Coarray Fortran Tutorial: Parallel Programming in Fortran 2018

Damian Rouson
Berkeley Lab

The International Conference for High Performance Computing, Networking, Storage, and Analysis 2023 Tutorial

go.lbl.gov/sc23
Acknowledgements

This presentation includes efforts on the part of contributors to the GASNet-EX, Matcha, and OpenCoarrays software libraries and members of the Computer Languages and Systems Software (CLaSS) Group and our collaborators:

Amir Kamil, Dan Bonachea, Paul Hargrove, Tobias Burnus, Alessandro Fanfarillo, Daniel Ceils Garza, Ethan Gutmann, Jeff Hammond, Peter Hill, Paul Hargrove, Dominick Martinez, Katherine Rasmussen, Soren Rasmussen, Brad Richardson, Sameer Shende, David Torres, Andre Vehreschild, Jordan Welsman, Nathan Weeks, Yunhao Zhang

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation’s exascale computing imperative.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231, as well as This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
Introduction to Coarray Fortran ("CAF")
• Motivation: Why Fortran, CAF philosophy
• SPMD parallel execution: Images
• PGAS data structures: Coarrays
• Example Application: Matcha
• Compiling and running "Hello, world!"

Break

A deeper dive
Why Fortran Matters

Weather & Climate

Intermediate Complexity Atmospheric Research (ICAR) Model
Courtesy of Ethan Gutmann, NCAR

Nuclear Energy

U.S. Nuclear Regulatory Commission File Photo

Aerospace

FUN3D Mesh Adaptation for Mars Ascent Vehicle, Courtesy of Eric Nielsen & Ashley Korzun, NASA Langley
The underlying philosophy of our design is to make the smallest number of changes to the language required to obtain a robust and efficient parallel language without requiring the programmer to learn very many new rules.


**Seminal paper:**

Single Program Multiple Data (SPMD) parallel execution
— Synchronized launch of multiple “images” (process/threads/ranks)
— Asynchronous execution except where program explicitly synchronizes
— Error termination or synchronized normal termination

```
program main
  implicit none
  print *, "Hello from image ", this_image(), "of", num_images()
end program
```
1. After the creation of a fixed number of images, each image’s first “segment” (sequence of statements) executes.
2. Image control statements totally order segments executed by a single image and partially order segments executed by separate images.
Partitioned Global Address Space (PGAS)

Coarrays:
- Distributed data structures — greeting
- Facilitate Remote Memory Access (RMA) — line 15

```fortran
program main
  !! One-sided communication of distributed greetings
  implicit none
  integer, parameter :: max_greeting_length=64, writer = 1
  integer image
  character(len=max_greeting_length) :: greeting[*] ! scalar coarray

  associate(me => this_image(), ni=>num_images())

  write(greeting,*),"Hello from image",me,"of",ni ! local (no "[]")
  sync all ! image control

  if (me == writer) then
    do image = 1, ni
      print *,greeting[image] ! one-sided communication: "get"
    end do
  end if

end associate
end program
```
Coarrays

- Non-allocatable (static):
  ```
  character(len=max_greeting_length) :: greeting[*]
  ```

- Dynamically allocatable:
  ```
  real(rkind), allocatable :: halo_x(:,::)[*]
  ```

- Derived type components:
  ```
  type global_field_t
      real, allocatable :: values_(:)[*]
  end type
  ```

- Local coarrays:
  ```
  subroutine gather_image_numbers
      integer, allocatable :: images(:,::)[*]
      allocate(images(num_images())[*])
  end subroutine
  ```

- Derived type coarrays:
  ```
  type payload_list_t
      type(payload_t), allocatable :: payloads(:)
  end type

  type(payload_list_t), allocatable :: mailbox[*]
  ```

A coarray is a data entity that has nonzero corank; it can be directly referenced or defined by other images. It may be a scalar or an array.

For each coarray on an image, there is a corresponding coarray with the same type, type parameters, and bounds on every other image of a team in which it is established.

=> Symmetric memory
if intrinsic-type coarray

}  

Allow for asymmetric memory
Application:
— Matcha: Motility Analysis of T Cells in Activation
— Matching the speed & turning angle distributions to observed T cells, simulations can explore large spatial volumes and parameter spaces.

Programming models:
— Coarray halo exchanges in a 3D diffusion PDE solver.
— Do concurrent for automatic GPU offloading

Highlights:
— This tutorial’s 2D heat equation solver was the prototype for the 3D diffusion solver.

https://go.lbl.gov/matcha

Compiling & Running hello.f90
Introduction to Coarray Fortran (“CAF”)

Break

A deeper dive

- Heat equation:
  - Numerical algorithm
  - Abstract calculus design pattern
  - Halo exchanges
  - Performance analysis

- CAF Overview:
  - Image enumeration
  - Synchronization
  - Collective subroutines
  - Events
  - Example: FEATS task scheduler
Heat Equation

\[ \frac{\partial T}{\partial t} = \alpha \nabla^2 T \]

\[ \{T\}^{n+1} = \{T\}^n + \Delta t \cdot \alpha \cdot \nabla^2 \{T\}^n \]

T = T + dt \ast \text{alpha} \ast \text{.laplacian.} \ T
Heat Equation

\[ \frac{\partial T}{\partial t} = \alpha \nabla^2 T \]

\[ \{T\}^{n+1} = \{T\}^n + \Delta t \cdot \alpha \cdot \nabla^2 \{T\}^n \]

\[ T = T + dt \times \text{alpha} \times \text{laplacian} \cdot T \]

local objects

pure user-defined operators

```
cd fortran
make run-heat-equation
```
Halo Exchange

```
116  real(rkind), allocatable :: halo_x(:,,:)[:]
117  integer, parameter :: west=1, east=2

134  me = this_image()
135  num_subdomains = num_images()
137  my_nx = nx/num_subdomains + merge(1, 0, me <= mod(nx, num_subdomains))

232  subroutine exchange_halo(self)
233     class(subdomain_2D_t), intent(in) :: self
234     if (me>1) halo_x(east,:)[:me-1] = self%s_(:1,:)
235     if (me<num_subdomains) halo_x(west,:)[:me+1] = self%s_(my_nx,:)
236  end subroutine
```
Loop-Level Parallelism

188 do concurrent(j=2:ny-1)
189    laplacian_rhs%s_(i, j) = &
190    (halo_left(j) - 2*rhs%s_(i, j) + rhs%s_(i+1,j))/dx_**2 + &
191    (rhs%s_(i, j-1) - 2*rhs%s_(i, j) + rhs%s_(i ,j+1))/dy_**2
191 end do
Coarray Fortran began as a syntactically small extension to Fortran 95:
— Square-bracketed “cosubscripts” distribute & communicate data
Integration with other features:
— Array programming: colon subscripts
— OOP: distributed objects
Minimally invasive:
— Drop brackets when not communicating
Communication is explicit:
— Use brackets when communicating
Image Enumeration

🔥 Obtaining an image index:

\texttt{this\_image([team])} \quad \texttt{image\_index(coarray, sub, team\_number)}
\texttt{this\_image(coarray [,team])} \quad \texttt{image\_index(coarray, sub, team)}
\texttt{this\_image(coarray, dim [,team])} \quad \texttt{image\_index(coarray, sub)}

🔥 Obtaining an image count:

\texttt{num\_images()} \quad \texttt{image\_index(a, [3]), image\_index(b, [0, 0])}
\texttt{num\_images(team)} \quad \texttt{print \*, image\_index(a, [3]), image\_index(b, [0, 0])}
\texttt{num\_images(team\_number)} \quad \texttt{print \*, lcobound(a), ucobound(a)}
Synchronization

アイメージバリアーや“meet-ups”:

```plaintext
sync all(stat, errmsg)
sync images(image-set, stat, errmsg)
allocate()  }
for coarrays only, including implicit (de)allocation at end of a block or procedure
deallocate()

stop  stop_code (integer or character codes allowed)
end  program

call move_alloc(from, to) with coarray arguments.

Any statement causing an implicit coarray deallocation by completing a block or procedure.

Deprecated by Metcalf, Reid & Cohen (2018):

sync memory(stat, errmsg)
```
Other Image Control Statements

**Locks:**

\[
\text{lock(lock-variable, errmsg)}
\]

\[
\text{unlock(lock-variable, stat, errmsg)}
\]

**Critical blocks:**

\[
\text{critical(stat, errmsg)}
\]

**Teams**

\[
\text{form team(team_number, team_variable)}
\]

\[
\text{change team(team_value, ...)}
\]

**Events**

\[
\text{event post(event-variable, stat, errmsg)}
\]

\[
\text{event wait(event-variable, stat, errmsg)}
\]
Collective Subroutines

Behavior:

— Successful execution of a collective subroutine performs a calculation on all the images of the current team and assigns a computed value on one or all of them.

— If it is invoked by one image, it shall be invoked by the same statement on all active images of its current team in segments that are not ordered with respect to each other.

— Corresponding references participate in the same collective computation.

Complete list:

— co_sum(a, result_image, stat, errmsg)
— co_max(a, result_image, stat, errmsg)
— co_min(a, result_image, stat, errmsg)
— co_broadcast(a, source_image, stat, errmsg)
— co_reduce(a, operation, result_image, stat, errmsg)
**co_sum**

```plaintext
co_sum(a, result_image, stat, errmsg)
```

*Argument a*
- shall be of numeric type,
- shall have the same shape, type, & type parameter values, in corresponding references.
- shall not be a coindexed object
- is an `intent(inout)` argument

*Argument result_image (optional)*
- shall be of scalar type `integer`
- is an `intent(in)` argument

- If present, it shall be present on all images of the current team, have the same value on all images of the current team, and shall be an image index of the current team
co\_sum

Team 1

Image 1

\[ a(1:4)[1] \]

\[ \begin{array}{c}
0 \\
5 \\
3 \\
6 \\
\end{array} \]

Image 2

\[ a(1:4)[2] \]

\[ \begin{array}{c}
2 \\
6 \\
5 \\
1 \\
\end{array} \]

Image 3

\[ a(1:4)[3] \]

\[ \begin{array}{c}
3 \\
4 \\
9 \\
7 \\
\end{array} \]

co\_sum(a)

Team 2

Image 4

\[ a(1:4)[4] \]

\[ \begin{array}{c}
3 \\
4 \\
5 \\
1 \end{array} \]

Image 5

\[ a(1:4)[5] \]

\[ \begin{array}{c}
2 \\
4 \\
3 \\
9 \end{array} \]

Image 6

\[ a(1:4)[6] \]

\[ \begin{array}{c}
4 \\
4 \\
0 \\
1 \end{array} \]

co\_sum(a)
**co_broadcast**

```c
co_broadcast(a, source_image, stat, errmsg)
```

**Argument** `a`
- shall have the same shape, dynamic type, & type parameter values, in corresponding references.
- shall not be a coindexed object
- is an `intent(inout)` argument
- successful execution causes `a` to become defined as if by intrinsic assignment on all images in the current team with the value of `a` on the `source_image`

**Argument** `source_image`
- shall be of scalar type `integer`
- is an `intent(in)` argument
- If present, it shall be present on all images of the current team, have the same value on all images of the current team, and shall be an image index of the current team
co_broadcast

Team 1

Image 1: \[ a(1:4)[1] \]
\[ \begin{bmatrix} 1 & 5 & 3 & 6 \end{bmatrix} \]

Image 2: \[ a(1:4)[2] \]
\[ \begin{bmatrix} 2 & 6 & 5 & 1 \end{bmatrix} \]

Image 3: \[ a(1:4)[3] \]
\[ \begin{bmatrix} 3 & 4 & 9 & 7 \end{bmatrix} \]

Team 2

Image 4: \[ a(1:4)[4] \]
\[ \begin{bmatrix} 3 & 4 & 5 & 1 \end{bmatrix} \]

Image 5: \[ a(1:4)[5] \]
\[ \begin{bmatrix} 2 & 4 & 3 & 9 \end{bmatrix} \]

Image 6: \[ a(1:4)[6] \]
\[ \begin{bmatrix} 4 & 4 & 0 & 1 \end{bmatrix} \]

co_broadcast(a,1)
Argument `a`

- shall be `intent(inout)`, non-polymorphic and not coindexed
- shall have the same shape, dynamic type, & type parameter values, in corresponding references.
- becomes the result of applying the reduction `operation` to values of `a` in the corresponding references, and likewise on an element-wise basis if `a` is an array

Argument `operation`

- shall implement an associative operation via a `pure` function with two arguments

Argument `result_image`

- shall be of scalar integer, `intent(in)` argument
- if present, it shall have the same value on all images of the current team and shall be an image index of the current team
Hands-on co_reduce

module co_all_m
  implicit none
  interface
    module subroutine co_all(a)
      implicit none
      logical, intent(inout) :: a
    end subroutine
  end interface
end module

submodule(co_all_m) co_all_s
  implicit none
  contains
  module procedure co_all
    call co_reduce(a, and)
  contains
    pure function and(lhs, rhs)
      result(lhs_and_rhs)
      logical, intent(in) :: lhs, rhs
      logical lhs_and_rhs
      lhs_and_rhs = lhs .and. rhs
    end function
  end procedure
end submodule

program main
  use co_all_m, only : co_all
  implicit none
  logical :: operand = .true.
  associate(me=>this_image())
  call co_all(operand)
  if (me==1) print *, operand
  if (me==num_images()) operand = .false.
  call co_all(operand)
  if (me==1) print *, operand
end associate
end program

https://github.com/sourceryinstitute/sourcery
program heat_equation

!! Parallel finite difference solver for the 2D, unsteady heat conduction partial differential equation
use subdomain_2D_m, only : subdomain_2D_t
use iso_fortran_env, only : int64
use kind_parameters_m, only : rkind
implicit none
type(subdomain_2D_t) T
integer, parameter :: nx = 4096, ny = nx, steps = 50
real(rkind), parameter :: alpha = 1._rkind
real(rkind) T_sum
integer(int64) t_start, t_finish, clock_rate
integer step

call T%define(side=1._rkind, boundary_val=1._rkind, internal_val=2._rkind, n=nx)! Initial/boundary cond.
call T%allocate_halo_coarray ! Implicit synchronization
associate(dt => T%dx() * T%dy() / (4*alpha)) ! set time step

call system_clock(t_start)
do step = 1, steps
   call T%exchange_halo ! put subdomain boundary values on neighboring images
   sync all
   T = T + dt * alpha * laplacian. T ! asynchronous parallel user-defined operators
   sync all
end do

end associate
T_sum = sum(T%values()) ! local sum
call co_sum(T_sum, result_image=1) ! distributed collective sum

call system_clock(t_finish, clock_rate)
if (this_image()==1) then
   print *, "walltime: ", real(t_finish - t_start, rkind) / real(clock_rate, rkind)
   print *, "T_avg = ", T_sum/(nx*ny)
end if
end program
Compiling and Running the Heat Equation Solver

cuf23-tutorial:
Events
Hello, world!

Performance-oriented constraints:
- Query and wait must be local.
- Post and wait are disallowed in do concurrent constructs.

Pro tips:
- Overlap communication and computation.
- Wherever safety permits, query without waiting.
Segment Ordering: 

Events

An intrinsic module provides the derived type `event_type`, which encapsulates an `atomic_int_kind` integer component default-initialized to zero.

- An image increments the event count on a remote image by executing `event_post`.
- The remote image obtains the post count by executing `event_query`.

<table>
<thead>
<tr>
<th>Image Control</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>event_post</code></td>
<td><code>atomic_add 1</code></td>
</tr>
<tr>
<td><code>event_query</code></td>
<td>defines count</td>
</tr>
<tr>
<td><code>event_wait</code></td>
<td><code>atomic_add -1</code></td>
</tr>
</tbody>
</table>
FEATS: Framework for Extensible Asynchronous Task Scheduling

Execution:
- In each team, establish one scheduler image and one or more compute images.
- Schedulers post task_assigned events to compute images in an order that respects dependencies in a directed acyclic graph (DAG).
- Compute images post ready_for_next_task events to scheduler.
- A task_payload_map_t abstraction maps task identifiers to locations in a payload_t mailbox coarray.

Initial target applications:
- NASA’s Online Tool for the Assessment of Radiation in Space (OLTARIS)
- NCAR’s Intermediate Complexity Atmospheric Research (ICAR) model: work-sharing/work-stealing.
- Fortran Package Manager: parallel builds.
FEATS:
Framework for Extensible Asynchronous Task Scheduling

Execution:
- In each team, establish one scheduler image and one or more compute images.
- Schedulers post task_assigned events to compute images in an order that respects dependencies in a directed acyclic graph (DAG).
- Compute images post ready_for_next_task events to scheduler.
- A task_payload_map_t abstraction maps task identifiers to locations in a payload_t mailbox coarray.

Initial target applications:
- NASA’s Online Tool for the Assessment of Radiation in Space (OLTARIS)
- NCAR’s Intermediate Complexity Atmospheric Research (ICAR) model: work-sharing/work-stealing.
- Fortran Package Manager: parallel builds.
Demo
CAF at Scale: Magnetic Fusion

**Application focus:**
- The shift phase of charged particles in a tokamak simulation code

**Programming models studied:**
- CAF + OpenMP or
- Two-sided MPI + OpenMP

**Highlights:**
- Experiments on up to 130,560 processors
- 58% speed-up of the CAF implementation over the best multithreaded MPI shifter algorithm on largest scale
- “the complexity required to implement … MPI-2 one-sided, in addition to several other semantic limitations, is prohibitive.”

Applications studied:

- Magnetohydrodynamics (MHD)
- 3D Fast Fourier Transforms (FFTs) used in infinite-order accurate spectral methods
- Multigrid methods with point-wise smoothers requiring fine-grained messaging

Programming models studied:

- CAF or
- One-sided MPI-3

Highlights:

- Simulations on up to 65,536 cores
- “… CAF either draws level with MPI-3 or shows a slight advantage over MPI-3.”
- “CAF and MPI-3 are shown to provide substantial advantages over MPI-2.
- “CAF code is of course much easier to write and maintain…”

CAF at Scale: Weather

Application:
- European Centre for Medium Range Weather Forecasts (ECMWF) operational weather forecast model

Programming models studied:
- CAF or
- Two-sided MPI

Highlights:
- Simulations on > 60K cores
- Performance improvement from switching to CAF peaks at 21% around 40K cores

Development and performance comparison of MPI and Fortran Coarrays within an atmospheric research model

Extended Abstract
Soren Rasmussen, Ethan D Gutmann, Brian Friesen, Damian Rouson, Salvatore Filippone, Irene Moulitsas

ABSTRACT

Cray applications (The Intermediate Complexity Atmospheric Research (ICAR) model) allow an opportunity to compare the costs and performance of the Message Passing Interface (MPI) versus coarray Fortran. Two methods of communication across processes are compared in this paper. The application uses two-way parallel communication of halo regions, which is performed with either MPI or coarrays. The MPI communication is done using non-blocking two-sided communication, while the coarray library is implemented using a one-sided MPI or OpenSHMEM communication backend. We examine the development costs in addition to strong and weak scalability results to understand the performance costs.

1 INTRODUCTION

1.1 Motivation and Background

In high-performance computing, MPI has been the de facto method for inter-node communication across a system's nodes for many years. MPI v1 was released in 1994 and research and development has continued across academia and industry. A method in Berkeley in 2000, known as coarray Fortran, was introduced to express the parallel programming model for distributed memory platforms in an extension to Fortran that was introduced by Robert W.巢 in 1998 [1]. Coarray Fortran, like MPI, is a single-program, multiple-data (SPMD) programming technique. Coarray Fortran's single program is replicated across multiple processors that are not co-located within the same memory.

Coming Soon to a Computer Screen Near You

☕ Fortran 2023
  - Reductions in do concurrent
  - Notified access for remote coarray data

☕ Fortran 202Y (Y ~ 8)
  - Type-safe generic programming
  - Task-based parallel programming