An HPF Compiler for the IBM SP

The SPMDizer of the IBM product HPF compiler

HPF -- *H*igh *P*erformance *F*ortran

- HPF was a parallel Fortran Dialect whose development was spearheaded by Mehrotra (NASA Langley), Zima and Chapman (Vienna), Kennedy (Rice) and Fox (Syracuse, now at IU)
- Was the hot language in the early/mid 1990s
- DEC, PGI and IBM were major commercial implementations

Motivation

- Distributed memory was *the* programming model of the 90s
  - attack of the killer micros
  - scaling to large number of processors
- Shared memory automatic parallelization was recognized as very hard
- Data distribution added to this complexity

Make the programmer handle distribution

- Let the programmer handle the distribution of data
  - templates
  - logical processor grids
  - distributions of arrays onto templates and processor grids
- Remainder of the program can be written in an sequential Fortran 90 style

Let the compiler handle parallelization and communication

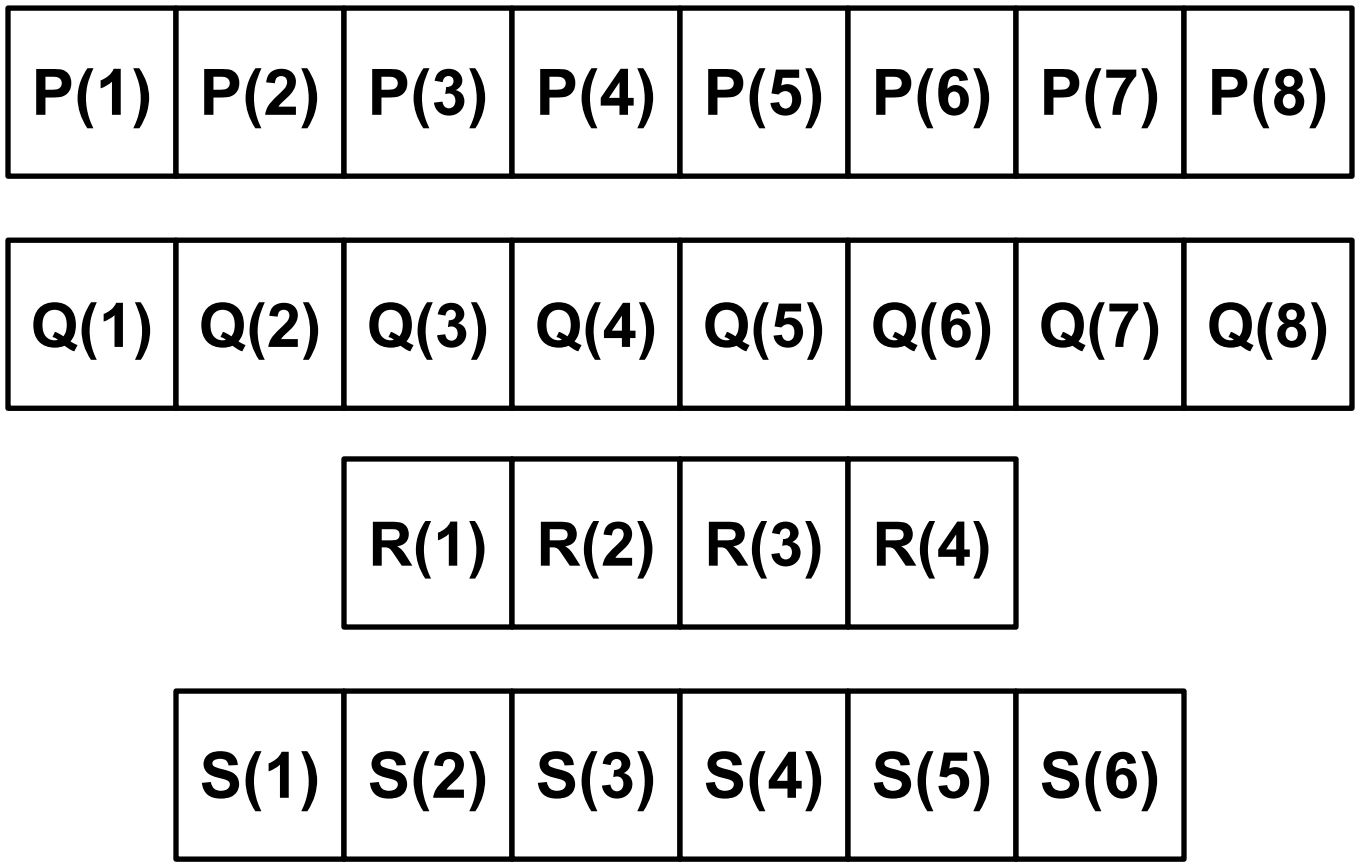
- Driven by data distribution
- Computation scheduled by having processors compute values of data that is distributed onto them (i.e. that they *own)*
- Communication is determined by what needs to be communicated to computed owned values
- Loops needing no communication inside the loop are parallel
- ***Ultimately, just as hard as shared memory auto-parallelization***

Processor grids

P(0,0)	P(0,1)	P(0,2)	P(0,3)
P(1,0)	P(1,1)	P(1,2)	P(1,3)
P(2,0)	P(2,1)	P(2,2)	P(2,3)
P(3,0)	P(3,1)	P(3,2)	P(3,3)

!HPF$ PROCESSOR P(4,4)

Processor Grids




```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
!HPF$ PROCESSORS Q(8)
!HPF$ PROCESSORS :: R(4), S(6)
```

Distributing data onto processor grids

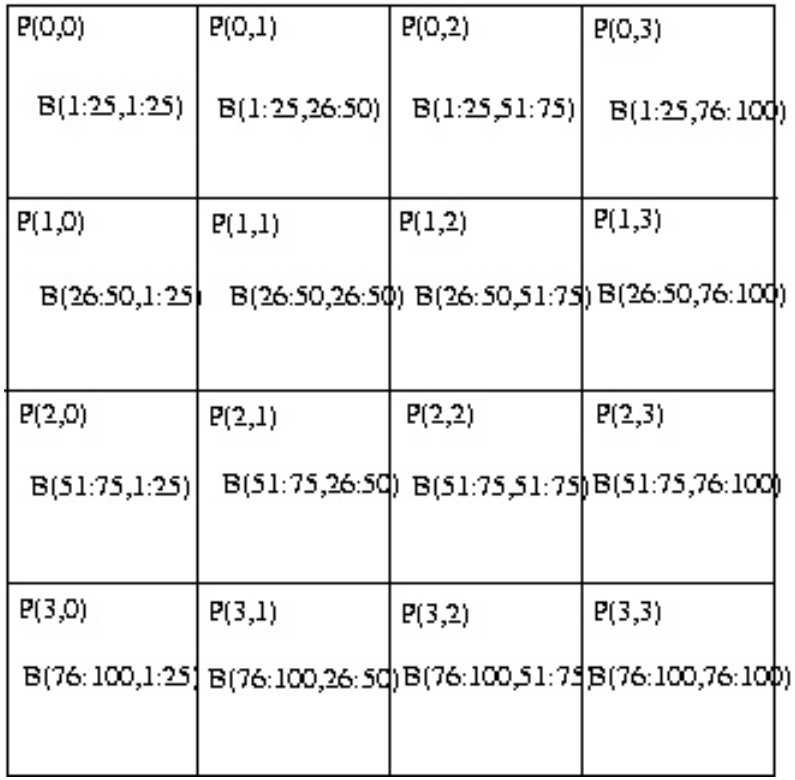
B(1,1)	B(1,2)	. . .	B(1,100)
B(2,1)	B(2,2)	...	(2,100)
...	...	...	...
B(100,1)	B(100,2)	...	B(100,100)

P(0,0)	P(0,1)	P(0,2)	P(0,3)
P(1,0)	P(1,1)	P(1,2)	P(1,3)
P(2,0)	P(2,1)	P(2,2)	P(2,3)
P(3,0)	P(3,1)	P(3,2)	P(3,3)

integer B(100,100)

!HPF$ PROCESSOR P(4,4)

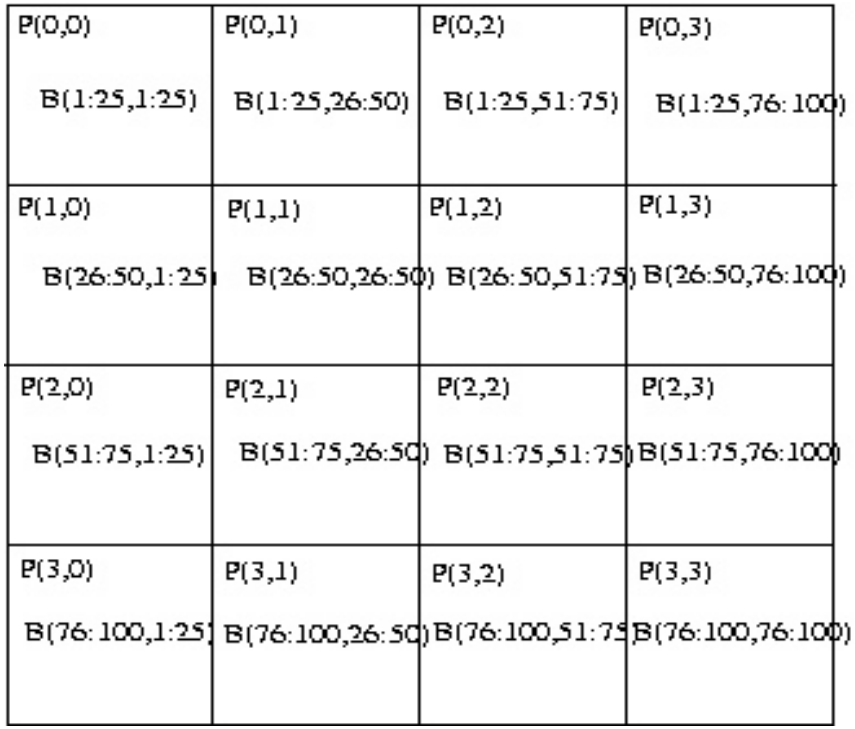
BLOCK DISTRIBUTED



integer B(100,100)

!HPF$ DISTRIBUTED B(BLOCK,BLOCK)

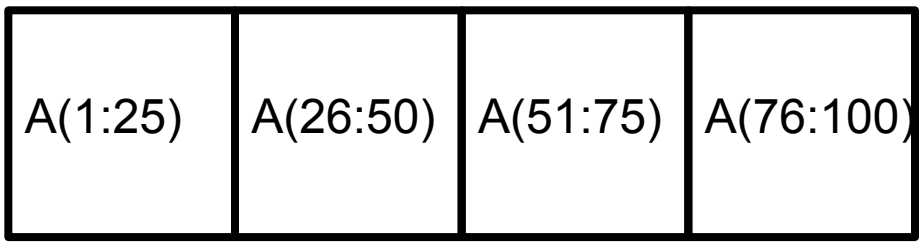
BLOCK DISTRIBUTED




```
      integer B(100,100)
!HPF$ PROCESSORS P(4,4)
!HPF$ DISTRIBUTE B(BLOCK,BLOCK) ONTO P
```

Default processor grids

- If not processor grid is specified, then a linear grid with however many processors there are at runtime is specified (for a one-D array). Finding the actual processor grid size a challenge.



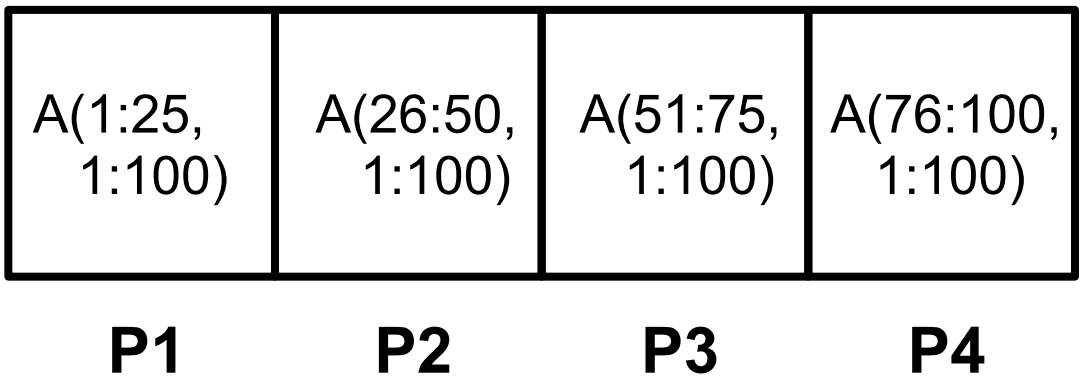
integer A(100)

!HPF$ DISTRIBUTE A(BLOCK)

Distributions

- Each dimension can be distributed separately
- BLOCK, CYCLIC, BLOCK(k), REPLICATED ("*") all supported

Rank of processor grid smaller than the rank of array




```
      FLOAT A(100,100)
!HPF$ Processor P(4)
!DISTRIBUTE A(BLOCK,*) ONTO P
```

Rank of processor grid larger than rank of array

A(1:25)	A(26:50)	A(51:75)	A(76:100)
A(1:25)	A(26:50)	A(51:75)	A(76:100)
A(1:25)	A(26:50)	A(51:75)	A(76:100)
A(1:25)	A(26:50)	A(51:75)	A(76:100)

```
FLOAT A(100)
!HPF$ Processor P(4,4)
!DISTRIBUTE A(BLOCK,*) ONTO P
```

ALIGNMENT

- Sometimes we want to distribute one array and *align* other arrays with it.

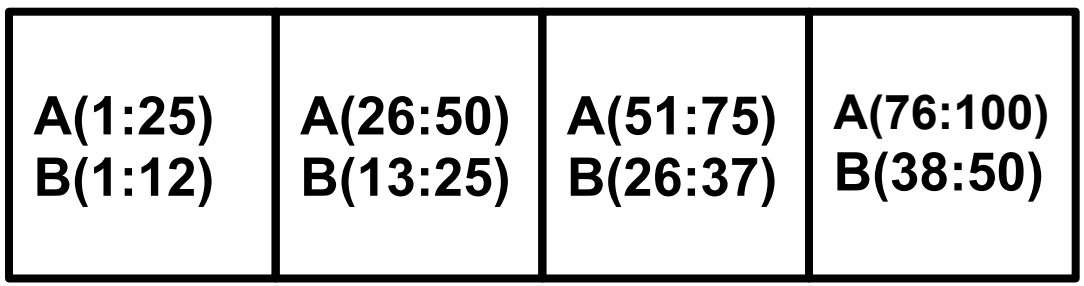
```
do i = 1, 50
   a(2*i) = a(2*i) + b(i)
end do
```

```
!HPF$ PROCESSOR P(4)
!HPF$ DISTRIBUTE B(BLOCK) ONTO P(4)
!HPF$ ALIGN A(2*I) WITH B(I)
```

ALIGNMENT

```
do i = 1, 50
   a(2*i) = a(2*i) + b(i)
end do
```

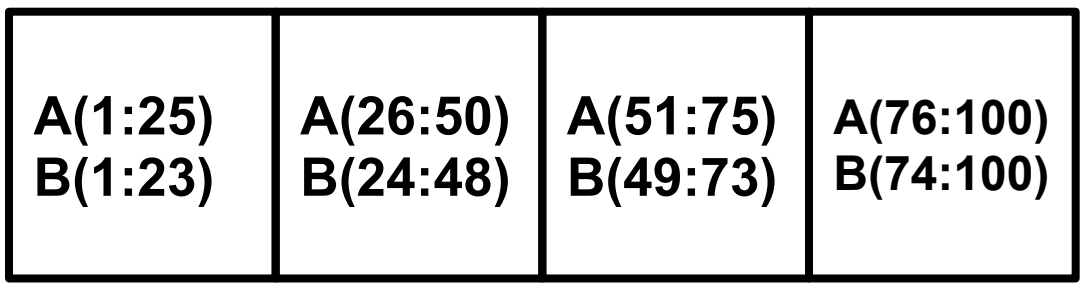
```
!HPF$ PROCESSOR P(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P(4)
!HPF$ ALIGN B(I) WITH A(2*I)
```



ALIGNMENT

```
do i = 1, 50
   a(i) = a(i) + b(i-2)
end do
```

```
!HPF$ PROCESSOR P(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P(4)
!HPF$ ALIGN B(I) WITH A(i+2)
```



Templates

- Templates can be thought of as virtual targets to align with, i.e. an array of nothing
- Templates, in turn, are distributed onto processor grids or aligned with other templates

TEMPLATE example

- Sometimes we want to distribute one array and *align* other arrays with it using a template

```
do i = 1, 50
   a(i) = a(i) + b(i-2)
end do
```

```
!HPF$ PROCESSOR P(4)
!HPF$ TEMPLATE T(52)
!HPF$ ALIGN A(I) WITH  T(I)
!HPF$ ALIGN B(I) WITH T(I+2)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P
```

TEMPLATE transpose example

- Sometimes we want to distribute one array and *align* other arrays with it using a template.

```
do i = 1, 50
  do j = 1,50
    a(i,j) = a(i,j) * b(j,i)
  end do
end do
```

```
!HPF$ PROCESSOR P(4,4)
!HPF$ TEMPLATE T(50,50)
!HPF$ ALIGN A(I,J) WITH  T(I,J)
!HPF$ ALIGN B(J,I) WITH T(J,I)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
```

TEMPLATE transpose example

A numbering of blocks of an array block distributed.

This notation is used only to make it easier to type in the block names without worrying about specific elements of an array α in the block.

α1	α2	α3	α4
α5	α6	α7	α8
α9	α10	α11	α12
α13	α14	α15	α16

TEMPLATE transpose example

```
do i = 1, 50
  do j = 1,50
    a(i,j) = a(i,j) * b(j,i)
  end do
end do
```

a1	a2	a3	a4
a5	a6	a7	a8
a9	a10	a11	a12
a13	a14	a15	a16

```
!HPF$ PROCESSOR P(4,4)
!HPF$ TEMPLATE T(50,50)
!HPF$ ALIGN A(I,J) WITH  T(I,J)
!HPF$ ALIGN B(J,I) WITH T(J,I)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
```

TEMPLATE transpose example

```
do i = 1, 50
  do j = 1,50
    a(i,j) = a(i,j) * b(j,i)
  end do
end do
```

b1	b5	b9	b13
b2	b6	b10	b14
b3	b7	b11	b15
b4	b8	b12	b16

```
!HPF$ PROCESSOR P(4,4)
!HPF$ TEMPLATE T(50,50)
!HPF$ ALIGN A(I,J) WITH  T(I,J)
!HPF$ ALIGN B(J,I) WITH T(J,I)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
```

TEMPLATE transpose example

Template allows all arrays aligned with it to be distributed together.

a1 b1	a2 b5	a3 b9	a4 b13
a5 b2	a6 b6	a7 b10	a8 b11
a9 b3	a10 b7	a11 b11	a12 b15
a13 b4	a14 b8	a15 b12	a16 b16

```
!HPF$ PROCESSOR P(4,4)
!HPF$ TEMPLATE T(50,50)
!HPF$ ALIGN A(I,J) WITH  T(I,J)
!HPF$ ALIGN B(J,I) WITH T(J,I)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
```

Distribution functions

- Let α be an array, τ a template, π a processor grid, and various f_i be functions mapping elements of α onto whatever α is distributed onto/aligned with.

Distribution functions

- Let α, β be an array, τ a template, π a processor grid, and various f_i be functions mapping elements of α onto whatever α is distributed onto/aligned with.

!HPF$ ALIGN $\alpha(i)$ WITH $\beta(i+1)$ // $i' = f_1(i+1)$

!HPF$ ALIGN $\beta(i)$ WITH $\tau(2*i)$ // $i' = f_2(2*i)$

!HPF$ DISTRIBUTE τ(BLOCK) // f_3

$f_3(f_2(f_1(i)))$ is the distribution function

Distribution functions

!HPF$ ALIGN $\alpha(i)$ WITH $\beta(i+1)$ // f_1

$\alpha(i)$ aligned with element $\beta(i+1)$

!HPF$ ALIGN $\beta(i)$ WITH $\tau(2*i)$ // f_2

$\alpha(i)$ aligned with element $\tau(2*(i+1))$

Given by $f_2(f_1(i))$

!HPF$ DISTRIBUTE τ(BLOCK) // f_3

Elements of τ are block distributed onto the default processor grid using the appropriate formula f_3

$f_3(f_2(f_1(i)))$ is the final distribution function

Architecture of the Compiler

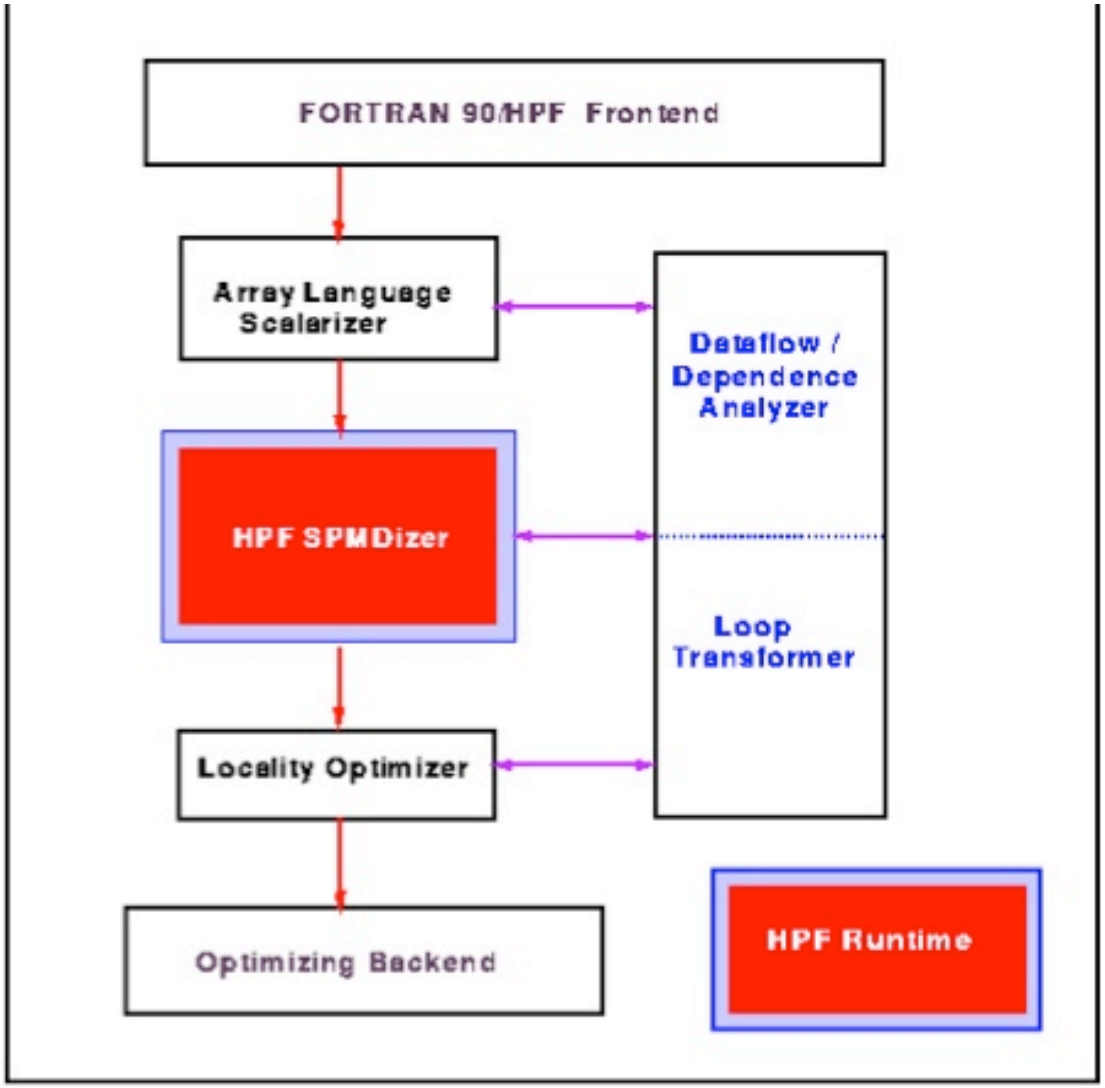


Figure 1: Architecture of the pHPF compiler

Architecture of the SPMDizer

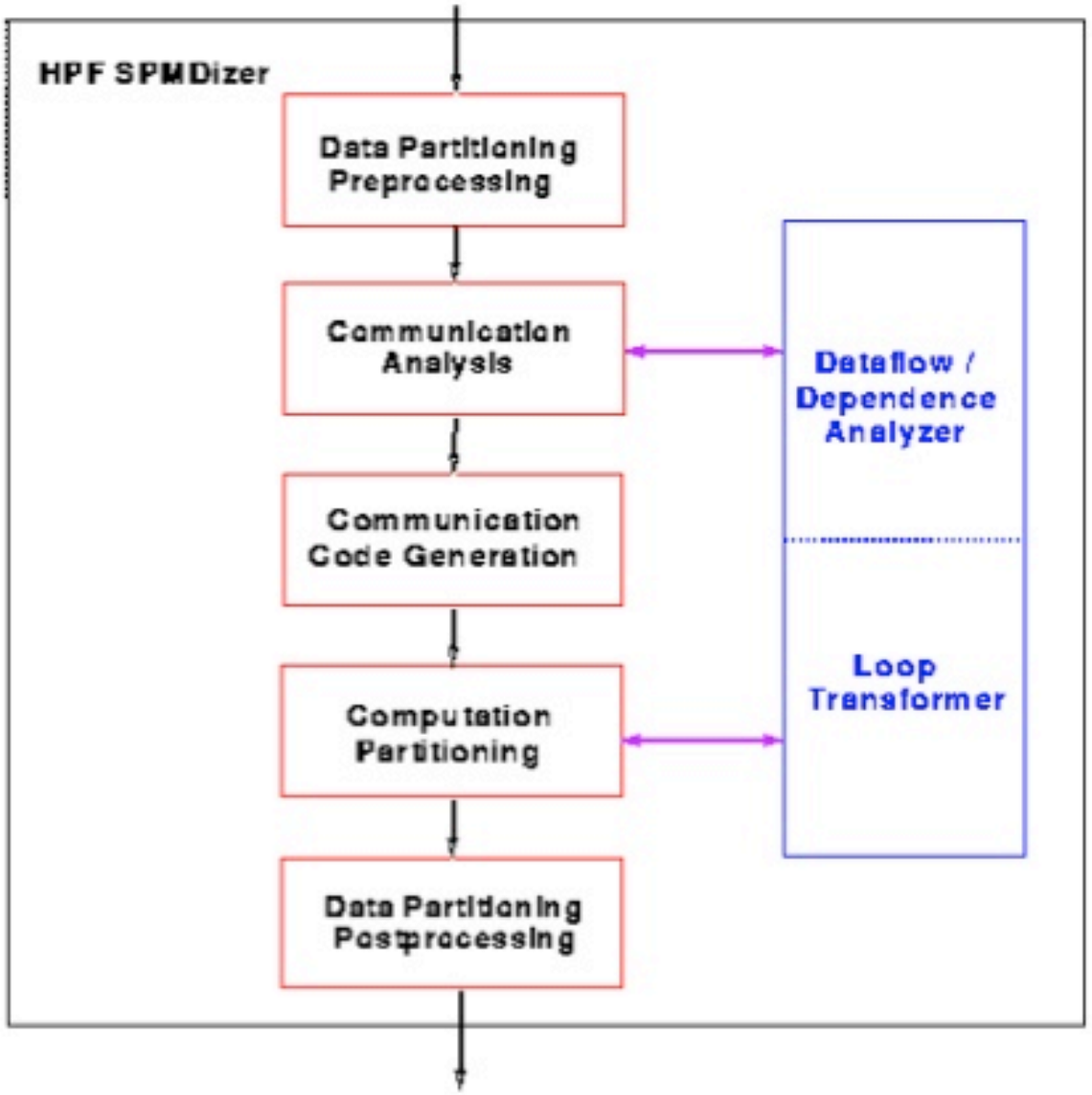


Figure 2: Architecture of the pHPF SPMDizer

Determining distributions

- The default is to let the programmer do this.
- Research by Uli Kremer (at Rice, now Maryland) and others aimed to do this automatically
  - Search a space of possible optimizations to determine which distribution incurred the least expense for data movement
  - Problem made harder by allowing redistribution

The owner's computes rule

- Developed by Pingali (UT Austin) and Anne Rogers (U. Chicago)
- Processor owning the target of an operation performs the operations
- a(i) = a(i) + b(i-2) - processor owning a(i) performs the computation
- Compiler must determine
  - what elements of *a* it owns
  - what elements of *a* and *b* it needs to compute its *a* values
  - generate communication to acquire those *a* and *b* elements and send elements to other processors
  - only execute the iterations of the loop nest that cause each processor to compute its owned elements

One way to determine communication

- Look at the pattern of *lhs* references and *rhs* references with respect to the distribution.
- The *distribution function* is useful for this
- Some amount of pattern matching also used.

Consider the program . . .

```
// assume 4 processors
float a(16), b(16)
!HPF$ ALIGN a(i) with b(i)
!HPF$ DISTRIBUTE \
          a(BLOCKED)
do i = 1, n
   a(i) = . . .
   do j = 1, n
      b(j) = a(j) . . .
   end
end
```

- P1 will compute a(1:4) and b(1:4)
- P1 only needs elements a(1:4)
- No communication needed

Consider the program . . .

```
// assume 4 processors
float a(16), b(16)
!HPF$ DISTRIBUTE \
          a(BLOCKED)
!HPF$ DISTRIBUTE \
          b(*)
do i = 1, n
   a(i) = . . .
   do j = 1, n
      b(j) = a(j) . . .
   end
end
```

- b replicated over the processors
- Every processor has a copy of the b array
- P1 will compute a(1:4) and b(1:16)
- P1 needs elements a(1:16)
- P1 needs to receive elements a(1:16)
- P1 needs to send elements a(1:4) to all processors

Consider the program . . .

```
// assume 4 processors
float a(16), b(16)
!HPF$ DISTRIBUTE \
          a(BLOCKED)
!HPF$ DISTRIBUTE \
          b(*)
do i = 1, n
  a(i) = . . .
  do j = 1, n
    b(j) = a(j) . . .
  end
end
```

- P1 needs to receive elements a(1:16)
- P1 needs to send elements a(1:4) to all processors
- All-to-all communication - allows a collective operation to be picked.

Consider the program . . .

```
// assume 4 processors
float a(16), b(16)
!HPF$ DISTRIBUTE \
          a(BLOCKED)
!HPF$ ALIGN a(i) WITH b(i)

do i = 1, n
   a(i) = a(i+1)
end
```

- P2 will compute a(5:8)
- P1 needs elements a(6:9)
- P1 needs to receive element a(9)
- P1 needs to receive a(9) and send a(5) to its lower neighbor
- Point-to-point operation -- send/ receive communication

Send/recv communication

```
// assume 4 processors
float a(16), b(16)
!HPF$ DISTRIBUTE \
          a(BLOCKED)
!HPF$ DISTRIBUTE B(*)
do i = 1, n
   a(i) = . . .
   do j = 1, n
      b(j) = a(j) . . .
   end
end
```

- P1 needs to receive elements a(1:16)
- P1 needs to send elements a(1:4) to all processors
- All-to-all communication - allows a collective operation to be picked.

Look for the following patterns

- one-to-one (or one-to-a-few): send/recv message passing communication
- one-to-many: bcast or scatter, depending on whether all elements sent to all, or some sent to some
- many-to-one: gather or reduce operation
- many-to-many: allgather, allscatter or alltoall communication

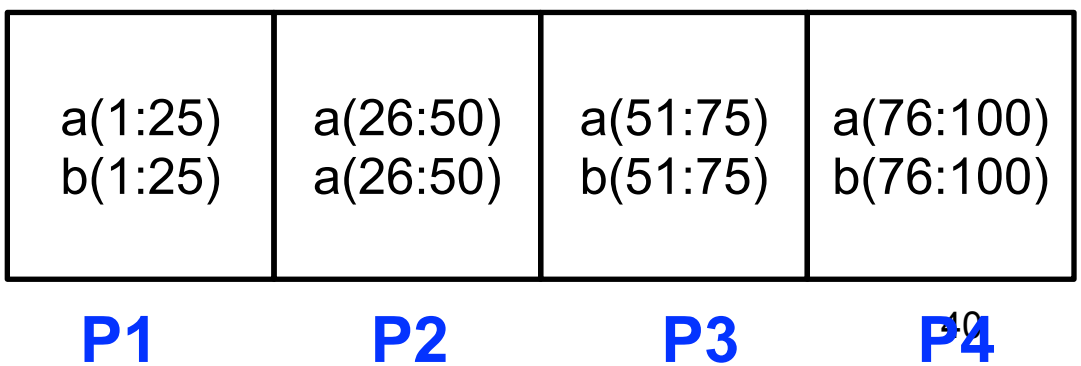
Scheduling strategy

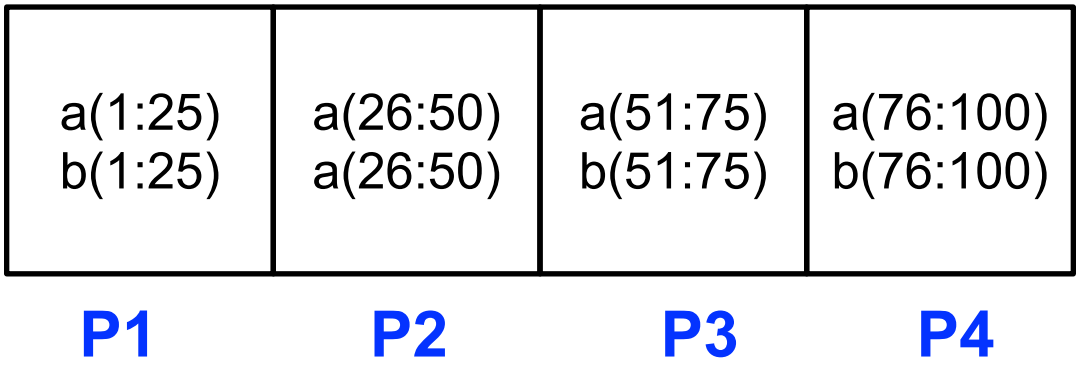
- Figure out the iterations needed by each Pi for each *lhs* reference using owner's computes
- Iteration space for Pi is
  - union of all of these (need to execute all required iterations)
  - intersected with the original loop bounds (can't execute more iterations that originally desired)
- Put guards around references to ensure it executes only for its iterations

Scheduling computation

```
      float a(100), b(100)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE a(BLOCK) \
      on P(4)

do i = 1, 100
   a(i) = . . .
   if (i .gt. 0) then b(i-1) = . . .
end do
```

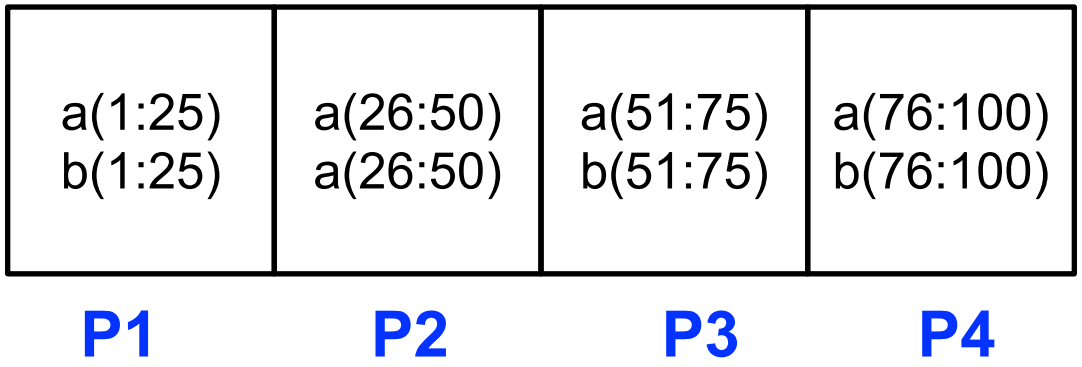





```
      float a(100), b(100)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE a(BLOCK) \
      on P(4)

do i = 1, 100
   // P1 executes i=1..25; P2 i=26..50; P3 i=51..75; P3 i=76..100
   a(i) = . . .
   // P1 executes i=2..26; P2 i=27..51; P3 i=52..76; P3 i=77..101
   if (i .gt. 0) then b(i-1) = . . .
end do
```

Figure out the iterations needed by each Pi for each *lhs* reference using owner's computes (intersection of written and owned elements)

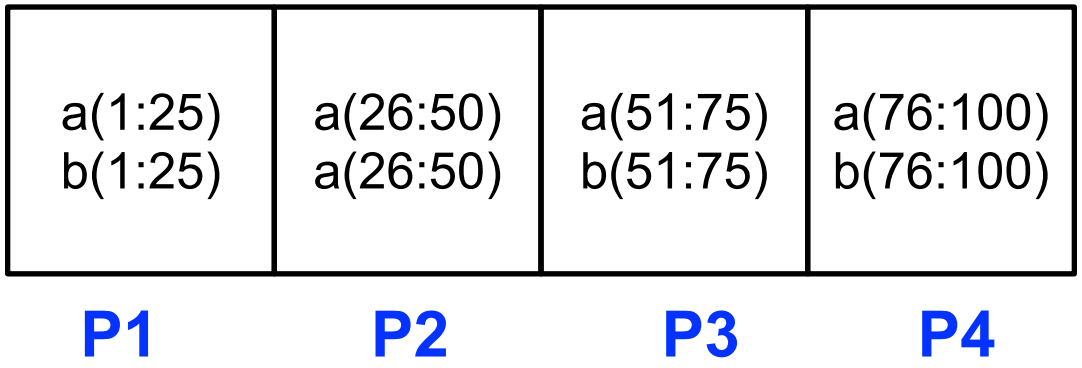


Iteration space for Pi is union of iterations spaces for all references

```
        float a(100), b(100)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE a(BLOCK) \
        on P(4)
lb = min(lb(aRef),lb(bRef) // min(1,2) for P1, min(51,52) for P3
ub = max(lb(aRef),lb(bRef) // max(25,26) for P1, max(75,76) for P3
do i = 1, 100
   // P1 executes i=1..25; P2 i=26..50; P3 i=51..75; P3 i=76..100
   a(i) = . . .
   // P1 executes i=2..26; P2 i=27..51; P3 i=52..76; P3 i=77..101
   if (i .gt. 0) then b(i-1) = . . .
end do
```

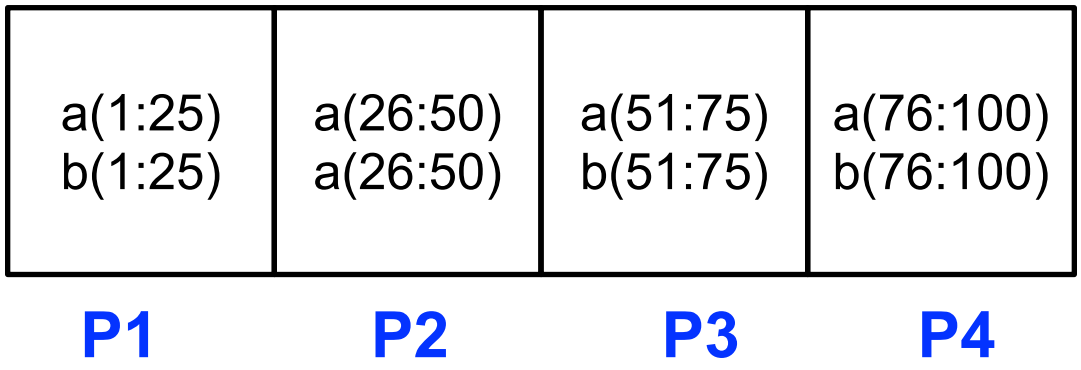
aRef

bRef



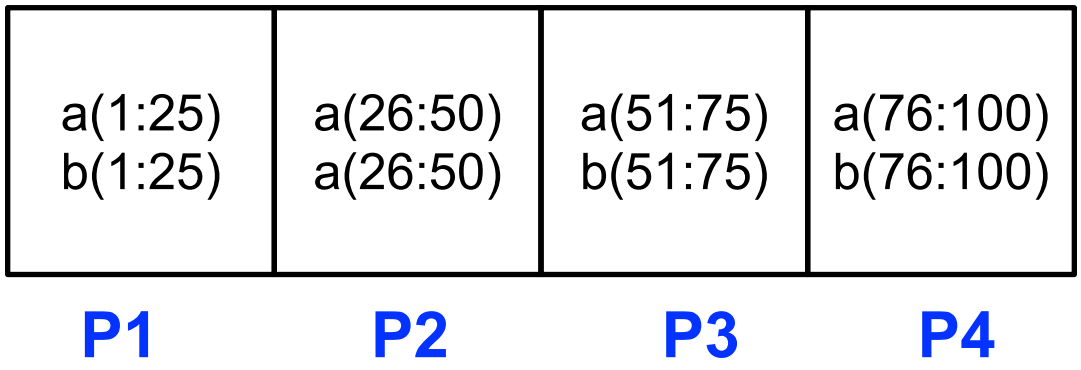
Iteration space for Pi is union of iterations spaces for all references

```
        float a(100), b(100)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE a(BLOCK) \
        on P(4)
lb = min(lb(aRef),lb(bRef) // min(1,2) for P1, min(51,52) for P3
ub = max(lb(aRef),lb(bRef) // max(25,26) for P1, max(75,76) for P3
do i = lb, ub
   // P1 executes i=1..25; P2 i=26..50; P3 i=51..75; P3 i=76..100
   a(i) = . . .
   // P1 executes i=2..26; P2 i=27..51; P3 i=52..76; P3 i=77..101
   if (i .gt. 0) then b(i-1) = . . .
end do
```



Iteration space for Pi is intersected with the original loop bounds (can't execute more iterations that originally desired)

```
      float a(100), b(100)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE a(BLOCK) \
      on P(4)
lb = max(1,min(lb(aRef),lb(bRef)) // max(1,min(1,2)) for P1
ub = min(100,max(lb(aRef),lb(bRef)) // min(100,max(100,101)) for P4
do i = lb, ub
   // P1 executes i=1..25; P2 i=26..50; P3 i=51..75; P3 i=76..100
   a(i) = . . .
   // P1 executes i=2..26; P2 i=27..51; P3 i=52..76; P3 i=77..101
   if (i .gt. 0) then b(i-1) = . . .
end do
```



Put guards around references to ensure it executes only for its iterations

```
      float a(100), b(100)
!HPF$ ALIGN a(i) WITH b(i)
!HPF$ DISTRIBUTE a(BLOCK) \
      on P(4)
lb = max(1,min(lb(aRef),lb(bRef)) // max(1,min(1,2)) for P1
ub = min(100,max(lb(aRef),lb(bRef)) // min(100,max(100,101)) for P4
do i = 1, 100
   // P1 executes i=1..25; P2 i=26..50; P3 i=51..75; P3 i=76..100
   if (i ∈ lb(aRef) ... ub(aRef)) then a(i) = . . .
   // P1 executes i=2..26; P2 i=27..51; P3 i=52..76; P3 i=77..101
   if (i ∈ lb(bRef) ... ub(bRef)) and (i .gt. 0) then b(i-1) = . . .
end do
```

An example

```
   integer A(100)
   integer B(100,100)
!HPF$ DISTRIBUTED B(BLOCK,BLOCK)
!HPF$ ALIGN A(i) WITH B(1,i)
   B = SPREAD(A,1,100)
   END
```

Distribution and alignment functions allow the compiler to express which processor an element is one, and which elements are on a processor, as functions that can be manipulated as solved.

Scalarization

- Want to turn as much code as possible into “scalar” code, i.e. operations are on single elements.
- Allows uniform compiler internals over code that was originally vector code and code that was originally scalar code.

```
      integer A(100)
      integer B(100)
!HPF$ DISTRIBUTED B(BLOCK,BLOCK)
!HPF$ ALIGN A(I) WITH B(1,I)
      B = SPREAD(A,1,100)
      END
```

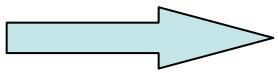
becomes

```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    B(i_6,i_5) = a(i,5)
  end do
end do
```

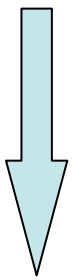
Plus distribution information

An early "gotcha" with F90

```
integer A(100)
integer B(100)
     a(1:100) = b(1:100) + a(1:100)
END
```



```
do i_5 = 1, 100,1
     t(i_5) = b(i_5) + a(i_5)
     a(i_5) = t(i_5)
  end do
end do
```



really want

F90 semantics say execution as if

i. the *rhs* computation is performed and placed in a temporary
ii. the temporary is written into the *lhs*
iii. enforces anti-dependences

```
do i_5 = 1, 100,1
     a(i_5) = b(i_5) + a(i_5)
  end do
end do
```

Now what?

- Schedule, or spread, computation across processors
  - Owner computes rule the most popular way to do this
  - Processor that owns the LHS datum computes values stored into the LHS datum
- Values on the RHS that reside on a different processor than the LHS owner must be communicated to the LHS owner.

SPMD code for the Spread

```
      integer, pointer :: A, B

!     DATA PARTITIONING

      call hpf_get_numprocs(2,numprocs,pid)
      global_bounds(1) = 1
      global_bounds(2) = 100
      global_bounds(3) = 1
      global_bounds(4) = 100
      blocksize(1) = ((100 + numprocs(1)) - 1) / numprocs(1)
      blocksize(2) = ((100 + numprocs(2)) - 1) / numprocs(2)
      iown_lbound(1) = 1 + blocksize(1) * pid(1)
      iown_ubound(1) = blocksize(1) + iown_lbound(1) - 1
      iown_lbound(2) = 1 + blocksize(2) * pid(2)
      iown_ubound(2) = blocksize(2) + iown_lbound(2) - 1
      call hpf_allocate(B, global_bounds, blocksize ...)
      ...
      call hpf_allocate(A,...)

!     COMMUNICATION

      cb_section(1) = iown_lbound(2)
      cb_section(2) = min0(iown_ubound(2),100)
      cb_section(3) = 1
      call hpf_allocate_computation_buffer(buffer,cb_section,...)
      if (pid(2) .le. 99 / blocksize(2) .and. pid(1) .le. 99 / blocksize(1)
     &  .or. pid(1) .eq. 0) then
        send_section(1) = iown_lbound(2)
        send_section(2) = min0(iown_ubound(2),100)
        send_section(3) = 1
        ....
        call hpf_bcast_section(A,send_section, buffer...)
      end if

!     LOOPS SHRUNK BY COMPUTATION PARTITIONING

      do i_8=iown_lbound(2),min0(iown_ubound(2),100),1
        do i_9=iown_lbound(1),min0(iown_ubound(1),100),1
          B(i_9,i_8) = buffer(i_8)
        end do
      end do

      call deallocate(buffer)
      end
```

Figure 5: SPMDized SPREAD program

Don't try to read this, but this is the final SPMD code for the original example program.

Three kinds of code:

1. Data partitioning bookkeepting code
2. Communication code
3. Shrunked computation loop to achieve parallelism

SPMDization

Enforce the owner compute's rule

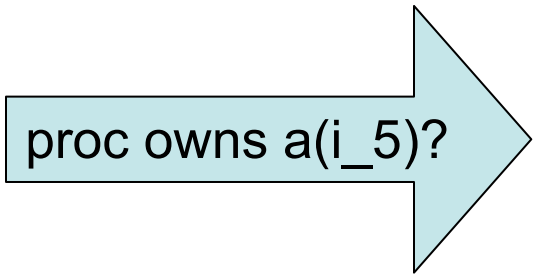
```
do i_5 = 1, 100,1
   do i_6 = 1,100,1
      b(i_6,i_5) = a(i,5)
   end do
end do
```

p = 0 … numproc-1

```
do i_5 = 1, 100,1
   do i_6 = 1,100,1
      if (I_6 .ge. p*blocksize(1) + 1 .and.
          I_6 .le. (p+1)*blocksize(1) .and.
          I_5 .ge. p*blocksize(2) + 1 .and.
                   I_5 .le. lb+blocksize(2)-1) then
          b(i_6,i_5) = a(i_5)
      end
   end do
end do
```

Fetch remote data

```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    if (I_6 .ge. p*blocksize(1) + 1.and.
    I_6.le.lb+blocksize(1)-1.and.
    I_5.ge. p*blocksize(2) + 1.and.
    I_5.le.lb+blocksize(2)-1) then
    B(i_6,i_5) = a(i_5)
    end
  end do
end do
```




```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    a_owner(1) = 1
    a_owner(2) = (i_5+numprocs(2)-1)/numprocs(2)
    if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2)) then
      send(a(i_5), ...); send(a(i_5), ...); ... send(a(i_5), ...)
      abuff = a(i_5) /* this allows a uniform reference to a(i_5) */
    end
    if (i_6 .ge. p*blocksize(1) + 1.and.
        i_6.le.lb+blocksize(1)-1.and.
        i_5.ge. p*blocksize(2) + 1.and.
        i_5.le.lb+blocksize(2)-1) then
        if !(pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2))
        then
            receive(abuff)
        end
        B(i_6,i_5) = abuff
        end
    end
  end do
end do
```

Fetch remote data

```
do i_5 = 1, 100,1
   do i_6 = 1,100,1
      if (I_6 .ge. p*blocksize(1) + 1.and.
      I_6.le.lb+blocksize(1)-1.and.
      I_5.ge. p*blocksize(2) + 1.and.
      I_5.le.lb+blocksize(2)-1) then
      B(i_6,i_5) = a(i_5)
      end
   end do
end do
```

Put *a(i_5)* in *abuff* so if we own it, and receive it into *abuff* if we don't own it. Allows single name for owner and non-owner

```
do i_5 = 1, 100,1
   do i_6 = 1,100,1
      a_owner(1) = 1
      a_owner(2) = (i_5+numprocs(2)-1)/numprocs(2)
      if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2)) then
         send(a(i_5), …); send(a(i_5), …); … send(a(i_5), …)
         abuff = a(i_5) /* this allows a uniform reference to a(i_5) */
      end
      if (i_6 .ge. p*blocksize(1) + 1.and.
          i_6.le.lb+blocksize(1)-1.and.
          i_5.ge. p*blocksize(2) + 1.and.
          i_5.le.lb+blocksize(2)-1) then
          if !(pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2))
          then
             receive(abuff)
          end
          B(i_6,i_5) = abuff
          end
      end
   end do
end do
```

Fetch remote data

```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    if (I_6 .ge. p*blocksize(1) + 1.and.
    I_6.le.lb+blocksize(1)-1.and.
    I_5.ge. p*blocksize(2) + 1.and.
    I_5.le.lb+blocksize(2)-1) then
    B(i_6,i_5) = a(i_5)
    end
  end do
end do
```

If we are in-bounds and don't own *a(i_5)* receive it into *abuff*

```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    a_owner(1) = 1
    a_owner(2) = (i_5+numprocs(2)-1)/numprocs(2)
    if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2)) then
      send(a(i_5), …); send(a(i_5), …); … send(a(i_5), …)
      abuff = a(i_5) /* this allows a uniform reference to a(i_5) */
    end
    if (i_6 .ge. p*blocksize(1) + 1.and.
        i_6.le.lb+blocksize(1)-1.and.
        i_5.ge. p*blocksize(2) + 1.and.
        i_5.le.lb+blocksize(2)-1) then
        if !(pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2))
        then
          receive(abuff)
        end
        B(i_6,i_5) = abuff
        end
    end
  end do
end do
```

Fetch remote data

```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    if (I_6 .ge. p*blocksize(1) + 1.and.
    I_6.le.lb+blocksize(1)-1.and.
    I_5.ge. p*blocksize(2) + 1.and.
    I_5.le.lb+blocksize(2)-1) then
    B(i_6,i_5) = a(i_5)
    end
  end do
end do
```

do the assign of *abuff* into the appropriate element of *b(i_6,i_5)*

```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    a_owner(1) = 1
    a_owner(2) = (i_5+numprocs(2)-1)/numprocs(2)
    if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2)) then
      send(a(i_5), …); send(a(i_5), …); … send(a(i_5), …)
      abuff = a(i_5) /* this allows a uniform reference to a(i_5) */
    end
    if (i_6 .ge. p*blocksize(1) + 1.and.
        i_6.le.lb+blocksize(1)-1.and.
        i_5.ge. p*blocksize(2) + 1.and.
        i_5.le.lb+blocksize(2)-1) then
        if !(pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2))
        then
          receive(abuff)
        end
        B(i_6,i_5) = abuff
        end
    end
  end do
end do
```

This code is very inefficient

```
do i_5 = 1, 100,1
  do i_6 = 1,100,1
    a_owner(1) = 1
    a_owner(2) = (i_5+numprocs(2)-1)/numprocs(2)
    if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2)) then
      send(a(i_5), …); send(a(i_5), …); … send(a(i_5), …)
      abuff = a(i_5)
    end
    if (i_6 .ge. p*blocksize(1) + 1.and.
        i_6.le.lb+blocksize(1)-1.and.
        i_5.ge. p*blocksize(2) + 1.and.
        i_5.le.lb+blocksize(2)-1) then
        if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2))
        then
          receive(abuff)
        end
        B(i_6,i_5) = abuff
    end
  end do
end do
```

- every iteration executed on every processor
  - All processors do work of whole program within a constant factor.
- Data communicated element by element
- Much redundant computation that appears to be loop independent

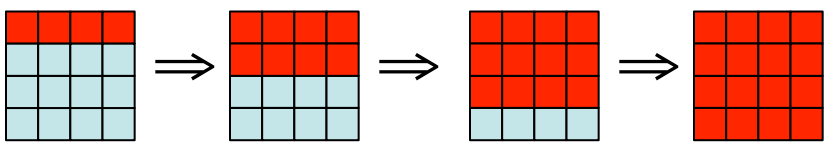
Shrink the loop bounds

```
i_6lb = min(1,p*blocksize(1) + 1)
i_6ub = max(100,i_6ilb+blocksize(1)-1)
i_5lb = min(1,p*blocksize(2) + 1)
i_5ub = max(i_5lb+blocksize(2)-1)
do i_5 = i_5lb, i_5ub, 1
   do i_6 = i_6lb, i_6ub, 1
      a_owner(1) = 1
      a_owner(2) = (i_5+numprocs(2)-1)/numprocs(2)
      if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2)) then
         send(a(i_5), …)
         abuff = a(i_5)
      end
      if then
          if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2))
          then
              receive(abuff)
          end
          B(i_6,i_5) = abuff
      end
   end do
end do
```

Each processor only does part of the work

Recognize the communication

```
i_6lb = min(1,p*blocksize(1) + 1)
i_6ub = max(100,i_6ilb+blocksize(1)-1)
i_5lb = min(1,p*blocksize(2) + 1)
i_5ub = max(i_5lb+blocksize(2)-1)
a_owner(1) = 1
a_owner(2) = (i_5+numprocs(2)-1)/numprocs(2)
if (pid(1).eq.a_owner(1).and.pid(2).eq.a_owner(2)) then
   bcast(abuff, part of a owned)
end
do i_5 = i_5lb, i_5ub, 1
   do i_6 = i_6lb, i_6ub, 1
      if then
         B(i_6,i_5) = abuff(i_5)
       end
   end do
end do
```



Computation consists of some startup code, and that part of the loop iteration space where the processor owns the LHS datum.

If the startup code is relatively fast, we get very good speedups.

Communication generation

- Look at subscript patterns and mapping information to see what the pattern is (c is constant, i, j, k are index variables). Assume data is aligned in what follows.
  - $a(...,i,...) = ... \ b(...,c,...)$ broadcast
  - $a(...,i,...) = ... \ b(...,i\text{-}c,...)$ shift
  - $a(...,c,...) = ...b(...,i,...)$ reduction
- Also perform optimizations

Communication optimizations
Loop distribution

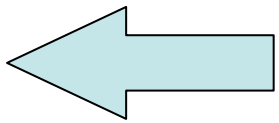
```
!HPF Align D(I) with A(i,1)
   do i = 1, n
      D(i) = D(i) + s * B(i)
      Communication for D(I)
      do j = 1, n
         A(i,j)=A(i,j)+D(i)
      end do
End do
```

Do communiation once instead of n times.

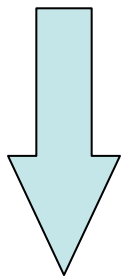
```
!HPF Align D(I) with A(i,1)
   do i = 1, n
      D(i) = D(i) + s * B(i)
   enddo
   Communication for D(1:n)
   do i = 1, n
      do j = 1, n
         A(i,j)=A(i,j)+D(i)
      end do
End do
```

Communication Optimizations Message Coalescing

```
!HPF Distribute (block,block) :: A,B
  Shift communication for (B(i-1,j))
  Shift communication for (B(i-1,j-1))
  do j = 2, n
    do I = 2, n
      A(i,j) - f (B(i-1,j), B(i-1,j-1))
    enddo
  enddo
```



can combine into one message



```
Shift communication for (B(i-1,j) U (B(i-1,j-1))
do j = 2, n
  do I = 2, n
    A(i,j) - f(B(i-1,j), B(i-1,j-1))
  enddo
enddo
```

Communication optimizations wavefront

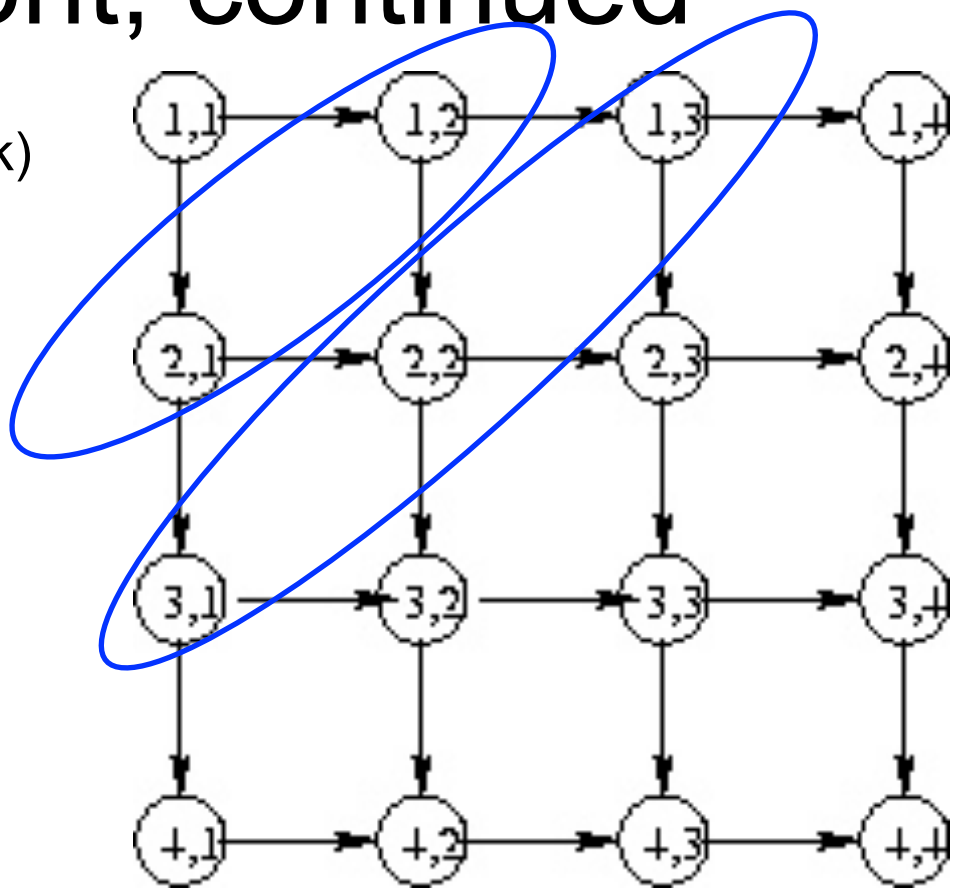
```
$HPF! Distribute A(block,block)
  do j = 2, n
    do I = 2, n
      a(i,j) = f(A(i-1,j),A(i,j-1))
    end do
  end do
```

Naïve communication generation puts communication inside both the i and j loops because of the true dependence.

```
$HPF! Distribute A(block,block)
  do j = 2, n
    receive(a2buff,…)
    do I = 2, n
      receive(a1buff, …)
      a(i,j) = f(a1buff,a2buff)
      send(a(I,j), …)
    end do
    send(a(1,j),…)
  end do
```

Communication optimizations wavefront, continued

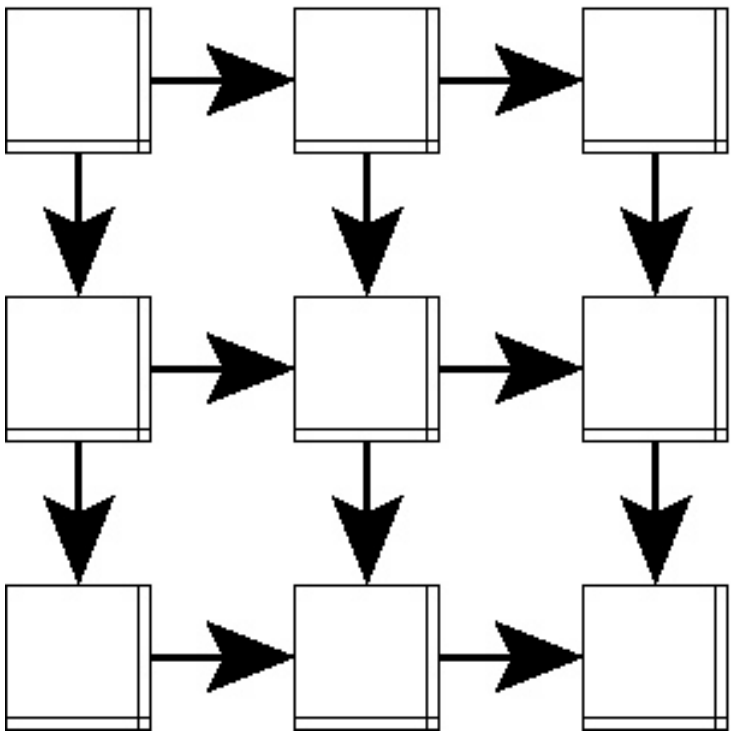
```
$HPF! Distribute A(block,block)
   do j = 2, n
      do I = 2, n
         a(i,j) = f(A(i-1,j),A(i,j-1))
      end do
   end do
```



Note that there is parallelism

Communication optimizations wavefront, continued

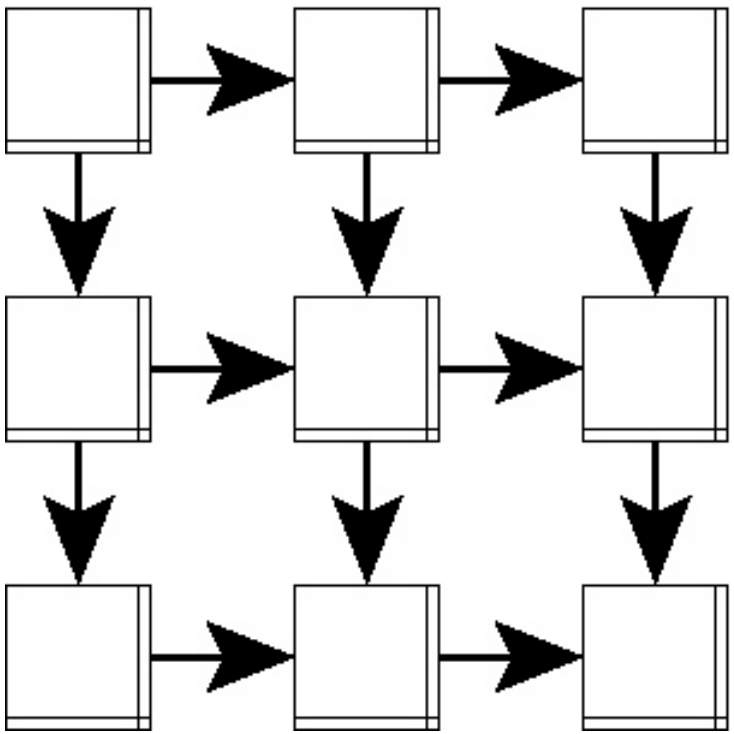
```
$HPF! Distribute A(block,block)
  do j = 2, n
    receive(abuff1,…)
    receive(abuff2,…)
    do i = 2, n
      a(i,j) = 𝓕(abuff2(i-1,j),abuff1(i,j-1))
    end do
    shift(A,down,…)
    shift(A,across,…)
  end do
```



Each shift communicates a strip of A, not just a single element.

Communication optimizations wavefront, continued

```
$HPF! Distribute A(block,block)
  do j = 2, n
    receive(abuff1,…)
    receive(abuff2,…)
    do i = 2, n
      a(i,j) = 𝒻(abuff2(i-1,j),abuff1(i,j-1))
    end do
    shift(A,down,…)
    shift(A,across,…)
  end do
```



Trade-off between granularity of parallelism and number of messages. Reduces message startup cost and latency overhead

The real code for SPREAD Data Partitioning

```
   integer, pointer : : A, B
!  DATA PARTITIONING
   call hpf_get_numprocs,numprocs,pid)
   global_bounds(1) = 1 // lower bounds dimension 1
   global_bounds(2) = 100 // upper bounds dimension 1
   global_bounds(3) = 1 // l lower bounds dimension 2
   global_bounds(4) = 100 // upper bounds dimension 1
   blocksize(1) = ((100+numprocs(1)-1)/numprocs(1) // blocksize dimension 1
   blocksize(2) = ((100+numprocs(2))-1)/numprocs(2) // blocksize dimension 2
   iown_lbound(1) = 1+blocksize(1)*pid(1) // LB owned elts dim 1
   iown_ubound(1) = blocksize(1)*iown_lbounds(1)-1 // UB owned elts dim 1
   iown_lbound(2) = 1+blocksize(2)*pid(2) // LB owned elts dim 2
   iown_ubound(2) = blocksize(2)*iown_lbounds(2)-1 // UB owned elts dim 2
   call hpf_allocate(B, global_bounds, blocksize, …) // allocate local space for B
   …
   call hpf_allocate(A, …) // allocate local space for A
```

The real code for SPREAD Communication

```
! COMMUNICATION
  cb_section(1) = iown_lbound(2) // communication buffer lower bound   cb_section(2) =
    min0(iown_lbound(2),100) // comm buffer upper bound
  cb_section(3) = 1 // comm buffer stride
  call hpf_allocate_computation_buffer(buffer, cb_section, …) // alloc comm buffer
  if (pid(2).le.99/blocksize(2).and.pid(1).le.99/blocksize(1).or.pid(1).eq.0) then
    send_section(1) = iown_lbound(2)
    send_section(2) = min0(iown_ubound(2), 100)
    send_section(3) = 1
    …
    call hpf_bcast_section(A, send_section, buffer, …)
  end if
```

The real code for SPREAD Computation

```
!  LOOPS SHRUNK BY COMPUTATION PARTITIONING
   do i_8=iown(lbound(2), min0(iown_ubound(2),100),1
     do i_9=iown_lbound(1),min0(iown_ubound(1),100),1
       B(i_9,i_8) = buffer(i_8)
     end do
   end do
   call deallocate(buffer)
   end
```

Performance

Program	Serial	1 proc	2 proc	4 proc	8 proc	16 proc	32 proc
Grid(b,b,*)	1 (43.6)	1.01	2.01	4.02	7.92	15.47	30.49
Grid*(b,b,*)	1.00	1.02	2.02	4.03	7.98	14.09	30.28
Grid MPL	1.00	1.00	2.00	3.98	7.89	15.78	30.43
Tomcatv(*,b)	1 (40.06	0.89	1.76	3.29	6.36	11.34	17.65
Ncar(*,b)	1 (14.34)	0.89	2.13	3.88	6.91	12.12	18.33
Ncar(b,b)	1	1.01	1.71	3.74	6.75	12.20	19.32
Ncar MPL	1	1.14	2.28	4.53	8.82	16.62	31.10
X42(*,b)	1 (0.84)	1.03	1.95	3.65	6.61	10.98	16.45
X42(b,b)	1	1.18	2.03	3.81	6.99	13.36	21.96
X42 MPL	1	1	1.98	3.85	7.50	13.77	24.70

How successful was HPF?

- Major hardware vendors had implementations (IBM, DEC, Portland Group (PGI) for others)
- Good performance on regular programs
- Lots of research funding from both industry and NSF
  - Rice, Syracuse, UIUC, IBM, DEC had major research efforts
  - Dominated conferences
  - Do you want your language to be a research topic?

What killed HPF?

- Unrealistic expectations: *just put in data distributions and we will do the rest*
- Often bad performance initially
- Unpredictable performance: slight change in program led to major, inexplicable changes in performance
- Different subset for every vendor
- Could not handle irregular programs well