Outline

1. Introduction to Data-Parallelism
2. Fortran 90 Features
3. HPF Parallel Features
4. HPF Data Mapping Features
5. Parallel Programming in HPF
6. HPF Version 2.0

Data-Parallel Statements

- **Data parallelism emphasizes having many fine-grain operations, such as computations on every element of an array**
- **HPF has several ways to exploit data parallelism:**
  - **Array expressions:** Taken from Fortran 90
  - **`FORALL:`** Tightly-coupled parallel execution based on the structure of an index space
    - **`PURE:`** Procedures without side effects that may be called in `FORALL`
  - **`INDEPENDENT`**`:` Assertion that iterations do not interfere with each other
  - **HPF library** and intrinsics: Extended from Fortran 90

The Single-Statement FORALL

- **Syntax:**
  - `FORALL (` *index-spec-list* `[,` *mask-expr* `] )` *forall-assignment*
  - *index-spec* is *int-variable* = *triplet-spec*
  - *forall-assignment* is ordinary assignment or pointer assignment
- **Semantics:**
  - Equivalent to array assignment in Fortran 90
  - For each value of indices, check the mask
  - Compute right-hand sides for unmasked values
  - Make assignments to left-hand sides for unmasked values
  - Multiple assignments to the same location are not standard-conforming (i.e. are undefined)
- ***Note: FORALL is not a general-purpose parallel loop!***

CRPC

The Multi-Statement FORALL

- **Syntax:**
  - `FORALL ( `*index-spec-list* [`,` *mask*])
      forall-body-list
`END FORALL`
  - *forall-body* can be a *forall-assignment,* `FORALL`, or `WHERE`
- **Semantics:**
  - Multi-statement `FORALL` is shorthand for a series of single-statement `FORALL`s
  - Multi-statement `FORALL`s can be nested to produce more complex iteration spaces
  - Each bottom-level assignment statement is completed before the next one starts
- ***Note: FORALL is not a general-purpose parallel loop!***

An Example of FORALL

Initially,

a = [0, 1, 2, 3, 4]

b = [0, 10, 20, 30, 40]

c = [-1, -1, -1, -1, -1]

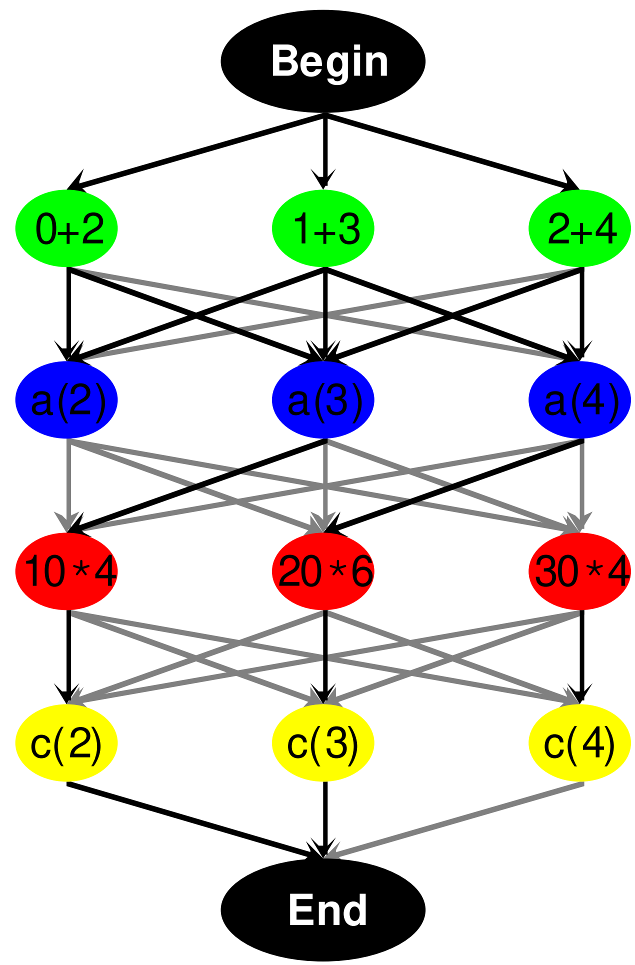
```
FORALL ( i = 2:4 )
   a(i) = a(i-1) + a(i+1)
   c(i) = b(i) * a(i+1)
END FORALL
```

Afterwards,

a = [0, 2, 4, 6, 4]

b = [0, 10, 20, 30, 40]

c = [-1, 40, 120, 120, -1]



An Example of DO

Initially,

a = [0, 1, 2, 3, 4]

b = [5, 6, 7, 8, 9]

c = [10, 20, 30, 40, 50]

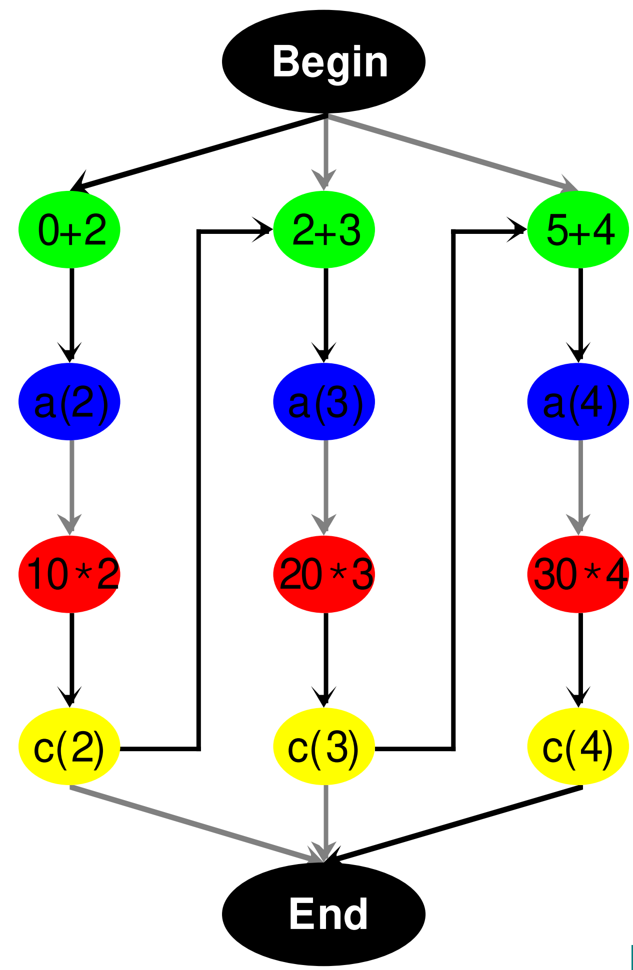
```
DO i = 2, 4
   a(i) = a(i-1) + a(i+1)
   c(i) = b(i) * a(i+1)
END DO
```

Afterwards,

a = [0, 10, 60, 40, 4]

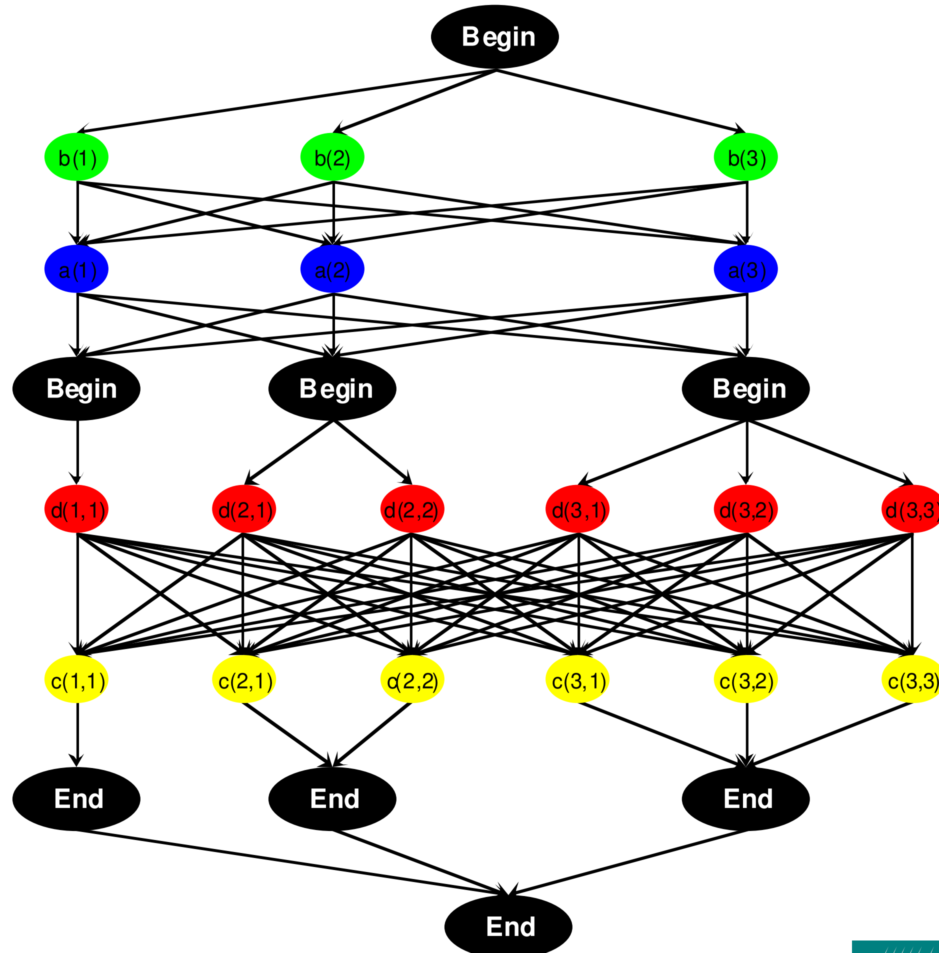
b = [5, 6, 7, 8, 9]

c = [10, 60, 420, 320, 50]



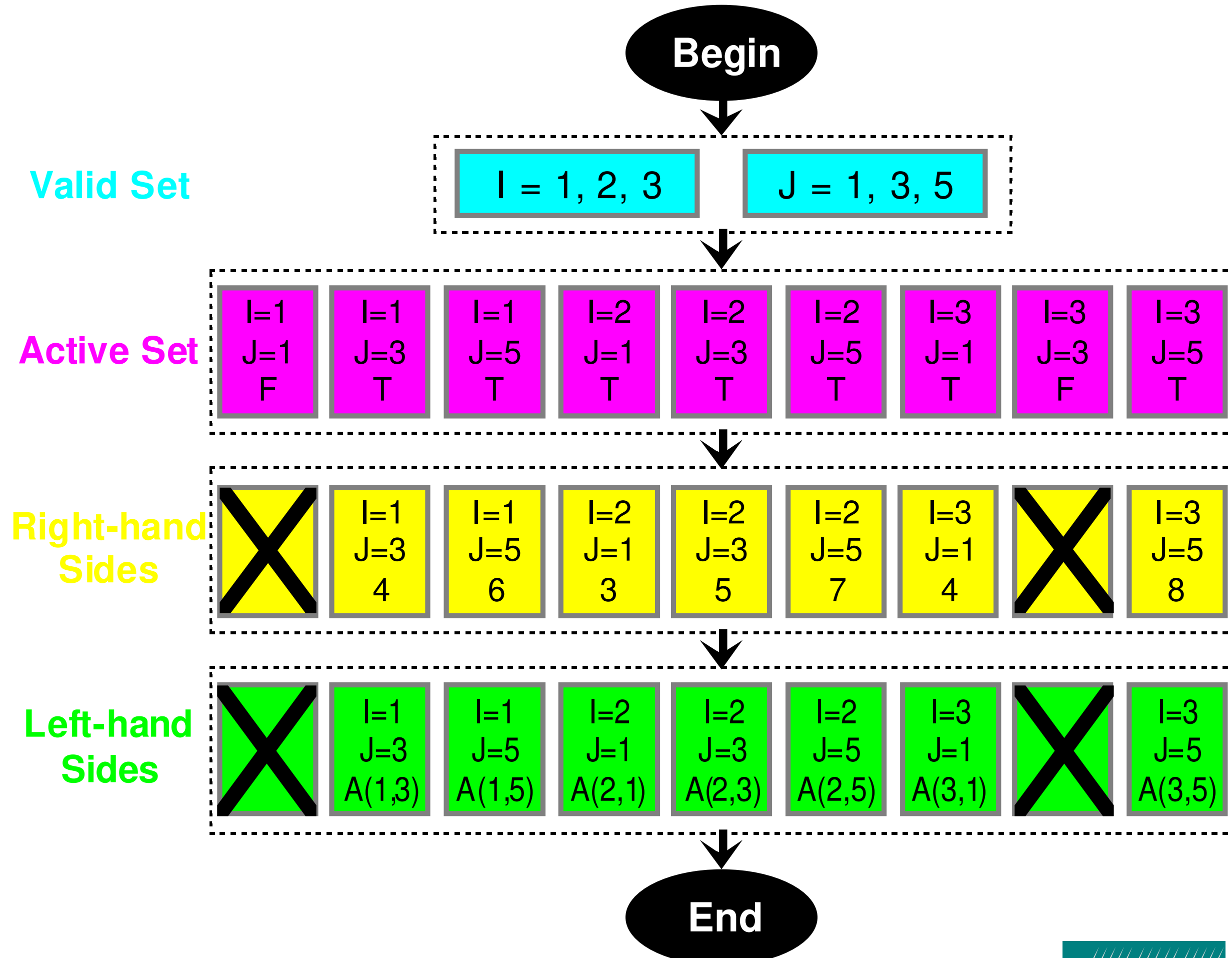
An Example of Nested FORALLs

```
FORALL ( i = 1:3 )
   a(i) = b(i)
   FORALL ( j = 1:i  )
      c(i,j) = d(i,j)
   END FORALL
END FORALL
```



An Example of Masked FORALL

```
FORALL ( i=1:3, &
     j=1:5:2, &
     i.NE.j )
   a(i,j) = i+j
END FORALL
```



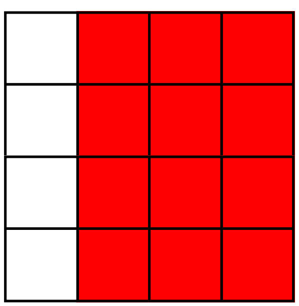
Why Use FORALL?

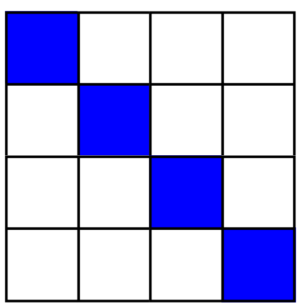
Assignments to array sections

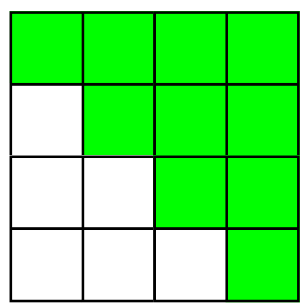
```
FORALL ( i = 1:4, j = 2:4 ) a(i,j) = a(i,j-1)
FORALL ( i = 1:4 ) a(i,i) = a(i,i) * scale
FORALL ( i = 1:4 )
   FORALL ( j=i:4 ) a(i,j) = a(i,j) / a(i,i)
END FORALL
FORALL ( i=1:4 )
   FORALL (j=ilo(i):ihi(i)) x(j) = x(j)*y(i)
END FORALL
```

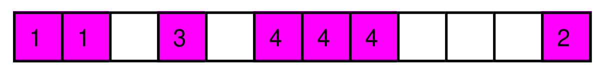
Calculating based on a subscript

```
FORALL (i=0:n,j=0:n) a(i,j) = SQRT(1.0*(i*i+j*j))/n
```










```
ilo(1:4) = [1,12,4,6]
ihi(1:4) = [2,12,4,8]
```

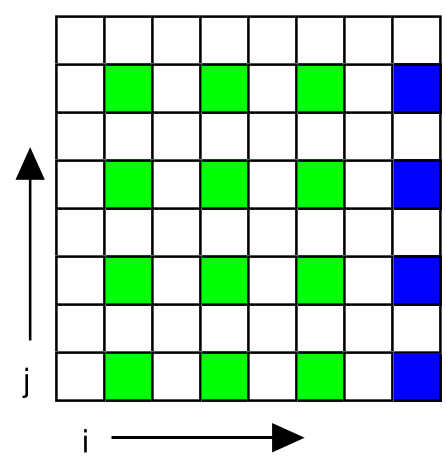
Typical Uses of FORALL

Anywhere array statements were used

```
FORALL ( i=0:nx, j=0:ny ) u(i,j) = fact*(u(i,j)-avg)
```

But periodic boundaries are difficult (no wraparound…)

```
FORALL ( j = 0:7:2 )
   FORALL ( i=2:7:2 ) r(i-1,j)=f(i,j)
   r(7,j) = f(0,j)
END FORALL
```



CRPC

Determinate Behavior of FORALL

- **Consider the statement:**
  - `FORALL ( i = 1:n ) a(ix(i)) = a(i)`
- **If `ix` has no repeated values (e.g. `ix` is a permutation), this is well-defined**
  - Note that `a(i)` is always the “old” value, not the new one computed elsewhere in the `FORALL`
- **If `ix` has repeated values (e.g. `ix(i)=i/2`), this is not defined by HPF**
  - **The compiler may take any action it feels appropriate…**
  - Assigning one of the possible values is appropriate
  - Reporting an error is appropriate
  - Assigning a random number is appropriate

Some Implications of FORALL

- **There is lots of fine-grain parallelism**
  - All right-hand sides of the same statement in body
  - All left-hand sides of the same statement in body
  - However, synchronization may be needed between RHS and LHS or between statements
  - However, data copying may be needed to implement parallelism
- **Syntax (almost) implies determinate behavior**
  - Exception: Attempted assignments of multiple values to one location
  - This was intentional

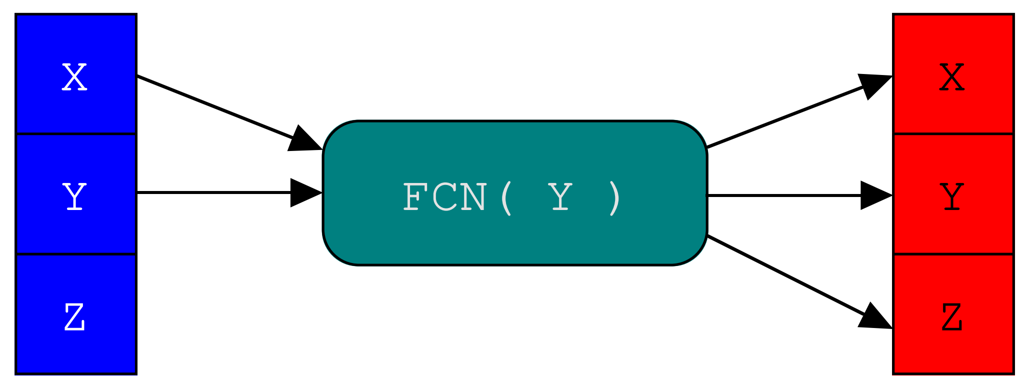
PURE Functions

- **`PURE` functions have no side effects**
- **Syntactic constraints:**
  - Global variables and dummy arguments cannot be used in any context that may cause the variable to become defined
    - Left hand side of assignment
    - `DO` index, `ASSIGN`, `ALLOCATE`
    - Actual argument with `INTENT(OUT)`
    - Targets of pointer assignments (due to later use of pointers)
    - Full list of restrictions is too long to fit on this slide!
  - No external I/O or file operations
  - Only inherited distribution/alignment of dummies and locals
- **Intrinsic functions are `PURE`**

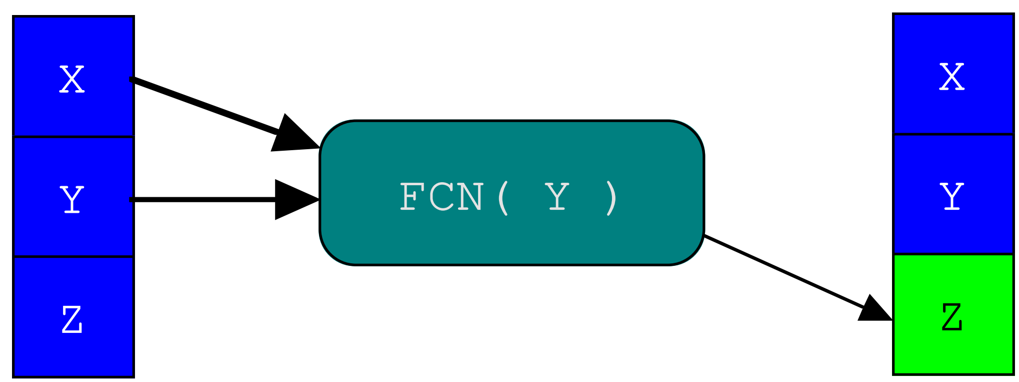
PURE Functions in Pictures

```
COMMON X
REAL Y, Z
Z = FCN( Y )
```

Normal **FCN**



PURE FCN



PURE Functions and FORALL

- **`PURE` functions are the only ones that can be invoked from a `FORALL`**
- **Safe to do this because they have no side effects**
- **Useful to do this because there are things you cannot do (directly) in a `FORALL`**
  - Conditionals and iteration
    - E.g., Do point-wise iteration this way
  - Local variables
    - E.g., Handle temporaries this way

Why Use PURE Functions?

- **Elemental functions**
  - Intrinsics
  - Equations of state, etc.
  - Note: A row can be an element!
- **Pointwise iteration**
  - Mandelbrot sets
  - Pointwise Newton iterations

PURE for Mandelbrot Sets

```
! The caller (Explicit interface not shown)
FORALL ( i=1:n, j=1:m )
    k(i,j) = mandelbrot( CMPLX((i-1)*1.0/(n-1), &
        (j-1)*1.0/(m-1)), 1000 )
END FORALL

! The callee
PURE INTEGER FUNCTION mandelbrot(x, itol)
COMPLEX, INTENT(IN) :: x
INTEGER, INTENT(IN) :: itol
COMPLEX xtmp
INTEGER k
    k = 0
    xtmp = -x
    DO WHILE (ABS(xtmp)<2.0 .AND. k<itol)
        xtmp = xtmp*xtmp - x
        k = k + 1
    END DO
    mandelbrot = k
END FUNCTION mandelbrot
```

Avoiding the PURE Function in Mandelbrot

```
k0 = 0
FORALL ( i=1:n, j=1:m )
    x(i,j) = CMPLX((i-1)*1.0/(n-1),(j-1)*1.0/(m-1))
    k(i,j) = 0
    xtmp(i,j) = -x(i,j)
    mask(i,j) = .TRUE.
END FORALL
DO WHILE (ANY(mask(1:n,1:m)) .AND. k0<1000)
    FORALL ( i=1:n, j=1:m, mask(i,j))
        xtmp(i,j) = xtmp(i,j)*xtmp(i,j)-x(i,j)
        k(i,j) = k(i,j) + 1
        mask(i,j) = ABS(xtmp(i,j))<2.0
    END FORALL
    k0 = k0 + 1
END DO
```

An Impure Function

```
REAL FUNCTION polluted(w, x, y)
   REAL, INTENT(IN) :: w
   REAL, INTENT(IN) :: x(10)
   REAL, TARGET     :: y(100)
   INTEGER, SAVE :: last = 1
   REAL, POINTER :: z
   INTEGER num_call
   REAL, TARGET :: lookup
   COMMON /GLOBAL/ num_call, lookup
   INTERFACE
      PURE SUBROUTINE bin_search(a, b, i)
      REAL, INTENT(IN)         :: a
      REAL, INTENT(INOUT)      :: b(100)
      INTEGER, INTENT(INOUT) :: i
      END SUBROUTINE bin_search
   END INTERFACE
   CALL bin_search(w, lookup, last)
   z => y( last:last+9 )
   num_call = num_call + 1
   polluted = SUM( x*z )
END FUNCTION polluted
```

CRPC

A Purified Function

```
PURE REAL FUNCTION clean(w, x, y)
   REAL, INTENT(IN) :: w
   REAL, INTENT(IN) :: x(10)
   REAL, INTENT(IN) :: y(100)
   INTEGER last
   REAL, POINTER :: z
   INTEGER num_call
   REAL, TARGET :: lookup
   COMMON /GLOBAL/ num_call, lookup
   INTERFACE
      PURE SUBROUTINE bin_search(a, b, i)
      REAL, INTENT(IN)       :: a
      REAL, INTENT(IN)       :: b(100)
      INTEGER, INTENT(INOUT) :: i
      END SUBROUTINE bin_search
   END INTERFACE
   last = 1
   CALL bin_search(w, lookup, last)
   clean = SUM( x * y( last:last+9 ) )
END FUNCTION clean
```

The INDEPENDENT Directive

- **Syntax:**
  - `!HPF$ INDEPENDENT [ , NEW(` *variable-list* `) ]`
- **Semantics:**
  - `INDEPENDENT` is an assertion that no iteration affects any other iteration in any way
  - `NEW` variables are treated as if they were allocated anew for each iteration (`DO` only)
  - Applied to a `DO`: states that there are no loop carried dependences (except for `NEW` variables)
  - Applied to a `FORALL`: states that no index point assigns to any location that another uses
  - **If the assertion is false, the program is not standard-conforming (i.e. results are not defined)**
- ***Note: INDEPENDENT is not a general parallel loop!***

An Example of INDEPENDENT

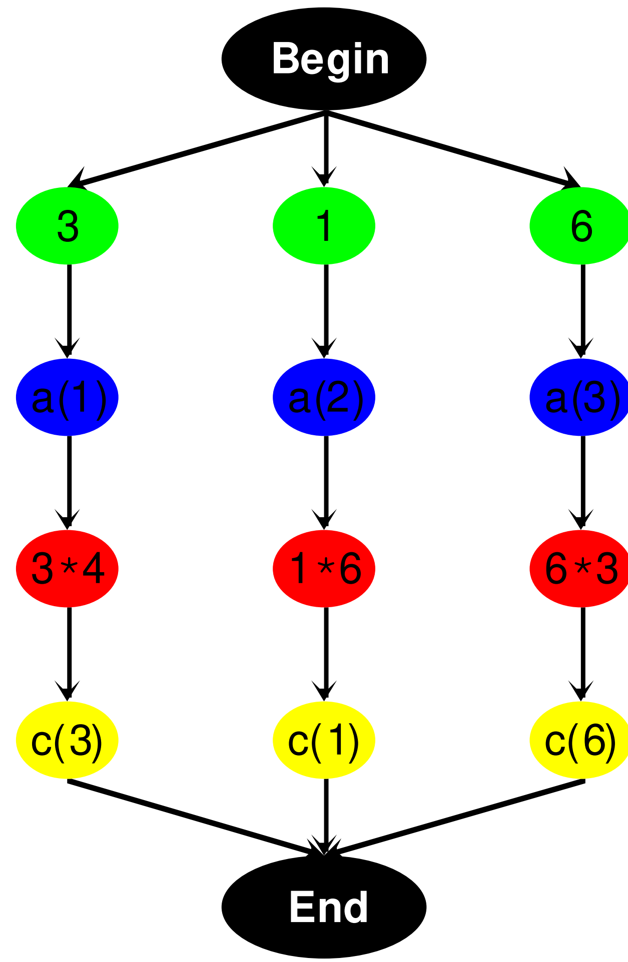
Initially,

a = [0, 2, 4, 6, 1, 3, 5, 7]

b = [6, 5, 4, 3, 2, 3, 4, 5]

c = [-1,-1,-1,-1,-1,-1,-1,-1]

```
!HPF$ INDEPENDENT
DO j = 1, 3
   a(j) = a(b(j))
   c(a(j)) = a(j)*b(a(j))
END DO
```



Afterwards,

a = [3, 1, 6, 6, 1, 3, 5, 7]

b = [6, 5, 4, 3, 2, 3, 4, 5]

c = [6,-1,12,-1,-1,18,-1,-1]

CRPC

Another Example of FORALL

Initially,

a = [0, 2, 4, 6, 1, 3, 5, 7]

b = [6, 5, 4, 3, 2, 3, 4, 5]

c = [-1,-1,-1,-1,-1,-1,-1,-1]

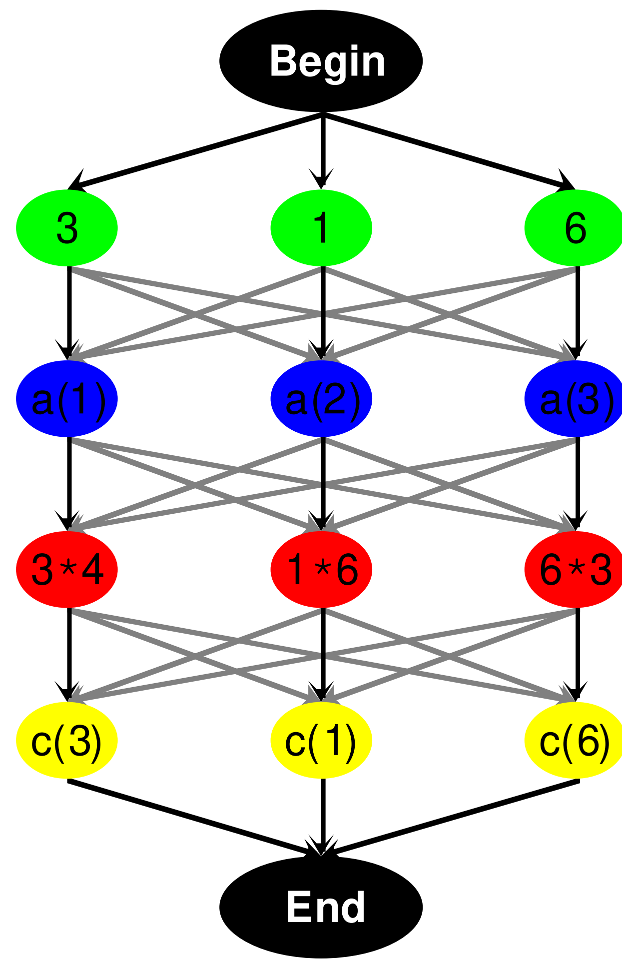
```
!HPF$ INDEPENDENT
FORALL ( j = 1:3 )
   a(j) = a(b(j))
   c(a(j)) = a(j)*b(a(j))
END FORALL
```

Afterwards,

a = [3, 1, 6, 6, 1, 3, 5, 7]

b = [6, 5, 4, 3, 2, 3, 4, 5]

c = [6,-1,12,-1,-1,18,-1,-1]



Another Example of FORALL With INDEPENDENT

Initially,

a = [0, 2, 4, 6, 1, 3, 5, 7]

b = [6, 5, 4, 3, 2, 3, 4, 5]

c = [-1,-1,-1,-1,-1,-1,-1,-1]

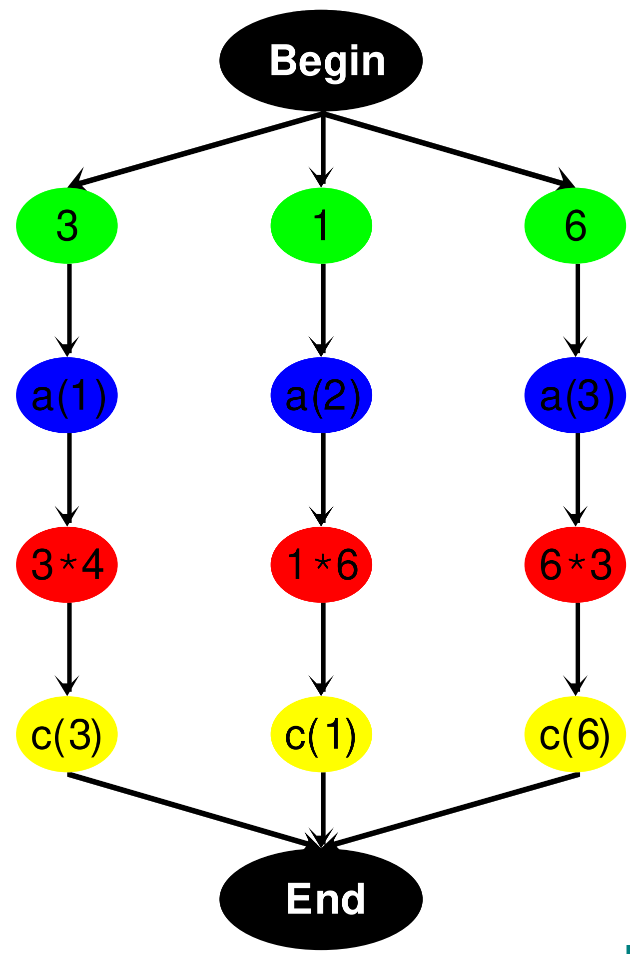
```
!HPF$ INDEPENDENT
FORALL ( j = 1:3 )
   a(j) = a(b(j))
   c(a(j)) = a(j)*b(a(j))
END FORALL
```

Afterwards,

a = [3, 1, 6, 6, 1, 3, 5, 7]

b = [6, 5, 4, 3, 2, 3, 4, 5]

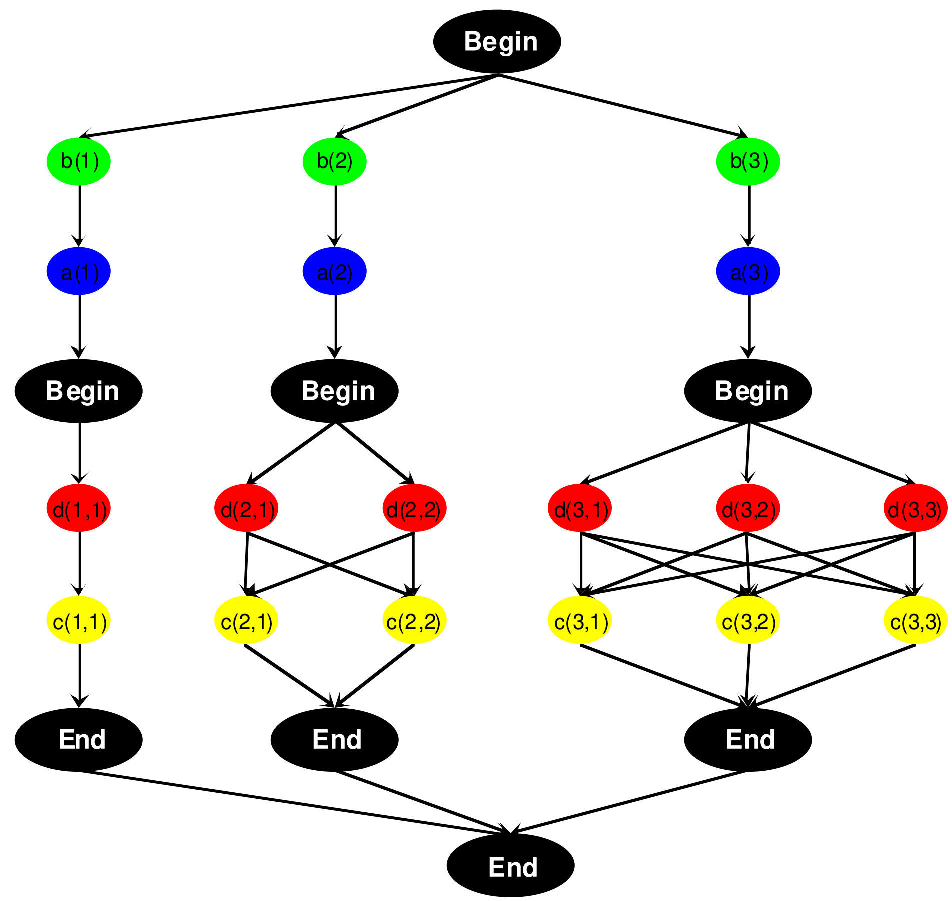
c = [6,-1,12,-1,-1,18,-1,-1]



CRPC

An Example of Nested INDEPENDENT FORALLs

```
!HPF$ INDEPENDENT
FORALL ( i = 1:3 )
   a(i) = b(i)
   FORALL ( j = 1:i )
      c(i) = d(i)
   END FORALL
END FORALL
```



The INDEPENDENT Directive: More Details

- **The Fundamental Rule:**
  - If one iteration writes to an object, others cannot read or write it
- **Things that write to an object:**
  - Assignment, `ASSIGN`, `DO` index, …
    - To the object itself
    - To an aggregate that contains it
    - Through a pointer to it
  - Input/Output statements
    - Write the file pointer (except `INQUIRE`)
    - `READ` assigns to its input list
- **Things that read an object**
  - Uses in expressions (as you expect)
  - Input/Output statements
    - Read the file pointer (always)

Examples of Correct INDEPENDENT Assertions

Always true

```
!HPF$ INDEPENDENT
FORALL (i=2:n-1) a(i)=b(i-1)+b(i)+b(i+1)

!HPF$ INDEPENDENT, NEW(j)
DO k = 2, m-1, 2
   !HPF$ INDEPENDENT, NEW(vl,vr)
   DO j = 2, n-1, 2
      vr = x(j,k) - x(j-1,k)
      vl = x(j+1,k) - x(j,k)
      x(j,k) = x(j,k) + 0.5*(vr-vl)
   END DO
END DO
```

Some compilers will catch these on their own; some won't

Examples of Incorrect INDEPENDENT Assertions

`INDEPENDENT` does not handle reductions

```
!HPF$ INDEPENDENT
DO i = 1, n
   x = x + a(i)*a(i)
END DO
```

`INDEPENDENT` does not know about higher-level correctness

```
DO WHILE (err > err_tol)
   !HPF$ INDEPENDENT
   DO i = 2, n-1
      b(i) = a(i)
      a(i) = 0.5 * (a(i-1) + a(i+1))
      b(i) = ABS(b(i) - a(i))
   END DO
   err = MAXVAL(b(2:n-1))
END DO
```

Example of Data-Dependent INDEPENDENT Assertion

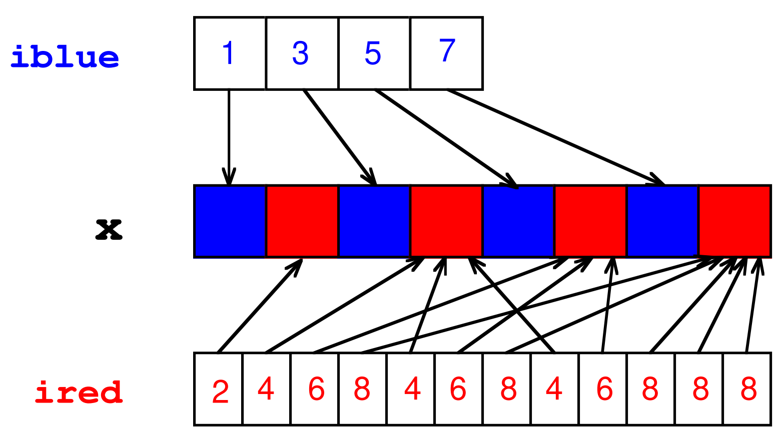
Sometimes true

```
!HPF$ INDEPENDENT, NEW(j, n1)
DO i = 1, nblack
   n1 = iblue(i)
   DO j = ibegin(n1), ibegin(n1+1)-1
      x(n1) = x(n1) + y(j)*x(ired(j))
   END DO
END DO
```

Example of Data-Dependent INDEPENDENT Assertion

Sometimes true

```
!HPF$ INDEPENDENT, NEW(j, n1)
DO i = 1, nblack
   n1 = iblue(i)
   DO j = ibegin(n1), ibegin(n1+1)-1
      x(n1) = x(n1) + y(j)*x(ired(j))
   END DO
END DO
```

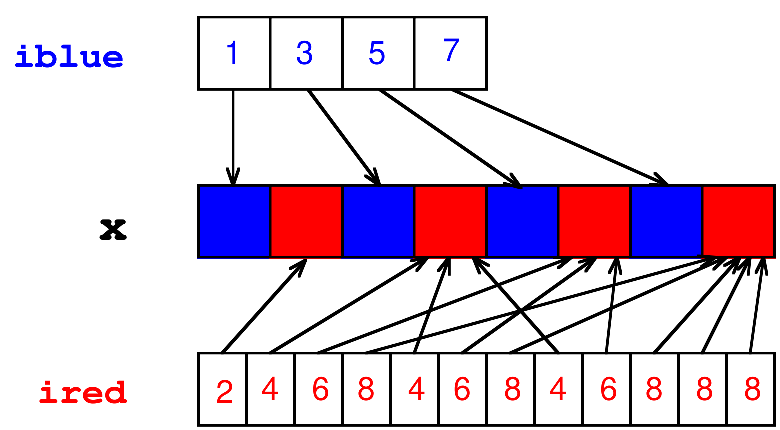


True

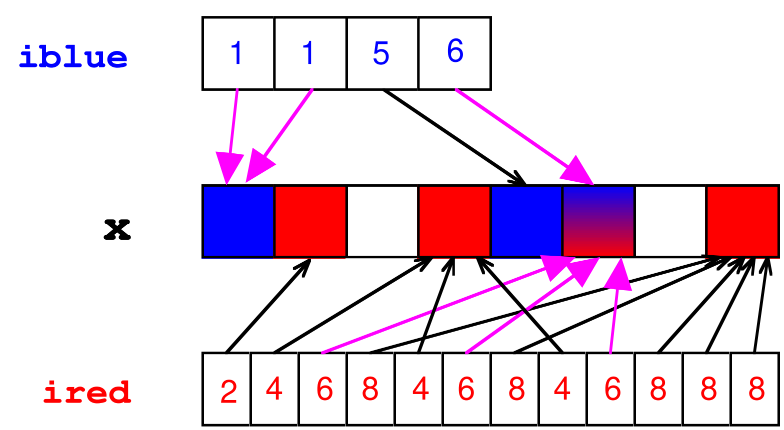
Example of Data-Dependent INDEPENDENT Assertion

Sometimes true

```
!HPF$ INDEPENDENT, NEW(j, n1)
DO i = 1, nblack
   n1 = iblue(i)
   DO j = ibegin(n1), ibegin(n1+1)-1
      x(n1) = x(n1) + y(j)*x(ired(j))
   END DO
END DO
```



True



False

CRPC

Possibly Correct INDEPENDENT Assertion

```
! Correct if Fermat's Last Theorem is true
!HPF$ INDEPENDENT, NEW(ix, iy, n, z, zi)
      DO i = 1, 10
         forever: DO
            READ (i, '(2I12,I3)') ix, iy, n
            IF (ix<=0 .OR. iy<=0 .OR. n<=2) &
              EXIT forever
            z = (ix**n + iy**n) ** (1.0/n)
            WRITE (i+10, '(E18.6)') z
            zi = FLOOR( z )
            IF (zi == z) THEN
              PRINT *, 'Fermat was wrong!'
              PRINT *, 'Notify Andrew Wiles!'
              GOTO 100
            END IF
         END DO
      END DO
100   CONTINUE
```

Why Use INDEPENDENT?

Yet another way to do (some) array assignments

```
!HPF$ INDEPENDENT
DO i = 1, n
    a(i) = b(i)
END DO
```

Express application-dependent information

```
! “colors” don't interfere with each other
DO i = 1, ncolor
    !HPF$ INDEPENDENT, NEW(i1,i2,f12)
    DO ix = color_beg(i), color_end(i)
        i1 = icolor(ix,1)
        i2 = icolor(ix,2)
        f12 = w(i1)-w(i2)
        x(i1) = x(i1) + f12
        x(i2) = x(i2) - f12
    END DO
END DO
```

Typical Use of INDEPENDENT

Express *application-dependent* information

```
! Meshes stored in 1-D array don't overlap
!HPF$ INDEPENDENT
DO k = 1, n_mesh
   CALL update(x(:),neighbor(:,ibeg(k):iend(k)))
END DO

SUBROUTINE update(x, nbr)
REAL x(:), tmp
INTEGER nbr(2,:), i
   DO i = 1, UBOUND(nbr,2)
      tmp = flux( x(nbr(1,i)), x(nbr(2,i)) )
      x(nbr(1,i)) = x(nbr(1,i)) + tmp
      x(nbr(2,i)) = x(nbr(2,i)) - tmp
   END DO
END
```

INDEPENDENT as an Assertion

```
!HPF$ INDEPENDENT
DO I = 1, N
   A(INDX(I)) = B(I)
END DO
DO J = 1, N
   HIST(INDX(J)) = HIST(INDX(J)) + 1
END DO
```

<u>The programmer thinks:</u>

"If `INDX` has repeated values, then the algorithm will converge no matter which one is assigned to `A`. So it's safe to mark the I loop `INDEPENDENT`."

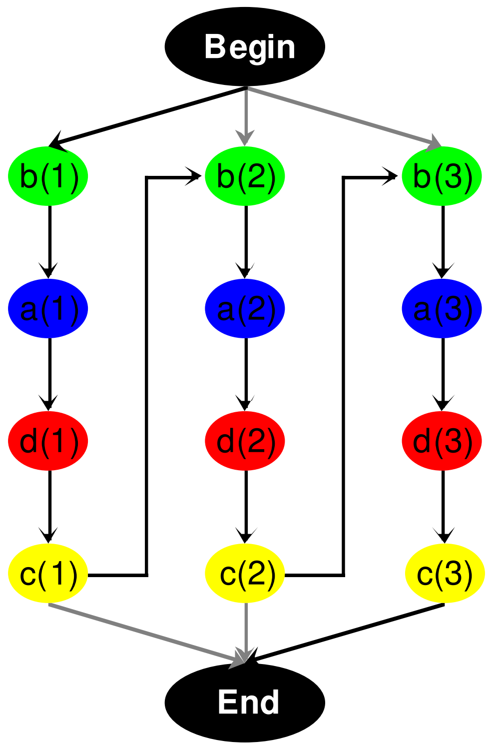
<u>The compiler thinks:</u>

"Since there's no dependence, `INDX` must be a permutation. I can compute `HIST` with a bitwise OR reduction instead of integer additions."

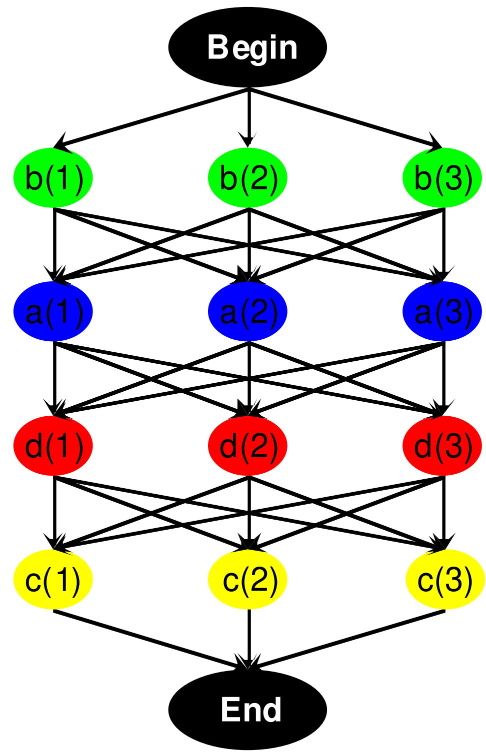
Some Implications of INDEPENDENT

- **Iterations of an `INDEPENDENT` can execute atomically in parallel**
  - Communication/synchronization can be done outside the loop
  - No way to force synchronization inside the loop
- **A conflict is a conflict, even if it doesn't matter**
  - "It will eventually converge to the same thing": not allowed
- **Behavior matters, not syntax**
  - OK to call a function that behaves like `PURE` for this case
- **Different iterations can read/write to different files (I/O units)**
  - But not to the same file

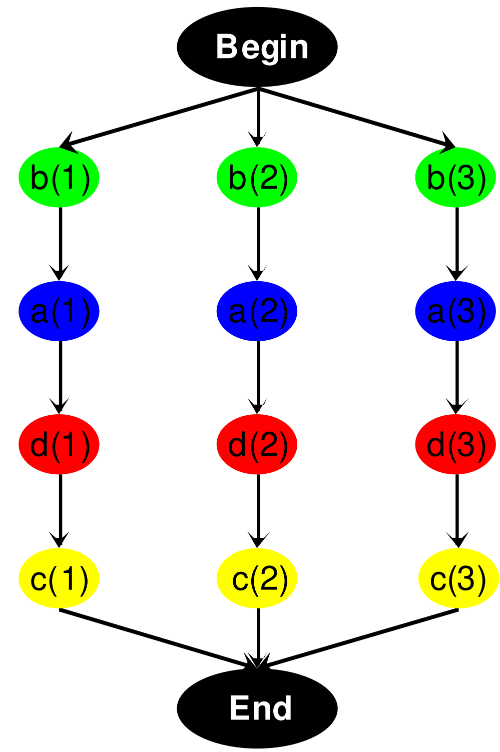
In Summary: DO, FORALL and INDEPENDENT




```
DO i = 1, 3
  a(i) = b(i)
  c(i) = d(i)
END DO
```




```
FORALL (i = 1:3)
  a(i) = b(i)
  c(i) = d(i)
END FORALL
```




```
!HPF$ INDEPENDENT
DO i = 1, 3
  a(i) = b(i)
  c(i) = d(i)
END DO
```

The HPF Library and New Intrinsics

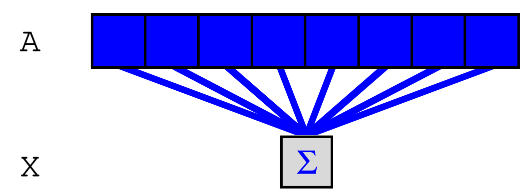
- Extended intrinsics: `MAXLOC, MINLOC`
- One elemental intrinsic: `ILEN`
- System inquiry intrinsics: `NUMBER_OF_PROCESSORS`
- New reduction functions: `IAND`
- Combining-scatter functions: `SUM_SCATTER`
- Prefix reduction functions: `SUM_PREFIX`
- Sorting functions: `GRADE_UP`
- Bit manipulation functions: `POPCNT`
- Data distribution inquiry subroutines: `HPF_DISTRIBUTION`

Examples of HPF Library

- **Many implement data-parallel operations that are not elemental**

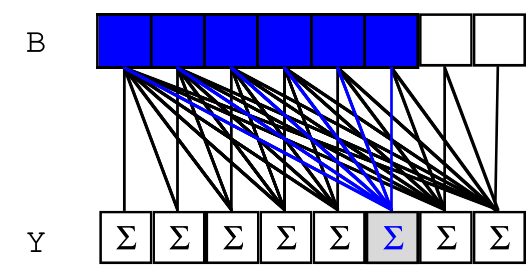
`X = SUM(A)`

$$x = \sum_{i=1}^{n} a_i$$



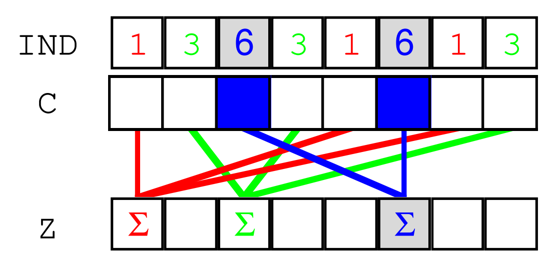
`Y = SUM_PREFIX(B)`

$$y_j = \sum_{i=1}^{j} b_i$$



`Z = SUM_SCATTER(C, Z, IND)`

$$z_j = \sum_{i \ni ind_i = j} c_i$$



Why Use the HPF Library?

- **If you need the functions…**
  - Round out the set of reductions
    - All built-in associative, commutative functions
  - Partial reductions (prefix reductions) for all reductions
  - Sorting
- **Functions were chosen for the library because**
  - They are useful
  - They are data-parallel
  - They are hard to implement efficiently by hand

Typical Uses of HPF Library

Accumulations through indirection arrays

```
x = SUM_SCATTER(  flux, x, nbr(1,1:n) )
x = SUM_SCATTER( -flux, x, nbr(2,1:n) )
! Equivalent to the following
DO i = 1, n
   x(nbr(1,i)) = x(nbr(1,i)) + flux(i)
   x(nbr(2,i)) = x(nbr(2,i)) - flux(i)
END DO
```

Manipulating array-based sparse structures

```
inum(1:n) = MAX( iend(1:n)-ibeg(1:n)+1, 0 )
ibeg_new(1:n) = SUM_PREFIX(inum(1:n)) + 1
iend_new(1:n) = ibeg_new(1:n)+inum(1:n)-1
! Moving the data left as exercise for reader
```

EXTRINSIC Procedures

- **`EXTRINSIC` is an escape mechanism for calling other paradigms from HPF code**
- **On the caller (HPF) side:**
  - There must be an explicit interface with `EXTRINSIC` directive
  - Remapping occurs to guarantee that arguments meet any distribution specifications
  - System synchronizes all processors before the call
  - System calls “local” routine on every processor
- **On the callee (non-HPF) side:**
  - `INTENT(IN)` and `INTENT(OUT)` must be obeyed
  - If variables are replicated, caller must make them consistent before return
  - Processors can access their own section of distributed arrays

Hints for Using Data Parallel Statements

- **Use `FORALL` to extend array operations**
  - More shapes
  - User-defined elemental functions
  - Better than array syntax?
- **Beware of hidden overheads**
  - Intra-statement overheads
  - Inter-statement synchronizations
  - Complex pure functions
- **Assert `INDEPENDENT` only when it is true**
  - This has implications for debuggers and programming environments!