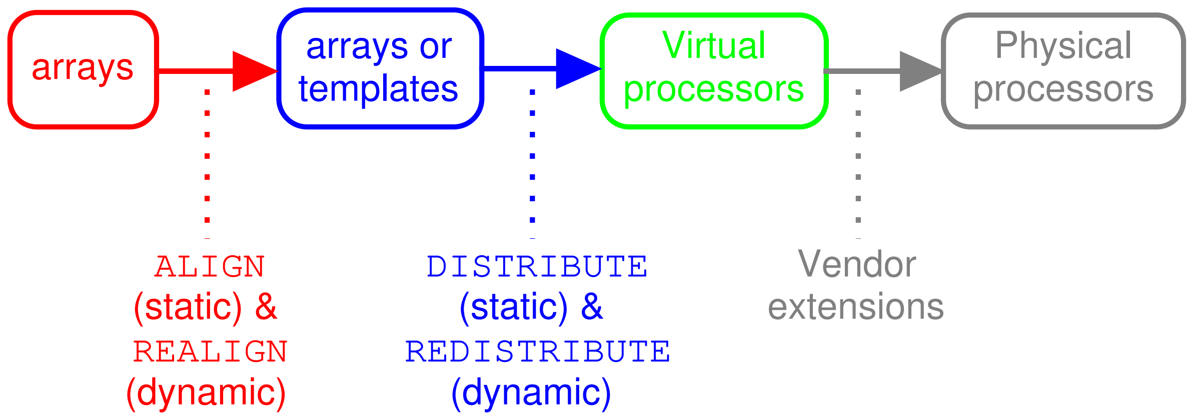
Outline

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 Mapping



- **Data mapping complements data parallelism by placing data for parallel access**
- **HPF uses a two-phase data mapping:**
  - **`ALIGN:`** Creates a relationship between objects
  - **`DISTRIBUTE:`** Partitions an object between processors
  - Vendors may define further levels of mapping

Data Mapping, cont.

- **Goals:**
  - Avoiding Contention: Data updated in parallel should be on different processors
  - Locality of Reference: Data used together should be on the same processor
- **These goals are sometimes in conflict**
  - To avoid all contention, put every data item on its own processor
  - To maximize locality, put all data on one processor
  - HPF gives you the tools to resolve these conflicts, but doesn't solve them for you

The DISTRIBUTE Directive

- **Syntax:**
  - `!HPF$ DISTRIBUTE` *array* `(`*dist-format-list*`)` [`ONTO` *procs*]
  - `!HPF$ DISTRIBUTE(`*dist-format-list*`)` [`ONTO` *procs*]`::`*array-list*
- **Semantics:**
  - Each dimension of *array* is divided according to the corresponding pattern in *dist-format-list*
  - `ONTO` (if present) names the processor array to distribute on
- **Options for dist-format**
  - (Let N be the number of elements.)
  - (Let P be the number of processors.)
  - `BLOCK` : Contiguous pieces of size N/P on each processor
  - `CYCLIC` : Every Pth element to the same processor
  - `CYCLIC(`K`)` : Every Pth block of size K to the same processor
  - `*` : Dimension not distributed

CRPC

Examples of DISTRIBUTE

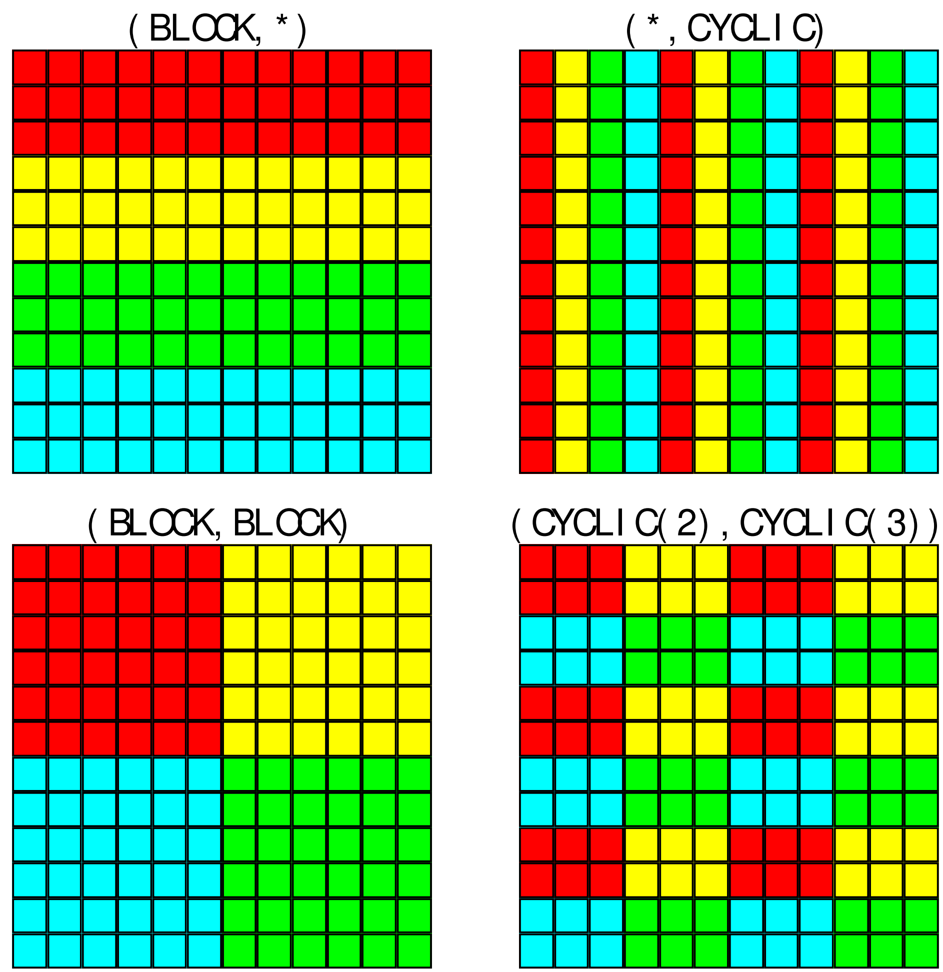
```
REAL W(12,12),X(12,12),Y(12,12),Z(12,12)

!HPF$ DISTRIBUTE W(BLOCK,*)

!HPF$ DISTRIBUTE X(*,CYCLIC)

!HPF$ DISTRIBUTE Y(BLOCK,BLOCK)

!HPF$ DISTRIBUTE Z(CYCLIC(2),CYCLIC(3))
```



Why Use DISTRIBUTE?

- **Compilation is based on the data distribution**
  - Computations will execute in parallel if
    - They are conceptually parallel (e.g. array operations)
    - The data is partitioned (e.g. by `DISTRIBUTE`)
- **Communication and synchronization are based on the data distribution**
  - `BLOCK` reduces surface-to-volume ratio
  - `CYCLIC` (and `CYCLIC(K)`) improves load balance
  - `*` keeps things on one processor

DISTRIBUTE and Communication

- **Communication (data movement) happens when two data items on different processors must be brought together**
  - Assume `a(n)`, `a(m)` are on different processors
    - `a(n) = a(m)` – Communicate one element
    - `x = a(n) + a(m)` – Communicate one of {`a(n)`,`a(m)`}
    - `a(n) = a(n)` – No communication
- **How communication is accomplished is a system problem**
  - Depends on data mapping (`DISTRIBUTE` and `ALIGN`)
  - Depends on data access (subscripts of arrays)
  - Depends on implementation (whose compiler?)

CRPC

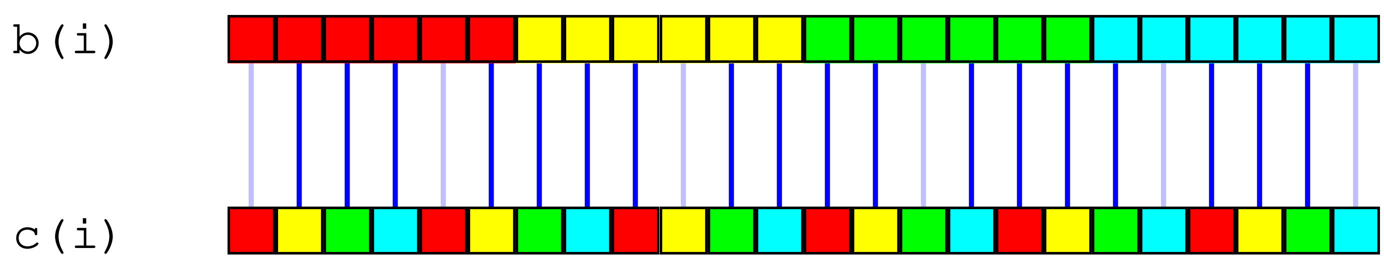
Rules of Thumb for Communication

- **`BLOCK` is good for local (e.g. nearest-neighbor) communication**
  - Look for subscripts like `a(i+1,j-1)`
  - Look for intrinsics like `CSHIFT`
  - Warning: `BLOCK` depends on array size
- **`CYCLIC` has non-obvious locality**
  - Look for subscripts like `x(i+jmp)`, where `jmp` is a multiple of the number of processors
  - For example, `jmp` may be a power of 2 on a hypercube machine
  - `CYCLIC` always balances the memory load
- **`CYCLIC(K)` has some of the advantage of each**
- **Strides are expensive on any distribution**
  - But some special cases are worth recognizing
- **Broadcasts are equally expensive on any distribution**
- **Communication between different distributions is very expensive**

Quantifying Communication

- **Basic idea:**
  - Match elements that are combined
  - Data volume is the number of matches that cross processor boundaries
  - Communication start-ups are one of
    - The number of distinct ordered pairs of processors *[parallel]*
    - The same as data volume *[sequential]*

```
REAL b(24), c(24)
!HPF$ DISTRIBUTE b(BLOCK), c(CYCLIC)
FORALL (i=1:24) b(i) = c(i)
```

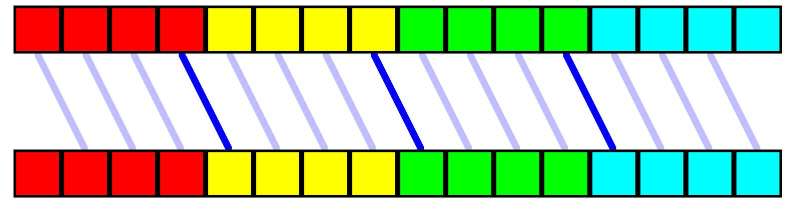


Quantifying Communication: BLOCK

b(i)

`FORALL (i=LB:UB) b(i)=b(i+K)`

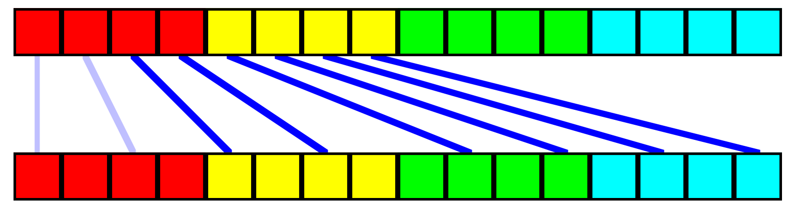
b(i+K)



b(i)

`FORALL (i=LB:UB) b(i)=b(A*i+K)`

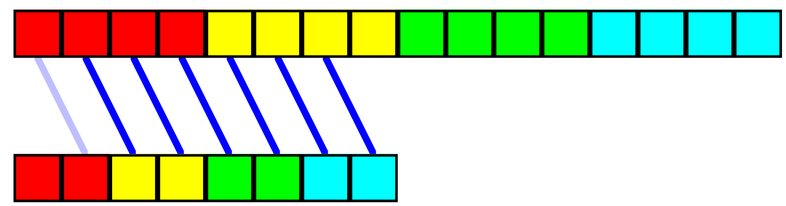
b(A*i+K)



b(i)

`FORALL (i=LB:UB) b(i)=b2(i+K)`

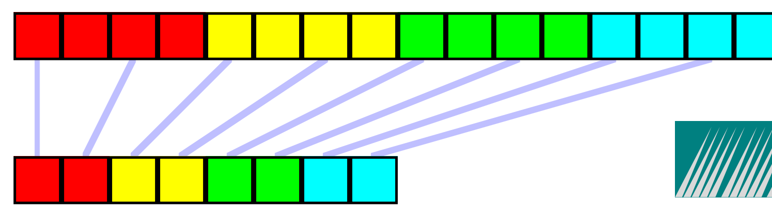
b2(i+K)



b(A*i+K)

`FORALL (i=LB:UB) b2(i)=b(A*i+K)`

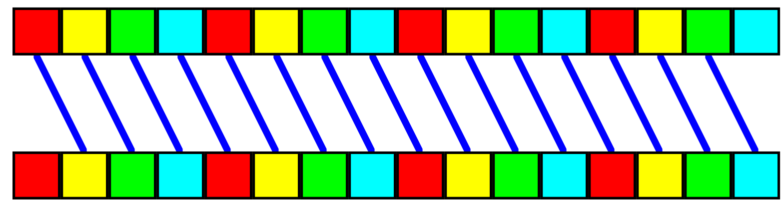
b2(i)



Quantifying Communication: CYCLIC

```
FORALL (i=LB:UB) c(i)=c(i+K)
```

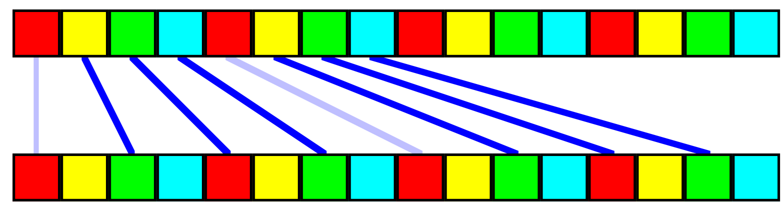
c(i)



c(i+K)

```
FORALL (i=LB:UB) c(i)=c(A*i+K)
```

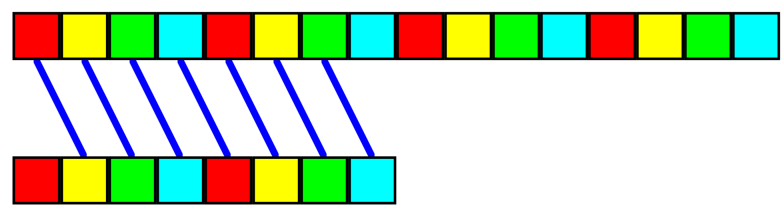
c(i)



c(A*i+K)

```
FORALL (i=LB:UB) c(i)=c2(i+K)
```

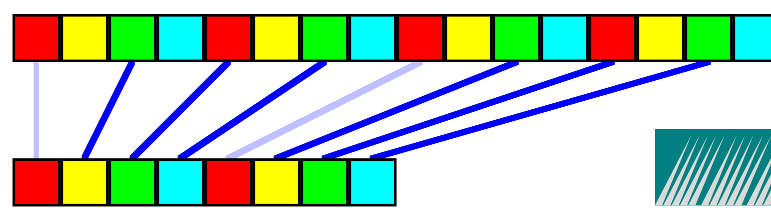
c(i)



c2(i+K)

```
FORALL (i=LB:UB) c2(i)=c(A*i+K)
```

c(A*i+K)



cs(i)

Quantifying Communication: Other Cases

- **`CYCLIC(K)`**
  - Quantify as if `BLOCK` on many processors (As if each processor stores `K` elements)
  - Remove communications between processors that are mapped together (due to divisibility)
- **Distinct distributions**
  - In general, communicate almost all elements of array
  - In general, each processor communicates with all others
- **Indirection arrays (and other complex subscripts)**
  - Can't be handled at compile time
  - But efficient runtime methods can minimize data volume and reduce contention (at the cost of some preprocessing)

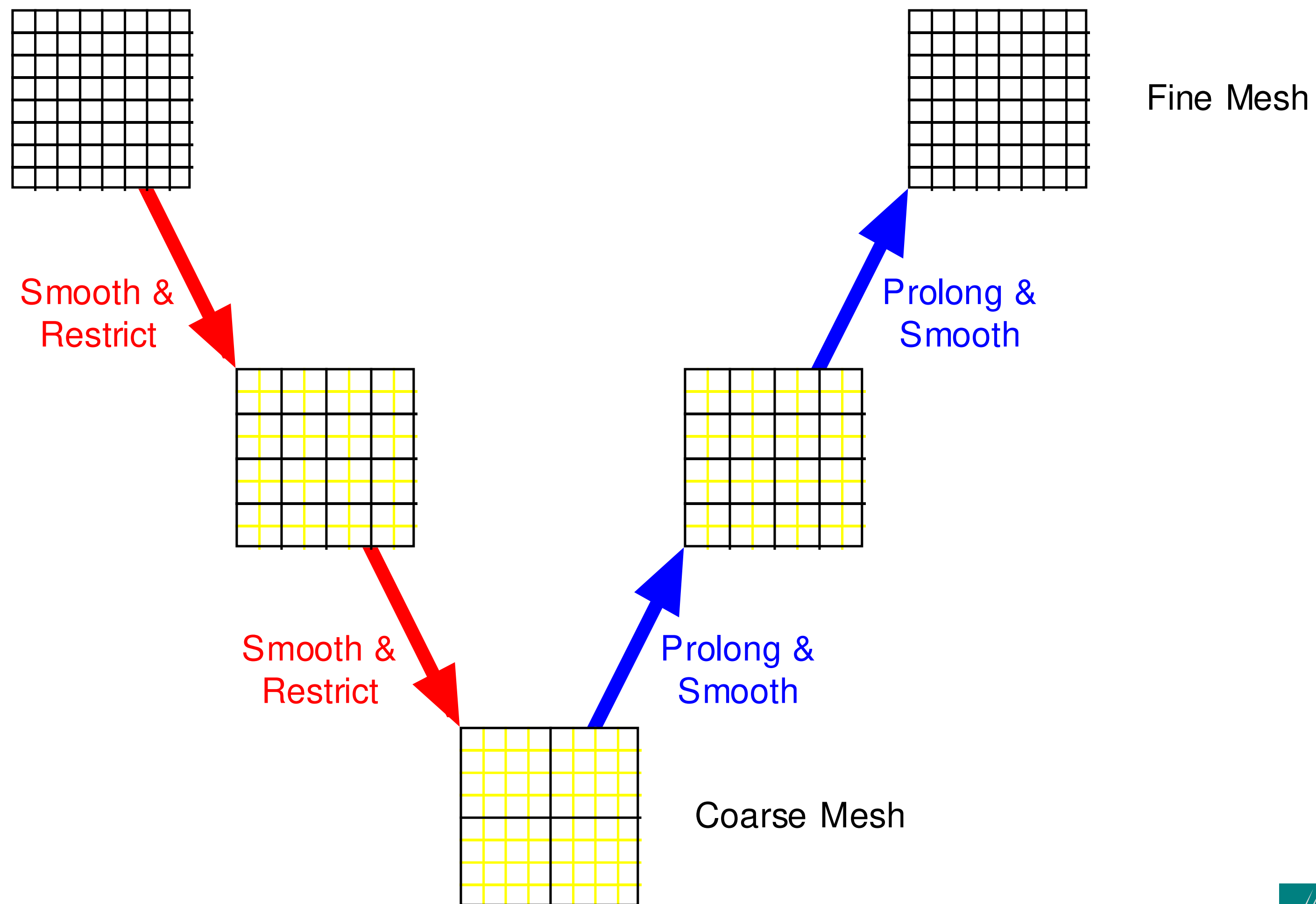
Other Communication Topics

- **Intrinsics and HPF library**
  - Most generate regular collective communications patterns
    - `CSHIFT` ⇒ circular shift
    - `TRANSPOSE` ⇒ all-to-all
    - `SUM, SUM_PREFIX, SUM_SUFFIX` ⇒ reduction tree
    - `SUM_SCATTER` ⇒ irregular pattern
  - Efficient collective communications methods are known
  - Compilers that don’t use them won’t stay in business
- **Precise communication information**
  - Compiler analysis and optimization makes it very difficult to determine or control
  - A real opportunity for programming tools

Typical Uses of DISTRIBUTE

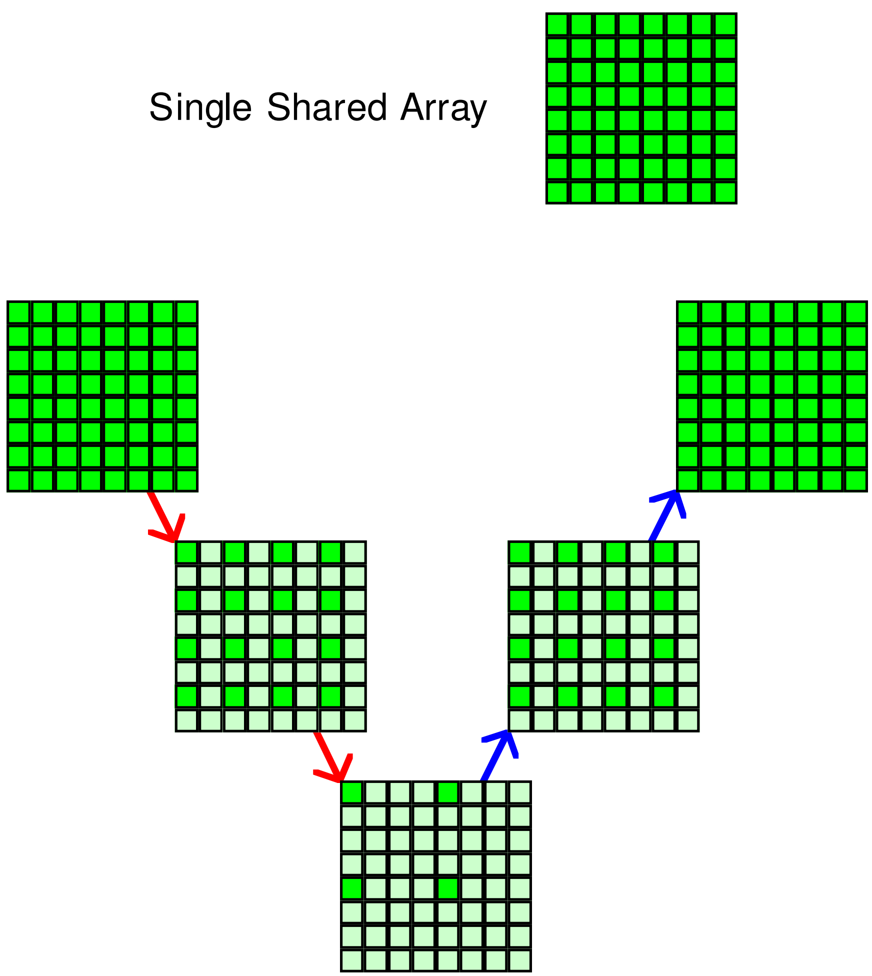
- **Nearest-neighbor relaxation**
  - `BLOCK` in dimension(s) with most parallelism
  - `CYCLIC(K)` may help with load balancing
- **Dense linear algebra**
  - `CYCLIC(K)` in one or two dimensions
- **Indirection array data structures**
  - "No silver bullet"
  - `BLOCK`, with careful numbering of data elements?

Multigrid: A Complicated Case for DISTRIBUTE

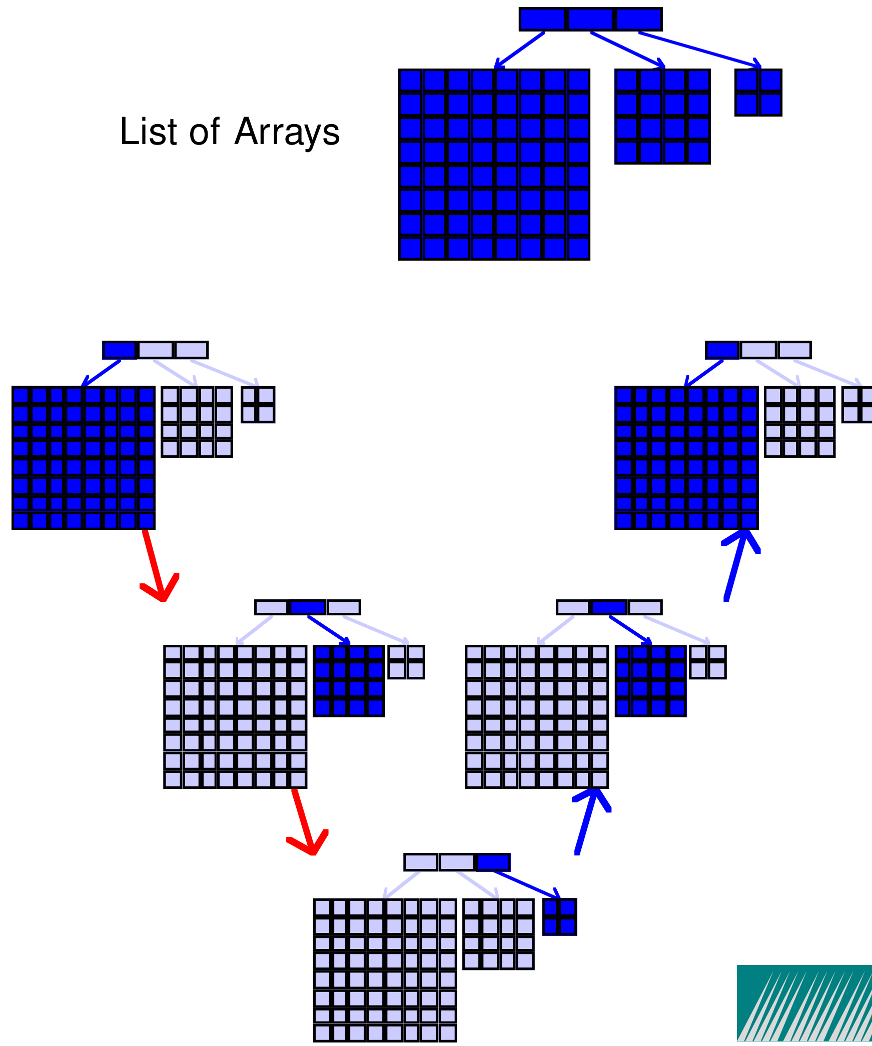


Possible Data Structures for Multigrid

Single Shared Array



List of Arrays

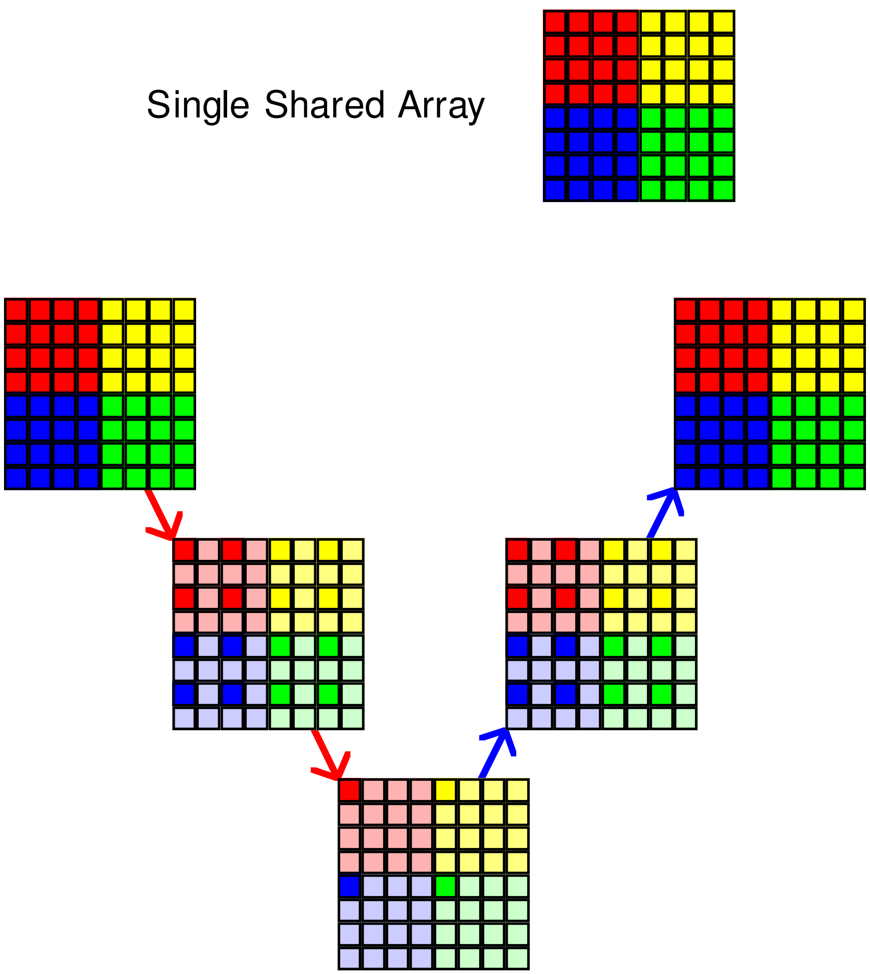


Possible DISTRIBUTE Patterns for Multigrid

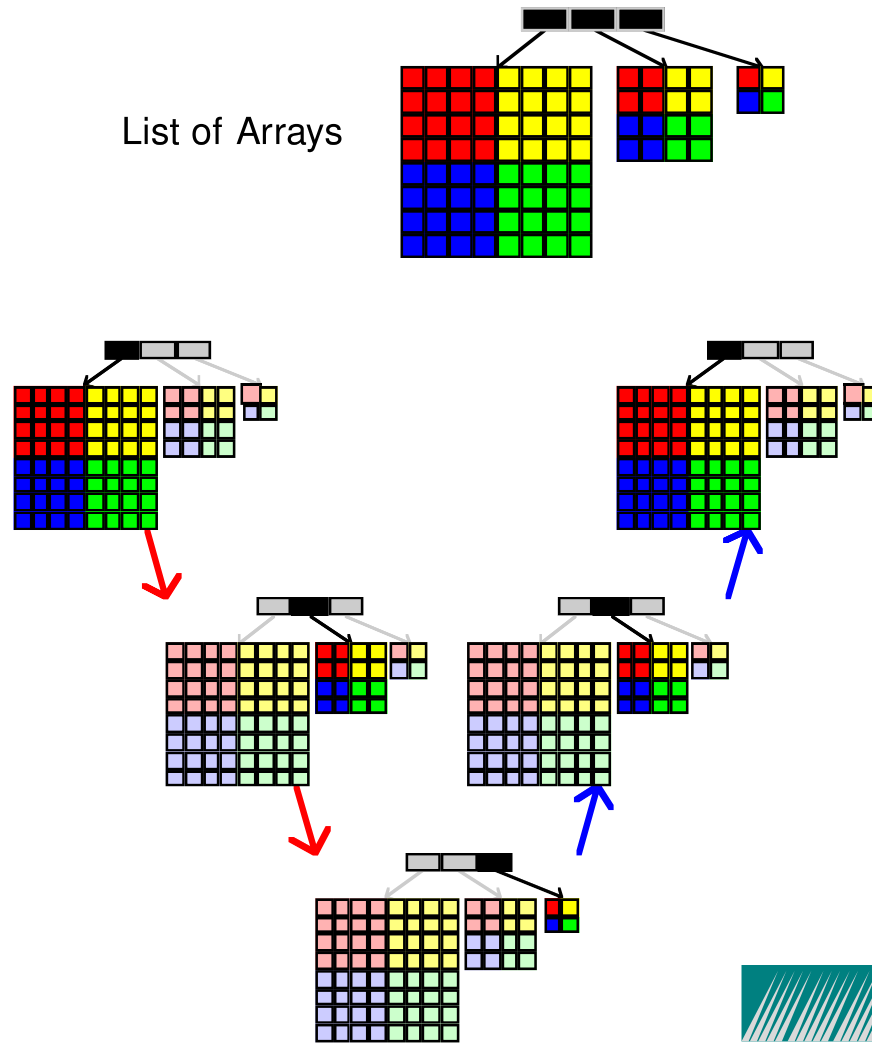
- **Single shared array for all levels**
  - Smoothing: All `CSHIFT` operations
  - Prolongation and Restriction: All `CSHIFT` operations
  - Conclusion: Use some form of `BLOCK`
    - If start-up cost is high: 1-D `BLOCK` (in longest dimension)
    - Otherwise: 2-D `BLOCK` may be better
- **List of arrays, one per level**
  - Smoothing: All `CSHIFT` operations
  - Prolongation and Restriction: copies between different-sized arrays
  - Conclusion: Use some form of `BLOCK` on each level
    - May want to change for coarser grid levels

Possible DISTRIBUTE Patterns for Multigrid (cont.)

Single Shared Array



List of Arrays

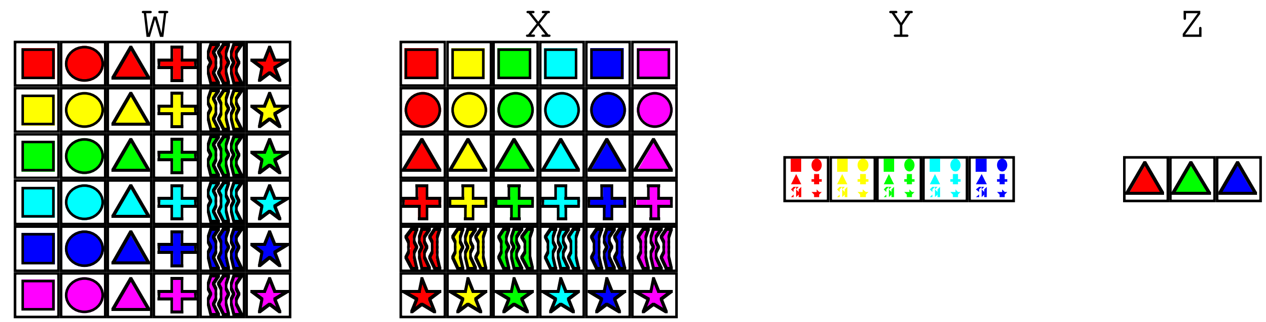


The ALIGN Directive

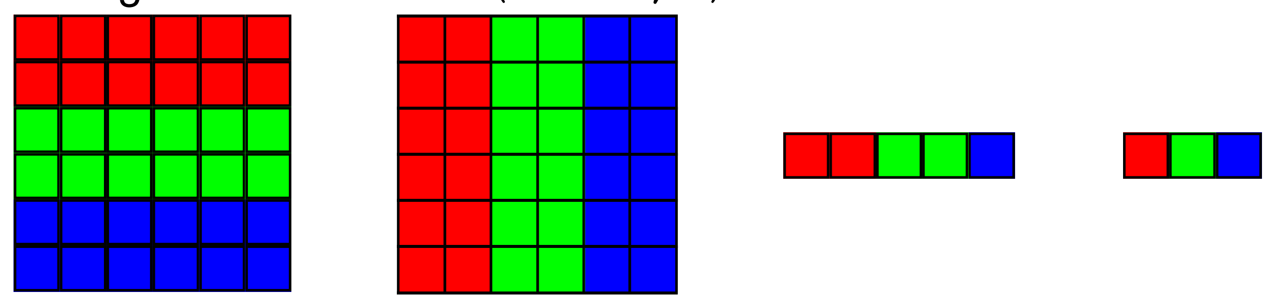
- **Syntax:**
  - `!HPF$ ALIGN` *array* `(`*source-list*`) WITH` *target* `(`*subscript-list* `)`
  - `!HPF$ ALIGN (`*source-list*`) WITH` *target* `(`*subscript-list*`) ::` *array-list*
- **Semantics:**
  - Creates a relationship between *array* and *target* so that for all values of the *source-list* variables, *array(source-list)* and *target(subscript-list)* are stored on the same processor
  - Only *target* can be distributed explicitly
- **Options for *subscript-list***
  - Linear function of one *source-list* variable
  - Triplet notation
    - Must match "`:`" in source-list
    - Element-wise matching as in array assignment
  - `*` (Replication)

Examples of ALIGN

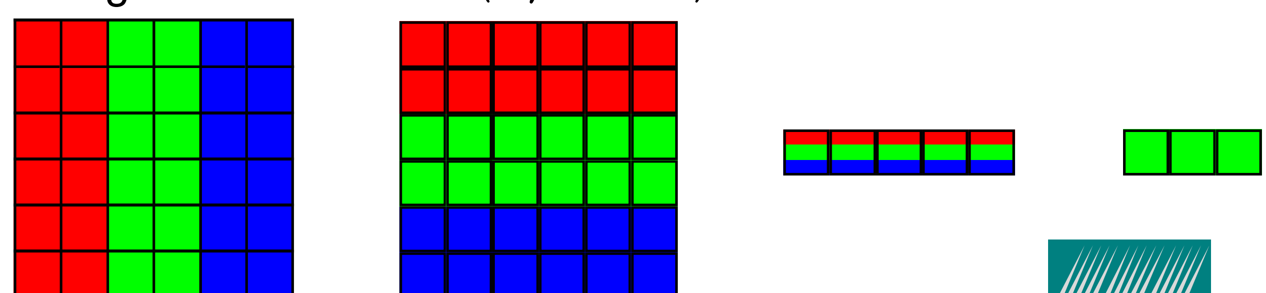
```
REAL W(6,6),X(6,6),Y(5),Z(3)
!HPF$ALIGN X(I,J)WITH W(J,I)
!HPF$ALIGN Y(K)WITH W(K,*)
!HPF$ALIGN Z(L)WITH X(3,2*L-1)
```



Using DISTRIBUTE W(BLOCK,*)



Using DISTRIBUTE W(*,BLOCK)



ALIGN and Communication

- **Communication (data movement) happens when two data items on different processors must be brought together**
- **`ALIGN` relates array elements, ensuring they are mapped together**
  - `!HPF$ ALIGN a(i) WITH b(i+1)`
    - No communication for `a(10) = b(11)`
    - No information about `a(10) = b(10)`
- **You can also think of this as modifying `DISTRIBUTE`**
  - Substitute the `ALIGN` subscripts before doing the communication analysis

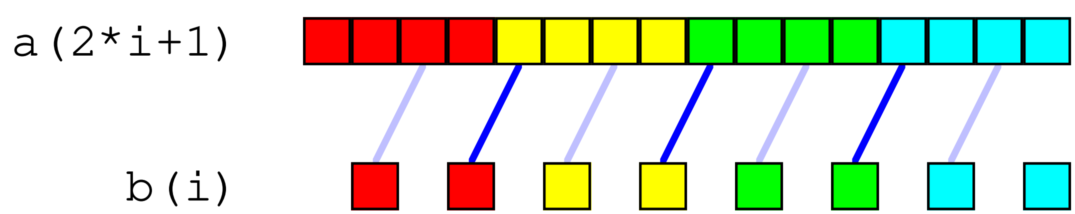
Quantifying Communication: ALIGN

```
REAL a(16), b(8)
!HPF$ ALIGN b(i) WITH a(2*i)
!HPF$ DISTRIBUTE a(BLOCK)
FORALL (j=1:7) b(i) = a(2*i+1)
```

$\Rightarrow$ `FORALL (j=1:16) b´(2*j) = a(2*j+1)`

$\Rightarrow$ `FORALL (j´=2:32:2) b´(j´) = a(j´+1)`

$\Rightarrow$ Communicate at block boundaries



Why Use ALIGN?

- **"Copy" distributions**
  - `DISTRIBUTE` one array as the "master" for data layout
  - `ALIGN` other arrays to it
  - Just modify one line to change all the distributions
    - This will be common when porting codes!
- **Off-by-one problems**
  - Sometimes boundaries get in the way of `DISTRIBUTE`
- **Differing array sizes**
  - Will the compiler handle this better?

```
REAL a(16), b(8)
!HPF$ ALIGN b(i) WITH a(2*i)
!HPF$ DISTRIBUTE a(BLOCK)
```

  - Or this?

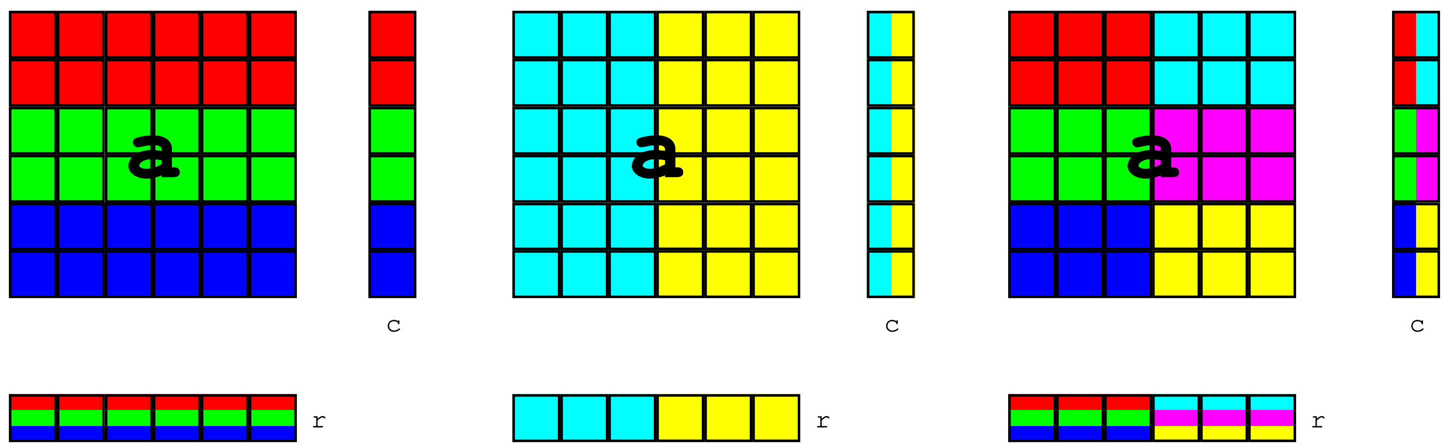
```
REAL a(16), b(8)
!HPF$ DISTRIBUTE a(BLOCK), b(BLOCK)
```

Typical Uses of ALIGN

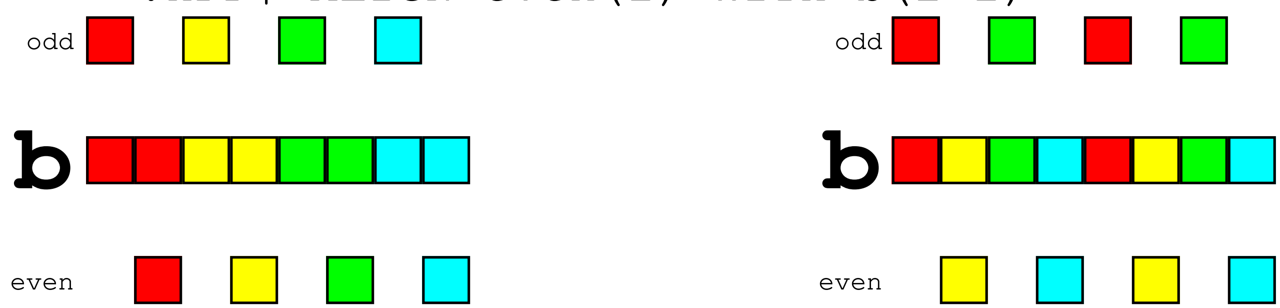
- **Align major arays together**
  - If all arrays represent quantities in the same physical space, with the same discretization, `ALIGN` them
  - If arrays are accessed differently (e.g. to represent different physics), `ALIGN` based on those access groups
- **More complex cases:**
  - Multigrid with lists of arrays: Use `ALIGN` with stride 2 between levels
  - Other cases: Often handled by choosing one “main” array and using a variety of `ALIGN` directives
  - Finding good `ALIGN` and `DISTRIBUTE` patterns for completely irregular codes is still a research problem

Advanced ALIGN Uses

```
!HPF$ ALIGN c(i) WITH a(i,*)
!HPF$ ALIGN r(i) WITH a(*,i)
```




```
!HPF$ ALIGN  odd(i) WITH b(2*i-1)
!HPF$ ALIGN even(i) WITH b(2*i)
```



Dynamic Data Mapping

- **One data mapping is not always appropriate for an entire program**
  - Different behavior in different phases
  - “First sweep in the X dimension, then sweep in the Y dimension”
  - `ALLOCATABLE` arrays can change size
- **Therefore, HPF needs executable `DISTRIBUTE` and `ALIGN`**
  - Called `REALIGN` and `REDISTRIBUTE`
  - `DYNAMIC` attribute (compare to `ALLOCATABLE`)

The DYNAMIC Directive

- **Syntax:**
  - `!HPF$ DYNAMIC` *entity-decl-list*
  - `!HPF$ DYNAMIC ::` *entity-decl-list*
- **Semantics:**
  - Any array in *entity-decl-list* can be used in a `REALIGN` or `REDISTRIBUTE` directive
  - `DYNAMIC` appears only in the declarations section
  - `DYNAMIC` arrays can also appear in `ALIGN` or `DISTRIBUTE` to get initial distributions)

The REALIGN Directive

- **Syntax:**
  - `!HPF$ REALIGN` *array* `(`*source-list*`)` `WITH` *target* `(`*subscript-list*`)`
  - `!HPF$ REALIGN (`*source-list*`)` `WITH` *target* `(`*subscript-list*`) ::` *array-list*
- **Semantics:**
  - Creates a **dynamic** relationship between *array* and *target* so that for all values of the *source-list* variables, *array(source-list)* and *target(subscript-list)* are stored on the same processor
  - *Array* must have the `DYNAMIC` attribute
  - Ends any previous `ALIGN` or `DISTRIBUTE` for *array*
  - Communicates data already in *array* to its new home
  - Lasts until another `REALIGN` or `REDISTRIBUTE` for the same *array*
    - I.e., Not static scope

CRPC

The REDISTRIBUTE Directive

- **Syntax:**
  - `!HPF$ REDISTRIBUTE` *array* (*dist-format-list*) [`ONTO` *processors*]
  - `!HPF$ REDISTRIBUTE` (*dist-format-list*) [`ONTO` *processors*] `::` *array-list*
- **Semantics:**
  - **Dynamically** divides each dimension of *array* according to the corresponding pattern in *dist-format-list*
  - Also changes the mappings of anything already aligned to *array*
    - I.e., *Array* was previously the *target* in an `ALIGN` or `REALIGN`
  - *Array* must have the `DYNAMIC` attribute
  - Communicates data to its new processor home
  - Lasts until another `REALIGN` or `REDISTRIBUTE` for the same *array*

CRPC

Typical Use of Dynamic Data Mapping

Picking mappings based on input data

```
!HPF$ DYNAMIC u
!HPF$ ALIGN WITH x(:,:) :: y(:,:), z(:,:)
IF (n1>n2) THEN
   !HPF$ REDISTRIBUTE x(BLOCK,*)
ELSE
   !HPF$ REDISTRIBUTE x(*,BLOCK)
END IF
! Note: Compiler doesn't know precise data
!       mappings of any array at this point
```

CRPC

Dynamic Data Mapping in Other Applications

- **FFT, ADI, and other directional sweeps**
  - (RE)`DISTRIBUTE BLOCK` in parallel dimension for first sweep
  - Perform sweep
  - `REDISTRIBUTE BLOCK` in the next dimension
  - Perform sweep, …
- **Cyclic reduction and other recursive doubling methods**
  - (Only if number of processors *P* is a power of 2)
  - (RE)`DISTRIBUTE BLOCK`
  - Reduce by step 1, then 2, … to *P*/2
  - `REDISTRIBUTE CYCLIC`
  - Reduce by *P*, 2**P*, …

Data Mapping in Subroutine Calls

- **Some subroutines require data to use a specific mapping, so actual arguments must be remapped**
- **Some subroutines can use any mapping, so actual arguments should be passed in place**
- **Sometimes the programmer knows the incoming data mappings, sometimes not**
- ***HPF has options to say all of this!***
- **Any remappings are undone on procedure return**

Mapping Options for Dummy Arguments

- **`DISTRIBUTE`**
  - “`*`” instead of *dist-format-list* or `ONTO` clause indicates any incoming distribution is acceptable (i.e. leave data in place)
  - “`*`” before *dist-format-list* or ONTO indicates data should stay in place, and guarantees that the actual has this distribution
  - If you are passing array subsections, the possible distributions are limited
- **`ALIGN`**
  - “`*`” instead of or before *target* with similar meanings to above
- **`INHERIT`**
  - A new attribute, allowing references back to the index space of the actual (i.e. the whole array, rather than the subset passed)

CRPC

Typical Mappings of Dummy Arguments

Option 1: Pass entire arrays every time

```
SUBROUTINE smooth(r,u,f,nx,ny)
REAL r(nx,ny), u(nx,ny), f(nx,ny)
!HPF$ DISTRIBUTE u*(BLOCK,BLOCK)
!HPF$ ALIGN WITH *u(i,j) :: r(i,j), f(i,j)
```

Option 2: Pass array subsections in place

```
SUBROUTINE smooth(r,u,f)
REAL r(:,:), u(:,:), f(:,:)
!HPF$ INHERIT r, u, f
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) :: r, u, f
```

Dummy Arguments in Other Applications

- **Libraries**
  - Prescriptive mapping ("remap actual") if a specific mapping is needed
  - Transcriptive mapping ("take anything") for embarrassingly parallel
  - Descriptive mapping ("this is coming") for internal routines, probably after checking distribution of (transcriptively mapped) user arguments
- **Non-reusable modules**
  - Descriptive mapping probably fastest
  - Prescriptive mapping probably safer

Other Mapping Features

- **`PROCESSORS`**
  - Define size and shape of processors array
  - Only guaranteed when size of processors array equals `NUMBER_OF_PROCESSORS()`
  - `!HPF$ PROCESSORS` *name*`(`*shape-spec-list*`)`
- **`TEMPLATE`**
  - Defines an index domain for alignment and distribution, but no memory
  - `!HPF$ TEMPLATE` *name*`(`*shape-spec-list*`)`
- **`ALLOCATABLE` Arrays and `POINTER` Targets**
  - `ALIGN` and `DISTRIBUTE` take effect (and have expressions evaluated) when the array is allocated
  - If used as the target of `REALIGN`, then the array must be allocated when `REALIGN` takes effect.

Storage and Sequence Association

- **Generally, if an array can be storage or sequence associated then it may not be explicitly mapped**
- **A variable is sequential if it is**
  - Part of a sequential `COMMON`
  - `EQUIVALENCE`d to something
  - An assumed-size array
  - Part of a Fortran 90 derived type with the `SEQUENCE` attribute, or
  - Declared by `!HPF$ SEQUENCE`
- **`COMMON` blocks may be nonsequential if they are consistently declared**
- **If an array is reshaped at procedure call, the actual and dummy must be sequential**
- **Nonsequential arrays can always be explicitly mapped**
- **Sequential arrays can be explicitly mapped if they exactly cover their aggregate variable group**

Hints for Using Data Mapping

- **`ALIGN` data based on physical domains**
  - Arrays with the same domain should be aligned
  - Arrays not physically connected should not
- **`DISTRIBUTE` based on parallelism**
  - Optimal performance comes from parallel operations on a distributed axis
- **Pick distribution pattern based on communications**
  - `BLOCK` generally good for local stencils and fully-filled arrays
  - `CYCLIC` and `CYCLIC(K)` generally good for load balancing and triangular loops
  - Conflicts require compromises, remapping, complex compilers, or new algorithms
- **`REALIGN` with care; `REDISTRIBUTE` with extreme care**
  - Computation performed must outweigh communication for the remapping