Additional Parallel Features in Fortran

An Overview of ISO/IEC TS 18508

Dr. Reinhold Bader
Leibniz Supercomputing Centre
Introductory remarks

Technical Specification – a „Mini-Standard“
- permits implementors to work against a stable specification
- will be eventually integrated with mainline standard (ISO/IEC 1539-1)
- modulo „bug fixes“ (e.g., issues with semantics that are identified during implementation)

Purpose of TS 18508:
- significantly extends the parallel semantics of Fortran 2008 (only a baseline feature set was defined there)
- extensive re-work of some parallel features pulled from Fortran 2008 during its development
  - many improvements based on the concepts developed in the group of John Mellor-Crummey at Rice University
- new feature: resiliency (controversial)
- however: parallel I/O is (somewhat unfortunately) not covered

Current TS draft
- DTS submitted for SC22 vote

Download from [http://bitly.com/sc22wg5](http://bitly.com/sc22wg5) ➔ 2015 ➔ N 2056
Recall coarray programming model (1)

- **Coarray declaration**
  - **symmetric** objects

  ```fortran
  integer :: b(3)
  integer :: a(3)[*]
  ```

- **Execute with 4 images**

  ```plaintext
  Image 1  2  3  4
  ```

- **Cross-image addressing**

  ```fortran
  if (this_image() == p) &
  b = a(:,)[q]
  ```

  "pull" (vs. "push")

  **one-sided communication**

  between images p and q

  - simplest case
  - coindexed reference

  Difference between A and B?

  ```plaintext
  ```

  ```plaintext
  B(1)  B(1)  B(1)  B(1)
  B(2)  B(2)  B(2)  B(2)
  B(3)  B(3)  B(3)  B(3)
  ```

©2015 LRZ

Additional Parallel Features in Fortran
Recall coarray programming model (2)

Asynchronous execution

\[ a = \ldots \]
\[ \text{if (this_image() == p) \& \&} \]
\[ b = a(:)[q] \]

\[ a = \ldots \]
\[ \text{if (this_image() == p) \& \&} \]
\[ \text{sync all} \]
\[ b = a(:)[q] \]

Image control statements

\[ a = \ldots \]
\[ \text{if (this_image() == p) \& \&} \]
\[ b = a(:)[q] \]

\[ a = \ldots \]
\[ \text{if (this_image() == p) \& \&} \]
\[ \text{sync all} \]
\[ b = a(:)[q] \]

- causes race condition \( \rightarrow \) violates language rules
- enforce segment ordering: \( q_1 \) before \( p_2 \), \( p_1 \) before \( q_2 \)

programmer's responsibility
Weaknesses of existing synchronization concept

- **Global barrier must be executed collectively**
  - All images must wait until barrier is reached
  - Load imbalanced applications may suffer more performance loss than necessary

- **Symmetric synchronization is overkill**
  - The ordering of \( p_1 \) before \( q_2 \) is not needed
  - Image \( q \) therefore might continue without waiting

- **Therapy:** TS 18508 introduces a **lightweight, one-sided** synchronization mechanism – **Events**

```fortran
use, intrinsic :: iso_fortran_env
type(event_type) :: ev[*]
```

facilitates producer/consumer scenarios

image subset synchronization (context-unsafe!) or mutual exclusion can also be used, but are still too heavyweight.

special opaque derived type; all its objects must be coarrays
Synchronization with Events

**Image q executes**

\[
a = \ldots \\
event \text{ post ( ev[p] )}
\]

- and continues **without** blocking

**Image p executes**

\[
event \text{ wait ( ev )}
\]

\[
b = a( :) [q]
\]

- the WAIT statement blocks until the POST has been received

---

**One sided segment ordering**

- \( q_1 \) ordered before \( p_2 \)
- no other ordering implied
- no other images involved

**EVENT_QUERY intrinsic**

- read event count without synchronization

---

- event variable has an internal counter with default value zero; its updates are **exempt** from the segment ordering rules ("atomic updates")
The dangers of over-posting

Scenario:
- Image p executes
  \[
  \text{event post ( ev[q] )}
  \]
- Image q executes
  \[
  \text{event wait ( ev )}
  \]
- Image r executes
  \[
  \text{event post ( ev[q] )}
  \]

Question:
- what synchronization effect results?

Answer: 3 possible outcomes
- which one happens is **indeterminate**!

⚠️ Avoid over-posting from multiple images!

Case 1: \(p_1\) ordered before \(q_2\)

Case 2: \(r_1\) ordered before \(q_2\)

Case 3: ordering as given on next slide
Multiple posting done correctly

Why multiple posting?
- Example: halo update

\[ p = q - 1 \quad q \quad r = q + 1 \]

Correct execution:
- Image \( p \) executes
  \[ fm(:,1)[q] = \ldots \quad \text{event post } ( ev[q] ) \]
- Image \( r \) executes
  \[ fm(:,n)[q] = \ldots \quad \text{event post } ( ev[q] ) \]

Image \( q \) executes

\[ \text{event wait } ( ev, \text{UNTIL_COUNT} = 2 ) \]
\[ \ldots = fm(:, :) \]

\( p_1 \) and \( r_1 \) ordered before \( q_2 \)

This case is enforced by using an UNTIL_COUNT
Atomic operations (1)

**Limited exception:**
- permit operations on coarrays from different images without synchronization
- for **scalars** of some **intrinsic** datatypes,

```
integer(atomic_int_kind)
logical(atomic_logical_kind)
```

- and via invocations of atomic subroutines only

**Fortran 2008:**
- `atomic_define(atom, value)`
  ```
  atom[q] := value
  ```
- `atomic_ref(value, atom)`
  ```
  value := atom[q]
  ```

**Added by TS18508:**
- `atomic_add(atom, value)`
  ```
  atom[q] := atom[q] + value  \text{ (integer)}
  ```
- `atomic_<and|or|xor>(...)`
  ```
  atom[q] := atom[q] <op> value  \text{ (logical)}
  ```
- `atomic_fetch_<op>(..., old)`
  ```
  incoming \text{ atom[q] assigned to OLD in addition to operation}
  ```
- `atomic_cas(atom, old, & compare, new)`
  ```
  compare and swap:
  old = atom[q]
  if (atom[q] == compare) atom[q] = new
  ```
Atomic operations (2)

Use for specifically tailored synchronization:

```fortran
integer(atomic_int_kind) :: x[*] = 0, z
integer :: q
q = ... ! same value on each image
sync memory
call atomic_add(x[q], 1)
if (this_image() == q) then
  wait: do
    call atomic_ref(z, x)
    if (z == num_images()) exit wait
  end do wait
sync memory
end if
```

Atomic operations do not imply segment ordering

- SYNC MEMORY statements are needed to assure $q_3$ is ordered against 1$^{st}$ segment of all images
Collective intrinsic subroutines (1)

- All collectives:
  - in-place → need to copy argument if original value is still needed
  - data arguments need not be coarrays; can be scalars or arrays
  - no segment ordering is implied by execution of a collective
  - must be invoked by all images (of current team)

- Data redistribution: CO_BROADCAST

```
type(matrix) :: xm
 :
call co_broadcast(A=xm, SOURCE_IMAGE=2)
```

©2015 LRZ

Additional Parallel Features in Fortran
**Collective intrinsic subroutines (2)**

### Reductions
- `co_max`, `co_min`, `co_sum`

![Execution sequence diagram](image)

```fortran
real :: a(2)
:call co_sum(a, RESULT_IMAGE=2)
```

- Without optional `RESULT_IMAGE`: result is assigned on **all** images
- Result for `CO_SUM` need not be exactly the same on all images

### General reduction facility
- User-defined binary operation (associative, commutative)

```fortran
interface
  pure function plus(x, y) result(r)
  import :: matrix
  type(matrix), intent(in) :: x, y
  type(matrix) :: r
end function
end interface
```

- Assignment to result: as if intrinsic (finalizers are executed for derived types if they exist)

```fortran
type(matrix) :: xm
:call co_reduce( A=xm, &
OPERATOR=plus, &
RESULT_IMAGE=2 )
```
Weaknesses of flat coarray model

Development of parallel library code
- coarrays are symmetric → memory management not flexible enough
- avoid deadlocks → obliged to do library call from all images
- collectives must be executed from all images

MPMD scenario: coupling of domain-specific simulation codes

Matching execution to hardware
- future systems likely are non-homogeneous (memory, core count)
- A unified hybrid programming model is desired → might use high internal bandwidth and fast synchronization of node architecture
Improving the scalability of the coarray programming model

- TS 18508 defines the concept of a team of images

- This provides additional syntax and semantics to
  - subdivide set of images into subsets that can independently execute, allocate/deallocate coarrays, communicate, and synchronize;
  - repeated (i.e., recursive and/or nested) subsetting is also permitted.

- Two essential mechanisms:
  - define the subsets
  - change the execution context to a particular subset

- Breaking composable parallelism where necessary
  - cross-team communication is also supported – as usual, with clear visual indication to the programmer
Setting up a team decomposition

**FORM TEAM statement**
- must be executed on all images of the current team
- synchronizes all images of that team

```fortran
form team ( id, team [, NEW_IMAGE=...] )
```

- integer supplies "color"
- resulting team of opaque type `team_type`

Option for programmer-defined image indexing inside new teams.
Example code

```
program coupled_systems
  use, intrinsic :: iso_fortran_env
  implicit none
  integer, parameter :: fluid = 1, structure = 2
  integer :: nf, id
  type(team_type) :: coupling_teams
  : nf = …
  if ( this_image() <= nf ) then
    id = fluid
  else
    id = structure
  end if

  form team ( id, coupling_teams )
  :
end program
```

- FORM TEAM does not by itself split execution
  - after the statement, regular execution continues on all images
Switching the execution context: the CHANGE TEAM block construct

Properties:
- at beginning, changes current team to become the one the executing image belongs to
- sets up an ancestor relationship between previous and new team
- at end of block, reverts to execution as ancestor team
- team-wide synchronization of images of each team at beginning and end of each block
- programmer is responsible for setting up appropriate control flow inside the block

Image indexing (including coindexing!) refers to current team
- order is processor dependent, unless the NEW_INDEX argument is specified in FORM TEAM

1 2 3 4 5

1 2 3

1 2

this_image() in initial team

form team

change team

end team

this_image() in team structure

return to original numbering

fluid and structure are sibling teams

©2015 LRZ Additional Parallel Features in Fortran
Adding a CHANGE TEAM block to the example

```
change team (coupling_teams)
block
  real, allocatable :: fl(:,:,,:), dfl(:,:,,:)[:]
  real, allocatable :: st(:,:,,:), dst(:,:,,:)[:]
  do
    select case( team_number() )
      case (fluid)
        if (...) allocate( fl(...), dfl(...)[*] )
        call process_fluid(fl, dfl, ...)
      case (structure)
        if (...) allocate( st(...), dst(...)[*] )
        call process_structure(st, dst, ...)
    end select
  end do
end block
end team
```

After FORM TEAM, the CHANGE TEAM block permits subsequent declarations.

New inquiry intrinsic: `team_number()`

Data only established in team "fluid":

- `fl`, `dfl` are allocated and established in this team.

Data only established in team "structure":

- `st`, `dst` are allocated and established in this team.

Fluid-structure interactions etc. (see later slide):

Deallocations are done here.
Cross-team data transfer

**Interaction between fluid and structure:**
- need to communicate across team boundaries

**Requires a coarray that is established in ancestor team**

**An addressing problem:**
- what is \( bd[4] \) in the initial team becomes \( bd[1] \) when the CHANGE TEAM starts executing \( \rightarrow \) team-local coindexing preserves composability 😊
- therefore, special syntax is needed for cross-team accesses
Extending the image selector:
Cross-team coarray references and definitions

```
real, allocatable :: bd(:,::)[:,]
```

**Example:**
- Statements below are executed on image 2 of the "fluid" team
- Sibling team syntax:
  ```
  ... = bd(:,::)[1,TEAM_NUMBER=structure]
  ```
- Ancestor team syntax:
  ```
  ... = bd(:,::)[4,TEAM=bd_team]
  ```

**Notes:**
- Both variants yield the same result in this situation
- Which to use depends on the image's knowledge of image indices and teams, and on the data assignment strategies.
- `bd_team` is an object of type `team_type`, to which `get_team()` assigns the value of the current team.
Dealing with the fluid-structure interaction (including necessary synchronization)

```fortran
real, allocatable :: bd(:, :, :)
type(team_type) :: bd_team
bd_team = get_team()
change team (coupling_teams)
  do
    ! deal with fluid and structure individually
    if (allocated(fl)) then ! executed by fluid
      do i=1, nimg
        bd(...) [img(i), TEAM_NUMBER=structure] = fl(...)
      end do
      sync team (bd_team)
      call process_interaction(bd, fl(...))
    end if
    ! analogous if block (with sync team) executed by structure
    if (...) exit
    sync team (bd_team)
  end do
end team
```
Teams and memory management

- Restrictions on coarray allocation and deallocation:
  - Coarrays cannot have „holes“ → in the current team, it is not permitted to deallocate a coarray that has been allocated in an ancestor team.
  - Avoid appearance of overlapping coarrays → all coarrays allocated while a `change team` block is executing are deallocated at the latest when the corresponding `end team` statement is reached (even if they have the `SAVE` attribute).
Fail-safe Execution (1): Behaviour after image failure

What happens in case an image fails?
- typical cause: hardware problem (DIMM, CPU, network link, …)
- Fortran 2008 (and all the rest of the HPC infrastructure): complete program terminates

TS18508: **optional** support for continuing execution
- images that are not directly impacted by partial failure might continue
- supported if the constant STAT_FAILED_IMAGE from ISO_FORTRAN_ENV is positive, unsupported if it is negative

---

**A failed image; it remains failed. Data become unavailable**

**sync all (STAT=sst)**

integer `sst` supplied with value STAT_FAILED_IMAGE on all images. Active images synchronize.
Fail-safe Execution (2):
Programmer‘s Responsibilities

- **Synchronization:** 
  - Without a STAT specifier on
    - image control statements (including ALLOCATE and DEALLOCATE),
    - collective, MOVE_ALLOC, or atomic subroutine invocations,
  - the program **terminates** if an image failure is determined to have occurred.

**With** a STAT specifier, active images **continue** execution,
- image control statements work as expected for these images,
- collective and atomic subroutine results are undefined

- **Data handling and Control flow:**
  - programmer must deal with **loss of data** on failed image, and
  - with side effects triggered by references and definitions of variables on failed images
  - FAILED_IMAGES intrinsic:
    - produces list of images known to have failed.

```fortran
integer, allocatable :: fl(:) :
  sync all (STAT=sst)
fl = FAILED_IMAGES()
```

Returns indices of at least the images that have failed up to the „sync all“
Referencing and defining objects

Reference to an object located on a failed image:
- Referencing image continues execution, but the object has a processor-dependent value.
- Example: statement executed on image 2
  
  \[ ... = a(:)[3, \text{STAT}=\text{sst}] \]

Definition of an object located on a failed image:
- Does not do anything, except setting a stat argument if present.
- Example: statement executed on image 2
  
  \[ a(:)[3, \text{STAT}=\text{sst}] = ... \]
Definition of an object performed by a failed image:

- Objects that would become defined by the failed image during execution of the segment in which failure occurred become **undefined**.

- Example: statement executed on image 3

\[ a(:,)[2] = \ldots \]

Difficulty of diagnosis: images that reference \( a[2] \) in a subsequent segment need to

- know the communication pattern, and hence
- identify image 3 as failed
FAIL IMAGE statement

- A statement that causes the images executing it to fail

- Enables testing of code that should execute in a fail-safe manner
  - might be executed conditioned on value returned by `random_number`
Thank you for your attention!

Any questions?