CS 267 Applications of Parallel Computers

Lecture 5: Data Parallel Processing

2/4/97

David E. Culler

http://www.cs.berkeley.edu/cs267/

Outline

- Recap
- Quick Evolution of Data-Parallel Machines
- Fortran 90
- HPF extensions

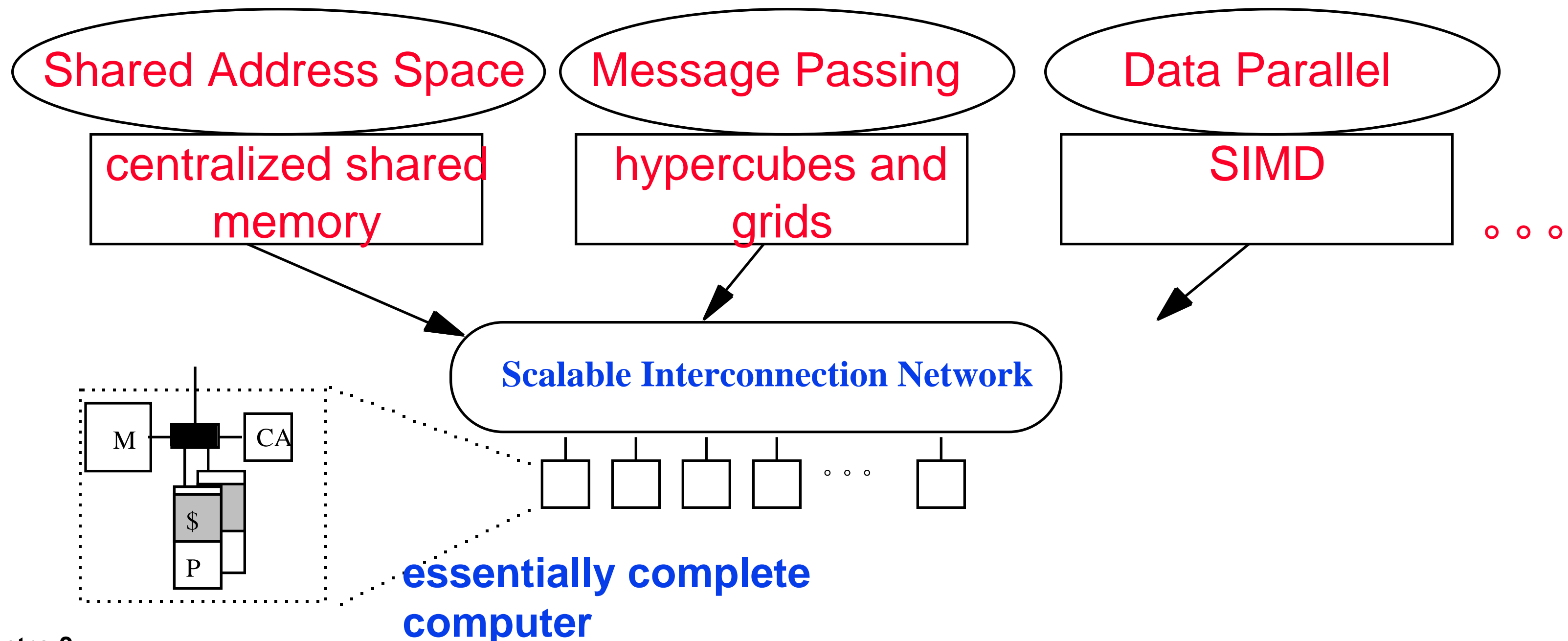
Recap: Historical Perspective

° Diverse spectrum of parallel machines designed to implement a particular programming model directly

° Technological convergence on collections of microprocessors on a scalable interconnection network

° Map any programming model to simple hardware

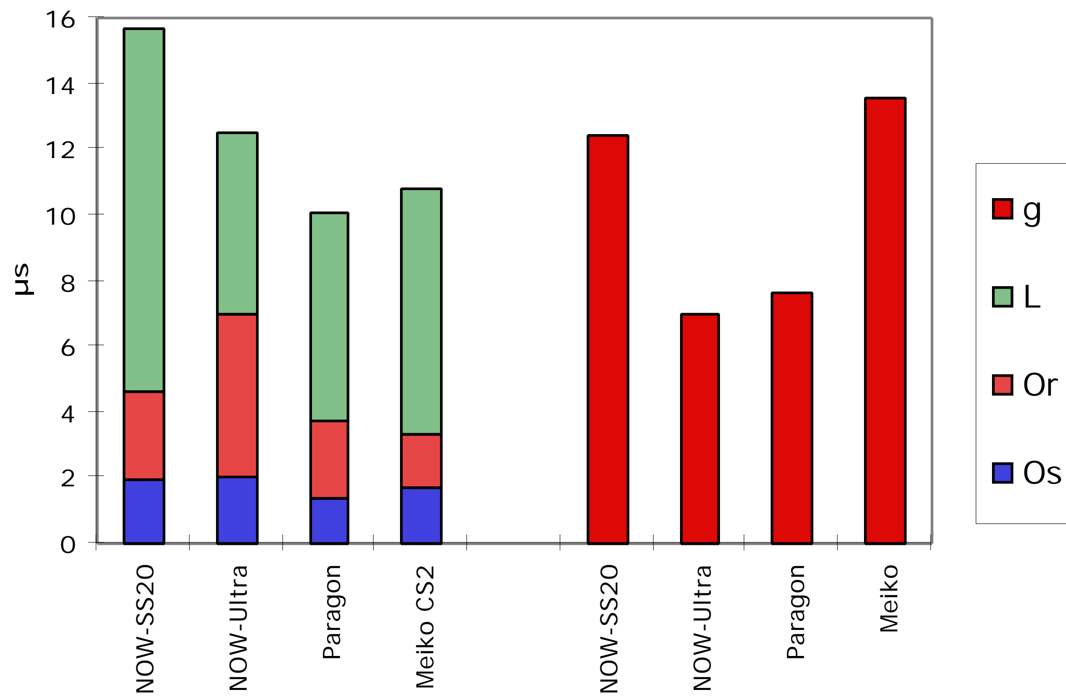
- with some specialization



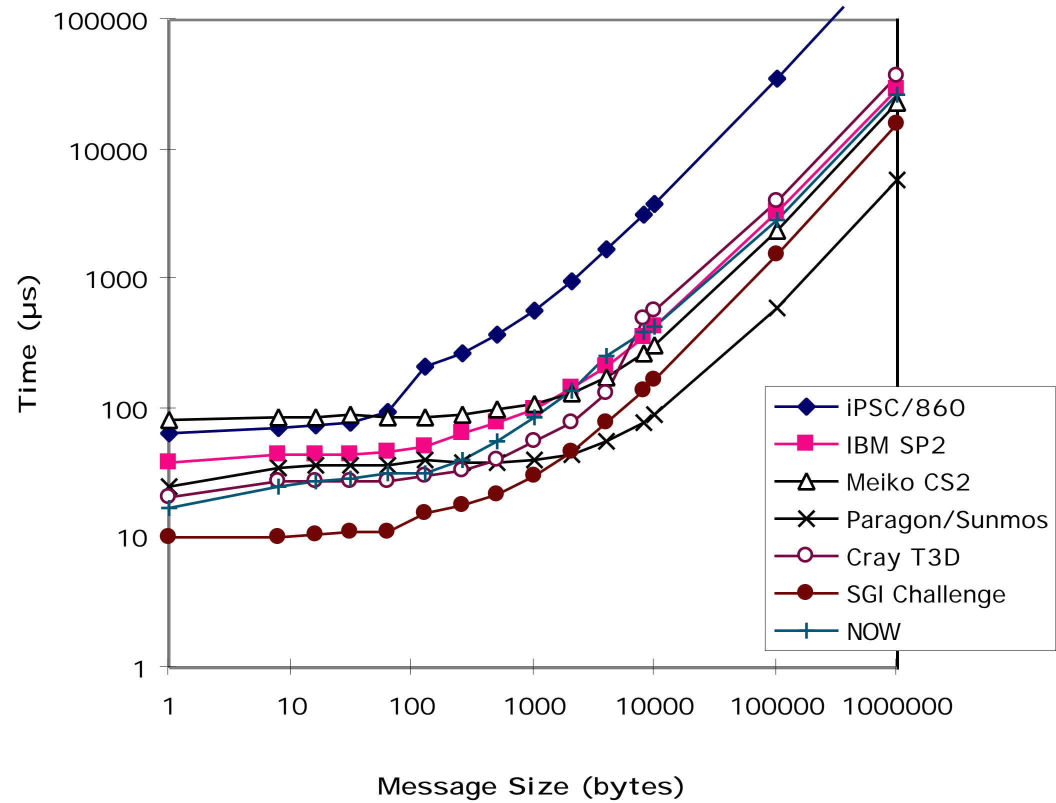
Where are things going

- High-end
  - collections of almost complete workstations/SMP on high-speed network
  - with specialized communication assist integrated with memory system to provide global access to shared data
- Mid-end
  - almost all servers are bus-based CC SMPs
  - high-end servers are replacing the bus with a network
    - Sun Enterprise 10000, IBM J90, HP/Convex SPP
  - volume approach is Ppro quadpack + SCI ring
    - Sequent, Data General
- Low-end
  - SMP desktop is here
- Major change ahead
  - SMP on a chip as a building block

Comparison of Base Message Event



Comparison of Message Passing Performance



MPI - thanks to Bill Saphir

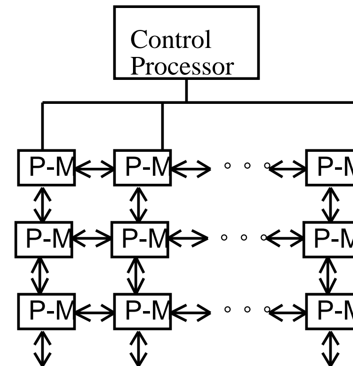
Data Parallel Architectures

° Programming model

- operations are performed on each element of a large (regular) data structure in a single step
- arithmetic, global data transfer

° processor is logically associated with each data element, general communication, and cheap global synchronization.

- driven originally by simple O.D.E.



Evolution and Convergence

° Rigid control structure (SIMD in Flynn's Taxonomy)

- SISD = uniprocessor, MIMD = multiprocessor

° Cost savings in centralized instruction sequencer

- 60's when CPU was a cabinet of eqipment
- mid 80's when 32-bit slices of datapath would just fit on a chip

° Simple, regular calculations usually have good locality

- realize on SM or MP machine with decent compiler
- may still require fast global synchronization

° Programming model converges with SPMD

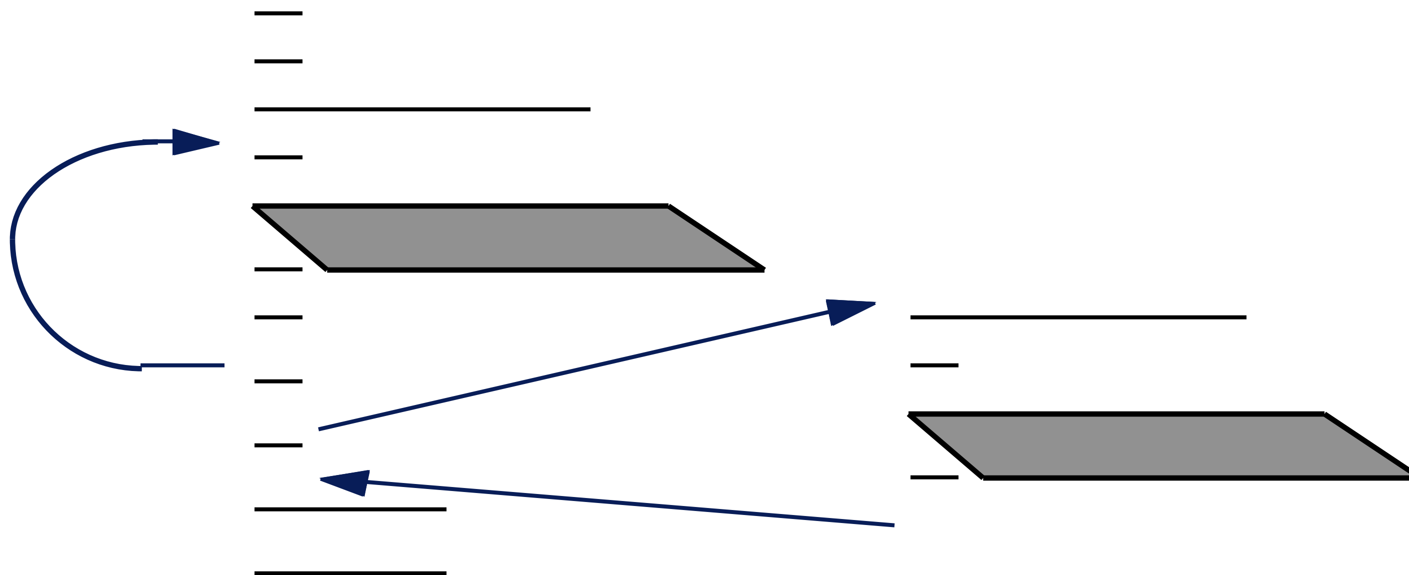
- single program multiple data

Basics of a Parallel Language

- How is parallelism expressed?
- How is communication expressed?
- How is synchronization expressed?
- What global / local data structures can be constructed?
- How do you optimize for performance?

Fortran 90 Execution Model

- Sequential composition of parallel (or scalar) statements
- Parallel operations on **arrays**



- Arrays have **rank** (# dimensions), **shape** (extents), **type** (elements)
  - and **layout**
- Communication implicit in array operations
- Configuration independent

Example: gravitational fish

```
    integer, parameter :: nfish = 10000
    complex fishp(nfish), fishv(nfish), force(nfish), accel(nfish)
    real    fishm(nfish)
    integer density(m,m)
. . .
    do (step = 1, nsteps)
       fishp   = fishp + dt*fishv
       call compute_current(force,fishp)
       accel   = force/fishm
       fishv   = fishv + dt*accel
       call compute_density(density,m,fishp,nfish)
    enddo
. . .
    subroutine compute_current(force,fishp)
    complex force(:),fishp(:)
    force = (3,0)*(fishp*(0,1))/(max(abs(fishp),0.01)) - fishp
    end
```

parallel assignment

pointwise parallel operator

Array Operations

Parallel Assignment

```
A = 0                       ! scalar extension
L = .TRUE.
B = [1,2,3,4]               ! array constructor
X = [1:n]                   ! real sequence [1.0, 2.0, . . .,n]
I = [0:100:4]               ! integer sequence [0,4,8,...,100]
C = [ 50[1], 50[2,3] ]      ! 150 ele'ts first 1s then repeated 2,3
D = C                       ! array copy
call CMF_random(A)          ! fill A with reals [0.0, 1.0]
call CMF_random(I,100)      ! ints from [0,99]
```

Binary array operators operate pointwise on **conformable** arrays

- have the same size and shape
- size is product of extent

Array Sections

Portion of an array defined by a triplet in each dimension

- **may appear wherever an array is used**

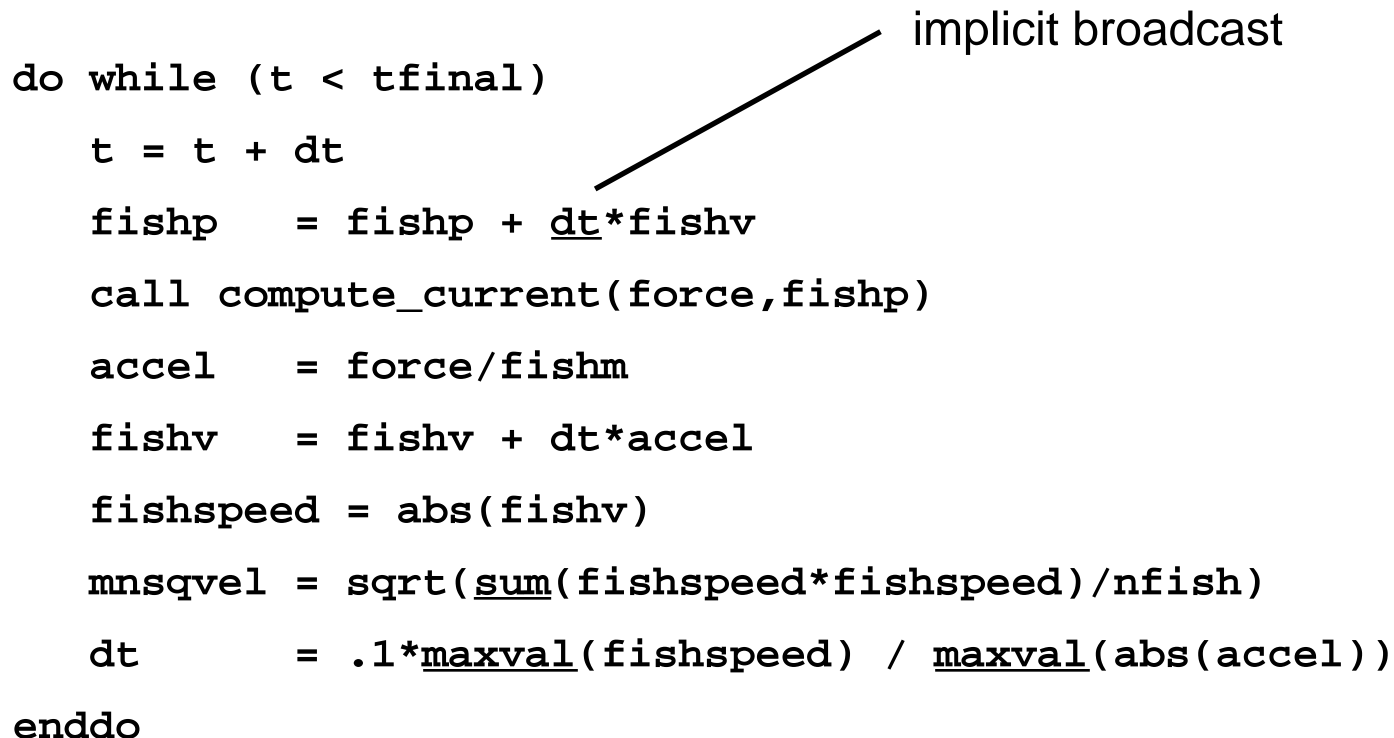
```
A(3)            ! third element
A(1:5)          ! first five elements
A(1:5:1)        ! same
A(:5)           ! same
A(1:10:2)       ! odd elements in order
A(10:2:-2)      ! even in reverse order
A(10:2:2)       ! []

B(1:2,3:4)      ! 2x2 block
B(1, :)         ! first row
B(:, j)         ! jth column
```

Reduction Operators

Reduce an array to a scalar under a binary operation

- sum, product
- minval, maxval
- count (number of .TRUE. elements of logical array)
- any, all

simplest form of communication




```
do while (t < tfinal)
   t = t + dt
   fishp    = fishp + dt*fishv
   call compute_current(force,fishp)
   accel    = force/fishm
   fishv    = fishv + dt*accel
   fishspeed = abs(fishv)
   mnsqvel = sqrt(sum(fishspeed*fishspeed)/nfish)
   dt       = .1*maxval(fishspeed) / maxval(abs(accel))
enddo
```

Conditional Operation

```
force = (3,0)*(fishp*(0,1))/(max(abs(fishp),0.01)) - fishp
```

could use

```
dist = 0.01
where (abs(fishp) > dist) dist = abs(fishp)
```

or

```
far = abs(fishp) > 0.01
where far dist = abs(fishp)
```

or

```
where (abs(fishp) .ge. 0.01)
    dist = abs(fishp)
elsewhere
    dist = 0.01
end where
```

No nested wheres. Only assignment in body of the where. The boolean expression is really a mask array.

General Parallel Assignment

FORALL (triplet spec) assignment

```
forall ( i = 1:n) A(i) = 0   ! same as A = 0
forall ( i = 1:n ) X(i) = i  ! same as X = [ 1:n ]
forall (i=1:nfish) fishp(i) = (i*2.0/nfish)-1.0

forall (i=1:n, j = 1:m) H(i,j) = i+j
forall (i=1:n, j = 1:m) c(i+j*2) = j
forall ( i = 1:n ) D(i) = C(i,i)

forall (i=1:n, j = 1:n, k = 1:n)
*     C(i,j) = C(i,j) + A(i,k) * B(k,j)   ! NO
```

LHS must use all variables in forall statement

Target need not be entire array

No more than one value for each element on the left

Only intrinsics (no user functions) on the right - CMF (see HPF pure)

Conditional (masked) intrinsics

Most intrinsics take an optional mask argument

```
funny_prod = product( A, A .ne. 0)
bigem = maxval(A, mask = inside )
```

Masks can also be used in the FORALL assignment (HPF)

```
forall ( i=1:n, j=1:m, A(i,j) .ne. 0.0 ) B(i,j) = 1.0 / A(i,j)
forall ( i=1:n, inside) A(i) = i/n
```

Subroutines

- Arrays can be passed as arguments.
- Shapes must match.
- Limited dynamic allocation
- Arrays passed by reference, sections by value (i.e., a copy is made)
  - HPF: either remap or inherit
- Can extract array information using inquiry functions

Implicit Communication

Operations on conformable array sections may require data movement

i.e., communication

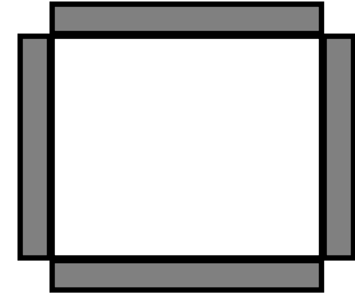
```
A(1:10, : ) = B(1:10, : ) +  B(11:20, : )
```

```
Parallel finite diference A'[i] = (A[i+1] - A[i])*dt
      A(1:n-1) = (A(2:n) - A(1:n-1)) * dt
```

Example: smear pixels

```
show(:,1:m-1) = show(:,1:m-1) + show(:,2:m)
show(1:m-1,:) = show(1:m-1,:) + show(2:m,:)
```

2D Electromagnetics



```
c   D[ex,t] =  c (  D[bz,y] - jx )
     ex(1:m-1,2:n-1) = ex(1:m-1,2:n-1) +
*          epsy * (bz(1:m-1,2:n-1) - bz(1:m-1,1:n-2)) - jx(1:m-1,2:n-1)
c   D[ey,t] =  c ( -D[bz,x] - jy )
     ey(2:m-1,1:n-1) = ey(2:m-1,1:n-1) -
*     epsx * (bz(2:m-1,1:n-1) - bz(1:m-2,1:n-1)) -  jy(2:m-1,1:n-1)
c   D[bz,t] =  c (  D[ex,y] - D[ey,x] )
     bz(1:m-1,1:n-1) = bz(1:m-1,1:n-1) + (
*     epsy * (ex(1:m-1,2:n) - ex(1:m-1,1:n-1)) -
*     epsx * (ey(2:m,1:n-1) - ey(1:m-1,1:n-1)) )
```

Global Communication

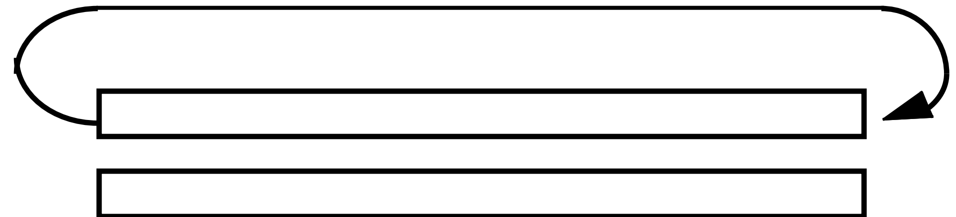
```
c(:, 1:5:2) = c(:, 2:6:2)     ! shift noncontiguous sections

D = D(10:1:-1)                ! permutation (reverse)

A = [1,0,2,0,0,0,4]
I = [1,3,7]
B = A(i)                      ! B = [1,2,4] ``gather''
C(I) = B                      ! C = A ``scatter'' (no duplicates on left)

D = A([1,1,3,3])              ! replication
```

Specialized Communication

```
CSHIFT( array, dim, shift)                    ! cyclic shift in one dimension
EOSHIFT( array, dim, shift [, boundary]) ! end off shift
TRANSPOSE( matrix )                           ! matrix transpose


SPREAD(array, dim, ncopies)
```

Example: nbody calculation

```
subroutine compute_gravity(force,fishp,fishm,nfish)
  complex force(:),fishp(:),fishm(:)
  complex fishmp(nfish),  fishpp(DSHAPE(fishp)), dif(DSIZE(force))
  integer k

  force  = (0.,0.)
  fishpp = fishp
  fishmp = fishm
  do k=1, nfish-1
    fishpp = cshift(fishpp, DIM=1, SHIFT=-1)
    fishmp = cshift(fishmp, DIM=1, SHIFT=-1)
    dif = fishpp - fishp
    force = force + (fishmp * fishm * dif / (abs(dif)*abs(dif)))
  enddo
  end
```



Scan Operations

```
forall (i=1:5) B(i) = SUM( A(1:i) )          ! forward running sum
forall (i=1:n) B(i) = SUM( A(n-i+1:n) )   ! reverse direction

dimension fact(n)
fact = [1:n]
forall (i=1:n) fact(i) = product( fact(1:i) )
```

or

CMF_SCAN_*op* (dest,source,segment,axis,direction,inclusion,mode,mask)

op = [add,max,min,copy,ior,iand,ieor]

CMF Homes and Layouts

- Machine is really a front-end sparc connected to a bunch of sparc nodes.
- Arrays have a **home**: FE or CM. Don't mismatch homes!
  - If you do parallel operations on it, it lives in the CM
  - If you specify a layout, it lives on the CM
- FORALL with a user function or intrinisic will usually end up on the front end.
- Arrays have a **layout**
  - Compiler views machine as a multidimensional grid of processors
  - A multidimensional array is placed over this grid in block of some shape.
  - Layout directive tells the compiler which dimensions should remain processor local.
  - If you give a layout directive, all subroutines that operate on the array must give the same one!

CMF Layout Example

```
C  Layout matrix as 8x8 grid of nxn blocks
      integer, parameter ::  p = 8
      integer, parameter ::  n = 64

      double precision, array(p,p,n,n) ::  A,B,C
CMF$  LAYOUT a(:NEWS,:NEWS,:SERIAL,:SERIAL)
CMF$  LAYOUT b(:NEWS,:NEWS,:SERIAL,:SERIAL)
CMF$  LAYOUT c(:NEWS,:NEWS,:SERIAL,:SERIAL)

      call CMF_describe_array(A)
```

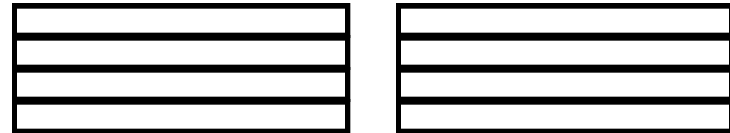
Stencil Calculations

```
nbrcnt(2:d-1,2:d-1) = fish(1:d-2,2:d-1) + fish(1:d-2,1:d-2) +
!                     fish(1:d-2,3:d) + fish(2:d-1,1:d-2) +
!                     fish(2:d-1,3:d) + fish(3:d,2:d-1) +
!                     fish(3:d,1:d-2) + fish(3:d,3:d)
```

How do you eliminate unnecessary data movement?

Blocking

```
subroutine compute_gravity(force,fishp,fishm,nblocks)
complex force(:,B),fishp(:,B),fishm(:,B)
complex fishmp(nblocks,B), fishpp(nblocks,B),dif(nblocks,B)
CMF$ layout force(:news, :serial), . . .
force  = (0.,0.)
fishpp = fishp
fishmp = fishm
do k=1, nblocks-1
  fishpp  = cshift(fishpp,  DIM=1, SHIFT=-1)
  fishmp = cshift(fishmp, DIM=1, SHIFT=-1)
  do j = 1, B
     forall (i = 1:nblocks) dif(i,:) = fishpp(i,j) - fishp(i,:)
     forall (i = 1:nblocks) force(i,:) = force(i,:) +
*        (fishmp(i,j) * fishm(i,:) * dif(i,:) / (abs(dif(i,:))*abs(dif(i,:))))
  end do
enddo
```



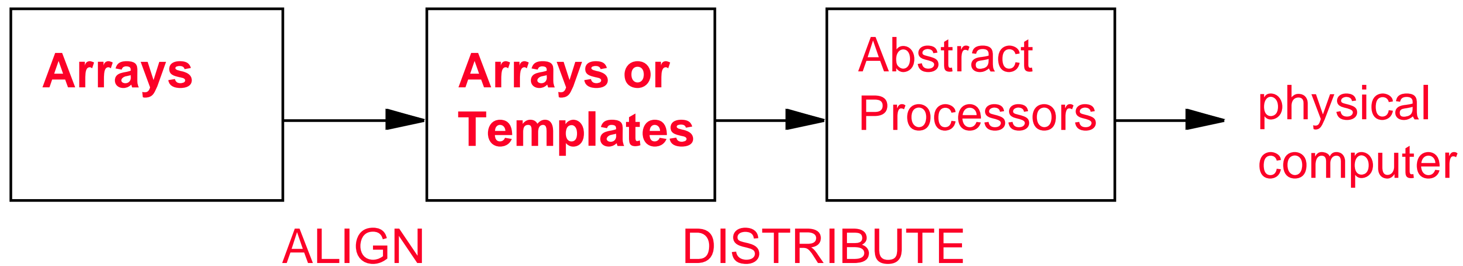
HPF Data Distribution (layout) directives

° **Can ALIGN arrays with other arrays to establish affinity**

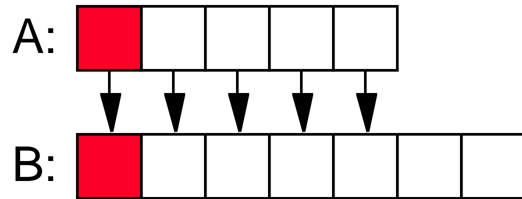
  • **elements that are operated on together**

° **Can DISTRIBUTE arrays over abstract processor grids**

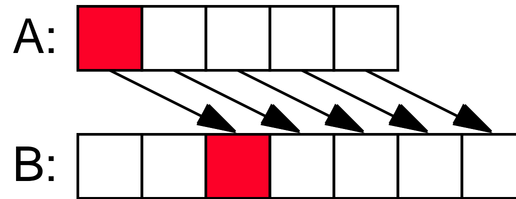
° **Compiler maps processor grids to physical procs.**



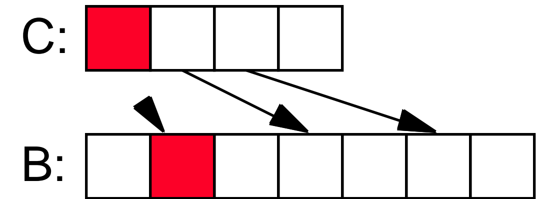
Alignment



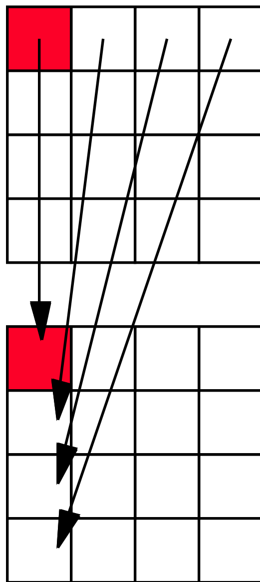
ALIGN A(I) WITH B(I)

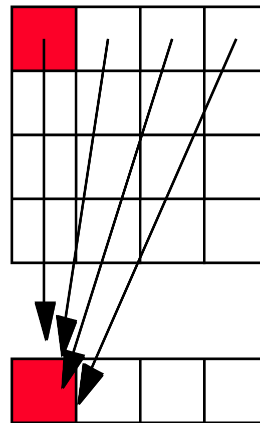


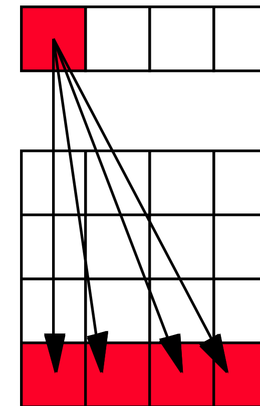
ALIGN A(I) WITH B(I+2)



ALIGN C(I) WITH B(2*I)







?

ALIGN D(:,*) with A(:)

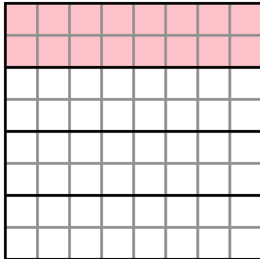
- collapse dimension

ALIGN A(:) with D(*,:)

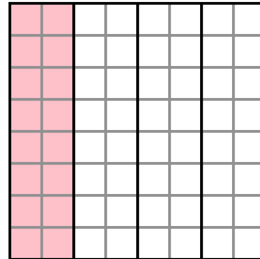
- replication

ALIGN D(i,j) WITH E(j,i)

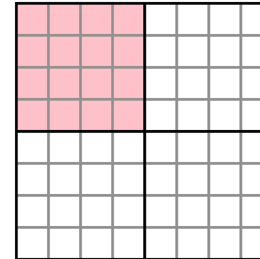
Layouts



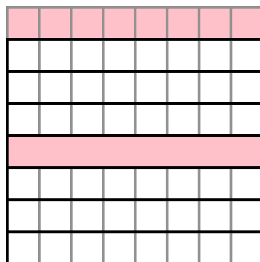
(Block, *)



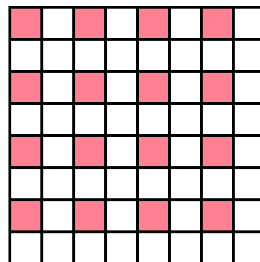
(*, Block)



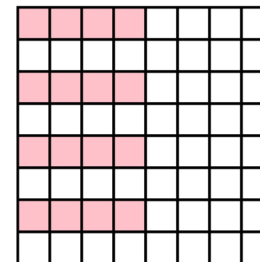
(Block, Block)



(Cyclic, *)



(Cyclic, Cyclic)



(Cyclic, Block)

Example

Declaring Processor Grids

```
!HPF$     PROCESSORS P(32)

!HPF$     PROCESSORS Q(4,8)
```

Distributing Arrays onto Processor Grids

```
!HPF$     PROCESSORS p(32)

            real D(1024), E(1024)
!HPF$     DISTRIBUTE D(BLOCK)

!HPF$     DISTRIBUTE E(BLOCK) ONTO p
```

Independent

° assert that the iterations of a do-loop can be performed independently without changing the result computed.

- in any order or concurrently

```
!HPF$   INDEPENDENT
          do i=1,n
            A(Index(i)) = B(i)
          enddo
```

Load balancing in Wator worlds?

- What is the current function was highly irregular?
- What if fish breed?
- How would you optimize away open ocean?
- How do you compute the fish density?

Positions

Fish per cell

Other Data Parallel Languages

- *LISP, C*
- NESL, FP
- PC++
- APL, MATLAB, . . .