

# Experiences with Coarrays

---

DEISA/PRACE Spring School, 30th March 2011

Parallel Programming with Coarray Fortran



# Overview

- Implementations
- Performance considerations
- Where to use the coarray model
- Examples of coarrays in practice
- References
- Wrapup

# Implementation Status

- History of coarrays dates back to Cray implementations
- Expect support from vendors as part of Fortran 2008
- G95 had multi-image support in 2010
- gfortran
  - work progressing (4.6 trunk) for single-image support
- Intel: multi-process coarray support in Intel Composer XE 2011 (based on Fortran 2008 draft)
- Runtimes are SMP, GASNet and compiler/vendor runtimes
  - GASNet has support for multiple environments (IB, Myrinet, MPI, UDP and Cray/IBM systems) so could be an option for new implementations

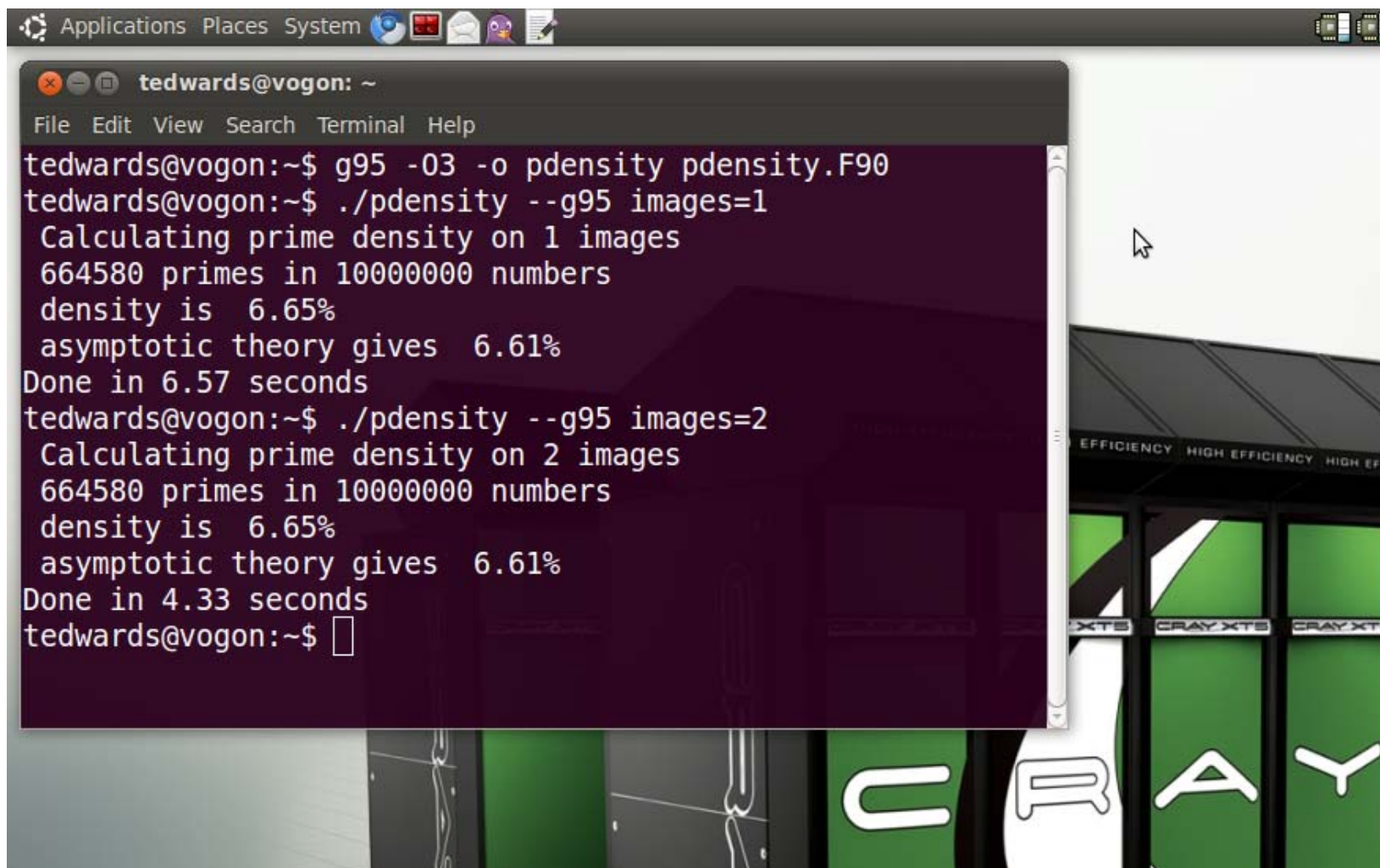
# Implementation Status (Cray)

- Cray has supported coarrays and UPC on various architectures over the last decade (from T3E)
- Full PGAS support on the Cray XT/XE
  - Cray Compiling Environment 7.0 – Dec 2008
  - Cray Compiler Environment 7.2 – Jan 2010
  - Full Fortran 2008 coarray support
  - Full Fortran 2003 with some Fortran 2008 features
- Fully integrated with the Cray software stack
  - Same compiler drivers, job launch tools, libraries
  - Integrated with Craypat – Cray performance tools
  - Can mix MPI and coarrays

# Implementations we have used

- Cray X1/X2
  - Hardware supports communication by direct load/store
  - Very efficient with low overhead
- Cray XT
  - PGAS (UPC,CAF) layered on GASNet/portals (so messaging)
  - Not that efficient
- Cray XE
  - PGAS layered on DMAPP portable layer over Gemini network hardware
  - Intermediate between XT and X1/2
- g95 on shared-memory
  - Using cloned process images on Linux

# g95 on Ubuntu



The screenshot shows a terminal window titled 'tedwards@vogon: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal output shows two runs of the 'pdensity' program using the 'g95' compiler. The first run with 'images=1' took 6.57 seconds. The second run with 'images=2' took 4.33 seconds. Both runs found 664,580 primes in 10,000,000 numbers, with a density of 6.65% and an asymptotic theory prediction of 6.61%.

```
tedwards@vogon: ~  
File Edit View Search Terminal Help  
tedwards@vogon:~$ g95 -O3 -o pdensity pdensity.F90  
tedwards@vogon:~$ ./pdensity --g95 images=1  
Calculating prime density on 1 images  
664580 primes in 10000000 numbers  
density is 6.65%  
asymptotic theory gives 6.61%  
Done in 6.57 seconds  
tedwards@vogon:~$ ./pdensity --g95 images=2  
Calculating prime density on 2 images  
664580 primes in 10000000 numbers  
density is 6.65%  
asymptotic theory gives 6.61%  
Done in 4.33 seconds  
tedwards@vogon:~$
```

# When to use coarrays

- Two obvious contexts
  - Complete application using coarrays
  - Mixed with MPI
- As an incremental addition to a (potentially large) serial code
- As an incremental addition to an MPI code (allowing use of MPI collectives)
- Use coarrays for some of the communication
  - opportunity to express communication much more simply
  - opportunity to overlap communication
- For subset synchronisation
- Work-sharing schemes

# Adding coarrays to existing applications

- Constrain use of coarrays to part of application
  - Move relevant data into coarrays
  - Implement parallel part with coarray syntax
  - Move data back to original structures
- Use coarray structures to contain pointers to existing data
- Place relevant arrays in global scope (modules)
  - avoids multiple declarations
- Declare existing arrays as coarrays at top level and through the complete call tree  
(some effort but only requires changes to declarations)



# Performance Considerations

- What is the latency?
- Do you need to avoid strided transfers?
- Is the compiler optimising the communication for target architecture?
  - Is it using blocking communication within a segment when it does not need to?
  - Is it optimising strided communication?
  - Can it pattern-match loops to single communication primitives or collectives?

# Performance: Communication patterns

- Try to avoid creating traffic jams on the network, such as all images storing to a single image.
- The following examples show two ways to implement an `ALLReduce()` function using coarrays

# AllReduce (everyone gets)

- All images get data from others simultaneously

```
function allreduce_max_allget(v) result(vmax)
  double precision :: vmax, v[*]
  integer i

  sync all

  vmax=v
  do i=1,num_images()
    vmax=max(vmax,v[i])
  end do
```

# AllReduce (everyone gets, optimized)

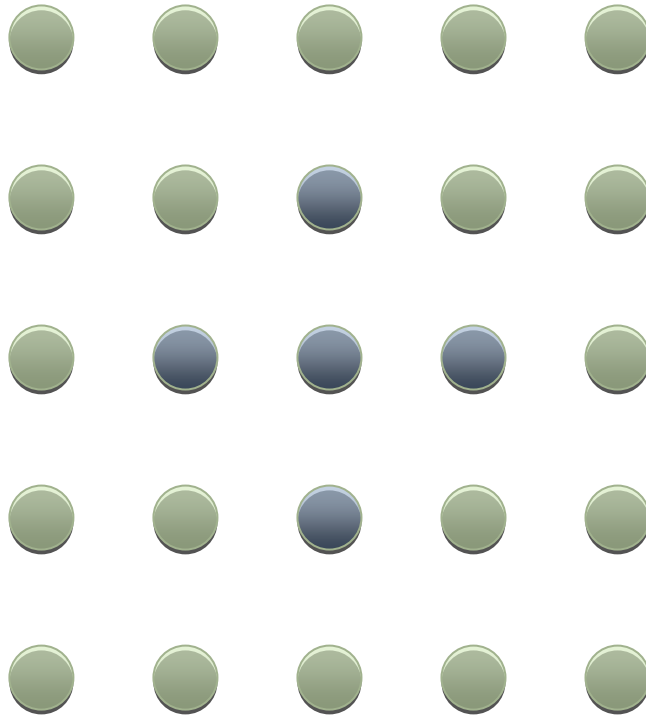
- All images get data from others simultaneously but this is optimized so communication is more balanced

```
!...  
sync all  
vmax=v  
do i=this_image()+1,num_images()  
    vmax=max(vmax,v[i])  
end do  
do i=1,this_image()-1  
    vmax=max(vmax,v[i])  
end do
```

- Have seen this much faster

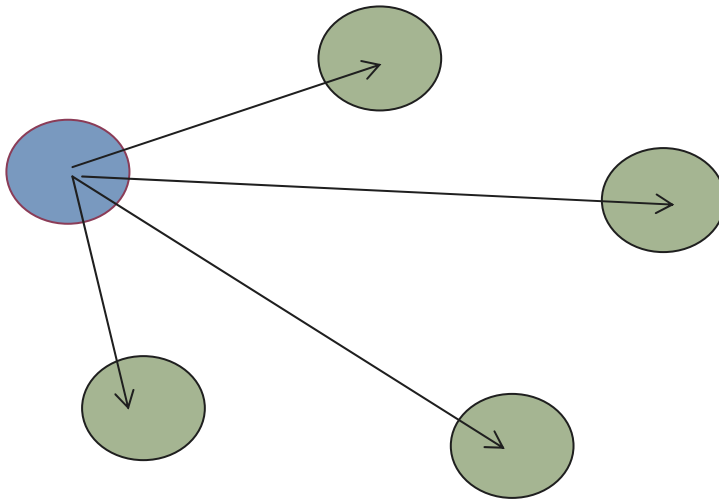
# Synchronization

- For some algorithms (finite-difference etc.) don't use `sync all` but pairwise synchronization using `sync images(image)`



# Synchronization (one to many)

- Often one image will be communicating with a set of images
- In general not a good thing to do but assume we are...

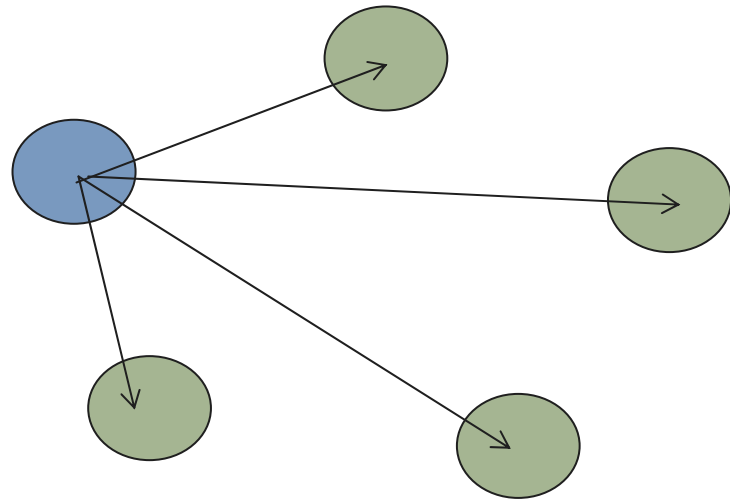


- Tempting to use `sync all`

# Synchronisation (one to many)

- If this is all images then could do

```
if ( this_image() == 1) then
  sync images(*)
else
  sync images(1)
end if
```

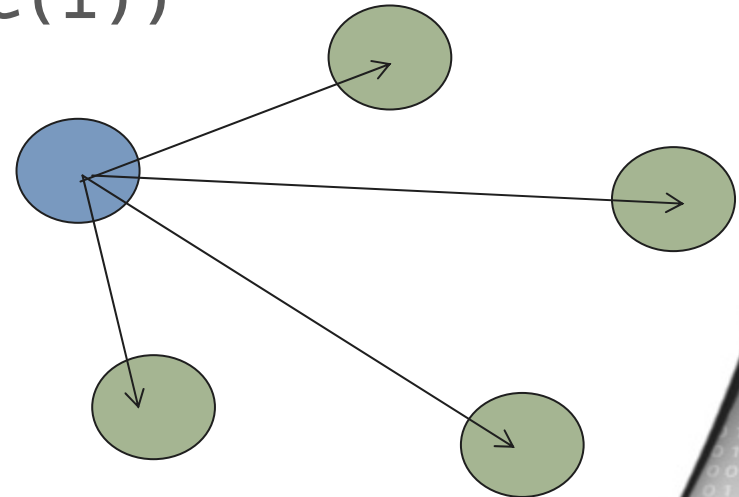


- Note that **sync all** is likely to be fast so is an alternative

# Synchronisation (one to many)

- For a subset use this

```
if ( this_image() == image_list(1)) then  
  sync images(image_list)  
else  
  sync images(image_list(1))  
end if
```



- instead of `sync images(image_list)`  
for all of them which is likely to be slower



# Collective Operations

- If your implementation does not have collectives and you need scalability to a large number of images
- Use MPI for the collectives if MPI+coarrays is supported
- Implement your own but this might be hard
  - For reductions of scalars a tree will be the best to try
  - For reductions of more data you would have to experiment and this may depend on the topology
- Coarrays can be good for collective operations where
  - there is an unusual communication pattern that does not match what MPI collectives provide
  - there is opportunity to overlap communication with computation

# Tools: debugging and profiling

- Tool support should improve once coarray takeup increases
- Cray Craypat tool supports coarrays
- Totalview works with coarray programs on Cray systems
- Scalasca
  - Currently investigating how PGAS support can be incorporated.

# Debugging Synchronisation problems

- One-sided model is tricky because subtle synchronisation errors change data
- TRY TO GET IT RIGHT FIRST TIME
  - look carefully at the remote operations in the code
  - Think about synchronisation of segments
  - especially look for early arriving communications trashing your data at the start of loops (this one is easy to miss)
- One way to test is to put sleep() calls in the code
  - Delay one or more images
  - Delay master image, or other images for some patterns

# Examples of coarrays in practice

- Work-sharing Examples
  - Puzzles
  - Many-body deformation code
- Distributed Remote Gather
- Example of incorporating coarrays into existing application

# Solving Sudoku Puzzles

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

# Going Parallel

- Started with serial code
- Changed to read in all 125,000 puzzles at start
- Choose work-sharing strategy
  - One image (1) holds a queue of puzzles to solve
  - Each image picks work from the queue and writes result back to queue
- Arbitrarily decide to parcel work as  
 $\text{blocksize} = \text{npuzzles} / (8 * \text{num\_images}())$

# Data Structures

```
use,intrinsic iso_fortran_env
```

```
type puzzle
```

```
integer :: input(9,9)
```

```
integer :: solution(9,9)
```

```
end type puzzle
```

```
type queue
```

```
type (lock_type) :: lock
```

```
integer :: next_available = 1
```

```
type(puzzle),allocatable :: puzzles(:)
```

```
end type queue
```

```
type(queue),save :: workqueue[*]
```

```
type(puzzle) :: local_puzzle
```

```
integer,save :: npuzzles[*],blocksize[*]
```

# Input

```
if (this_image() == 1) then
  ! After file Setup.
  inquire (unit=inunit,size=nbytes)
  nrecords = nbytes/10
  npuzzles = nrecords/9
  blocksize = npuzzles / (num_images()*8)

  write (*,*) "Found ", npuzzles, " puzzles."
  allocate (workqueue%puzzles(npuzzles))
  do i = 1, npuzzles
    call read_puzzles( &
&          workqueue%puzzles(i)%input,inunit, &
&          error)
  end do

  close(inunit)
```



# Core program structure

```
! After coarray data loaded
sync all

blocksize = blocksize[1]
npuzzles = npuzzles[1]

done = .false.
workloop: do

    ! Acquire lock and claim work

    ! Solve our puzzles

end do workloop
```

# Acquire lock and claim work

```
!   Reserve the next block of puzzles
```

```
lock (workqueue[1]%lock)
next = workqueue[1]%next_available
if (next <= npuzzles) then
    istart = next
    iend   = min(npuzzles, next+blocksize-1)
    workqueue[1]%next_available = iend+1
else
    done = .true.
end if

unlock (workqueue[1]%lock)
if (done) exit workloop
```

# Solve the puzzles and write back

```
!   Solve those puzzles

do i = istart,iend

  local_puzzle%input = &
    &   workqueue[1]%puzzles(i)%input

  call sudoku_solve &
    &   (local_puzzle%input,local_puzzle%solution)

  workqueue[1]%puzzles(i)%solution = &
    &   local_puzzle%solution

end do
```

# Output the solutions

```
! Need to synchronize puzzle output updates  
sync all
```

```
if (this_image() == 1) then
```

```
    open (outunit,file=outfile,iostat=error)
```

```
    do i = 1, npuzzles
```

```
        call write_puzzle &
```

```
&        (workqueue%puzzles(i)%input, &
```

```
&        workqueue%puzzles(i)%solution,outunit,error)
```

```
    end do
```

# More on the Locking

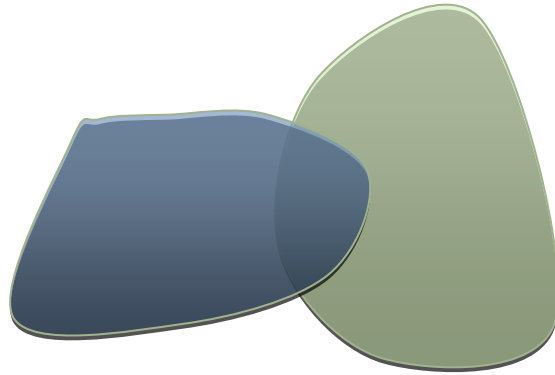
- We protected access to the queue state by lock and unlock
- During this time no other image can acquire the lock
  - We need to have discipline to only access data within the window when we have the lock
  - There is no connection with the lock variable and the other elements of the queue structure
- The unlock is acting like sync memory
  - If one image executes an unlock...
  - Another image getting the lock is ordered after the first image

# Summary and Commentary

- We implemented solving the puzzles using a work-sharing scheme with coarrays
- Scalability limited by serial work done by image 1
- I/O
  - Parallel I/O (deferred to TR) with multiple images running distributed work queues.
  - Defer the character-integer format conversion to the solver, which is executed in parallel.
- Lock contention
  - Could use distributed work queues, each with its own lock.

# Impact Application

- Study of impact and deformation of multiple bodies



- Bodies deform on contact

# Impact application: Challenges and approach

## Challenges

- Body contact is unpredictable
- Much more computation in contact regions
  - Very poor load balance limits scalability

## Approach

- Replicate contact data on all images (cores)
- Divide contact data into segments (e.g. 5 times as many segments as images)
- Use coarrays to assign data segments to images by atomically incrementing the global segment index
- Images process more or fewer segments, depending on the computational work in each segment



# Results

- After implementing the work-dispatch method in the contact algorithm

	Contact Algorithm fraction of entire code		Contact Algorithm
Code Version	Elapsed Time	Comm. & Sync. time	Load Imbalance
Original MPI	15%	75%	28%
Coarray version with work dispatch	10%	35%	1%

- Big improvement in load balancing

# Distributed remote gather

- The problem is how to implement the following gather loop on a distributed memory system

```
REAL      :: table(n), buffer(nelts)
INTEGER :: index(nelts)    ! nelts << n
...
DO i = 1, nelts
    buffer(i) = table(index(i))
ENDDO
```

- The array **table** is distributed across the processors, while **index** and **buffer** are replicated
- Synthetic code, but simulates “irregular” communication access patterns

# Remote gather: MPI implementation

- MPI rank 0 controls the index and receives the values from the other ranks

```
IF (mype.eq.0)THEN

    isum=0
    ! PE0 gathers indices to send out to individual PEs
    DO i=1,nelts
        pe =(index(i)-1)/nloc
        isum(pe)=isum(pe)+1
        who(isum(pe),pe) = index(i)
    ENDDO
    ! send out count and indices to PEs
    DO i = 1, npes-1
        CALL MPI_SEND(isum(i),1,MPI_INTEGER,i,10.
        IF(isum(i).gt.0)THEN
            CALL MPI_SEND(who(1,i),isum(i),...
        ENDIF
    ENDDO
    ! now wait to receive values and scatter them.
    DO i = 1,isum(0)
        offset = mod(who(i,0)-1,nloc)+1
        buff(i,0) = mpi_table(offset)
    ENDDO
    DO i = 1,npes-1
        IF(isum(i).gt.0)THEN
            CALL MPI_RECV(buff(1,i),isum(i),...
        ENDIF
    ENDDO

    DO i=nelts,1,-1
        pe =(index(i)-1)/nloc
        offset = isum(pe)
        mpi_buffer(i) = buff(offset,pe)
        isum(pe) = isum(pe) - 1
    ENDDO

ELSE !IF my_rank.ne.0

    ! Each PE gets the list and sends the values to PE0
    CALL MPI_RECV(my_sum,1,MPI_INTEGER,...
    IF(my_sum.gt.0)THEN
        CALL MPI_RECV(index,my_sum,MPI_INTEGER,...
        DO i = 1, my_sum
            offset = mod(index(i)-1,nloc)+1
            mpi_buffer(i) = mpi_table(offset)
        ENDDO
        CALL MPI_SEND(mpi_buffer,my_sum,...
    ENDIF

ENDIF
```

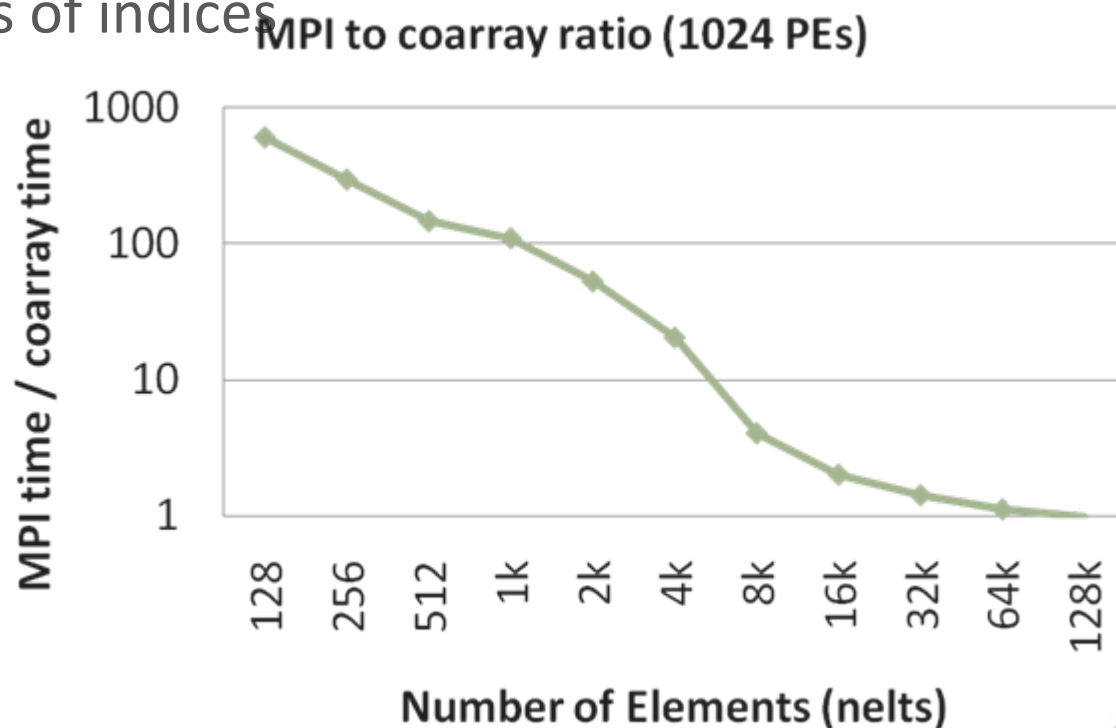
# Remote gather: coarray implementation (get)

- Image 1 gets the values from the other images

```
IF (myimg.eq.1) THEN
  DO i=1,nelts
    pe =(index(i)-1)/nloc+1
    offset = MOD(index(i)-1,nloc)+1
    caf_buffer(i) = caf_table(offset)[pe]
  ENDDO
ENDIF
```

# Remote gather: coarray vs MPI

- Coarray implementations are much simpler
- Coarray syntax allows the expression of remote data in a natural way – no need of complex protocols
- Coarray implementation is orders of magnitude faster for small numbers of indices



# Adding coarray with minimal code modifications

- Coarray structures can be inserted into an existing MPI code to simplify and improve communication without affecting the general code structure
- In some cases the usage of Fortran pointers can simplify the introduction of coarray variables in a routine without modifying the calling tree
- Code modifications are mostly limited to the routine where coarrays must be introduced
- A coarray is not permitted to be a pointer: however, a coarray may be of a derived type with pointer or allocatable components

# Example: adding coarrays to existing code

- In subroutine **sub2** we want to use coarrays for communication
- Array **x** is defined and allocated in the main program and is passed as argument to subroutine **sub1** and then to **sub2**

```
PROGRAM main
REAL, ALLOCATABLE :: x(:)
ALLOCATE (x(n))
CALL sub1(x,n)

SUBROUTINE sub1(x,n)
REAL :: x(n)
x(:) = ...
CALL sub2(x,n)

SUBROUTINE sub2(x,n)
REAL :: x(n)
CALL MPI_Irecv(n,rx,...
CALL MPI_Send(n,x,...
```

# Modify the routine where coarrays are used

- In **sub2** add the derived type **CAFP** which contains just a pointer and allocate a coarray variable **xp** of type **CAFP**
- Each image sets **xp** to point to array **x**
- Array **x** on image **p** can be referenced by **xp[p]%ptr**

```
SUBROUTINE sub2(x,n)                                ! Each image sets the
REAL, TARGET :: x(n)                                ! pointer and sync
! Add derived type CAFP                             xp%ptr => x(1:n)
TYPE CAFP                                           sync all
REAL, POINTER :: ptr(:)                             ! Coarray x on image k can
END TYPE CAFP                                       ! be referenced by the
                                                    ! derived type pointer
! Allocate xp of type CAFP                          ! rx = x[k]
TYPE(CAFP) :: xp[*]                                rx(1:n) = xp[k]%ptr(1:n)
```



# References

- <http://lacs.rice.edu/software/caf/downloads/documentation/nrRAL98060.pdf>- Co-array Fortran for parallel programming, Numrich and Reid, 1998
- <ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf>  
“Coarrays in the next Fortran Standard”, John Reid, April 2010
- Ashby, J.V. and Reid, J.K (2008). Migrating a scientific application from MPI to coarrays. CUG 2008 Proceedings. RAL-TR-2008-015  
See <http://www.numerical.rl.ac.uk/reports/reports.shtml>
- <http://upc.gwu.edu/> - Unified Parallel C at George Washington University
- <http://upc.lbl.gov/> - Berkeley Unified Parallel C Project

# Wrapup

Remember our first Motivation slide?

- Fortran now supports parallelism as a full first-class feature of the language
- Changes are minimal
- Performance is maintained
- Flexibility in expressing communication patterns

We hope you learned something and have success with coarrays in the future

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

# Acknowledgements

This material for is based on original content developed by EPCC at The University of Edinburgh for use in teaching the MSc in High Performance Computing. The following people contributed to its development:

Alan Simpson, Michele Weiland, Jim Enright and Paul Graham

The material was subsequently developed by EPCC and Cray with contributions from the following people:

David Henty, Alan Simpson ( EPCC)

Harvey Richardson, Bill Long, Roberto Ansaloni,  
Jef Dawson, Nathan Wichmann (Cray)

This material is Copyright © 2010 by The University of Edinburgh and Cray Inc.

