Experiences with Coarrays

Parallel Programming with Fortran Coarrays

MSc in HPC

David Henty, Alan Simpson (EPCC)
Harvey Richardson, Bill Long (Cray

Overview

- Implementations
- Performance considerations
- Where to use the coarray model
- Coarray benchmark suite
- 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 (development halted)
- gfortran
  - Introduced single-image support at version 4.6
  - Can use OpenCoarrays for multi-image support (gcc 5)
- 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

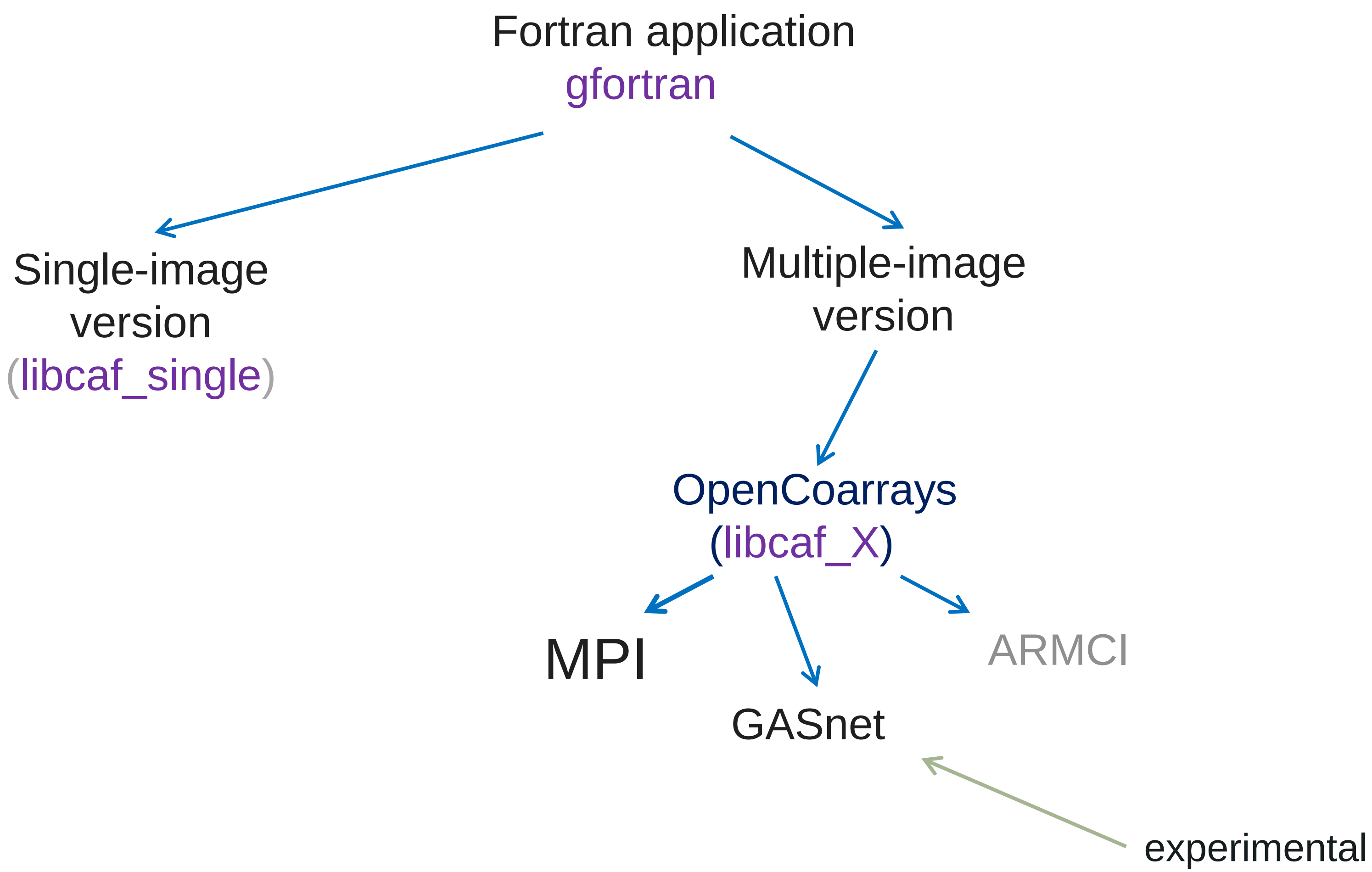
Open Source software stack for coarrays

- gfortran (from v5) supports Fortran 2008 coarrays along with broadcast/reduction collectives and atomics in TS18508
- Still some issues
- Single-image support (-fcoarray=single)
- Multi-image support via library (OpenCoarrays)

OpenCoarrays

- Provides a runtime to support coarrays
- Use –fcoarray=lib
- Implemented using:
  - MPI
  - GASnet
  - ARMCI (from Global Arrays)

OSS Stack for coarrays



So how can I use this myself?

Download binaries?

- I could not get that to work

Build it all yourself

- Build the sources on Linux
- Your distro (or the repository you use) may not provide elements that are new enough
- In my case (for Ubuntu 14.04 VM)
  I had to build/install:
  m4, g++, gcc (c,c++,gfortran), MPICH, Cmake, OpenCoarrays
- See backup slides for more details

Use the Sourcery Institute VM

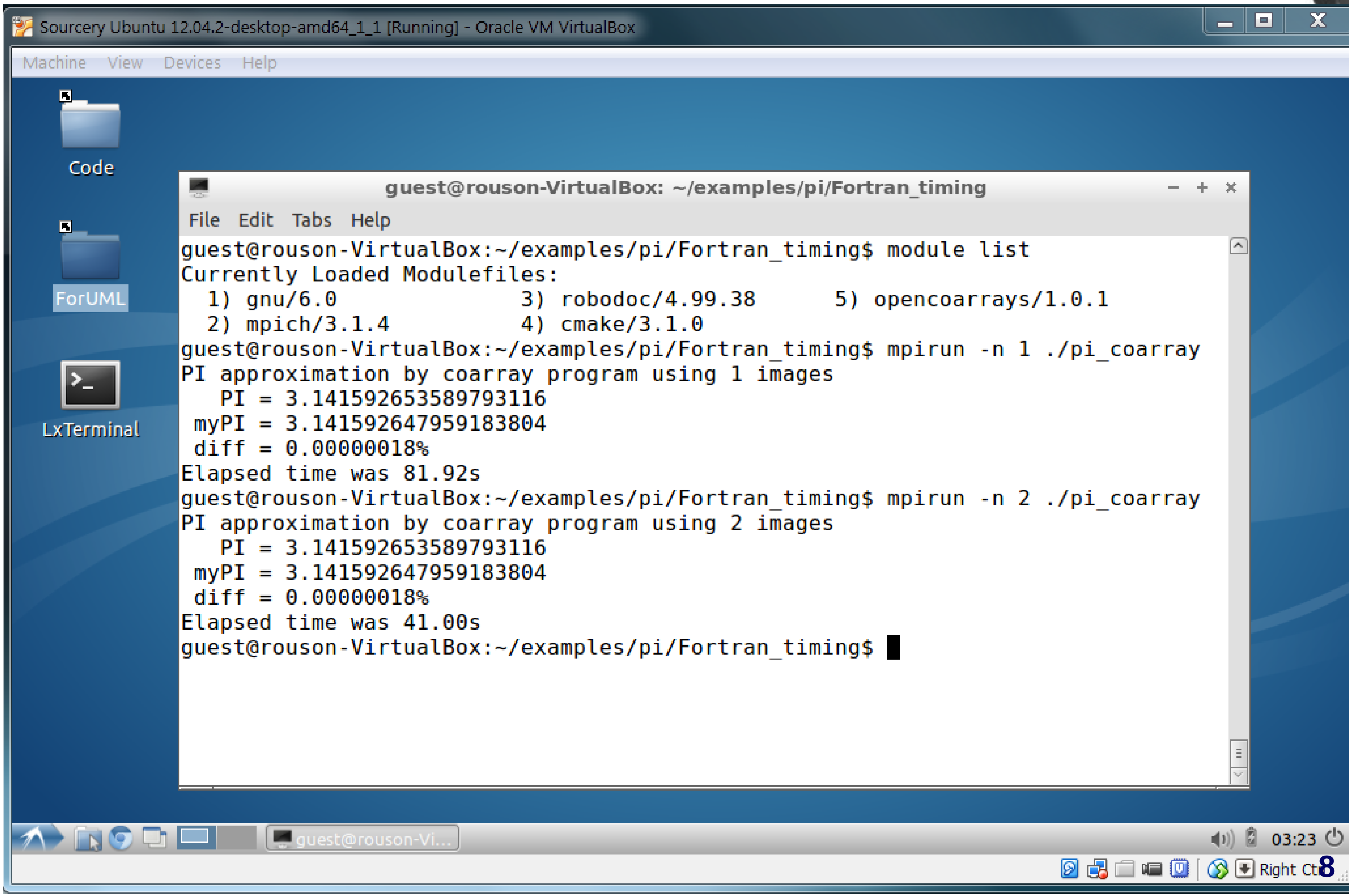
- Get from http://www.sourceryinstitute.org/
- This is a (4.3GB) pre-built VM appliance for VirtualBox

OpenCoarrays inside Ubuntu 14.04 VM

```
harvey@u14-04: ~/examples/pi/Fortran_timing
File  Edit  View  Search  Terminal  Help
harvey@u14-04:~/examples/pi/Fortran_timing$ make -f Makefile.Gnu pi_coarray
caf -o pi_coarray pi_coarray.f90 params.o -O2 -I ../timers/Fortran ../timers/For
tran/timer.a
harvey@u14-04:~/examples/pi/Fortran_timing$ mpirun -n 1 ./pi_coarray
PI approximation by coarray program using 1 images
   PI = 3.14159265358979311б
 myPI = 3.141592647959183804
 diff = 0.00000018%
Elapsed time was 80.92s
harvey@u14-04:~/examples/pi/Fortran_timing$ mpirun -n 2 ./pi_coarray
PI approximation by coarray program using 2 images
   PI = 3.141592653589793116
 myPI = 3.141592647959183804
 diff = 0.00000018%
Elapsed time was 40.74s
harvey@u14-04:~/examples/pi/Fortran_timing$ mpirun -n 4 ./pi_coarray
PI approximation by coarray program using 4 images
   PI = 3.141592653589793116
 myPI = 3.141592647959183804
 diff = 0.00000018%
Elapsed time was 21.02s
harvey@u14-04:~/examples/pi/Fortran_timing$ _
```

CRAY THE SUPERCOMPUTER COMPANY | epcc |

Sourcery Institute VM



Intel® Fortran Compiler

- Coarray support since Intel Composer XE 2011 (v12.0)
- Functionally complete in v13.0 a year later
- This used a distributed runtime based on Intel MPI
- Single-node support built-in (also MPI)
- Distributed (cluster) runtime needs the Cluster version of the current Parallel Studio product (licensing requirement)

- Intel are currently prioritising additional features

Intel Composer XE in Ubuntu VM

Applications Places System

harvey@u10-vbox: ~/fortran

File Edit View Search Terminal Help

```
harvey@u10-vbox:~/fortran$ ifort -coarray -O3 -o pdensity pdensity.f90
harvey@u10-vbox:~/fortran$ export FOR_COARRAY_NUM_IMAGES=1
harvey@u10-vbox:~/fortran$ ./pdensity
 Calculating prime density on           1 images
      539778 primes in     8000000 numbers
 density is  6.75%
 asymptotic theory gives  6.71%
Done in 15.61 seconds
harvey@u10-vbox:~/fortran$ export FOR_COARRAY_NUM_IMAGES=2
harvey@u10-vbox:~/fortran$ ./pdensity
 Calculating prime density on           2 images
      539778 primes in     8000000 numbers
 density is  6.75%
 asymptotic theory gives  6.71%
Done in 10.31 seconds
harvey@u10-vbox:~/fortran$ _
```

Cray Compilation Environment (CCE) Fortran

- Cray has supported coarrays and UPC on various architectures for nearly two decades (from T3E)
- Full PGAS support on the Cray XT/XE/XC
- CCE Fortran Compiler
  - ANSI/ISO Fortran 2008 compliant (since CCE 8.1 in 2012)
  - OpenMP 4.0, OpenACC 2.0
  - Coarray support integrated into the compiler
  - CCE 8.3/8.4 support TS29113 (interoperability) + collectives and atomics from new parallel 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

History of Cray PGAS runtimes

- 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/XC
  - PGAS layered on DMAPP portable layer over Gemini/Aries network hardware
  - Aries supports RDMA, atomic operations and has hardware support for barrier/reduction operations
  - Intermediate efficiency between XT and X1/X2

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 reuse of most of the existing code)
- 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)

CRAY THE SUPERCOMPUTER COMPANY | epcc |

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 no 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
```

CRAY THE SUPERCOMPUTER COMPANY | epcc |

AllReduce (everyone gets, optimized)

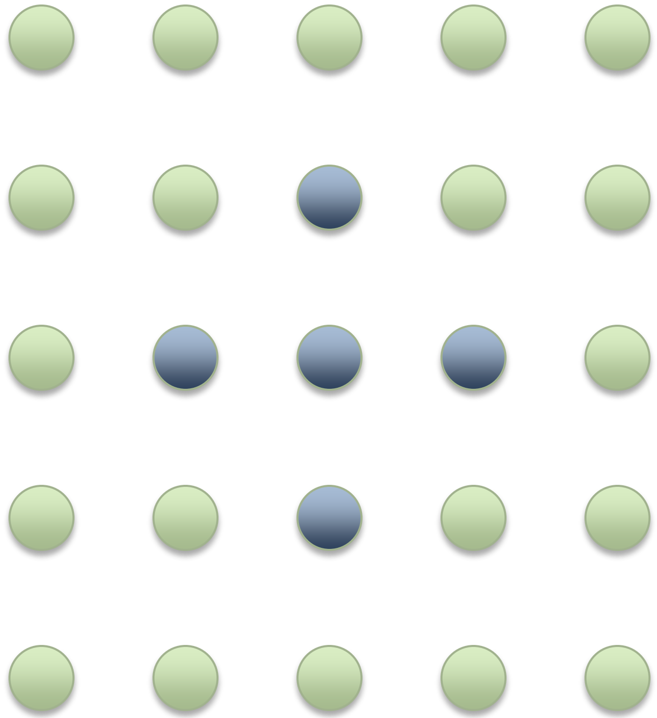
- All images get data from others simultaneously but this is optimized so communication is more balanced

```
!...
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

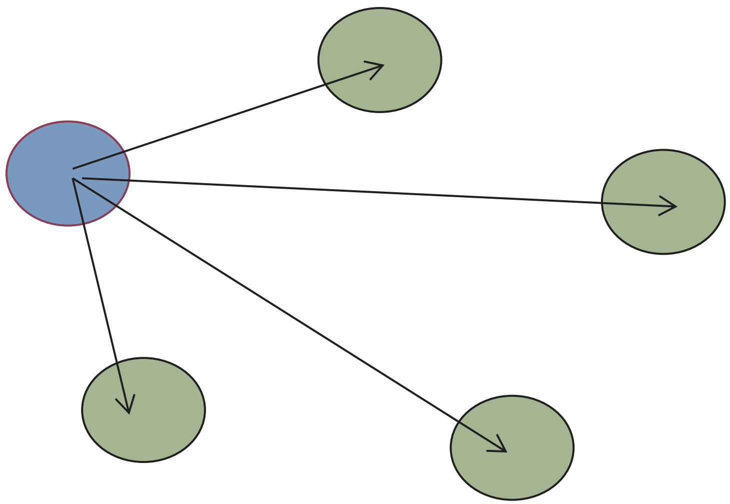
Synchronization

- 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...

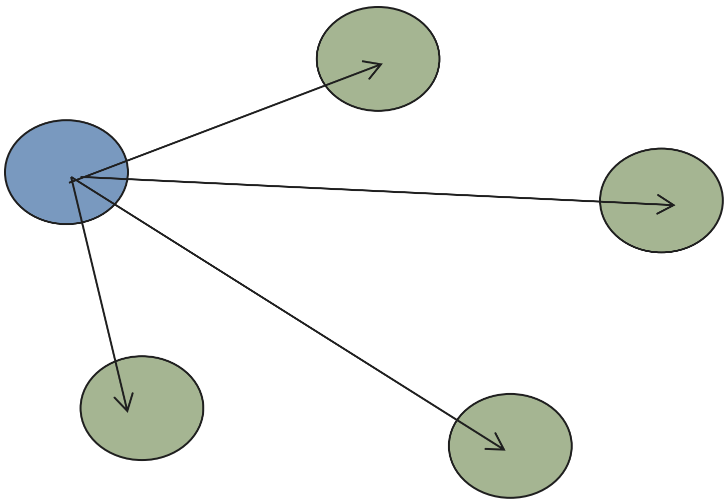


- Tempting to use **`sync all`**

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
```

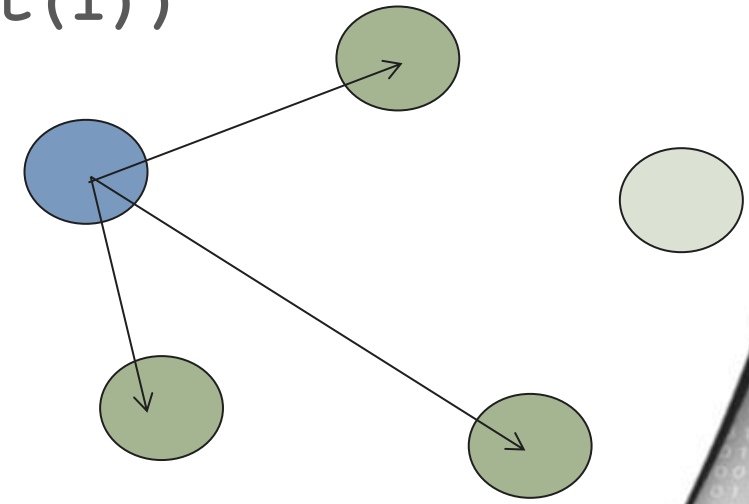


- Note that **`sync all`** is likely to be fast so is an alternative

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 you need scalability to a large number of images you may need to temporarily work around current lack of collectives
  - 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
- Allinea DDT
  - support for coarrays and UPC for a number of compilers is in public beta and will be in DDT 3.1

- Scalasca
  - Currently investigating how PGAS support can be incorporated.

Debugging Synchronisation problems

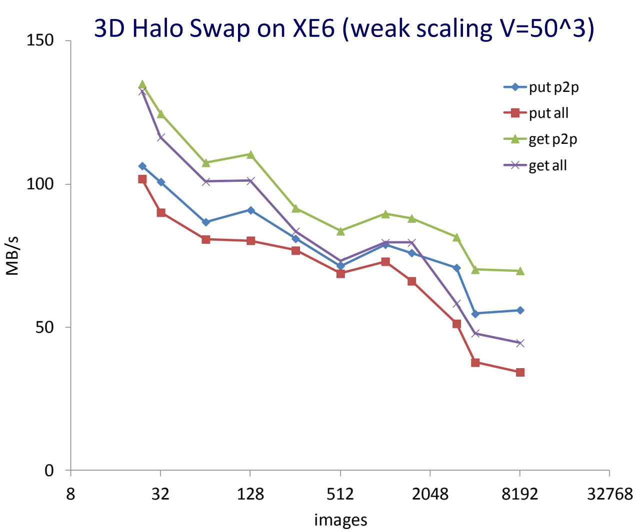
- One-sided model is tricky because subtle synchronisation errors change data
- TRY TO GET IT RIGHT FIRST TIME
  - look carefully at the remote operations in the code
  - Think about synchronisation of segments
  - especially look for early arriving communications trashing your data at the start of loops (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

Coarray Benchmark Suite

- Developed by David Henty at EPCC
- Aims to test fundamental features of a coarray implementation
- We don’t have an API to test (cf. IMB for MPI)
- We can test basic language syntax for communication of data and synchronization
- Need to choose communication pattern and data access
- There is some scope for a given communication pattern:
  - array syntax, loops over array elements
  - inline code or use subroutines
  - Choices can reveal compiler capabilities

Coarray Benchmark Suite…

- Useful to find out
- Basic latencies and bandwidths
- Language-specific aspects (strided transfers for example)
- Regimes where particular choices are best
  - For example how to do halo-swap
  - Does sync all beat sync images?



CRAY THE SUPERCOMPUTER COMPANY | epcc |

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
  blocksize = npuzzles /(8* 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[*]
```

CRAY THE SUPERCOMPUTER COMPANY | epcc

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
```

CRAY THE SUPERCOMPUTER COMPANY | epcc

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
```

THE SUPERCOMPUTER COMPANY

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 TS) 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.

CRAY THE SUPERCOMPUTER COMPANY | epcc

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
```

CRAY THE SUPERCOMPUTER COMPANY | epcc

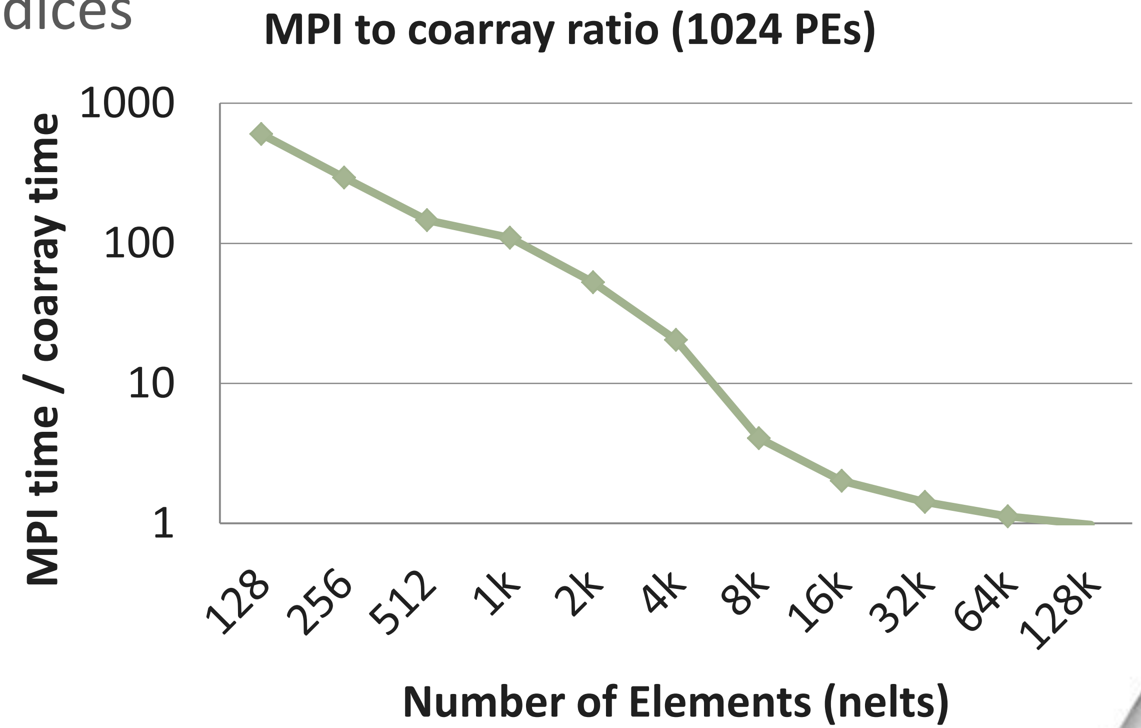
Remote gather: coarray implementation (get)

- 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

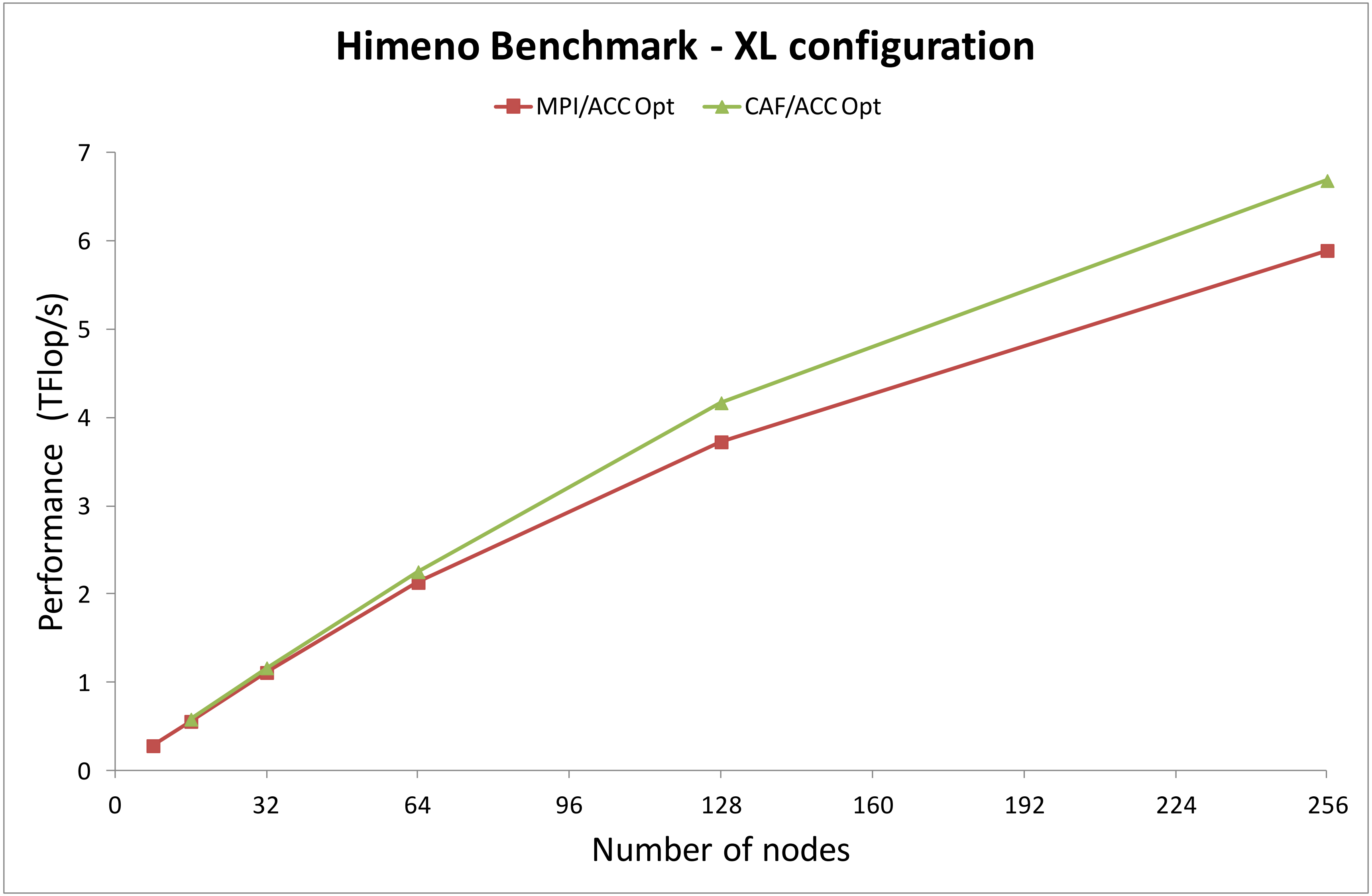


THE SUPERCOMPUTER COMPANY

HIMENO

- HIMENO Halo-Swap benchmark
- Uses Jacobi method to solve Poisson's equation
- Looked at a distributed implementation for GPUs
- When distributed this gives a stencil computation and halo-swap communication.
  - Used draft OpenMP GPU directives stencil computation
  - used MPI or coarrays for halo-swap between processes
- Coarray code for halo-swap was simple and was best performing of the optimized versions
- There is still scope to optimize the coarray version (reduce extra data copy)

HIMENO

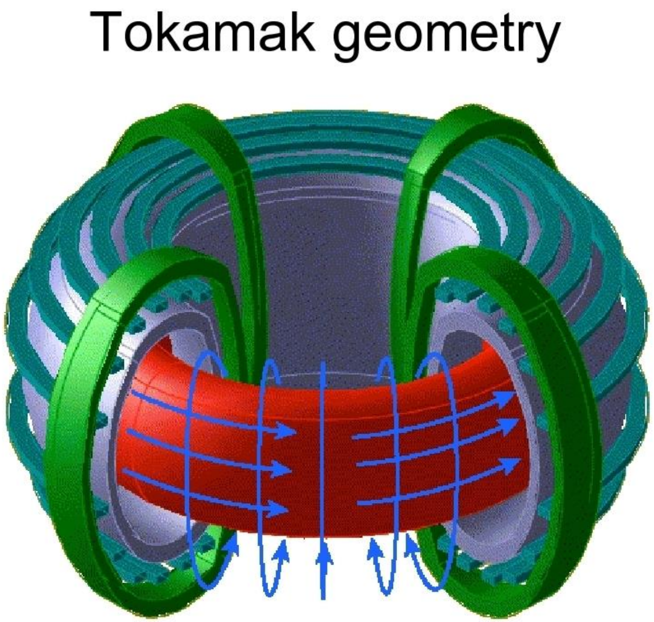


Himeno Benchmark - XL configuration
MPI/ACC Opt
CAF/ACC Opt
Performance (TFlop/s)
0
1
2
3
4
5
6
7
0
32
64
96
128
160
192
224
256
Number of nodes

Gyrokinetic Fusion Code

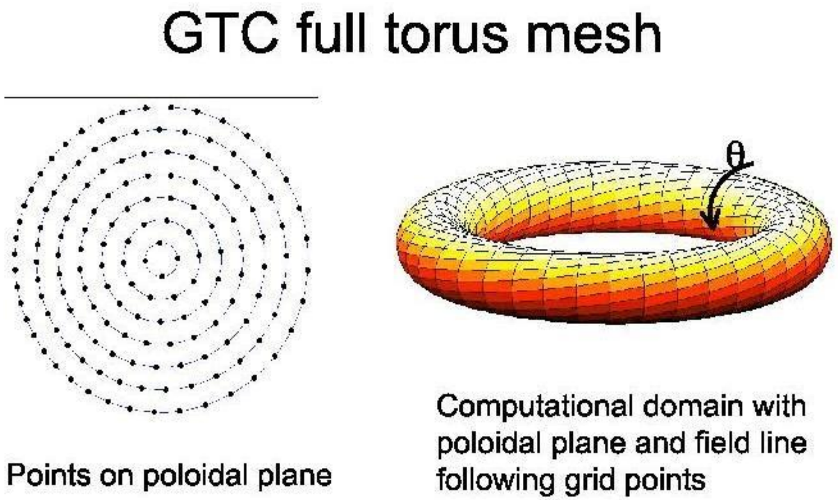
- Tokamak fusion code for transport of charged particles
- Involved were: Robert Preissl, Nathan Wichmann, Bill Long, John Shalf, Stephane Ethier, Alice Koniges
  from Lawrence Berkeley National Lab (LBNL), Cray Inc and Princeton Plasma Physics Laboratory (PPPL)
- Optimized Hybrid MPI/OpenMP kernels replace with PGAS (coarray)/OpenMP

Gyrokinetic Fusion Code



- Particle in Cell (PIC) approach to simulate motion of confined particles
- Motion caused by electromagnetic force on particle

Gyrokinetic Fusion Code



- Many particles stay in a cell for small time step but some don't
- Timestep chosen to limit travel to 4 cells away
- Departing particles stored in a buffer and when this is full the data is sent to the neighboring cell's incoming buffer
- Force fields recomputed once particles are redistributed
- Coarrays used to avoid coordinating the receive of the data: >40% improvement at 27,160 processes/images
- SC11 paper

ECMWF Integrated Forecasting system (IFS)

- IFS is ECMWF's production forecasting code
- Significant work undertaken to optimize transforms and transposes:
  - Overlap Legendre transforms and associated transpositions
  - Overlap Fourier transforms and associated transpositions
- Work was undertaken as part of the EU *Collaborative Research into Exascale Systemware, Tools and Applications* (CRESTA) project
- Monolithic MPI Alltoallv implementation replaced by coarray puts directly from some OpenMP loops
  (data sent as soon as available)
- Use of Fortran coarrays was natural choice for ECMWF
- Very promising scalability improvement for large cases

LTINV Recoding: From MPI to coarrays

```
!$OMP PARALLEL DO SCHEDULE(DYNAMIC,1) PRIVATE(JM,IM)
DO JM=1,D%NUMP
  IM = D%MYMS(JM)
  CALL
LTINV(IM,JM,KF_OUT_LT,KF_UV,KF_SCALARS,KF_SCDERS,ILEI2,IDIM1,&
   & PSPVOR,PSPDIV,PSPSCALAR ,&
   & PSPSC3A,PSPSC3B,PSPSC2 , &
   & KFLDPTRUV,KFLDPTRSC,FSPGL_PROC)
ENDDO
!$OMP END PARALLEL DO
DO J=1,NPRTRW
  ILENS(J) = D%NLTSFTB(J)*IFIELD
  IOFFS(J) = D%NSTAGT0B(J)*IFIELD
  ILENR(J) = D%NLTSGTB(J)*IFIELD
  IOFFR(J) = D%NSTAGT0B(D%MSTABF(J))*IFIELD
ENDDO
CALL MPL_ALLTOALLV(PSENDBUF=FOUBUF_IN,KSENDCOUNTS=ILENS,&
 & PRECVBUF=FOUBUF,KRECVCOUNTS=ILENR,&
 & KSENDDISPL=IOFFS,KRECVDISPL=IOFFR,&
 & KCOMM=MPL_ALL_MS_COMM,CDSTRING='TRMTOL:')
```

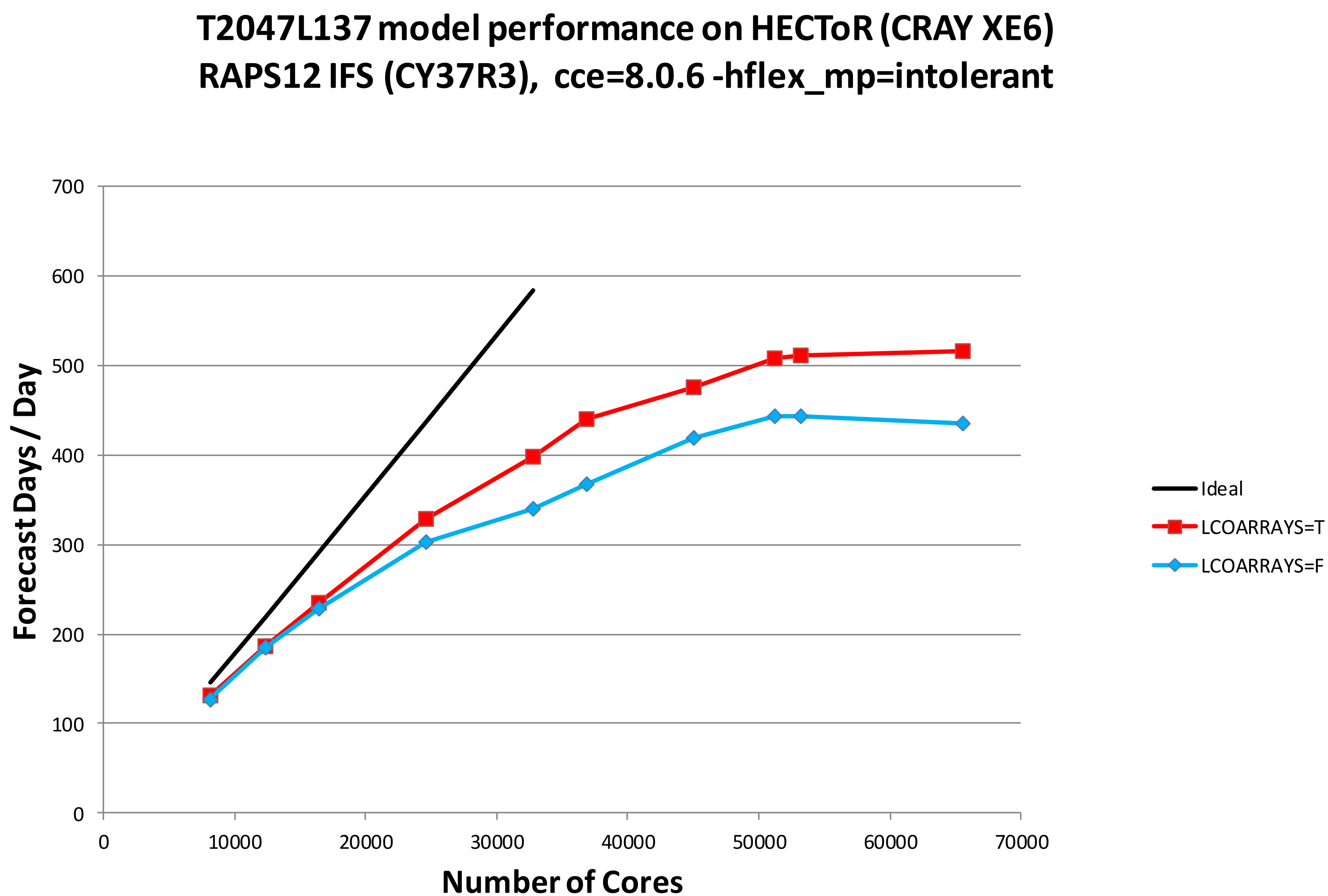
COMPUTE
COMMUNICATION

LTINV Recoding: From MPI to coarrays

```
!$OMP PARALLEL DO SCHEDULE(DYNAMIC,1)
  PRIVATE(JM,IM,JW,IPE,ILEN,ILENS,IOFFS,IOFFR)
DO JM=1,D%NUMP
  IM = D%MYMS(JM)
  CALL LTINV(IM,JM,KF_OUT_LT,KF_UV,KF_SCALARS,KF_SCDERS,ILEI2,IDIM1,&
    & PSPVOR,PSPDIV,PSPSCALAR ,&
    & PSPSC3A,PSPSC3B,PSPSC2 , &
    & KFLDPTRUV,KFLDPTRSC,FSPGL_PROC)
  DO JW=1,NPRTRW
    CALL SET2PE(IPE,0,0,JW,MYSETV)
    ILEN = D%NLEN_M(JW,1,JM)*IFIELD
    IF( ILEN > 0 )THEN
      IOFFS = (D%NSTAGT0B(JW)+D%NOFF_M(JW,1,JM))*IFIELD
      IOFFR = (D%NSTAGT0BW(JW,MYSETW)+D%NOFF_M(JW,1,JM))*IFIELD
      FOUBUF_C(IOFFR+1:IOFFR+ILEN)[IPE]=FOUBUF_IN(IOFFS+1:IOFFS+ILEN)
    ENDIF
    ILENS = D%NLEN_M(JW,2,JM)*IFIELD
    IF( ILENS > 0 )THEN
      IOFFS = (D%NSTAGT0B(JW)+D%NOFF_M(JW,2,JM))*IFIELD
      IOFFR = (D%NSTAGT0BW(JW,MYSETW)+D%NOFF_M(JW,2,JM))*IFIELD
      FOUBUF_C(IOFFR+1:IOFFR+ILENS)[IPE]=FOUBUF_IN(IOFFS+1:IOFFS+ILENS)
    ENDIF
  ENDDO
ENDDO
!$OMP END PARALLEL DO
SYNC IMAGES(D%NMYSETW)
FOUBUF(1:IBLEN)=FOUBUF_C(1:IBLEN)[MYPROC]
```

COMPUTE
COMMUNICATION

IFS scaling with coarrays



© 2012 ECMWF, Used by permission

References

- "Cray's Approach to Heterogeneous Computing",
  R. Ansaloni, A. Hart, Parco 2011 (to appear).
  - "The Himeno benchmark", *Ryutaro Himeno*, http://accc.riken.jp/HPC_e/himenobmt_e.html.
- "Multithreaded Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms", *Robert Preissl, Nathan Wichmann, Bill Long, John Shalf, Stephane Ethier, Alice Koniges*, SC11 best paper finalist
- "A PGAS implementation by co-design of the ECMWF Integrated Forecasting system," George Mozdzynski,, 15th Workshop on High Performance Computing in Meteorology, 1-5 October 2012.

References

- http://lacsi.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

The material for this tutorial is based on original content developed by EPCC of The University of Edinburgh for use in teaching their 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 © 2013
by The University of Edinburgh and Cray Inc.