Asynchronous Tasks in Fortran

Introduction

The principle of asynchrony is fundamental to many aspects of computing, and appears in many forms, many of which already overlap with the use of Fortran. Fortran supports asynchronous execution already via coarrays, since two images execute independent of one another unless explicitly synchronized. However, some use cases for asynchrony are a poor fit for coarrays, either due to the need for data to be distributed across images and explicitly communicated between them, or because the number of images is fixed and - except for termination - does not change throughout the execution of a program.

It is also possible to achieve asynchrony using DO CONCURRENT, but this assumes that an implementation generates the appropriate type of paralleism required for asynchronous execution. In any case, DO CONCURRENT imposes a number of restrictions on the code that can be executed within that scope, particularly that impure procedures are not permitted. This excludes the possibility of long-running and/or complex tasks that cannot be contorted to fit within the scope of a single DO CONCURRENT region.

Related Efforts

Ada, C and C++ are ISO languages with support for asynchronous execution in one form another, with features introduced in 1995, 2011 and 2011, respectively. We will briefly highlight the asynchronous parts of these languages below.

Within the domain of high-performance computing (HPC), the standardized directives known as OpenMP and OpenACC support asynchronous execution, which is used in Fortran applications today. The use cases for directive-based asynchronous execution in Fortran applications provide use cases for the features we are proposing.

Ada Tasks

Ada has tasks a first-class feature in the language. Ada tasks are similar to OS threads, in that they can act asynchronously relative to their parent, and can act on data from the parent scope, for example with a discriminate (argument) that is a pointer to data allocated and initialized elsewhere. In contrast to coarrays, Ada tasks can be created and destroyed within a program, tasks can create and wait on additional tasks, and tasks can access global variables.

https://dwheeler.com/lovelace/s13s1.htm

C Threads

The ISO C11 language introduced both threads and atomic operations, which are primitives that allow the implementation of asynchronous execution similar to Ada tasks. For example, the user can spawn a thread, provide it with pointers to data that it can manipulate. Because C provides nothing like Ada's protected objects, users must take care and use atomic operations when

multiple threads access data concurrently. Users can also built their own protected accesses using mutexes.

C++ Threads

C++11 added equivalent features to C11 - threads, atomics, mutexes - as well as related features like async, promise and future. In later versions of the standard, C++ has introduced parallel algorithms (e.g. std::for_each, which has the ability to behave like Fortran 2008 DO CONCURRENT).

OpenMP Tasks

OpenMP 3.1 introduced tasks on top of the existing threading model. The 3.1 version of tasks did not support dependencies, other than parent tasks waiting on child tasks. In OpenMP 4.0, task dependencies were introduced, which permitted the synchronization of tasks using memory locations as identifiers. The implementation of task dependencies is considerably more complicated, and is a cautionary tale for Fortran.

OpenACC Async

OpenACC has `async`, which provides a queue-like mechanism for allowing asynchronous execution of certain features. This is a natural match when OpenACC targets another device, such as a GPU, which naturally executes asynchronously relative to the host controlling it. OpenACC `async` does not allow interactions between different asynchonrous regions, and furthermore offers multiple streams of asynchrony, enumerated as integers. Because asynchronous regions cannot interact, it is legal for an implementation to ignore asynchrony and wait for completion of such regions before proceeding. This allows naive implementations or ones that only target CPUs, where asynchronous parallel execution might not be productive.

Motivating Examples

Basic Overlap

Modern computers have many different execution units, some which are fully general, like CPU cores, and others which are specialized. For example, Intel's latest server CPU (codename: Sapphire Rapids) has at least three different on-chip accelerators, including the Data Streaming Accelerator (DSA), which is capable of executing data parallel operations like fill, copy and compare faster than the CPU cores, and also asynchronously relative to them. While current Fortran compilers could utilize the DSA for `DO CONCURRENT`, the current semantics provide no mechanism for the programmer to encourage such a region to be executed on the DSA while returning control immediately to allow the CPU to execute the following code.
https://www.intel.com/content/www/us/en/developer/articles/technical/scalable-io-between-accelerators-host-processors.html

Recent NVIDIA GPUs also contain asynchronous copy engines within the processor, in addition to the asynchronous DMAs for copying outside of the GPU.

https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/ https://docs.nvidia.com/cuda/cuda-driver-api/api-sync-behavior.html https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Similar compute engines to DSA exist for the capability required by Fortran's `MATMUL`, for example. Processors from Apple, Intel and NVIDIA have dedicated matrix units, which are separate silicon from the general purpose logic and may be capable of executing independently, i.e. asynchronously.
https://nod.ai/comparing-apple-m1-with-amx2-m1-with-neon/ https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html

Even without special processing engines, all modern CPUs, including ones found in cheap cell phones, contain multiple cores. While Fortran permits programs to utilize parallelism across cores using coarrays and `DO CONCURRENT`, the former does not support shared-memory and thus requires unnecessary copies of data between images when cores can access the same data coherently, and the latter is limited to highly structured data parallelism and prohibits impure procedures. There are uncountable examples of less structured models for multicore parallelism, including OS threads, that can be expressed in terms of asynchronous tasks. For example, if one has N independent operations and N cores, annotating the operations as asynchronous allows them to run in parallel across cores. While it is possible for compilers to do such transformations, they rarely do, because proving the profitability is difficult and being too aggressive here risks offending the user. This is not unlike the possibility for autoparallelization of `DO` loops, and the reason why `DO CONCURRENT` exists to allow programmers to describe the desired behavior of a program to the compiler.
Finally, and of great relevance to the Fortran community, are the use of accelerators or coprocessors, which are attached to CPUs via a high-bandwidth interconnect, and which support a large portion of Fortran programming language. Recently, both NVIDIA and Intel have begun to support `DO CONCURRENT` on GPUs, but the current semantics do not permit the programmer to describe the asynchronous nature of GPU computing, which is default in native APIs like OpenCL, CUDA, HIP and SYCL (or Intel's Data Parallel C++). Fortran programs that use OpenMP or OpenACC can achieve asynchronous behavior in `DO CONCURRENT`, but not without stepping outside of the standard.

Asynchronous Communication

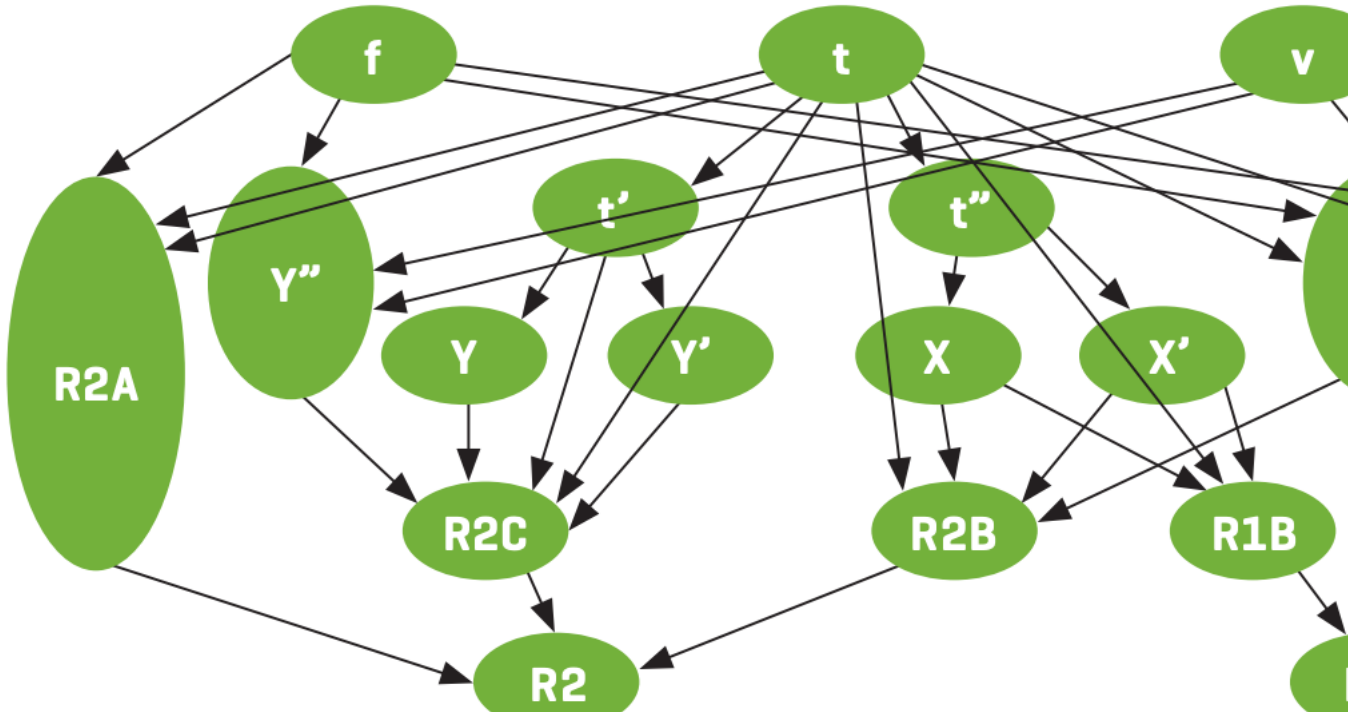
Asynchronous communication is regarded as an important tool for scaling distributed applications. Library-based communication such as MPI and OpenSHMEM includes non-blocking operations, which are allowed to behave asynchronously relative to the calling context. While coarray operations may be asynchronous between images, i.e. "one-sided", collective operations are synchronous - there is no mechanism to express a pipeline of collectives, or overlap of collectives with computation.

Quantum chemical many-body theory

This is a specific example of the basic overlap described above, which has been proven to be enormously beneficial to performance in a real-world scientific application.

Quantum chemical many-body theory requires the computation of a large number of terms (dozens to hundreds), many of which are independent of one another. In a Fortran implementation where `DO CONCURRENT` and intrinsics like `MATMUL` are capable of being executed asynchronously relative to the rest of the program, the availability of an asynchronous or task-like mechanism allows a more efficient implementation of this than is otherwise possible.
Shown below is the computational graph associated with one version of the populare CCSD method. It has been implemented using GPU parallelism, and asynchronous execution of data parallel kernels and matrix multiplication operations enables better utilization of the GPU and overlap of computation and data movement, leading to a 2.5x speedup relative to the synchronous GPU implementation.

Figure 3. The directed acyclic graph (DAG) representing data depe one formulation of the CCSD method. The vertex labels are not im



https://pubs.acs.org/doi/abs/10.1021/ct100584w https://dl.acm.org/doi/10.1145/2425676.2425687

Opportunities for Asynchrony in Fortran

There are at least four different types of opportunities for asynchrony in Fortran:

1. `DO CONCURRENT` execution on coprocessors, specialized or not.
2. Data parallel intrinsics such as `MATMUL`.
3. Coarray communications, especially collectives.
4. General code not included above.

We will describe the associated use cases in more detail below.

DO CONCURRENT

If DO CONCURRENT is executed a separate processor, as is available with the Intel and NVIDIA compilers today, then the implementation looks like this:
Before:

```
DO CONCURRENT (i=1:N) SHARED(X,Y,Z)
  Z(i) = X(i) + Y(i)
END DO
CALL toimia()
```

After:

```
COPY X(1:N) and Y(1:N) to GPU
ALLOCATE Z(1:N) on the GPU
INVOKE Z(:) = X(:) + Y(:)
WAIT on the GPU
COPY Z(1:N) to the CPU
CALL toimia()
```

In contrast, by adding the asynchronous feature of OpenACC, we will achieve the following:

Before:

```
!$ACC PARALLEL LOOP ASYNC
DO CONCURRENT (i=1:N) SHARED(X,Y,Z)
  Z(i) = X(i) + Y(i)
END DO
CALL toimia()
!$ACC WAIT
```

After:

```
COPY X(1:N) and Y(1:N) to GPU
ALLOCATE Z(1:N) on the GPU
INVOKE Z(:) = X(:) + Y(:)
CALL toimia()
WAIT on the GPU
COPY Z(1:N) to the CPU
```

Whichever of the time spent in `toimia()` or WAIT+COPY is lesser will be eliminated from the program execution time, because they will happen concurrently. The pointless time spent in WAIT will be replaced with something useful.

Data parallel operations

NVIDIA Fortran supports execution of numerous data parallel operations on GPUs, including MATMUL, TRANSPOSE, and array assignments. The underlying library implementation of these routines in CUDA is naturally asynchronous, but the Fortran compiler has no standard syntax to expose that to users. Today, it must rely on OpenACC directives, i.e. the async statement described above, to allow the programmer to express their intent and to coordinate one or more streams of execution that are permitted to execute asynchronously relative to one another.
It is not possible to solve this problem with DO CONCURRENT because the compiler may map these statements to the GPU, and Fortran provides no way to map DO CONCURRENT to different devices, or nest multiple DO CONCURRENT regions, where the outer one uses CPU threads and the inner one uses a GPU.

Asynchronous Communication

In the following program, there is no need to synchronize images between coarray collectives, or finish any of them before initiating the PRINT operation.

```
subroutine stuff(A,B,C,D)
implicit none
double, intent(inout) :: A, B, C
double, intent(in) :: D(:)
call co_sum(A)
call co_min(B)
call co_max(C)
print*,D
end subroutine stuff
```

In MPI-3, it is natural to express the independent nature of these operations using nonblocking collectives.

```
subroutine stuff(A,B,C,D)
use mpi_f08
implicit none
double, intent(inout) :: A, B, C
double, intent(in) :: D(:)
type(MPI_Request) :: R(3)
call MPI_Iallreduce(MPI_IN_PLACE, A, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD, R(1))
call MPI_Iallreduce(MPI_IN_PLACE, B, 1, MPI_DOUBLE, MPI_MIN,
MPI_COMM_WORLD, R(2))
call MPI_Iallreduce(MPI_IN_PLACE, C, 1, MPI_DOUBLE, MPI_MAX,
MPI_COMM_WORLD, R(3))
print*,D
call MPI_Waitall(3,R,MPI_STATUSES_IGNORE)
end subroutine stuff
```

An asynchronous syntax for coarray communications would allow Fortran compilers to exploit asynchronous communication libraries like MPI.

Other Asynchronous Operations

Any time two or more operations or regions of code contains no conflicting data references, they may be executed independently. In theory, compilers can recognize this automatically, but as nearly all forms of asynchronous execution require non-zero runtime overhead, compilers will not do so unless they can prove profitability. Proving both the lack of conflicting data references and profitability is extremely difficult. This is the reason why essentially all successful implementations of parallelism and asynchrony are explicit.

In the most explicit forms of asynchrony, the programmer creates a thread in their program and associates with it a procedure to execute, and conflicting data references, known as race conditions, must be dealt with explicitly using atomic operations or other synchronization primitives.

In Ada, the underlying mechanism of asynchrony is not exposed, and the programmer describes tasks, which will be implemented using threads or similar, by the Ada runtime library. Because Ada tasks are permitted to synchronous, e.g. using the rendezvous, they must be implemented in such a way that task synchronization cannot deadlock. This requires that tasks be mapped to

parallel execution agents or that the runtime library schedule tasks concurrently, e.g. using coroutines.

In contrast to Ada, OpenMP tasks are not required to use independent execution agents and therefore cannot synchronize with one another without the possibility of deadlock, in contrast to OpenMP threads. Similarly, OpenACC `async` does not require an independent execution agent and cannot synchronize with other asynchronous regions.
While Ada, C and C++ allow tasks/threads to synchronize, but OpenMP tasks and OpenACC async may not, they both permit essentially arbitrary code to execute within them, limited only by the semantics of concurrent data access. All of the above provide mechanics for describing the potential for concurrent data access, whether or not they permit such access to implement synchronization.

Syntax and Semantics for Fortran Asynchrony

The terms asynchronous, concurrent and parallel are heavily used in computing, not always in consistent ways. Because Fortran already defines ASYNCHRONOUS in the context of data, we will not reuse it to describe execution, at least as syntax. We will however, use the term to describe when execution units are independent of one another.

Inspired by Ada, the syntax `TASK` will be used to describe a region that is asynchronous relative to the parent task. However, like OpenMP tasks, Fortran tasks should not permit synchronization between tasks since this requires a more involved implementation. Like `DO CONCURRENT`, a `TASK` is a programmer hint that it is allowed, desired, and/or profitable to implement the contained code using an asynchronous mechanism, such as an OS thread or a dedicated execution unit that operates independently of the primary CPU computing logic.
A `TASK` extends `BLOCK`. It is expected that implementing `TASK` as `BLOCK` is legal, in the same way that implementing `DO CONCURRENT` as `DO` is legal, but that high-quality implementations will do more.
Because tasks - including the implicit task of the program - may execute independently of one another, the programmer must not make conflicting data accesses. Concurrent reads are not conflicting and always permitted. Writes that are concurrent with other reads or writes will have unspecified, implementation-defined behavior. Unlike `DO CONCURRENT`, which explicitly prohibits impure procedures, Fortran tasks may call any procedures. The burden is on the programmer to avoid conflicting data accesses, in the same way that coarray programs should not do so.
While tasks execute independently of their parent task, a sequence of tasks created by the same parent will execute in sequence relative to one another. In order to relax this behavior, tasks may be associated with different streams of execution. A stream of execution for tasks is labeled with an integer.

Streams of tasks are synchronized to the parent task via the `TASK WAIT [(stream-number[,stream-number]...)]` statement, where `stream-number` is a *scalar-integer-expression*. If no stream index is provided, the `TASK WAIT` statement synchronizes all streams.
The above syntax and semantics of streams is similar to OpenACC `async`, and thus is known to be implementable by multiple compilers. It is also similar to OpenMP constructs with `NOWAIT` or tasks without dependencies.
Ada tasks provide a number of important semantics to exclude problematic programs. For example, a parent task may not terminate while a child task has an outstanding data reference, since this would create incorrect and difficult to debug programs. Similarly, Fortran tasks

implicitly wait on child tasks unless the child task contains no references to data associated with the parent task.

("Concurrent and Real-Time Programming in Ada" by Alan Burns and Andy Wellings)

Fortran tasks need a way to describe the access properties of data. Locality specifiers are similar to what OpenMP and OpenACC use to describe the behavior or asynchronous/task regions and can be reused. A Fortran TASK region begins with LOCALITY statements such as the following:

```
TASK [(stream-number[,stream-number]...)]
  LOCALITY(SHARED) :: X(:) ! shared but conflicting accesses are
prohibited
  LOCALITY(LOCAL_INIT) :: IBEGIN, IEND, IX
  REAL :: A
  A = COS(IX)
  X(IBEGIN:IEND) = A
END TASK
```

The above task assigns to a range of the array X, as prescribed by the indices provided by the parent task.

The REDUCE locality specifier is not permitted in tasks. All data declared within a task cannot be accessed by the parent but may be accessed by child tasks, according to the child tasks locality statements.