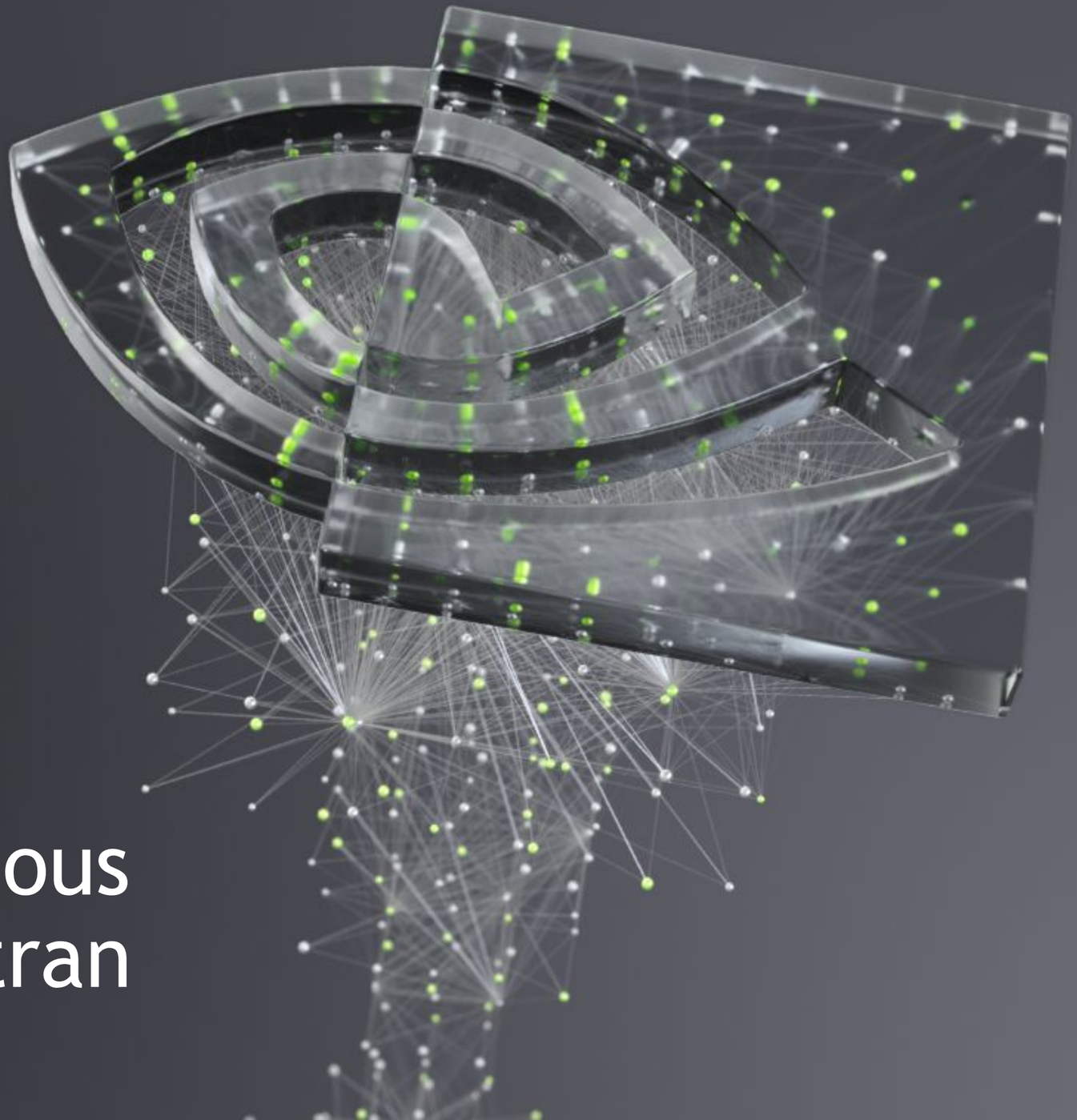


The Case for Asynchronous Task Parallelism in Fortran

Jeff Hammond and Jeff Larkin
NVIDIA HPC Group

Outline

- Overview of parallelism in Fortran
- Motivating use cases
- Prior art in OpenMP and OpenACC
- Code for a simple example
- Possible Fortran implementations

This talk is based on a blog post I wrote this summer:
https://github.com/jeffhammond/blog/blob/main/Fortrans_Missing_Parallelism.md

Parallelism in Fortran 2018

```
! fine-grain parallelism

! explicit
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! implicit
MATMUL
TRANSPOSE
RESHAPE
...
```

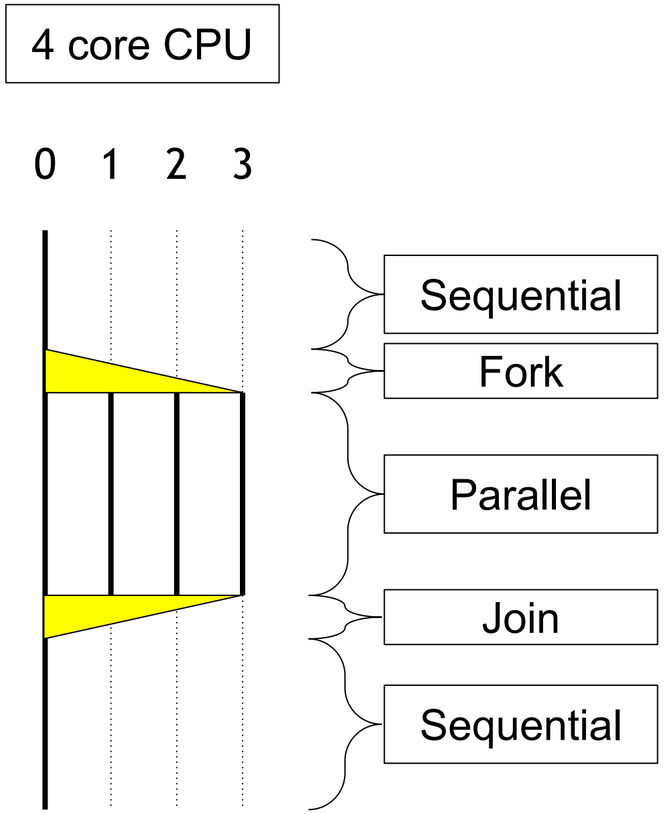
```
! coarse-grain parallelism

np = num_images()
n_local = n / np

! X, Y, Z are coarrays

do i=1,n_local
    Z(i) = X(i) + Y(i)
end do
sync all
```

Motivation for Asynchrony 1

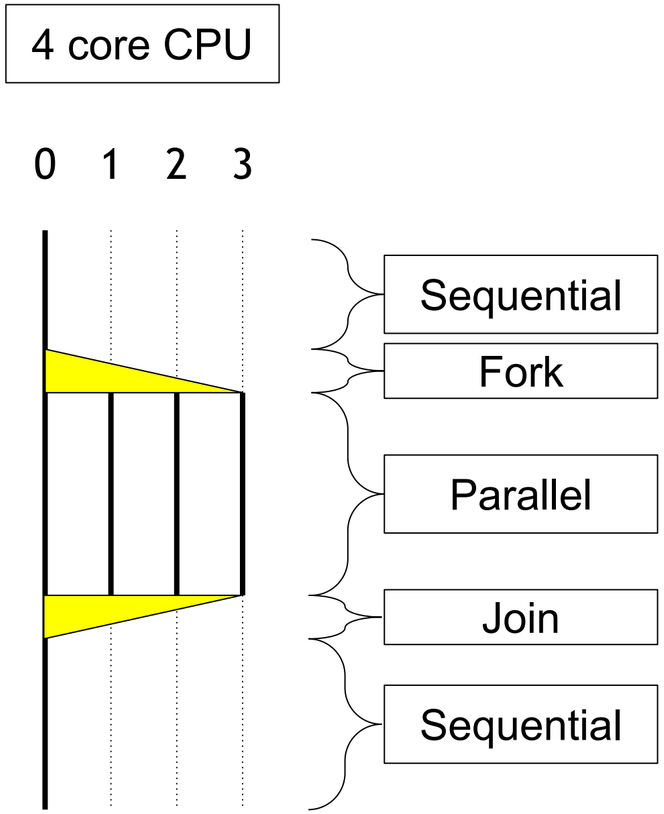



```
! sequential
call my_input(X,Y)

! parallel
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! sequential
call my_output(Z)
```

Motivation for Asynchrony 1

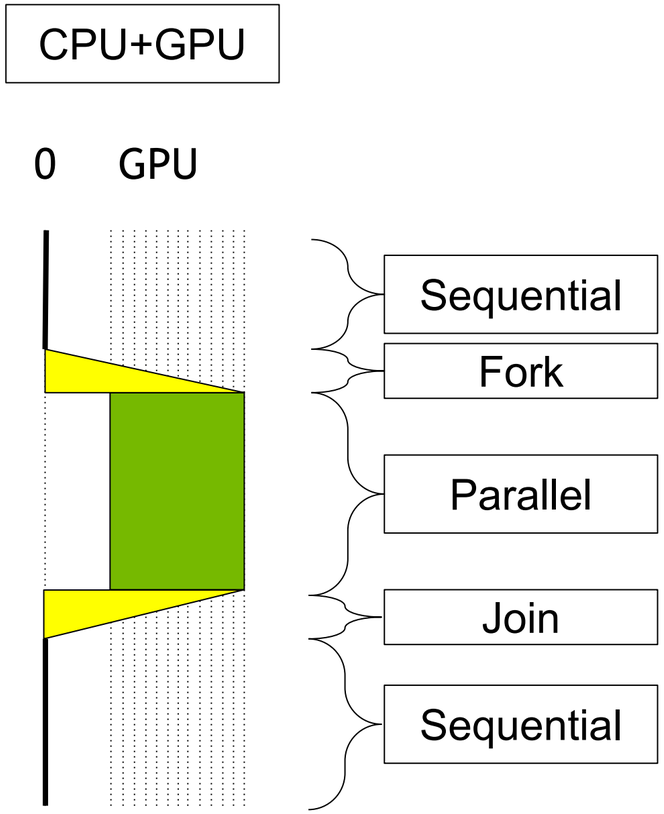



```
! sequential
call my_input(X,Y)

! parallel
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! sequential
call my_unrelated(A)
```

Motivation for Asynchrony 1

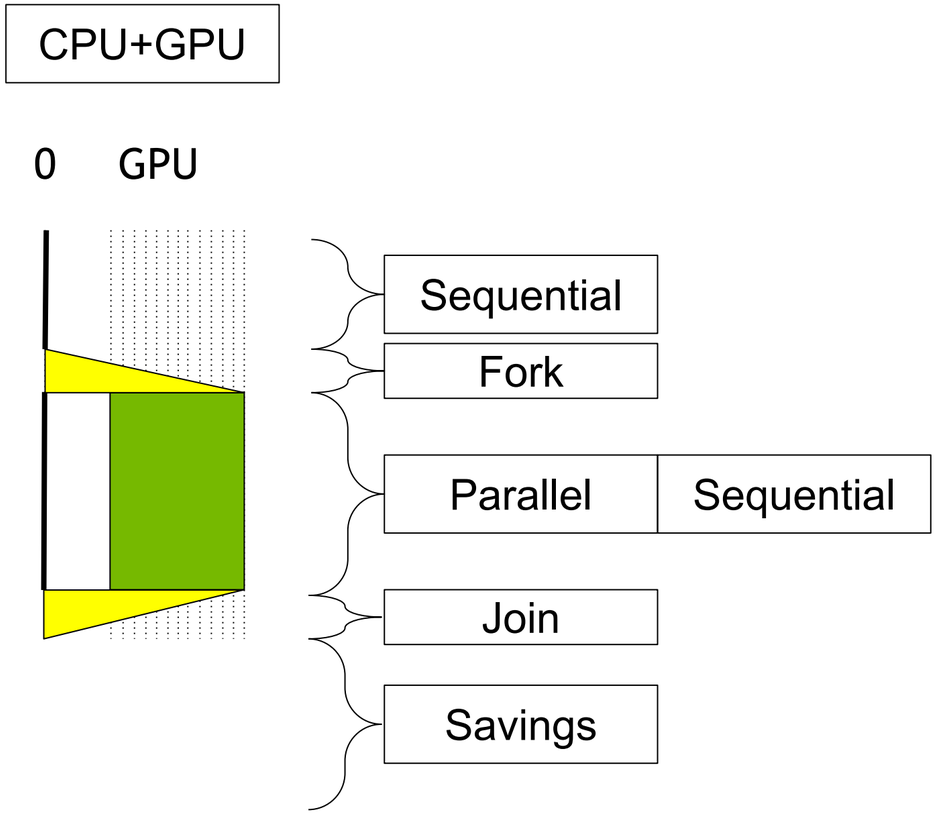



```
! sequential on CPU
call my_input(X,Y)

! parallel on GPU
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! sequential on CPU
call my_unrelated(A)
```

Motivation for Asynchrony 1

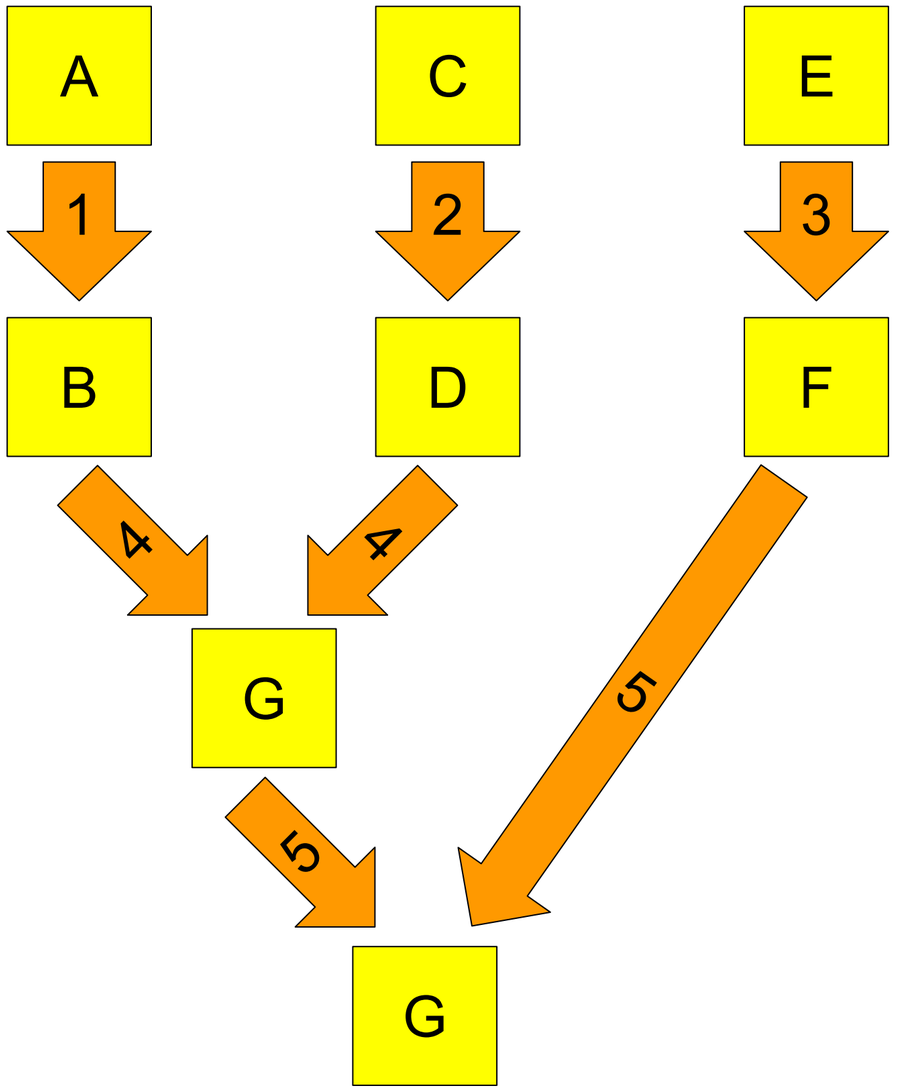



```
! sequential on CPU
call my_input(X,Y)

! parallel on GPU w/ async
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! sequential on CPU w/ async
call my_unrelated(A)
```

Motivation for Asynchrony 2 (synthetic)

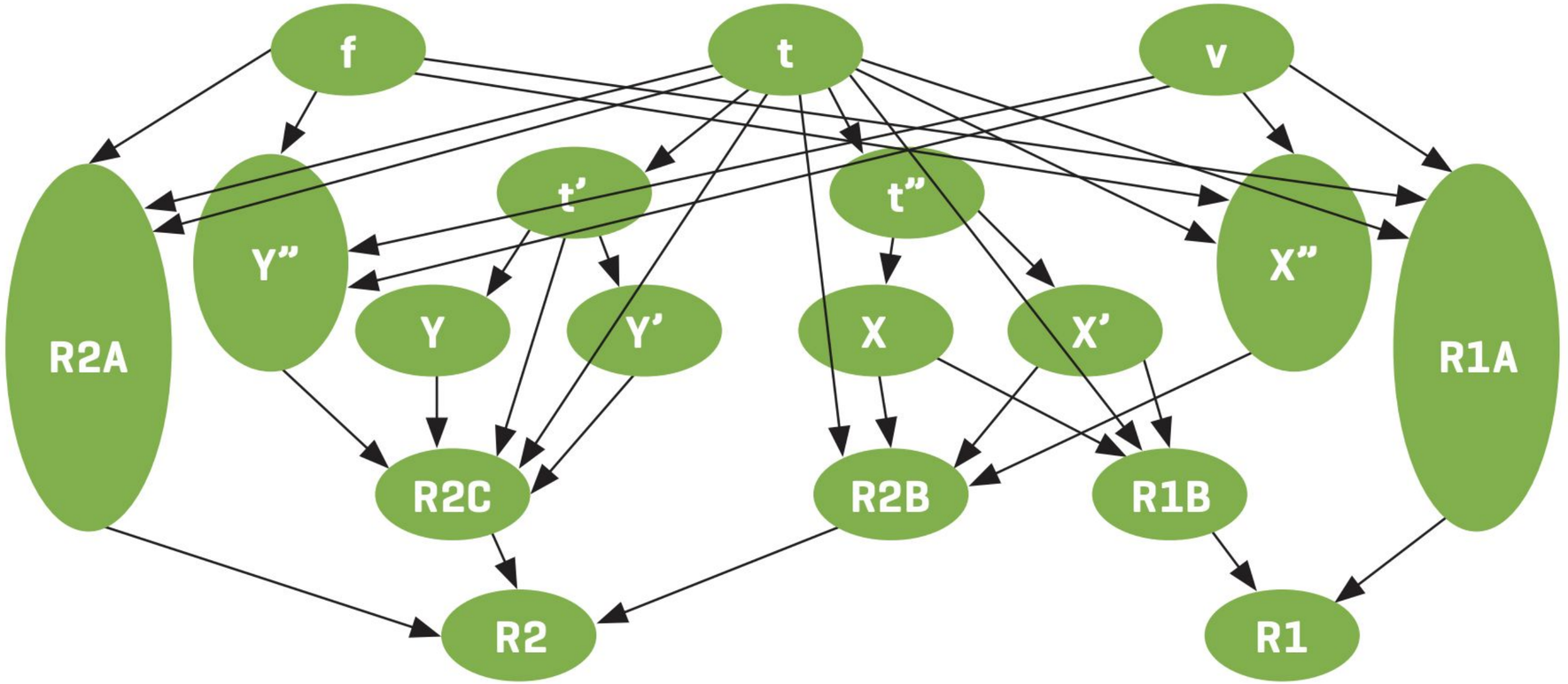



```
call sub1(IN=A,OUT=B)
call sub2(IN=C,OUT=D)
call sub3(IN=E,OUT=F)
call sub4(IN=B,IN=D,OUT=G)
call sub5(IN=F,IN=G,OUT=H)
! 5 steps require only 3 phases
```

Fortran compilers may be able to prove these procedures are independent but it is often impossible to prove that executing them in parallel is *profitable*.

Motivation for Asynchrony 2 (realistic)

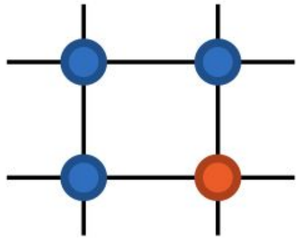
Figure 3. The directed acyclic graph (DAG) representing data dependencies within one formulation of the CCSD method. The vertex labels are not important.



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

See https://github.com/ParRes/Kernels p2p* for details...

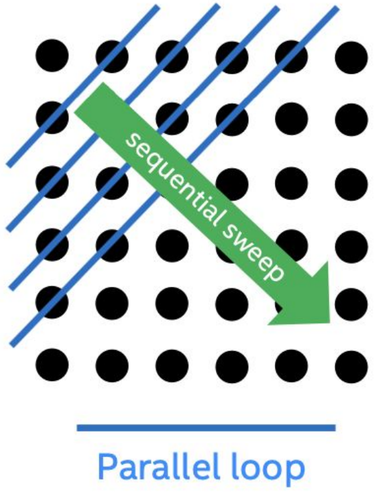
Motivation for Asynchrony 3

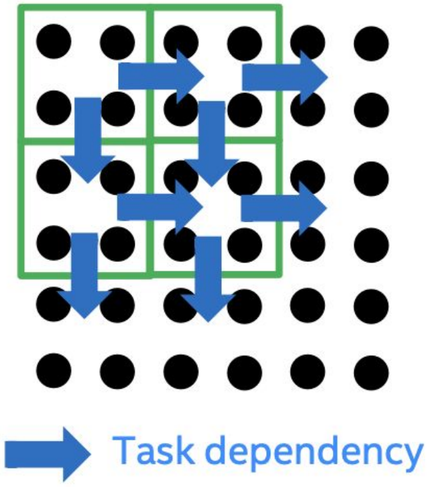


$$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$$

This pattern appears in a range of applications:

- Deterministic neutron transport
- Dynamic programming for sequence alignment
  e.g. Smith-Waterman/PairHMM (bioinformatics)
- Linear algebra (e.g.NAS LU benchmark)





Prior Art in OpenMP and OpenACC

Both of the popular directive-based models for parallel computing support asynchronous tasks in a range of operations.

OpenACC supports async and wait, with an implicit/default queue (stream) as well as explicit/numbered queues, and the ability to create dependency chains between operations, similar to CUDA streams.

OpenMP supports tasks with dependencies (and without). The syntax for dependencies is finer granularity - based on data references rather than queues - and the implementation may end up using a global queue as a result.

There are merits to both approaches, so the Fortran community will have to think about what form should be standardized.

These are examples of different things. Please don't try to compare them.

Prior Art in OpenMP and OpenACC

```
do i=1,n
  !$acc parallel loop async(i)
  do j=1,m
    ...
  enddo
enddo
do i=1,n
  !$acc parallel loop async(i)
  do j=1,m
    ...
  enddo
enddo
!$acc wait
```

```
!$omp parallel
!$omp master
do j=1,n
  do i=1,m
    !$omp task
    !$omp& depend(in:grid(i-1)) &
    !$omp& depend(out:grid(j))
    ...
    !$omp end task
  enddo
enddo
```

e.g. https://github.com/ParRes/Kernels/blob/default/FORTRAN/p2p-tasks-openmp.F90

Example

```
program main
  use numerot
  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
```

https://github.com/jeffhammond/blog/tree/main/CODE

```
module numerot
  contains
    pure real function yksi(X)
      real, intent(in) :: X(100)
      !real, intent(out) :: R
      yksi = norm2(X)
    end function yksi
    pure real function kaksi(X)
      real, intent(in) :: X(100)
      kaksi = 2*norm2(X)
    end function kaksi
    pure real function kolme(X)
      real, intent(in) :: X(100)
      kolme = 3*norm2(X)
    end function kolme
 end module numerot
```

A coarray implementation?

```
program main
  use numerot
  real :: A(100) ! each image has one
  real :: R
  A = 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().eq.1) print*,R
 end program main
```

Coarrays are designed to support distributed memory, hence are based on image-private data.

There is limited opportunity for shared-memory optimizations in such codes, as direct inter-image copies will be required.

One of the common motivations for task-based models is dynamic load-balancing, but coarrays provide no mechanism for doing this, so users will have to write their own, which they always do poorly.

A do concurrent implementation?

```
program main
  use numerot
  real :: A(100), B(100), C(100)
  real :: RA, RB, RC
  integer :: k
  A = 1;  B = 1;  C = 1
  do concurrent (k=1:3) ! reduction, someday
    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 implementation only supports independent tasks, and is likely completely useless when the implementation uses SIMD lanes or GPU threads for DO CONCURRENT (DC).

As with coarrays, the if (...eq...) is not scalable to more general examples. Do we want arrays of functions?

Both the coarray and DC are also *tedious and error prone*, which is a good justification for adding new language features.

What might Fortran tasks look like?

```
do i=1,n
  task block async(i)
    do j=1,m
      ...
    enddo
  end task block
enddo
task sync all
```

The block mechanism is used for scoping.

Prepending task implies this block scope is also a task, which can execute asynchronously until synchronized.

Important questions:

- Is everything (e.g. I/O) allowed to be in a task?
- How do tasks interact with shared state?

What might Fortran tasks look like?

```
do i=1,n
  task block async(i)
    type :: private
    do j=1,m
      ...
    enddo
  end task block
enddo
task sync all
```

The block mechanism is used for scoping.

Prepending task implies this block scope is also a task.

It is essential to be able to have task-private state, which is already covered by the block feature.

What might Fortran tasks look like?

```
real :: x
do i=1,n
  task block async(i) shared(x)
    type :: private
    do j=1,m
      ...
    enddo
  end task block
enddo
task sync all
```

We also want to be able to describe the intent of data outside of the task, so we could reuse locality specifiers from DO CONCURRENT.

Locality specifiers already match OpenMP syntax, and a related feature in Fortran, so they are likely to be intuitive to Fortran programmers.

Task reductions are supported by OpenMP now, but the concept is tricky.

Atomics would be nice but that’s a big bag of worms.

What might Fortran tasks look like?

```
real :: x
do i=1,n
  task call foo(i,x)
enddo
task wait
```

```
do i=1,n
  task call foo(i,x) async(mod(i,2))
enddo
task sync 0
...
task sync 1
```

Calling subroutines as tasks is useful, but they should be **pure** in order to have reasonable behavior.

The right syntax for this is not obvious, but we can solve that later.

Summary

Fortran has two great ways to write parallel code, but needs a third.

Shared-memory task parallelism is implemented in OpenMP, OpenACC, and in models associated with languages that aren't Fortran.

Task parallelism allows users to solve new types of problems and make better use of existing parallel features, especially DO CONCURRENT (e.g. when executing on GPUs).

Fortran tasks make new things possible and obviate the need for tedious and error prone implementations. They also reduce the need for non-standard extensions like OpenMP and OpenACC.

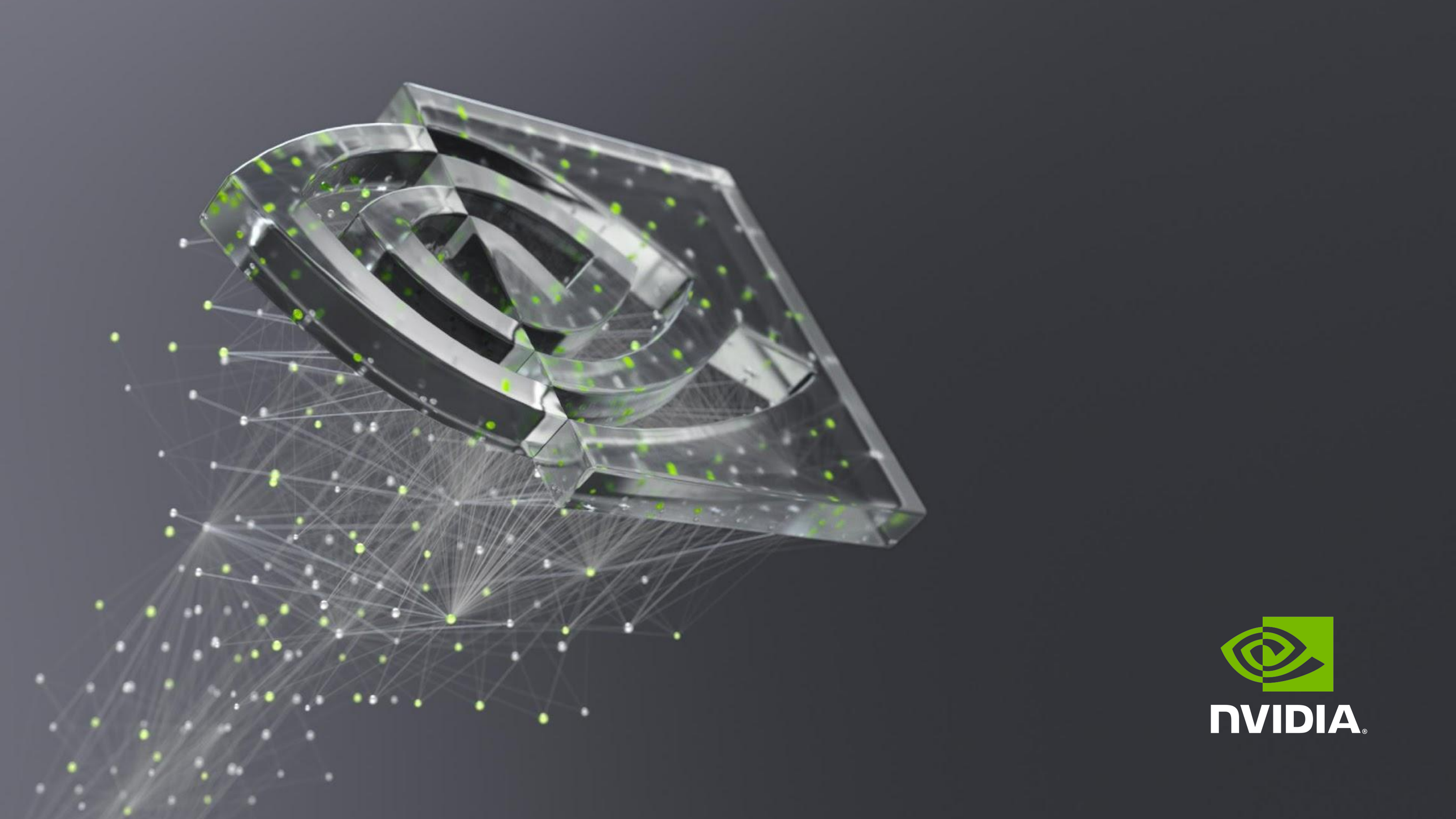
Please do not let whatever you don't like about my syntax to get in the way 🙂

Questions/Comments

Twitter: https://twitter.com/science_dot

Email: jeff_hammond@acm.org

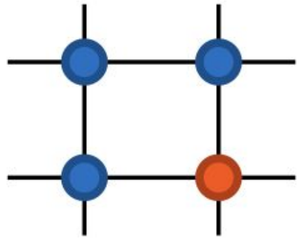
LinkedIn: https://www.linkedin.com/in/jeffhammond/


NVIDIA

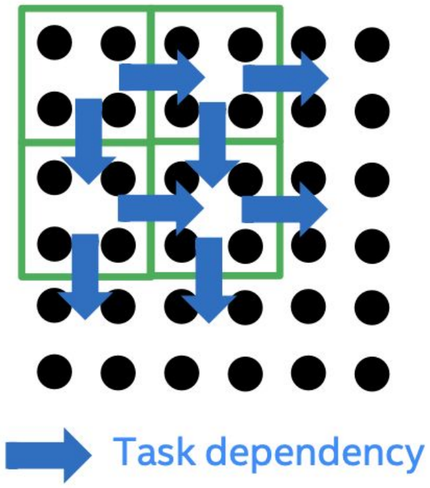
See https://github.com/ParRes/Kernels p2p* for details...

Motivation for Asynchrony 3

```
#pragma omp parallel
#pragma omp master
for (int i=1; i<m; i+=mc) {
  for (int j=1; j<n; j+=nc) {
    #pragma omp task \
            depend(in:grid[i-mc][j],grid[i][j-nc]) \
            depend(out:grid[i][j])
    for (int ii=i; ii<std::min(m,i+mc); ii++) {
      for (int jj=j; jj<std::min(n,j+nc); jj++) {
        A[ii][jj] = A[ii-1][jj] + A[ii][jj-1] - A[ii-1][jj-1];
      }
    }
  }
}
#pragma omp taskwait
```



$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$



This pattern appears in a range of applications:

- Deterministic neutron transport
- Dynamic programming for sequence alignment e.g. Smith-Waterman/PairHMM (bioinformatics)
- Linear algebra (e.g.NAS LU benchmark)