Modern Fortran: Objects and Coarrays

Robert W. Numrich

CUNY High-Performance Computing Center
City University of New York
New York, NY
Modern Fortran

- “The report of Fortran’s death is an exaggeration.” Mark Twain
- Even Latin is making a comeback in high schools and colleges.
- The current standard is Fortran 2008.
  - Not FORTRAN 77 anymore
  - Fortran 90, Fortran 95, Fortran 2003, Fortran 2008
- If you haven’t been paying attention, you have missed a lot.
Object-oriented
  - Objects
    - Classes: User-defined derived types
    - Type-bound data components
    - Type-bound procedures
    - Type constructors
    - Type finalization
  - Abstract types
  - Inheritance
  - Deferred procedure bindings
  - Overloaded generic procedures
  - Polymorphism
The coarray programming model is the first parallel model added to the language.

- Execution model
  - SPMD programming model (MPI rank ⇔ CAF image)
  - Explicit synchronization

- Memory model
  - Explicit data decomposition by the programmer
  - Explicit data movement by the programmer using codimensions
Compilers that support coarrays

- Vendors
  - Cray has supported coarrays for over ten years
  - Intel has released a compiler
  - IBM under development
  - Nothing from PGI, others

- Open Source Compilers
  - gfortran has started development
  - g95 has gone dormant

- Academic research compilers
  - Rice University project
  - University of Houston project
New languages

- We do not need new languages.
- We do need better compilers.
- We do need better run-time systems.
- We do need better hardware.
Limitations beyond control of the language

- Shared caches with unknown design
- Cache coherency protocols
- Memory partitioning algorithms used by the run-time system
- Overheads for spawning threads
- Bandwidth/latency to local memory
- Bandwidth/latency to remote memory
- Cache contention, thrashing
- Memory bus contention
- Memory bank contention
- TLB reach
Parallel programming models

- The main things that distinguish parallel programming models:
  - Global view vs. local view
  - How to go back and forth between the two views
  - Start with a single thread of control or start with multiple threads of control
  - Programmer synchronizes or the language synchronizes
  - Programmer writes higher level abstractions or the language provides them

- The coarray model:
  - Local view is default.
  - Programmer defines how to go back and forth between local and global views.
  - The coarray execution model is SPMD, asynchronous threads of control.
  - Programmer provides explicit synchronization.
  - Programmer writes distributed data structures.
The coarray parallel programming model

- SPMD with a fixed number of *virtual* images
  - A single program is replicated a fixed number of times.
  - `num_images()` returns the number of images at run-time
  - `this_image()` returns the local image index
  - The run-time system assigns physical memory to each image.
  - The run-time system assigns a physical core to an image to perform work on data in the image’s local memory.
  - Memory accesses across images are dereferenced by multi-rank codimensions.
    - All variables are local to an image.
    - Only variables declared with codimensions are visible across images.
    - Codimensions are a *logical* statement of how the programmer wants to decompose the problem.
  - Full barrier, partial barrier, locks.
  - `allocate` and `deallocate` of coarrays are collective across all images.
Assigning virtual images to cores

One-to-one
  ▶ one core to one image

Many-to-one
  ▶ many cores to one image (hybrid model with OpenMP threads)

One-to-many
  ▶ one core to many images (virtual processors Charm++, Thinking Machines)

Many-to-many
  ▶ many cores to many images (virtual processors combined with OpenMP threads)
Coarray syntax

\begin{verbatim}
real :: x(n)[*]
real :: y(n)

y(:) = x(:)  ! local read
y(:) = x(:)[p]  ! remote read from image p
x(:)[q] = y(:)  ! remote write to image q
\end{verbatim}
Non-blocking memory references

- Coarray syntax allows interleaving of computation and communication.
  - The hardware must be designed to allow this to happen.
  - The compiler must be able to generate code to make it happen.
- Writing code that does this is incompatible with object-oriented design.
real, allocatable :: a[:, :, :]

\[ p = \text{coresPerChip}() \]

\[ q = \text{chipsPerNode}() \]

\[ r = \text{nodesPerSystem}() \]

allocate(a[p, q, :])

\[ x = a \quad \text{! local reference} \]

\[ x = a[:, q, r] \quad \text{! on-chip reference} \]

\[ x = a[p, :, r] \quad \text{! on-node reference} \]

\[ x = a[p, q, :) \quad \text{! off-node reference} \]

- This requires interaction with run-time system to partition memory correctly.
Coarrays and GPUs

- Compilers should be able to generate code for GPUs (Cray, PGI).
- GPUs look a lot like long-vector machines such as Cyber 205, CM5, MASPAR etc.
  - The higher the peak the more unbalanced the machine.
- Two ways for the run-time system to define images:
  - An image is associated with a processor and its memory; the GPU with its memory is an accelerator. (acceleration)
  - An image is associated with a GPU and its memory; the processor is an auxiliary device for network and I/O support. (reverse acceleration)
- Extending the coarray model to GPUs is NOT being considered.
Coarray objects

```fortran
!-type Y
  real, allocatable :: x(:)
end type

!-type(Y) :: A[*]
real, allocatable :: y(:)
allocate(y(n)) ! local heap
allocate(A%x(n)) ! local heap

y(:)=A[p]%x(:) ! read data from remote heap on image p

Out=A%method(In) ! methods associated with objects
Out=A[p]%method(In) ! remote procedure invocation (NOT being considered)
```
Example Program

program LU
    use CafLib
    type(BlockR8Matrix) :: A[*]
    type(PivotVector) :: Pivot[*]
    type(DiskFile) :: file
    integer, parameter :: n=1000, block=100, p=4, q=8

    A = BlockMatrix(n,block,n,block,p,q)
    Pivot = PivotVector(A)
    file = DiskFile('filename')

    err = A%readBlockMatrix(file)
    err = A%LU(Pivot)
    err = A%writeBlockMatrix(file)

end program LU

▶ Compare this code with code needed to use ScaLapack!
Data Decomposition and Distribution via Maps

- Abstract Map
  - ObjectMap
    - VectorMap
      - DenseMatrixMap
        - DenseGridMap
      - SparseMatrixMap
        - SparseField
          - SparseGridMap
    - DenseField
Distributed fields

Abstract Field
- data(:,:)\[p,*]\n- getData()
- setData()

Distributed Field
- Map
- Grid
- integrator()
- exchange()

Abstract Grid
- LonLat Grid
- DistrLonLat Grid

.... passed to kernels ....

Abstract Map
- Iterator

Distributed Map
- co-arrays
- co-arrays
Driver-kernal, unified memory model

type(PhysicalField) :: A
type(Iterator) :: iterator
real,contiguous,pointer :: ptrA(:,:)

.....
A = PhysicalField(nx,ny,p,q)
..

do while(t < tMax)
  iterator = A%getGlobalIterator()
  do while(iterator%hasNext())
    ptrA => iterator%requestField()
    call kernel(ptrA)
    ptrA => iterator%releaseField()
  end do
end do
Summary

- Fortran is still the best language for scientific computing.
- All the features of a modern language, which people said they needed to write modern applications, are part of the language.
- The rich collection of high-performance kernels still work.
- The coarray model in F2008 is the first time a parallel programming model has been made a standard feature of the language.
- The coarray model is SPMD, similar to the MPI model.
  - Coarray syntax is much simpler than MPI syntax.
  - Problem decomposition is about the same difficulty as MPI.
  - Coarray synchronization is about the same difficulty as MPI synchronization.
- Coarrays work best on hardware with a true global address space.
- Object-oriented Fortran:
  - Will it save Fortran?
  - Will it kill Fortran?
M. Metcalf, J. Reid and M. Cohen, Modern Fortran Explained, Oxford University Press (2011)


