Requirements Paper

To: J3 J3/25-116

From:

Subject: Requirements for asynchronous tasks

Date: 2025-July-26

References: 22-169 (https://j3-fortran.org/doc/year/22/22-169.pdf), 23-174 (https://j3-fortran.org/doc/year/23/23-174.pdf)

Introduction

Meeting 230 (https://j3-fortran.org/doc/meeting/230) passed paper 23-174 (https://j3-fortran.org/doc/year/23/23-174.pdf), which provides use cases for asynchronous tasks. Three related Info papers were presented at Meeting 231 (https://j3-fortran.org/doc/meeting/231). Note, paper 23-232 (https://j3-fortran.org/doc/year/23/23-232.pdf) is an updated version of paper 23-174 (https://j3-fortran.org/doc/year/23/23-174.pdf).

- Paper 22-169 (https://j3-fortran.org/doc/year/22/22-169.pdf) "Fortran asynchronous tasks"
- Paper 23-232 (https://j3-fortran.org/doc/year/23/23-232.pdf): “Asynchronous Tasks in Fortran”
- Paper 23-245 (https://j3-fortran.org/doc/year/23/23-245.txt): “Asynchronous collective operations”
- Paper 23-246 (https://j3-fortran.org/doc/year/23/23-246.txt): “Concurrent tasks”

The current paper proposes requirements for the features described in the above papers.

REQUIREMENTS (IN PROGRESS)

R0. An asynchronous region is the dynamic scope executed by one thread of control. It will be outlined by the appropriate code block delineater. An asynchronous region may be executed independently from the surrounding execution context. Language defining these terms and concepts (e.g. thread, child, parent), will need to be added.

Example for illustrative purposes:

```
...
a = 10
async_region
  !-- The following statements are executed by a single
  !   control of thread
  b = a
  b = b * c
end async_region
d = a
...
```

R1. It should be valid for an implementation to wait for any asynchronous region to complete before continuing execution of its surrounding thread of control. We do not want programs to be able to deadlock if an implementation does not provide sufficient concurrency.

R2. It must be possible to express dependencies between multiple asynchronous regions.

R2.5 The dependencies should be structured such that each asynchronous region can have many upstream dependencies and a single shared identifier for its downstream dependencies.

In this example below, the `async_region` `A1` will not start until all `async_region` currently registered on `D1, D2, D3, D4` completes. Other `async_region` created after `A1` that has `D1` as a dependency will not start until `A1` completes.

```
! `in` and `inout` mutually exclusive (example 2a the survey paper)
A1 : async_region(dependinout=D1, dependin=[D2, D3, D4])
  ...
end async_region
```

R3. A construct should be provided to enforce synchronization of one or more dependency chains.

```
A1 : async_region
  ...
end async_region
...
A2 : async_region
  ...
end async_region
...
wait_on_async !-- current thread of control will wait all children
```

```
! `in` and `inout` mutually exclusive (example 2a in survey)
A1 : async_region(dependinout=D1, dependin=[D2, D3, D4])
  ...
end async_region
...
A2 : async_region(dependinout=D2)
  ...
end async_region
...
wait_on_async(D1) !-- only waits for async_region registered to D1
```

R4. Wording will be created such that race conditions can be avoided.

R5. Asynchronous regions shall not contain image control statements or calls to collective subroutines (at least in the initial version, we might add this later)

> Rationale: A given image initiates collective subroutines in a specific total order. Due to practical concerns, a strong requirement for correctness is that all images (at least within a particular team) initiate corresponding collective subroutines over that team in the *same* total order. This requirement is shared by collectives in other distributed models (e.g. MPI). It's unclear how best to ensure this property if collective subroutines can be initiated from two or more asynchronous regionss that are not totally ordered in the same way by all images in the team. A similar concern arises for image control statements that include collective image synchronization (SYNC ALL/SYNC TEAM/FORM TEAM/CHANGE TEAM/END TEAM and coarray ALLOCATE/DEALLOCATE), all of which must also be initiated in the same collective total order across the images in a particular team. In addition, Fortran's hierarchically scoped team model does not compose well with multiple independent threads of execution, because the team stack is effectively an image-wide global state property (e.g. what would it mean to CHANGE TEAM within two concurrent asynchronous regionss?).

R6. Dependency chains must be able to span different lexical scopes. E.g. one should be able to pass a dependency as an argument to a procedure.

R7. A mechanism to describe locality of data. In particular, capturing a copy of a variable from the enclosing scope is necessary for avoiding race condition. Explicitly marking data that is shared but read-only may also be helpful.

R8. Asynchronous tasks are not allowed to be "in-flight" when crossing segment boundaries.

> Rationale: See Questions section below.

NON-REQUIREMENTS

NR1. We do not provide a fine-grained mechanism to guard against access to shared data across asynchronous regions. E.g. no in-image ATOMIC or CRITICAL. A separate paper would be needed to provide this.

JUSTIFICATION

Tasking is a well-researched and widely implemented feature of asynchronous programming. Due to the increasing degree of parallelism available on modern machines, tasking is frequently required to saturate available resources. Additionally, because an increasing number of machines employ accelerators that
require offloading, tasking is needed to express the potential for overlapping of tasks to hide offload latencies and use all parts of the system.

A survey document of current solutions (https://hackmd.io/s1404_e3Qty6_qZWXzHZyA) has been compiled of existing approaches to inform the development of this proposal.