Subject: A problem concerning asynchronous input/output
From: Van Snyder

1 Introduction

There is a problem with asynchronous output as presently defined: If it is initiated for a non-saved local variable of a procedure, and the procedure returns, the variable might become undefined before the data transfer completes. There are three possibilities: (1) The ASYNCHRONOUS attribute implies or requires the SAVE attribute, which makes it unuseful in recursive procedures, (2) there is some machinery under the covers that prevents the undefinition in some other way, which makes implementation difficult, or (3) the program is required to WAIT before returning, which would be more simply and more safely enforced by a structured form of parallelism.

2 Proposal

Delete asynchronous input/output and the WAIT statement. Define the ASYNCHRONOUS attribute in terms of parallel execution. Provide a simple construct for multi-thread parallelism. The advent of cheap and ubiquitous multiprocessor computers makes support for parallel execution desirable. Since it is possible to provide the functionality of asynchronous input/output with multi-thread parallelism, but not vice-versa, multi-thread parallelism is more valuable than asynchronous input/output. The construct described below provides a natural linguistic mechanism for multi-thread parallel execution that is simpler than the present mechanism for asynchronous input/output, and avoids the danger described in section 1.

3 Replacement – structured parallelism

Define a new execution construct:

parallel-construct	**is**	[*parallel-construct-name* :] PARALLEL
		fork
		[*fork*] ...
		END PARALLEL [*parallel-construct-name*]
fork	**is**	FORK [*parallel-construct-name*]
		execution-construct
		[*execution-construct*] ...

Upon executing a PARALLEL statement the processor may divide the sequence of execution into a number of sequences not exceeding the number of FORK blocks. Each of these sequences begins execution at the start of a different FORK block; after finishing execution of one FORK block a sequence may continue into a different one, or may continue by executing the END PARALLEL statement. In any case, each FORK block is executed exactly once. After all of the FORK blocks are executed, all sequences of execution that were created by execution of the PARALLEL statement are condensed into a single sequence, and execution proceeds at the first statement after the END PARALLEL statement. Notice that this definition permits the processor to ignore the PARALLEL statement, all FORK statements, and the END PARALLEL statement.

A GO TO statement or an arithmetic IF statement within a parallel construct shall not have a branch target without that parallel construct or in a different fork block, and vice-versa. A RETURN statement shall not appear within a parallel construct. This requirement prevents the danger described in section 1, and avoids having to answer the question “So, how long does the activation record exist?”.

A variable shall have the ASYNCHRONOUS attribute if it or a subobject of it or a variable associated with it or a subobject of a variable associated with it may be accessed within more than one FORK block. The ASYNCHRONOUS attribute is not automatically deduced so as not to impose it when it is not necessary. Consider the following example of double-buffered input overlapped with processing. Even though references to the BUF variable appear in both FORK blocks, it should not have the ASYNCHRONOUS attribute because no element of BUF can be accessed simultaneously in two FORK blocks.

```
  type(myInput) :: BUF(blockSize,0:1)
  integer :: STAT, WHICH
...
  which = 0
  read ( inunit, iostat=stat ) buf(:,which)
  do while ( stat == 0 )
    parallel
    fork
      ! Process buf(:,which)
    fork
      ! Read into the side of BUF not being processed
      read ( inunit, iostat = stat ) buf(:,1-which)
    end parallel
    which = 1 - which
  end do
```

The number of forks is statically specified. If a dynamically determined number of forks is necessary, use FORALL. The iterations in FORALL are independent but qualitatively identical; in a PARALLEL construct the forks can be qualititively different.

4 Alternative replacement – unstructured parallelism

The proposal in section 3 is safer than the present asynchronous input/output mechanism, but has slightly less flexibility. The proposal here has the same danger and same flexibility as the present asynchronous input/output mechanism, but provides for multi-thread execution as well.

Ideally, define a new intrinsic type SEMAPHORE. Objects of type SEMAPHORE may be specified to have the DIMENSION, INTENT, OPTIONAL, PRIVATE, PUBLIC and SAVE attributes. Maybe ALLOCATABLE and POINTER are OK, too. They shall not appear in EQUIVALENCE or COMMON, or be components of sequence types. They implicitly have the ASYNCHRONOUS attribute. Neither assignment nor any operations are defined for objects of type SEMAPHORE. There are no constants of type SEMAPHORE. A semaphore variable indicates whether a parallel construct is active, or has completed. A processor may put whatever information is necessary in a variable of type SEMAPHORE. The present asynchronous input/output facility would benefit from using SEMAPHORE variables instead of INTEGER variables. Using integer semaphore variables, however, avoids the possibility of a conflict with a user-defined derived type.

Add the following execution construct:

```
parallel-construct        is  [ parallel-construct-name : ] PARALLEL ■
                              ■ ( semaphore-variable )
                                  execution-construct
                                  [ execution-construct ] ...
                              END PARALLEL [ parallel-construct-name ]
```

Upon executing a PARALLEL statement, the processor is permitted to divide the sequence of execution into two parallel sequences of execution. One of them shall initiate execution of the body of the PARALLEL construct, and the other shall continue execution at the first statement after the corresponding END PARALLEL statement. During execution of the parallel construct, the semaphore variable shall indicate that the parallel construct is active, and otherwise it shall not. If parallel sequences of execution are created, the sequence of execution that proceeds through the parallel construct shall cease to exist upon executing the corresponding END PARALLEL statement. A GO TO statement or an arithmetic IF statement within a parallel construct shall not have a branch target without that parallel construct, and vice-versa. A RETURN statement shall not appear within a parallel construct.

The processor may instead choose to ignore the PARALLEL statement and its corresponding END PARALLEL statement, except that while executing the body of the construct, the semaphore variable shall indicate that the construct is active and otherwise it shall not.

Define a WAIT statement that references a list of semaphore variables. Upon executing a WAIT statement, the sequence of execution in which it is encountered is suspended until the sequences of execution indicated by all semaphores named in the WAIT statement have completed. Of course, if a WAIT statement is executed in a sequence of execution associated with one of the semaphore variables it names, the program locks up – and there are other ways to lock up. A PARALLEL statement is considered to be preceeded by a WAIT statement that specifies the same semaphore.

It is not necessary, but it may be useful to define an intrinsic function, say ACTIVE (or COMPLETE), that takes a semaphore variable as an argument. The result type is LOGICAL. It is true (or false) if the argument indicates a parallel construct has not completed execution, and false (or true) otherwise.

5 Comparison

Unlike the unstructured parallelism proposal described in section 4, variables of type SEMAPHORE, a WAIT statement, and an ACTIVE intrinsic function are not needed by the structured parallelism proposal described in section 3. This simplicity comes at the price that dynamic parallelism is required to nest with control constructs in the same way that they are statically nested. Unstructured parallelism has the price that it is more difficult to define the temporal span of existence of an activation record (in case a RETURN statement is executed in one sequence of execution created in a procedure without waiting for the other sequences to complete. It also has the same danger as described in section 1 for asynchronous input/output.