We need a way to run things asynchronously within an address space.There are lots of use cases for this, one of which is running `DO CONCURRENT` regions on GPUs.

Summary

Starting in Fortran 2008, Fortran supports two forms of parallelism:

1. `DO CONCURRENT`, which supports loop-level data parallelism.
2. coarrays, which is form of PGAS.

This document will describe a third form of parallelism and argue that it should be supported by the Fortran language.

The third form of parallelism is shared-memory task parallelism, which supports a range of use cases not easily covered by 1 and 2.

Background Reading

The reader may wish to consult the following for additional context on this topic:

- *Patterns for Parallel Programming* by Timothy G. Mattson, Beverly Sanders and Berna Massingill
- *Task Parallelism By Example* from the Chapel Project (Slides)
- *OpenMP Tasking Explained* by Ruud van der Pas (Slides)
- *OpenMP Tasking* by Christian Terboven and Michael Klemm (Slides)
- *The Problem with Threads* by Edward A. Lee (Paper)

Motivating Example

Consider the following Fortran program:

```
module numerot
  contains
    pure real function yksi(X)
      implicit none
      real, intent(in) :: X(100)
      !real, intent(out) :: R
      yksi = norm2(X)
    end function yksi
    pure real function kaksi(X)
```

```
      implicit none
      real, intent(in) :: X(100)
      kaksi = 2*norm2(X)
    end function kaksi
    pure real function kolme(X)
      implicit none
      real, intent(in) :: X(100)
      kolme = 3*norm2(X)
    end function kolme
end module numerot

program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC

  A = 1
  B = 1
  C = 1

  RA = yksi(A)
  RB = kaksi(B)
  RC = kolme(C)

  print*,RA+RB+RC
end program main
```

Assuming that `yksi`, `kaksi`, `kolme` share no state, then all three functions can execute concurrently. How would we implement this in Fortran 2018?

One way is to use coarrays and assign each function to a different image:

```
program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: R

  A = 1
  B = 1
  C = 1

  if (num_images().ne.3) STOP

  if (this_image().eq.1) R = yksi(A)
  if (this_image().eq.2) R = kaksi(A)
  if (this_image().eq.3) R = kolme(A)

  SYNC ALL()

  call co_sum(R)
```

```
  if (this_image()) print*,R
end program main
```

While this works, this approach has many shortcomings. First, there is no way to share data directly between images - data must be explicitly copied using coarray operations. Second, images exist throughout the lifetime of the program (unless they fail) and thus the amount of parallelism is restricted to what is specified at runtime. Third, if there are many functions that can execute concurrently, many more than the number of images (which are likely to be processor cores or similar), then either the system will be oversubscribed or the user needs to implement scheduling by hand.

Dynamic load-balancing is nontrivial and should not be delegated to application programmers in most cases.

Another way to implement this program is to use DO CONCURRENT:

```
program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC
  integer :: k

  A = 1
  B = 1
  C = 1

  do concurrent (k=1:3)

    if (k.eq.1) RA = yksi(A)
    if (k.eq.2) RB = kaksi(B)
    if (k.eq.3) RC = kolme(C)

  end do

  print*,RA+RB+RC
end program main
```

This could work if the external functions are declared PURE, but DO CONCURRENT provides no means for dynamic load-balancing. The bigger problem is that Fortran implementations cannot agree on what form of parallelism DO CONCURRENT uses. Some implementations will use threads while others will use vector lanes. The latter is going to be useless for most purposes. Finally, the above is ugly and tedious - no one wants to write code like that to execute independent tasks.

The OpenMP/OpenACC Solution

There is a proven solution for Fortran task parallelism in OpenMP (4.0 or later) or OpenACC:

```
program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC

  A = 1
  B = 1
  C = 1

  !$omp parallel
  !$omp master

  !$omp task
  RA = yksi(A)
  !$omp end task

  !$omp task
  RB = kaksi(B)
  !$omp end task

  !$omp task
  RC = kolme(C)
  !$omp end task

  !$omp end master
  !$omp end parallel

  print*,RA+RB+RC
end program main

program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC

  A = 1
  B = 1
  C = 1

  !$acc async
  RA = yksi(A)
  !$acc end async

  !$acc async
  RB = kaksi(B)
  !$acc end async
```

```
  !$acc async
  RC = kolme(C)
  !$acc end async

  !$acc async wait

  print*,RA+RB+RC
end program main
```

These programs will execute regardless of the available hardware parallelism, including sequentially. OpenMP tasking is more powerful in some use cases than OpenACC, by allowing the user to create dependencies between tasks, which forces the runtime to do more work when scheduling.

This feature - tasks with dependencies - is not proposed for Fortran.

The Proposal for Fortran

Because OpenMP independent tasks is implemented in essentially all of the Fortran 2008 compilers, it is reasonable to assume that the design is portable. The goal here is to design a language feature for Fortran that is consistent with its existing semantics and syntax.

We consider the `BLOCK` construct to be an appropriate starting point, because it defines a scope, and scoping data is an essential part of defining task parallelism. Because we need more than just data scoping, we use the keyword `task_block` to tell the implementation that execution concurrency is both permitted and desirable.

```
program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC

  A = 1
  B = 1
  C = 1

  task_block
  RA = yksi(A)
  end task_block

  task_block
  RB = kaksi(B)
  end task_block

  task_block
```

```
  RC = kolme(C)
  end task_block

  task_sync all

  print*,RA+RB+RC
end program main
```

Non-trivial data issues

Obviously, very few programs can exploit concurrency where all data is strictly private. In `DO CONCURRENT`, locality specifiers are used to inform the implementation about whether data is shared, etc. (See this or this for details.)

Below we modify our program as if each function used a private scratch buffer. This is not the best way to allocate X, since X could be defined inside of the `task_block` scope or inside of the external function, but this is just an illustration of the syntax. We also add T, which could be a read-only lookup table, for example.

```
program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC
  real :: X(10)
  real :: T(1000)

  A = 1
  B = 1
  C = 1

  task_block local(X) shared(T)
  RA = yksi(A,X)
  end task_block

  task_block local(X) shared(T)
  RB = kaksi(B,X)
  end task_block

  task_block local(X) shared(T)
  RC = kolme(C,X)
  end task_block

  task_sync all

  print*,RA+RB+RC
end program main
```

Much like `DO CONCURRENT`, we should be able to write a fully explicit version using `default(none)`.

```
program main
  use numerot
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC
  real :: X(10)
  real :: T(1000)

  A = 1
  B = 1
  C = 1

  task_block local_init(A) shared(RA) local(X) shared(T)
  RA = yksi(A,X)
  end task_block

  task_block local_init(B) shared(RB) local(X) shared(T)
  RB = kaksi(B,X)
  end task_block

  task_block local_init(C) shared(RC)  shared(T)
  RC = kolme(C,X)
  end task_block

  task_sync all

  print*,RA+RB+RC
end program main
```

It might make sense to have a new locality specifier, `local_final` but since there might have been a reason why that was not added for `DO CONCURRENT`, we use the `shared` specifier to the result of this function.

Dependencies

Many applications where task parallelism will be used have dependencies between tasks.

For example, in our program, we can add a fourth function `nalja` that depends on

`yksi` and `kaksi`.

```
program main
  use iso_fortran_env, only : task_depend_kind
  use numerot
```

```
  implicit none
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC
  real :: X(10)
  real :: T(1000)
  type(task_depend_kind) :: DEP

  A = 1
  B = 1
  C = 1

  task_block depends_to(DEP)
  RA = yksi(A)
  end task_block

  task_block depends_to(DEP)
  RB = kaksi(B)
  end task_block

  task_block
  RC = kolme(C)
  end task_block

  task_block depend_from(DEP)
  RD = nalja(RA,RB)
  end task_block

  task_sync all

  print*,RC+RD
end program main
```

This syntax may not be ideal but it expresses the concept. In OpenMP, dependencies are expressed in the form of memory locations. Because this might be harder to implement in some scenarios, we propose an explicit opaque type that the implementation can use.

Known Shortcomings

Fortran lacks a memory model in the way that Java, C11 and C++11 do. We do not take a position on whether that is a good or bad thing, but instead attempt to make the fewest changes required to address hazards of concurrent data access by tasks. One obvious solution for tasks is to reuse the coarray atomic operations, although this may not be acceptable to the committee. However, requiring that tasks use atomic operations to access data that may be modified by another task is a straightforward solution to these hazards. Unfortunately, the overhead of coarray atomics may be

higher than acceptable for shared-memory uses, in which case a new syntax is required.

Acknowledgements

Thanks to the following people, who read this proposal or related material and may have provided feedback:

- Ondrej Certik
- Jeff Larkin