_nnodes()`
- To return the node ID of the process:
  - `nodeid = ga.cluster_nodeid()`



Cluster Information (cont.)

- To return the number of processors available on node inode:
  - `nprocs = ga.cluster_nprocs(inode)`
- To return the processor ID associated with node inode and the local processor ID iproc:
  - `procid = ga.cluster_procid(inode, iproc)`

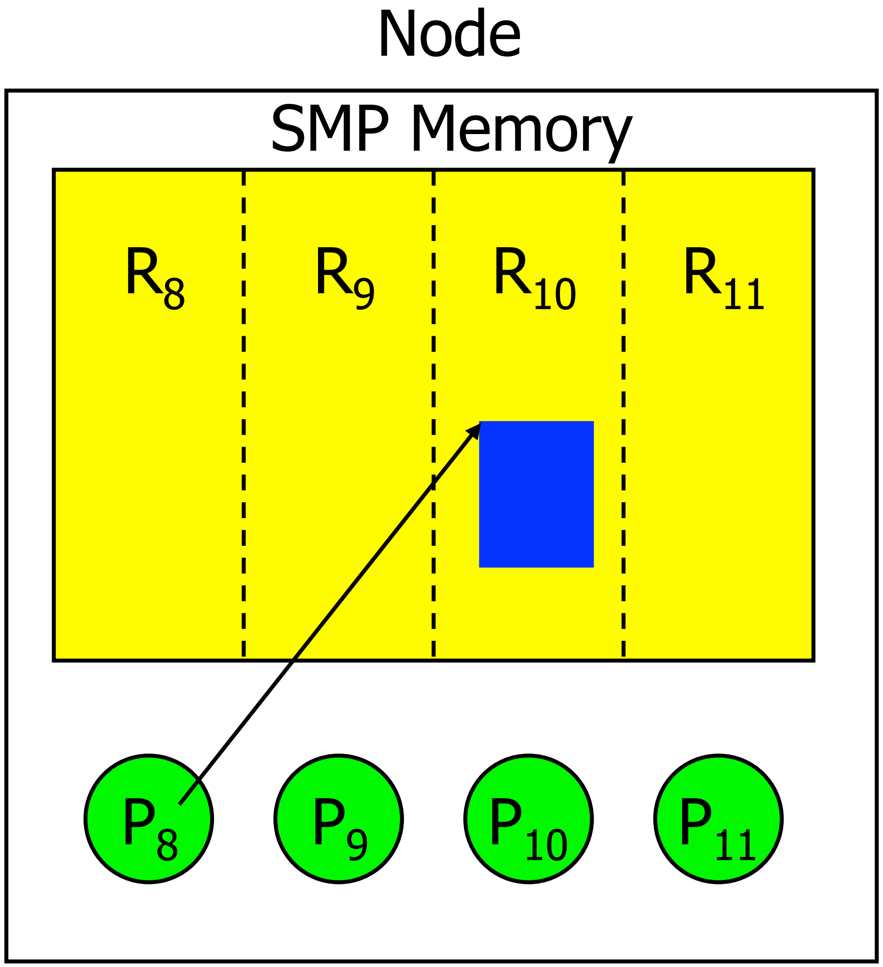


0(0) 1(1)
2(2) 3(3)

4(0) 5(1)
6(2) 7(3)

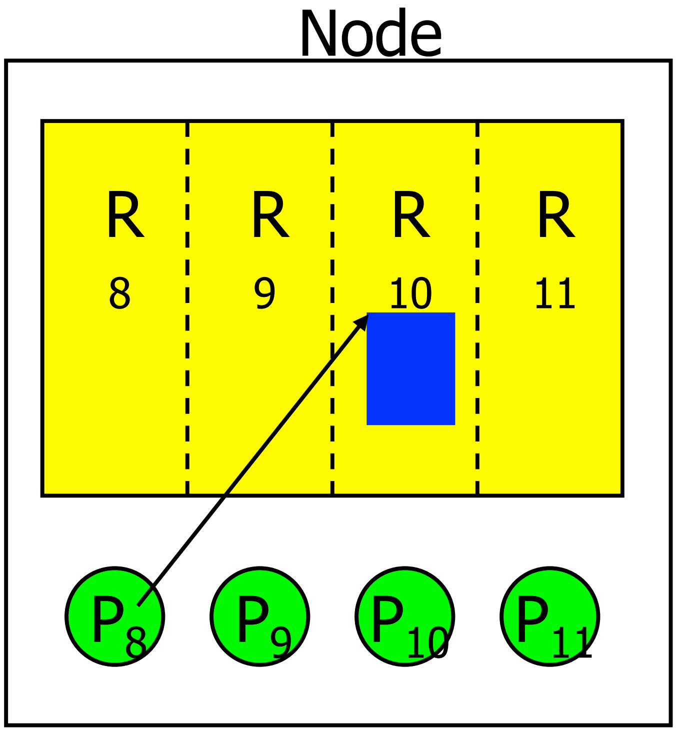
Accessing Processor Memory

```
if ga.nodeid() == 8:
    ga.access(g_a, proc=10)
```

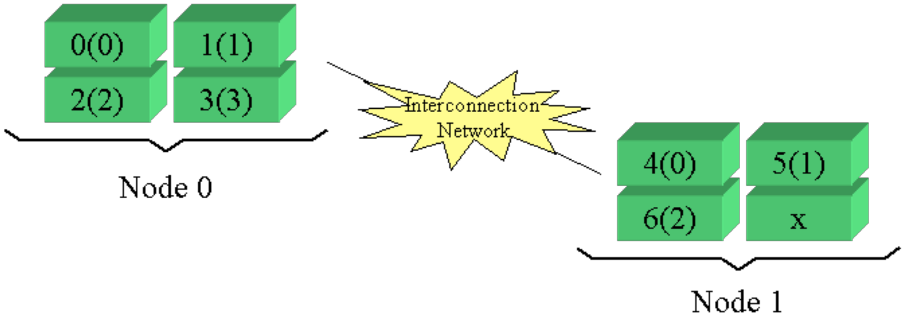


Example: access.py

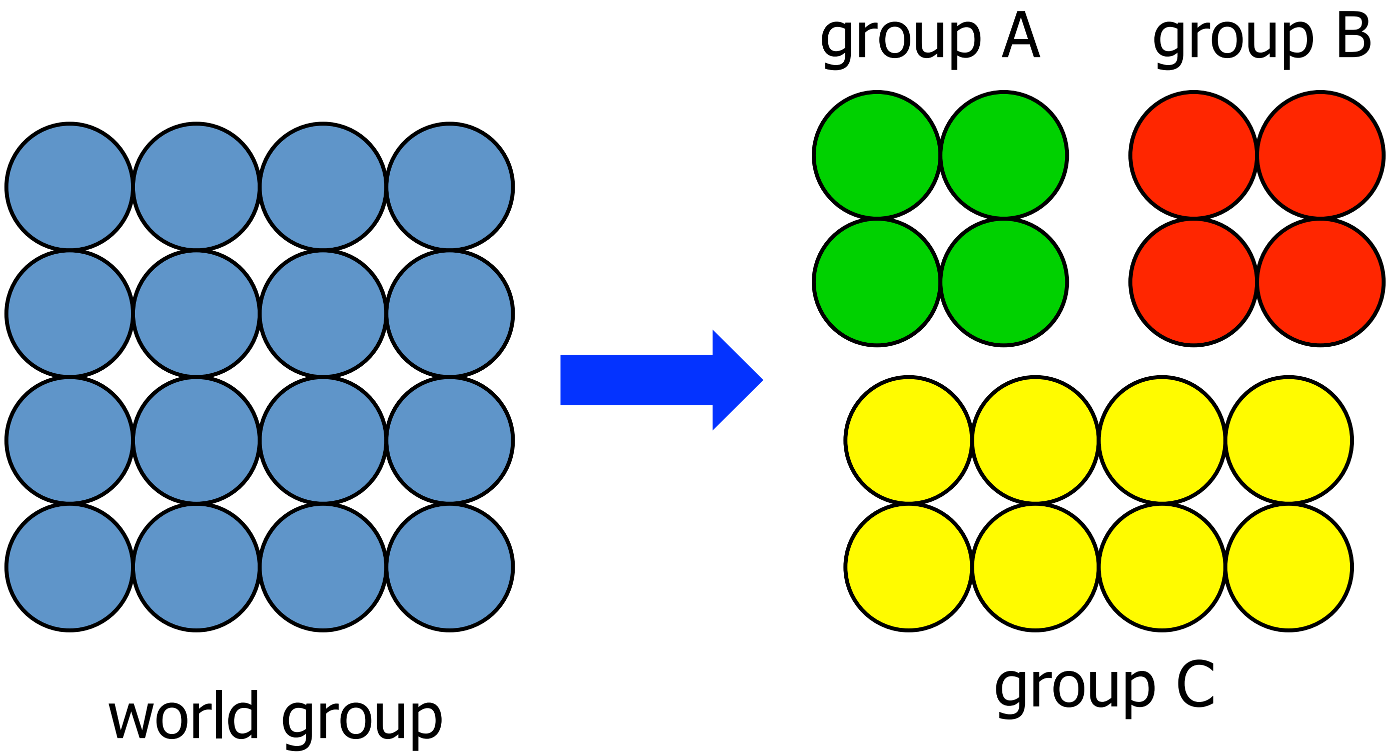
Using the cluster functions, have the master (zeroth) process on each cluster to sum the values of a global array.



- *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



Processor Groups



Processor Groups

- To create a new processor group:
  - `pgroup = ga.pgroup_create(list)`
- To assign a processor groups:
  - `g_a = ga.create(type, dims, name, chunk, pgroup=-1)`
- To set the default processor group
  - `ga.pgroup_set_default(p_handle)`
- To access information about the processor group:
  - `nnodes = ga.pgroup_nnodes(p_handle)`
  - `nodeid = ga.pgroup_nodeid(p_handle)`

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

Processor Groups (cont.)

- To determine the handle for a standard group at any point in the program:
  - `p_handle = ga.pgroup_get_default()`
  - `p_handle = ga.pgroup_get_mirror()`
  - `p_handle = ga.pgroup_get_world()`

Default Processor Group

```
# create subgroup p_a, run a parallel task
p_a = ga.pgroup_create(list)
ga.pgroup_set_default(p_a)
parallel_task()
ga.pgroup_set_default(ga.pgroup_get_world())
```

```
def parallel_task():
    p_b = ga.pgroup_create(new_list)
    ga.pgroup_set_default(p_b)
    parallel_subtask()
```

Take a shot at groups.py!

Creating Arrays with Ghost Cells

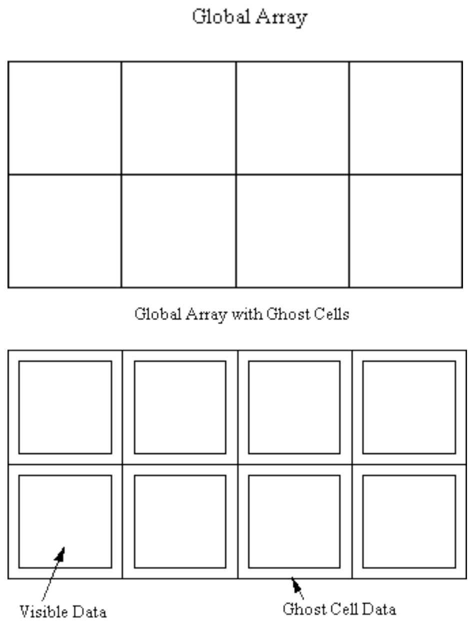
- To create arrays with ghost cells:
  - For arrays with regular distribution:

```
g_a = ga.create_ghosts(type, dims, width,
   name="", chunk=None, pgroup=-1)
```

  - For arrays with irregular distribution:

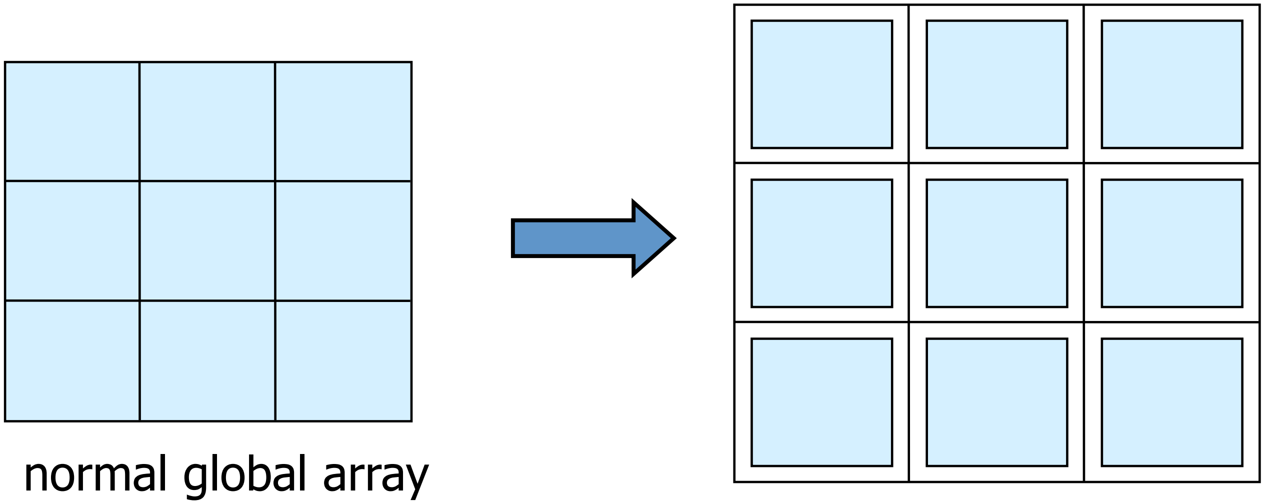
```
g_a = ga.create_ghosts_irreg(type, dims, width,
   block, map, name="", pgroup=-1)
```

integer width(ndim) - iterable of ghost cell widths [input]



Code

Ghost Cells



normal global array

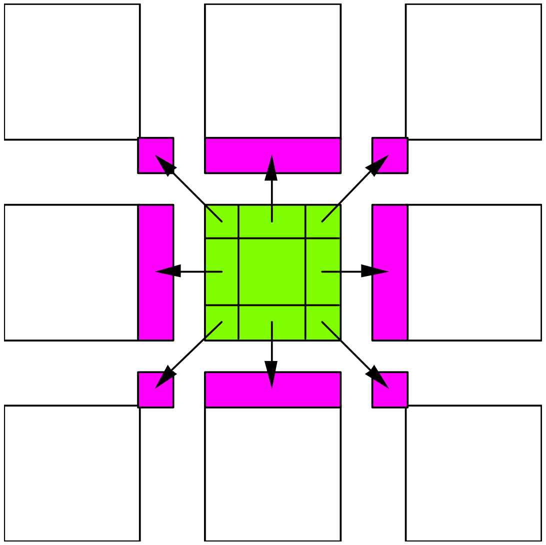
global array with ghost cells

Operations:

ga.create_ghosts	- creates array with ghosts cells
ga.update_ghosts	- updates with data from adjacent processors
ga.access_ghosts	- provides access to “local” ghost cell elements
ga.nbget_ghost_dir	- nonblocking call to update ghosts cells

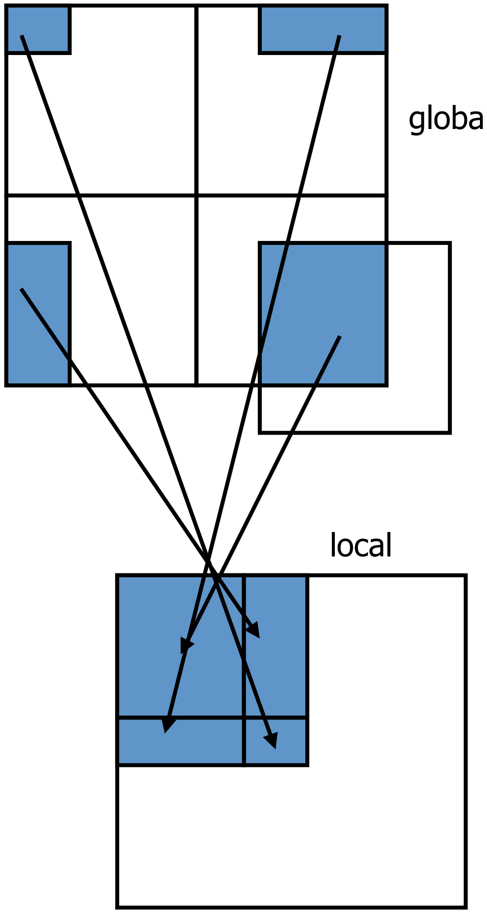
Ghost Cell Update

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*



```
ga.periodic_get(g_a,lo=None,hi=None,buf=None)
```

Periodic Get/Put/Accumulate

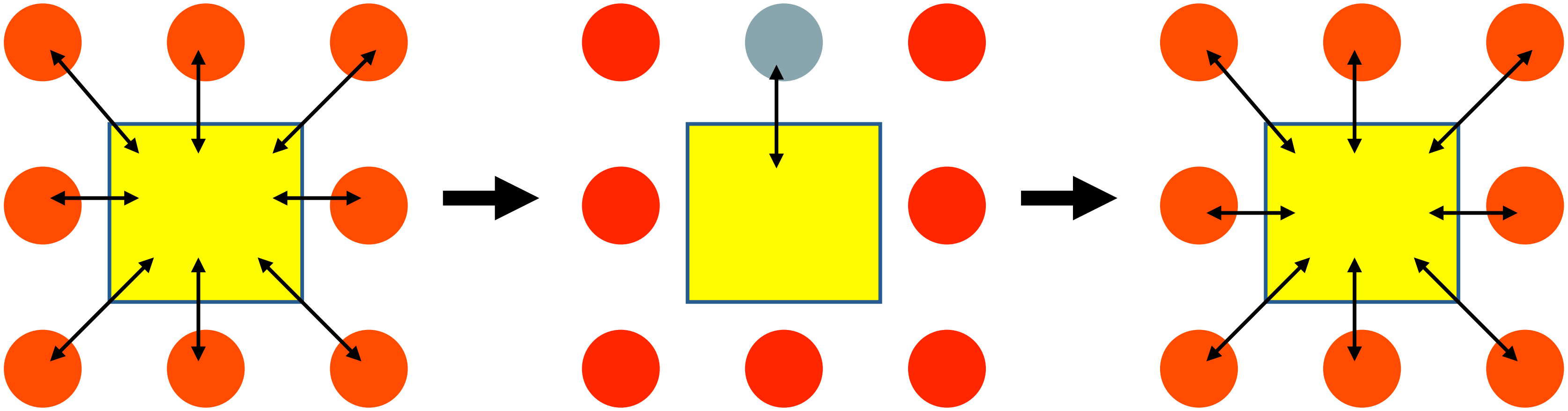
- `ndarray = ga.periodic_get(g_a, lo=None, hi=None, buffer=None)`
- `ga.periodic_put(g_a, buffer, lo=None, hi=None)`
- `ga.periodic_acc(g_a, buffer, lo=None, hi=None, alpha=None)`

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* function is:
  - `bool ga.create_mutexes(number)`

number - number of mutexes in mutex array [input]

Lock and Mutex (cont.)



Lock

Unlock

Lock and Mutex (cont.)

- The *destroy mutex* functions are:
  - `bool ga.destroy_mutexes()`
- The *lock* and *unlock* functions are:
  - `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* function is:
  - `ga.init_fence()`
- The *fence* function is:
  - `ga.fence()`

Synchronization Control in Collective Operations

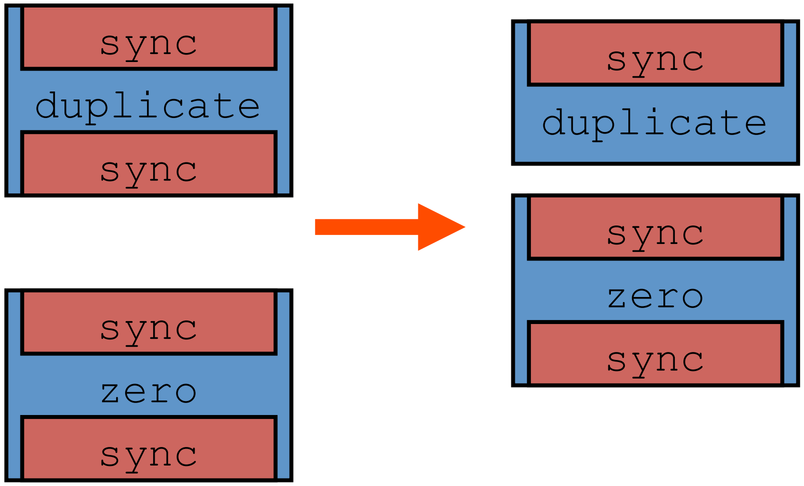
- To eliminate redundant synchronization points:

```
ga.mask_sync(prior_sync_mask, post_sync_mask)
```

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

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

```
ga.mask_sync(False,True)
ga.duplicate(g_a, g_b)
ga.zero(g_b)
```



sync
duplicate
sync

sync
zero
sync

sync
duplicate

sync
zero
sync

Linear Algebra

- To add two arrays:

```
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:

```
gemm(ta, tb, m, n, k, alpha, g_a, g_b, beta, g_c)
```

integer	g_a, g_b, g_c	- array handles	[input]
float/complex/int	alpha	- scale factor	[input]
float/complex/int	beta	- scale factor	[input]
bool	transa, transb		[input]
integer	m, n, k		[input]

Linear Algebra (cont.)

- To compute the element-wise dot product of two arrays:
  - Python has only one function: `ga.dot(g_a, g_b)`
  - This is not NumPy’s dot i.e. not matrix multiply

```
ga.dot(g_a, g_b,
  alo=None, ahi=None,
  blo=None, bhi=None,
  ta=False, tb=False)
```

Linear Algebra (cont.)

- To symmetrize a matrix:

```
ga.symmetrize(g_a)
```

- To transpose a matrix:

```
ga.transpose(g_a, g_b)
```

Linear Algebra – Array Patches (cont.)

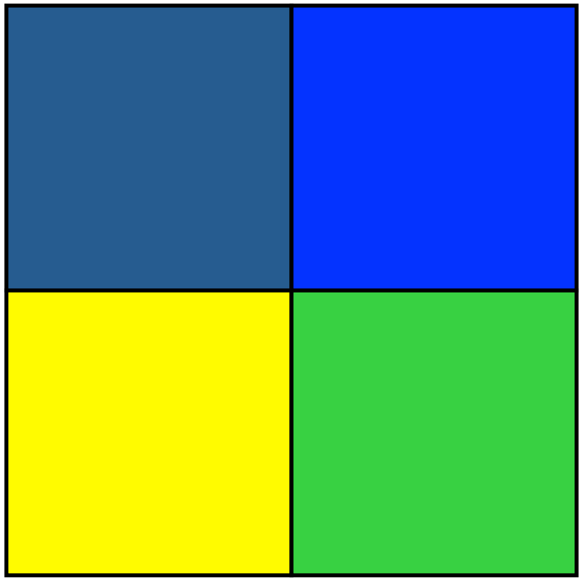
- To perform matrix multiplication:

```
 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)
```

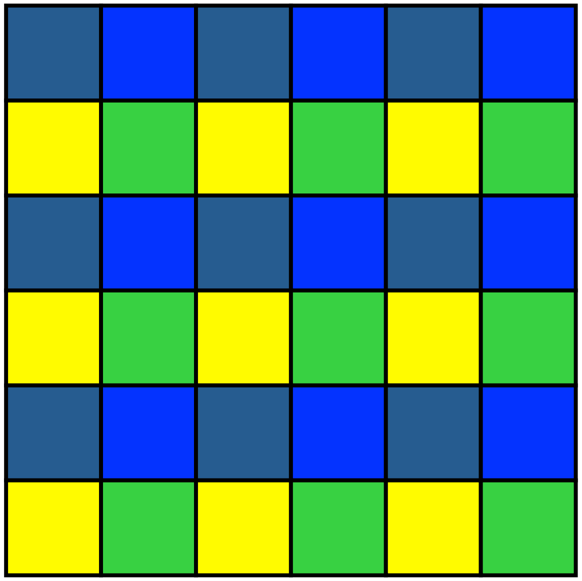
integer	g_a, ailo, aihi, ajlo, ajhi	patch of g_a	[input]
integer	g_b, bilo, bihi, bjlo, bjhi	patch of g_b	[input]
integer	g_c, cilo, cihi, cjlo, cjhi	patch of g_c	[input]
dbl prec/comp	alpha, beta	scale factors	[input]
character*1	transa, transb	transpose flags	[input]

Block-Cyclic Data Distributions

Normal Data Distribution

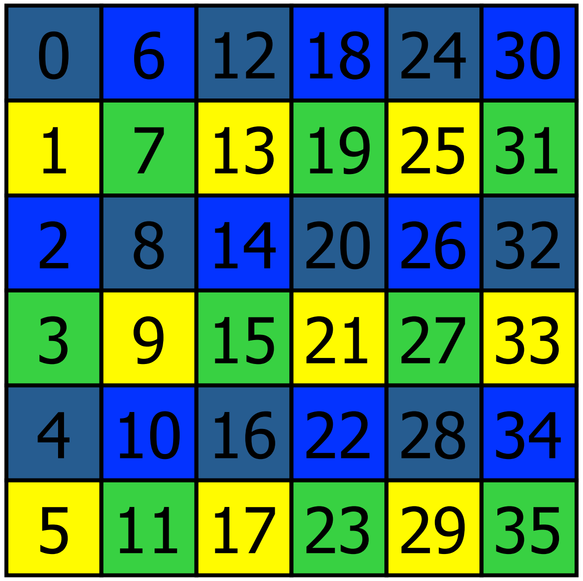


Block-Cyclic Data Distribution



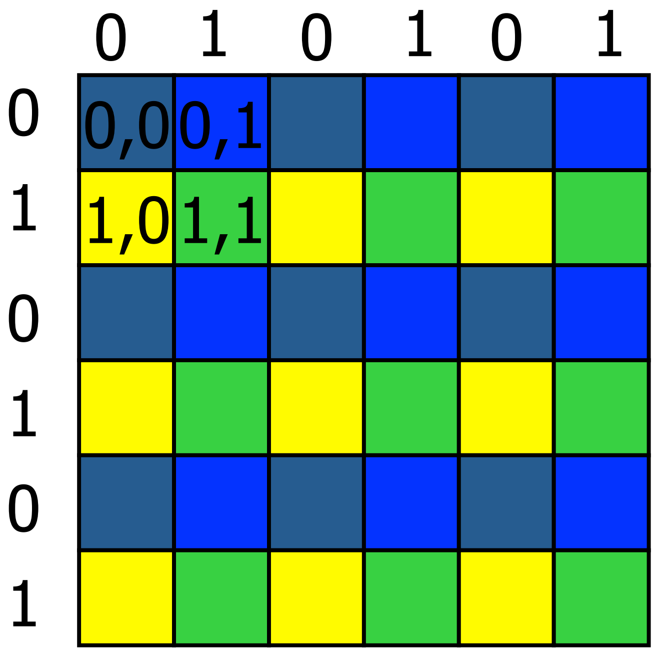
Block-Cyclic Data (cont.)

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 (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

New Interface for Creating Arrays

```
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)
```

Creating Block-Cyclic Arrays

- Must use new API for creating Global Arrays

```
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 must equal total number of processors)

Block-Cyclic Methods

- Methods for accessing data of individual blocks

```
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)
```

integer length - total size of blocks held on processor

integer idx - index of block in array (for simple block-cyclic distribution

integer 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

Example: ufunc.py

- Can you use `ga.access()` to generically reimplement a distributed NumPy unary ufunc?

Outline of the Tutorial

- Parallel Programming Models
- Overview of the Global Arrays Programming Model
- Intermediate GA Programming Concepts and Samples
- Advanced GA Programming Concepts and Samples
- Global Arrays in NumPy (GAiN)
  - Overview and Using GAiN
  - Differences with NumPy
  - Advanced GAiN and GA/GAiN interoperability

Overview of Global Arrays in NumPy (GAiN)

- All documented NumPy functions are collective
  - GAiN programs run in SPMD fashion
- Not all arrays should be distributed
  - GAiN operations should allow mixed NumPy/GAiN inputs
- Reuse as much of NumPy as possible (obviously)
- Distributed nature of arrays should be transparent to user
- Use owner-computes rule to attempt data locality optimizations

GAiN is Not Complete (yet)

- What's finished:
  - Ufuncs (all)
  - ndarray
  - flatiter
  - *numpy dtypes are reused!*
  - Various array creation and other functions:
    - zeros, zeros_like, ones, ones_like, empty, empty_like
    - eye, identity, fromfunction, arange, linspace, logspace
    - dot, diag, clip, asarray
- Everything else doesn't exist

How to Use GAiN

Change one line in your script:

```
#import numpy
import ga.gain as numpy
```

Run using the MPI process manager:

```
$ mpiexec -np 4 python script.py
```

Go ahead and write something using NumPy! Do you have an application already on your computer? Try to use GAiN as shown above.

GA/GAiN Interoperability

- `gain.from_ga(g_a)`
  - Won’t ga.destroy(g_a) when garbage collected
  - Allows custom data distributions
    - Block and block cyclic not currently supported by GAiN

Additional Examples to Try

1. Write a NumPy code, run it serially, then convert it to use GAiN.
2. Use process groups with GAiN.
3. Use process groups and ga.read_inc() with GAiN.
4. Is GAiN missing something you need?? WRITE IT.

This is it, folks! Thank you!!

jeff.daily@pnnl.gov

hpctools@googlegroups.com

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