High Performance Fortran

Tutorial

at

Europar'99

by Fabien COELHO

Fabien Coelho

- Engineering degree in 1993
- PhD on HPF Compilation in 1996
- researcher at cole des mines de Paris, France
  - algebraic transformations to improve performance
  - focus on scientific applications
  - PhD Student: Julien Zory
- visitor at IMEC, Belgium
  - Francky Catthoor's team
  - program transformations for low power
  - focus on embedded applications

Sources

- HPF language specifications

  `http://dacnet.rice.edu/Depts/CRPC/HPFF/`

- The HPF Handbook

  by Koelber, Loveman, Schreiber, Steele and Zosel

  MIT Press

Tutorial outline

- What is HPF? MPI? OMP?
- History of HPF
- Presentation of HPF
- Examples of codes
- Conclusion

MPI vs HPF vs OMP

What they have in common:

- 3 letters acronymes
  - Message Passing Interface
  - High Performance Fortran
  - Open Multi Processing
- to express parallelism
- based on standard languages
  - Fortran
  - C

Why Parallelism?

So as to improve performance of applications

- inside processors:
  - super-scalar/VLIW
    different instructions at the same time
  - pipelines, vector processors
    start next computation before previous is finished
  - replication of functionnal units
- network of processors (slow network/fast processors, 2 orders!)

Parallel Hardwares: Multiple Data

- Shared Memory vs Distributed Memory
  - SM: communications/addressing by OS/HW
  - DM: YOU take care (user/compiler, OS...)
    communication and addressing are hard
    management of data distribution among processors...
- Control flow: Single vs Multiple Instruction
  - SIMD: one processor + many computing elements
  - MIMD: multiple processors

Examples of parallel machines

general purpose machines: MIMD survived

- SM-MIMD: shared memory multi-processors

  SUN servers, Intel-based mp

  not scalable wrt #procs

- DM-MIMD: IBM SP2, Cray T3E, SGI Origin 2000

  scalable wrt #procs

special purpose machines: often SIMD (DSP, ASICS)

- DM-SIMD: CM2, MMX (in processor)

Parallel Programming

- Control Parallelism (based on threads/processes...)

  fork/join model, synchronizations...
  - MPI: processes are spawned
  - OMP: multi threading on loops
  - Tera computer...
- Data Parallelism
  - same computation on different elements
  - Fortran 90, HPF
  - simpler codes (one flow, SIMD-like)

Data Parallelism

- manipulation of arrays as scalars
- operation can be applied in parallel

```
program dataparallel
integer, parameter:: n=1000
real, dimension(n,n):: A, B, C
A = ..., B = ...
C = 0.0
do while (MAXVAL(ABS(C-B))>0.1)
   C = B ! old B is saved
   B = MATMUL(A, B) + B
end do
end
```

Control parallelism

- fork parallel communicating 'processes'
- beware of data sharing!
- join back...
- can be hidden in the language (OMP)

  the memory is shared among processors
- addressing issues handled by OS/HW

Control parallel example

```
      program controlparallelism
      real, dimension(n):: v
      integer:: count
      count = 0
!$omp parallel, do
      do i=1, n
         if (v(i).gt.floor) then
! synchronization needed...
!$omp    atomic
            count = count + 1
         endif
      enddo
!$omp end parallel
```

Control parallelism: message passing

- fork PROCESSES (a la Unix)
- library to send and receive messages
- addressing managed by programmer

MP example

```
      program messagepassing
      real, dimension(n/2):: v ! v(n)
      spawn(2,...)
      if (i am process 1) then
! v holds first half v(1) is v(1)
        senddatato(2)
        ...
      else if (i am process 2) then
! v holds second half, v(1) is V((n/2)+1)
        receivedatafrom(1)
        ...
      endif
```

Control vs Data parallelism

A dual philosophy:

- control parallel
  - distribute iterations
  - data will follow
- data parallel
  - distribute data
  - iterations will follow

Parallel programming and architectures

- Shared Memory, MIMD
  - OMP is the only standard! things are simple!
- Distributed Memory, MIMD
  - MPI: low level, everything by hand!
  - HPF: put the trouble on the compiler!

What is the trouble?

On a Distributed Memory parallel machine

- find available parallelism
  - program semantics must not be changed!
  - automatically detection of parallelism?
  - hints: language? assertions?
- must distribute data on processors!
  - otherwise no parallelism!
  - how? when? hints?
  - impact on addressing? on performance?
- non local data communication and store?

High Performance Fortran

- Data Parallel language
- based on Fortran 90/95

  the language of scientific computing
- plus directives (i.e. hints as comments)

  YOU are going to help!

  program still in Fortran
- a de facto standard

  not a ISO or ANSI standard

  however only one available!

  contribution to the standard (FORALL)

Fortran standard...

- dusty deck Fortran IV/66/77 code legacy
- engineers are used to Fortran
- software engineering improvements with Fortran 90
  - structured programming
  - data structures, allocate
  - better library of functions
    e.g. TRANSPOSE MATMUL
- other option: SISAL (functionnal)
- new Fortran 95: small improvements

History of HPF: the Forum

- DEC initiative end of 1991
- composition: vendors, researchers, users
- aim: to propose an industry standard
  - offer portability for users
  - as a commercial argument
- meetings

HPF versions...

- 03/1992-03/1993: HPF 1.0 + subset

  subset: selection of features, Fortran 77 based

- 04/1994-10/1994: HPF 1.1 + subset
- 01/1995-11/1996: HPF 2.0 + approved extensions

  AE: how shoud extensions should look like

  no more subset

- kernel HPF: efficient part of the language

Organization

- regular meetings in the USA
- specialized work groups AND mailing lists
  - data distribution, forall
  - Fortran 90, intrinsics, I/O...
- one regularly-participating organisation, one vote
- language defined by the majority...

Existing base

- Standards
  - Fortran 77
  - Fortran 90 (long awaited Fortran 8x)
  - PC Fortran (for share memory)
- Vendors products
  - CM-Fortran from Thinking Machine
  - MPP-Fortran
- Research prototypes
  - Fortran D (Ken Kennedy, Rice University)
  - Vienna Fortran (Hans Peter Zima, Vienna

University)

Ideas taken

For this existing base

- compilation directives seen as comments
- sequential program + data mapping
- data parallel array sections (F90)
- concepts of alignment and distribution

  i.e. two level mapping
- two semantics of parallel loops

Life of the Forum

- great battles: Templates or not Templates
- lost projects: I/O
- forgotten principles

  not only directives

  but extensions included in Fortran 95!
- forgotten dead lines;-)

HPF 1.0, May 3 1993

- a Fortran for parallel computers
- data parallel model
- based on Fortran 90
- data mapping directives
- parallel construcst/instructions
- intrinsic functions (reductions)
- official subset...

HPF 2.0, October 19, 1996

- hopefully the same language!
- specification fixes
- features deleted... since in ISO Fortran!
- features moved as approved extensions
- new approved extensions developped
- new feature included: REDUCTION
- no more subset

HPF Programming Model

- 2 factors influence parallel performance
  - available parallelism
  - needed communications
- HPF tunes the program:
  - parallel constructs
  - data mapping directives

Presentation of the language 2.0

- data mapping directives
  - data parallel application are data-driven
  - communications cost is prohibitive
  - emphasize data mapping
- parallelism!
  - implicit parallelism
  - parallel loopS
  - intrinsics
- other issues

Directives

- considered as Fortran comments
- can be declarative or executable
- can expand on several lines
  with F90 continuations &
- spaces are meaningful!

```
      program directive
!hpf$ here is a directive
      print *, 'hello'
*hpf$ another directive &
chpf$   spawned on 2 lines
      end
```

Data Mapping Model

- Group lead by Guy L. Steele Jr
  - was Thinking Machine Corp.
  - then at SUN for JAVA
- Aims
  - specify the mapping of arrays
  - onto the processors of a DM machine.
  - understandable intuitive for users
  - optimisable for compilers
  - suitable to applications

2/3 level mapping

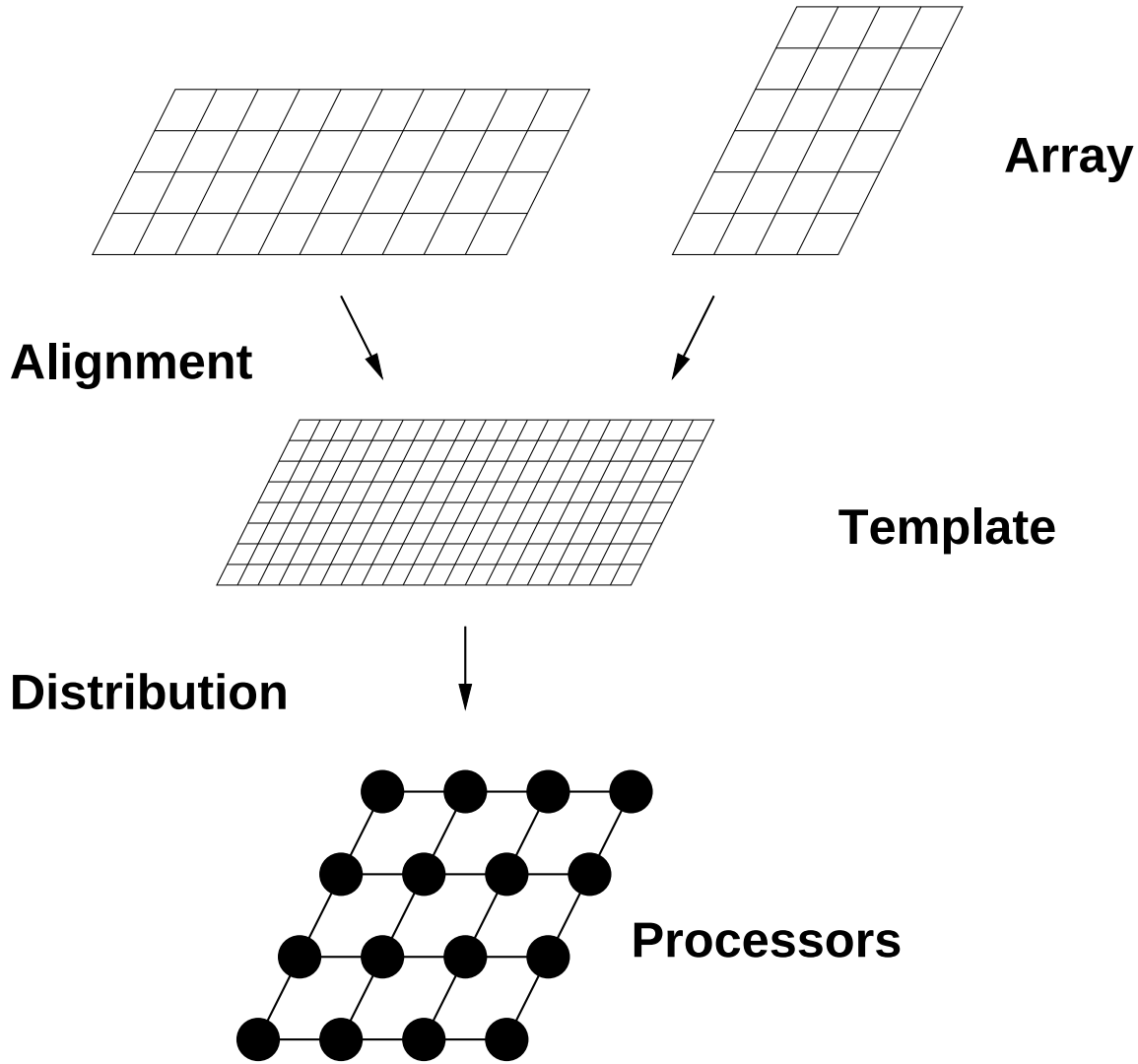
- Arrays are aligned with Templates
- Templates are distributed on Processors
- Processors may be mapped on physical processors

Only the 2 first levels are specified and used

4 directives:

ALIGN TEMPLATE DISTRIBUTE PROCESSORS

HPF Mapping Model


Array
Alignment
Template
Distribution
Processors

2 Level Mapping

load balancing and reduced communications

- Affine Alignment: ALIGN
  - relative data mapping

    'this array must be mapped as this one'
  - close to the application/problem

    decide data locality
- Regular distribution: DISTRIBUTE
  - distribute data on the processors, BLOCK or CYCLIC
  - close to the machine/architecture

Templates

TEMPLATE is a declaration directive

- array of nothing
- not allocated!
- used as an alignment target
- an array can be used as a template

```
!hpf$ template t(100), t2(n,m)
!hpf$ dimension(n,n), template :: &
!hpf$                     domain
```

Alignment

ALIGN is a declaration directive

- arrays are aligned with templates
- 2 array elements aligned with the same template element will be on the same processor
- data locality

```
!hpf$ ALIGN A(I,...) WITH T(affine(I),...)
```

Constraints

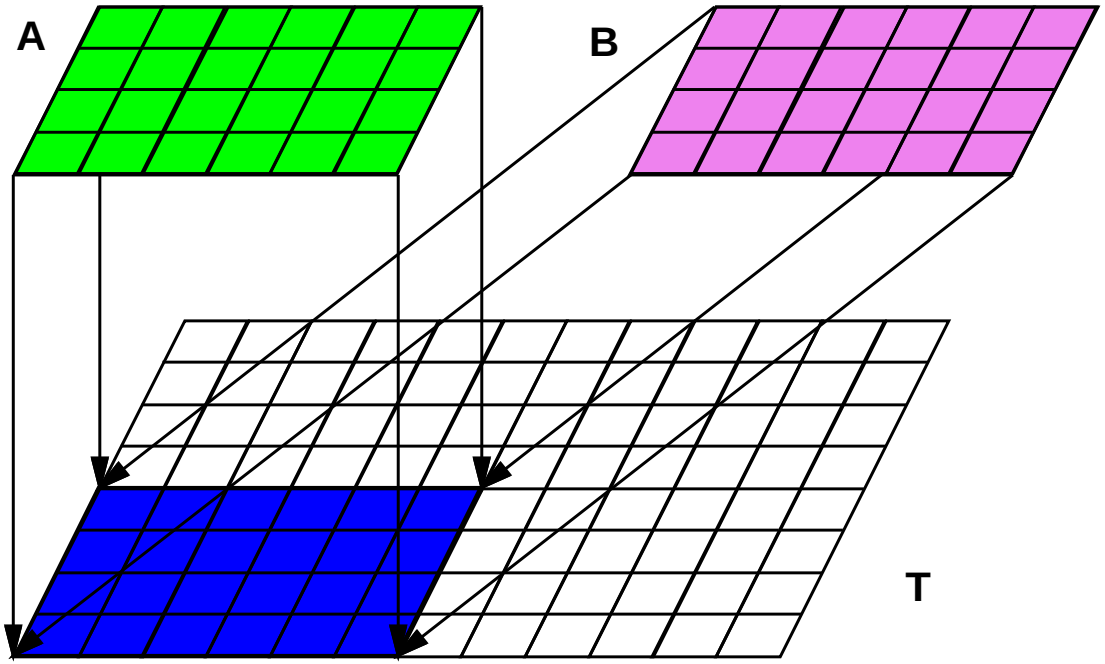
- Affine function
  - one dummy variable `a*I+b`
  - `a` and `b` are integers
- one use of dummy variables

  NO: `ALIGN A(I) WITH T(I,I)...`

Alignment expressivity

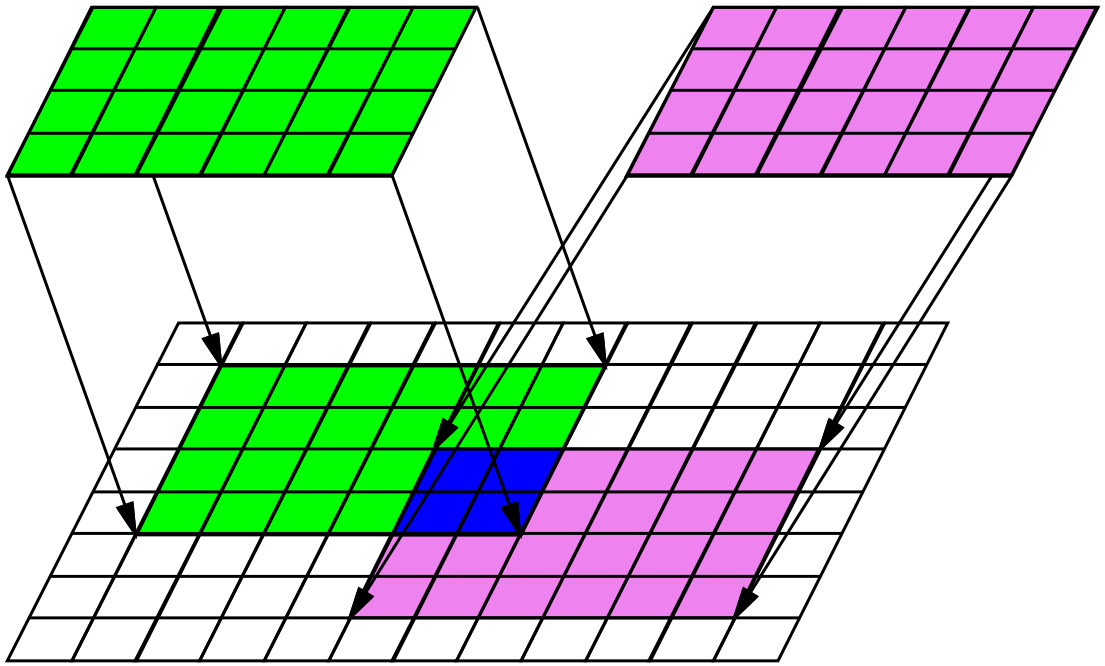
- direct mapping
- transposition
- reversing
- replication
- collapsing
- ...

Direct mapping



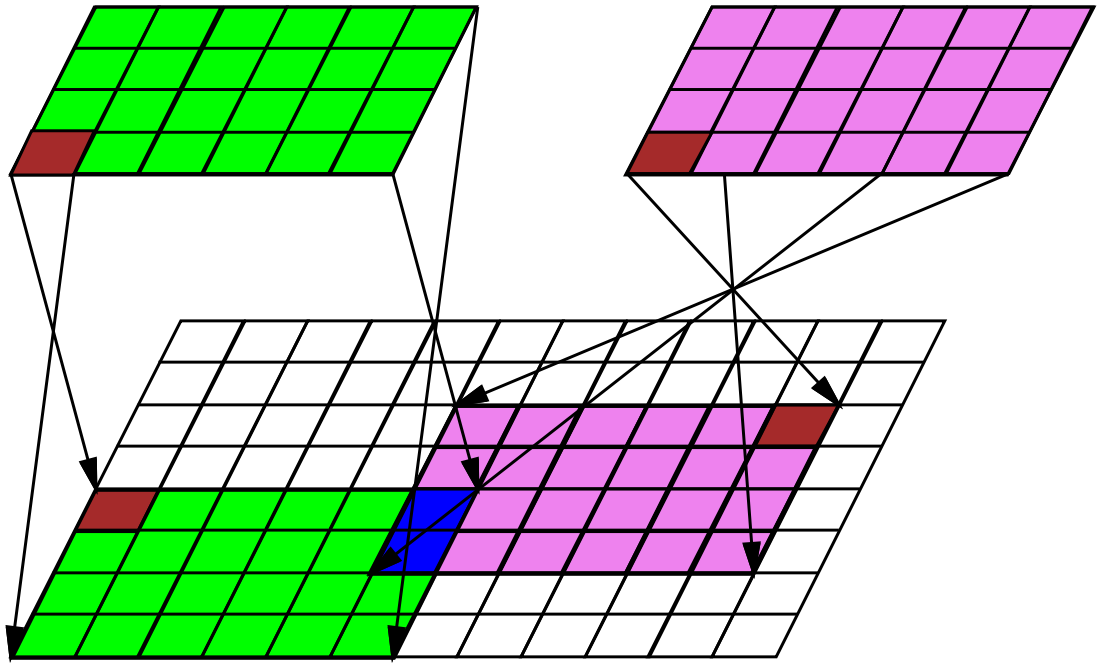

```
align A(i,j) with T(i,j)
align B with T
```

Shift



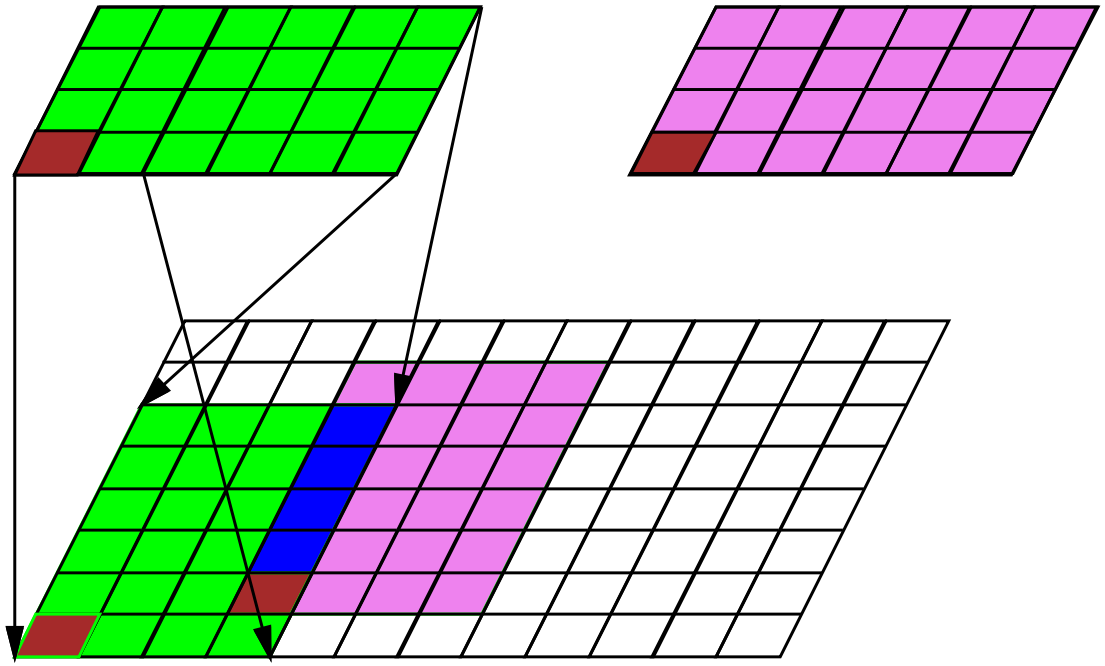
```
align A(i,j) with T(i+1,j+3)
align B(:,:) with T(6:11,2:5)
```

Reversing



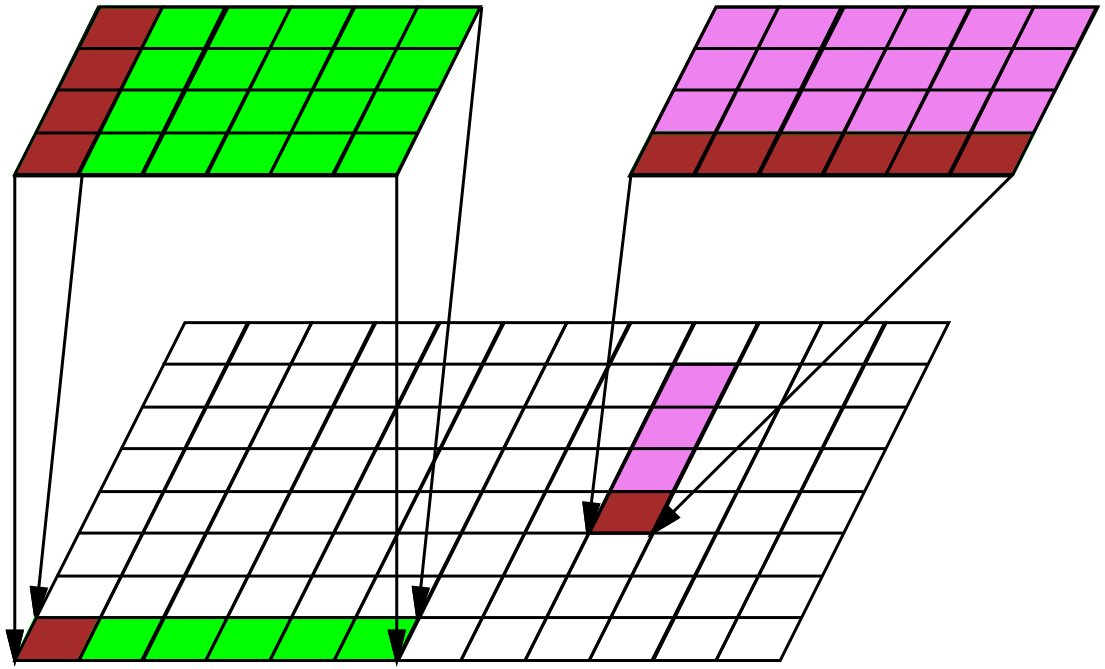
```
align A(i,j) with T(i,5-j)
align B(:,:) with T(11:6:-1,6:3:-1)
```

Transposition



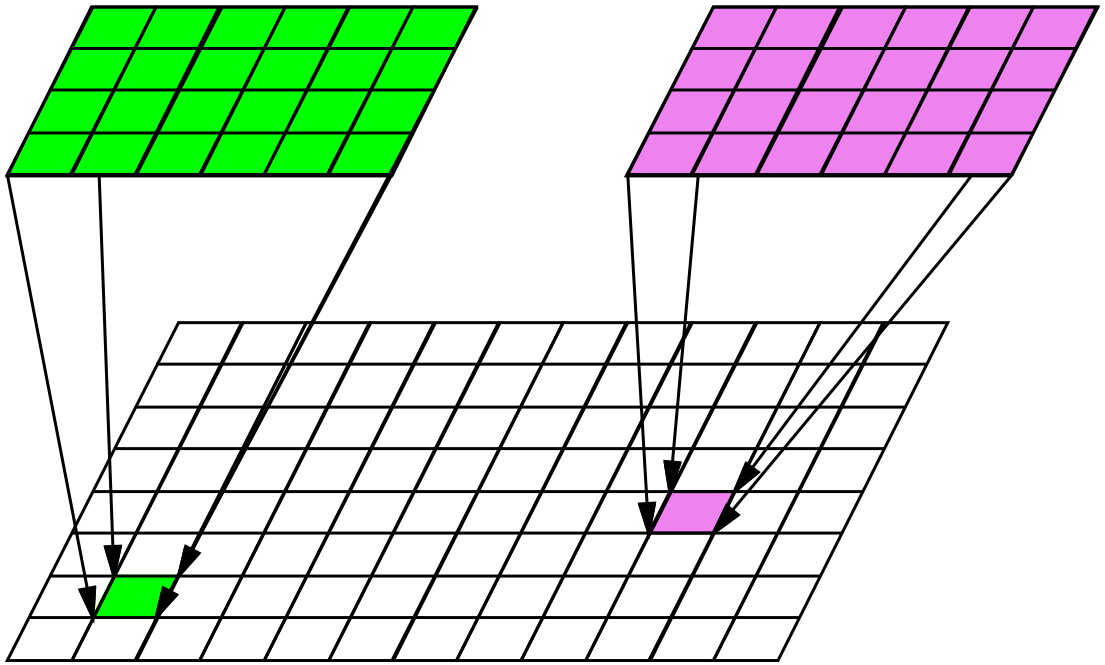
```
align A(i,j) with T(j,i)
align B(i,j) with T(j+3,i+1)
```

Collapsing



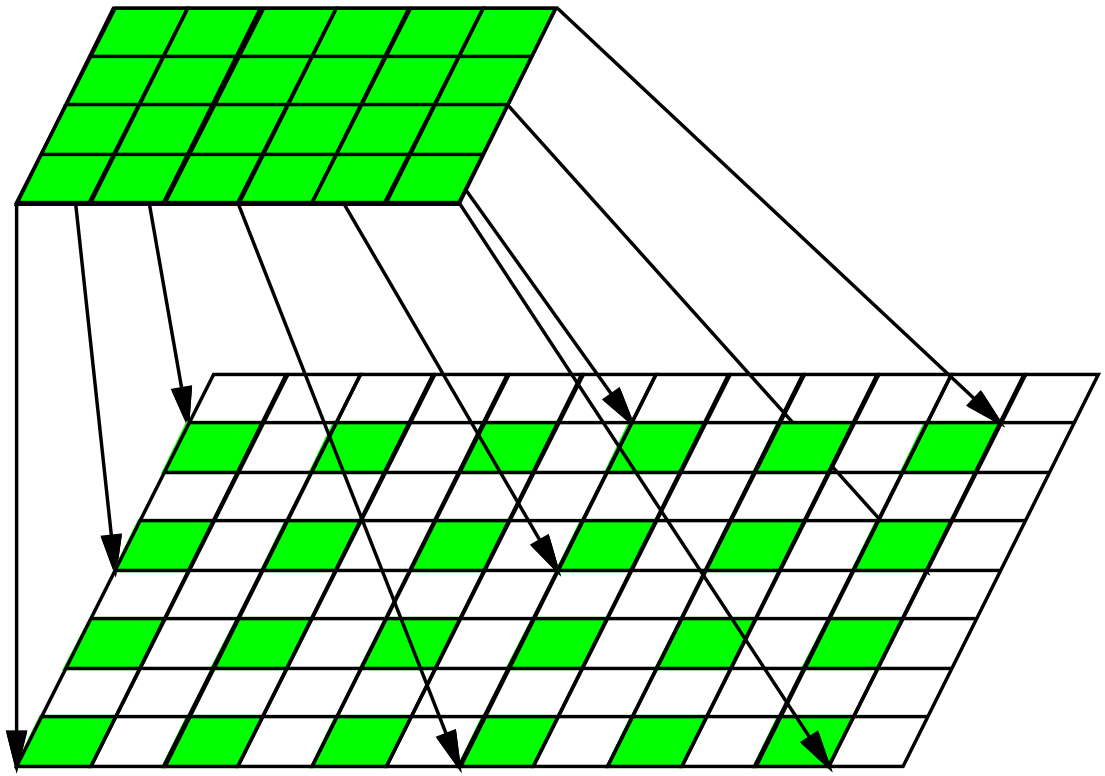
```
align A(i,*) with T(i,1)
align B(*,:) with T(9,4:7)
```

More collapsing



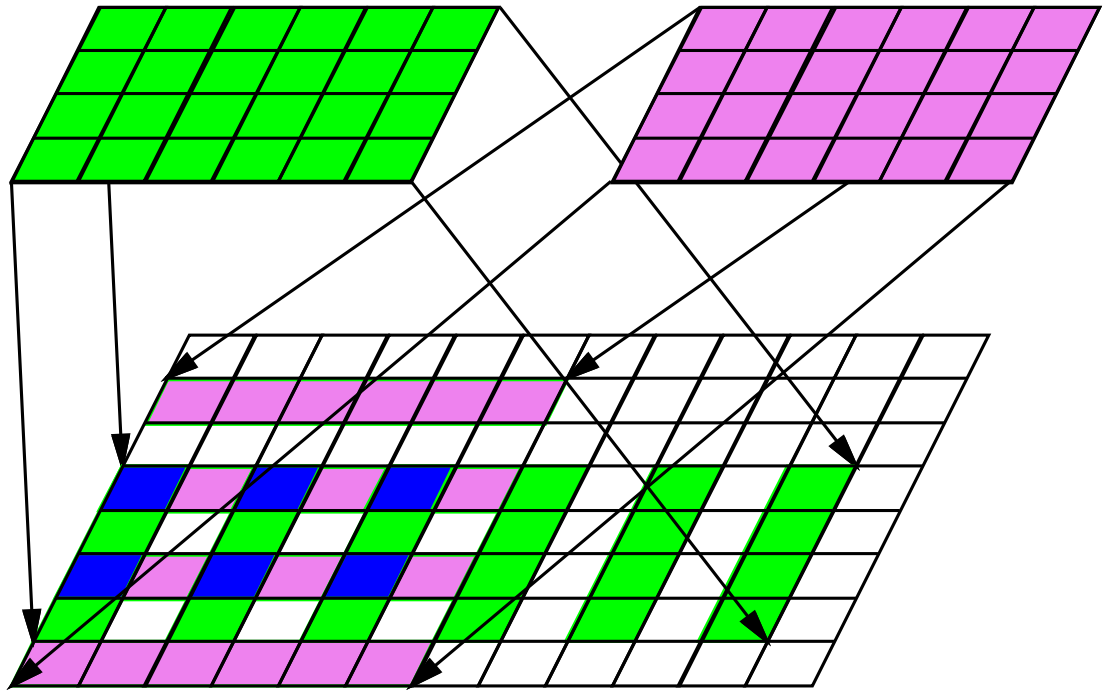
```
align A(*,*) with T(2,2)
align B(*,*) with T(10,4)
```

Scattering



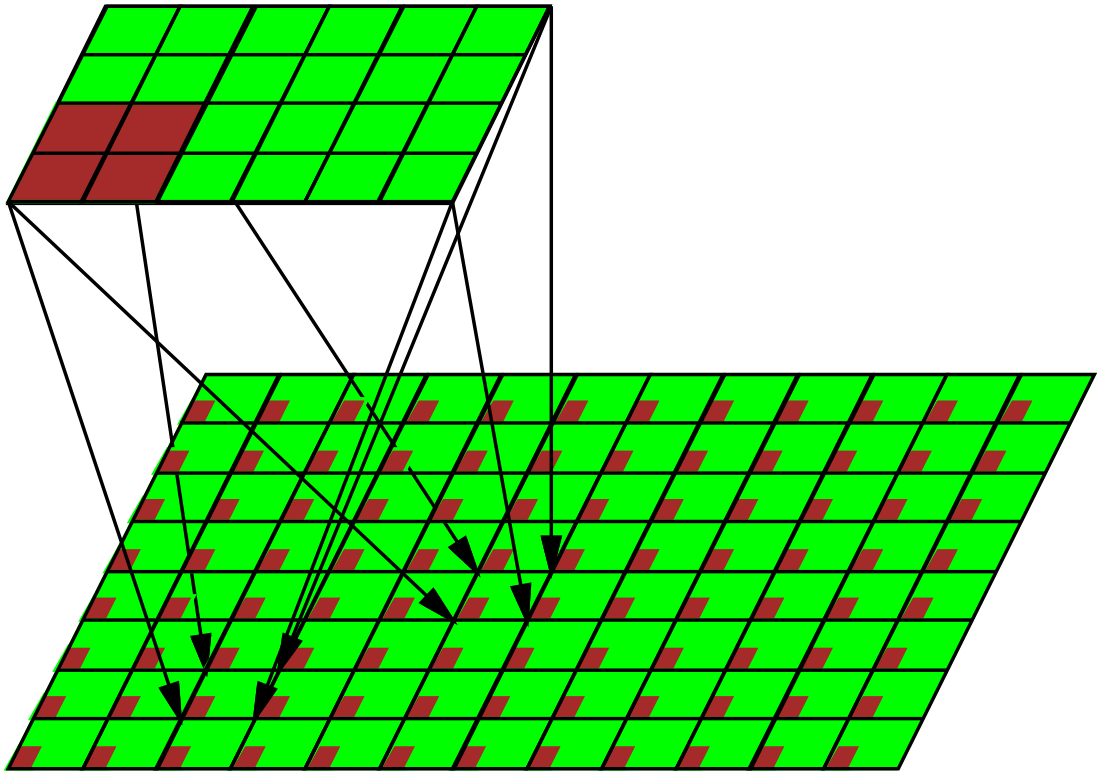
```
align A(i,j) with T(2*i-1,2*j-1)
```

Scattering



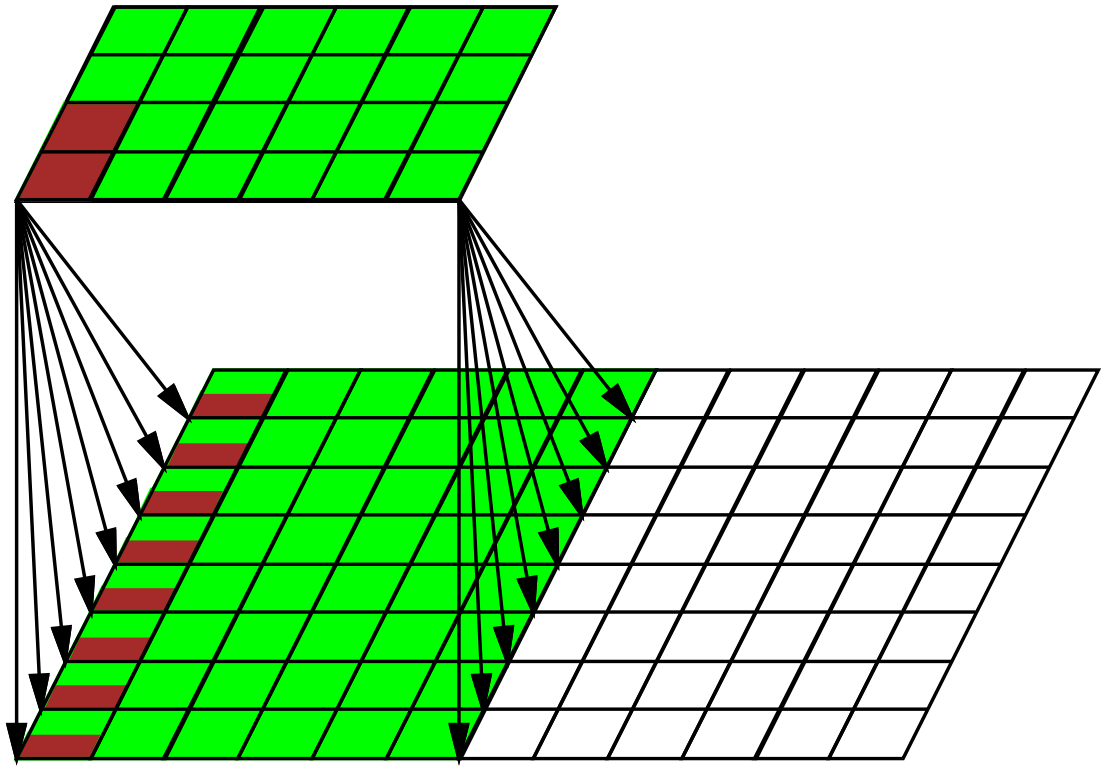
```
align A(i,j) with T(2*i-1,j+1)
align B(i,j) with T(i,2*j-1)
```

Replication



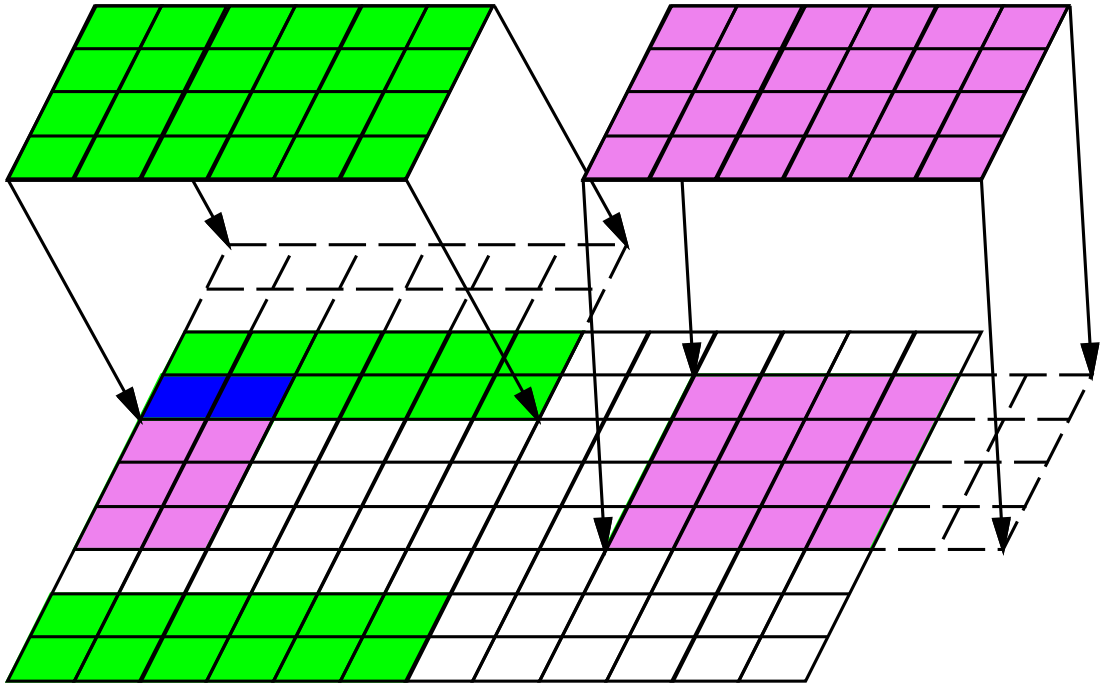
```
align A(*,*) with T(*,*)
```

Partial replication



```
align A(:,*) with T(:,*)
```

NO overlaps



```
align A(i,j) with T(i,j+6)    ! non
align B(:,:) with T(9:,4:7)   ! non
```

Processors

PROCESSORS is a declaration directive

- declares abstract processor arrays
- often correspond to actual processors
- can query the number of processors:

  `NUMBER_OF_PROCESSORS()`

```
!hpf$ PROCESSORS P(10), Q(4,4)
!hpf$ PROCESSORS Z(2:3,8)
!hpf$ PROCESSORS R(NUMBER_OF_PROCESSORS())
```

Distribution

DISTRIBUTE is a declaration directive

- templates are distributed on processors
- template elements are grouped together on a processor element

```
!hpf$ DISTRIBUTE T(BLOCK,*) ONTO P
!hpf$ DISTRIBUTE T2(CYCLIC,CYCLIC(10)) ONTO Q
```

Use of BLOCK and CYCLIC

BLOCK regular distribution for neighbor comms

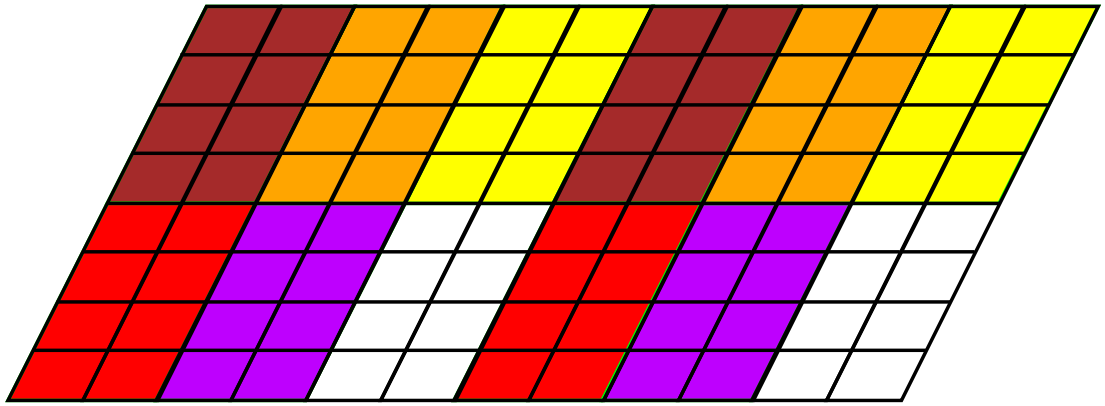
typical of stencil computations

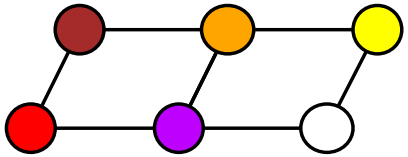
BLOCK(n) adjust block size

CYCLIC distribute load

CYCLIC(n) both load balance and neightbor comms.

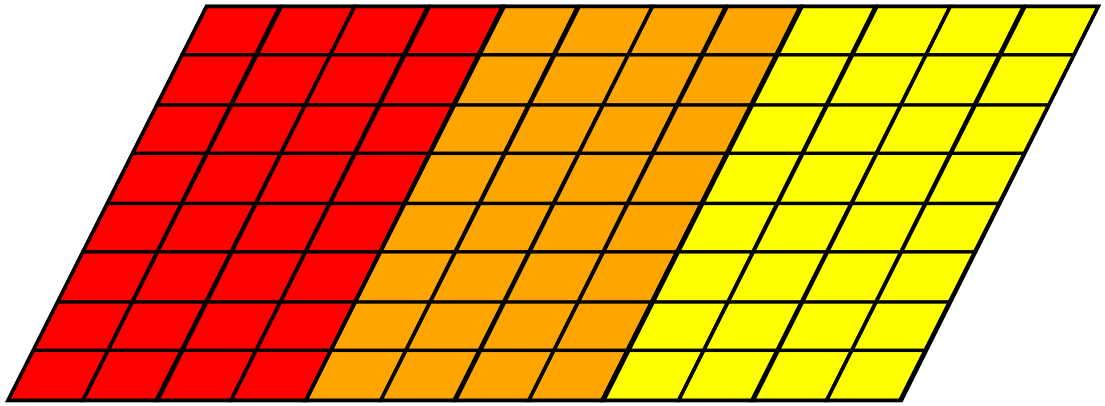
Distribution





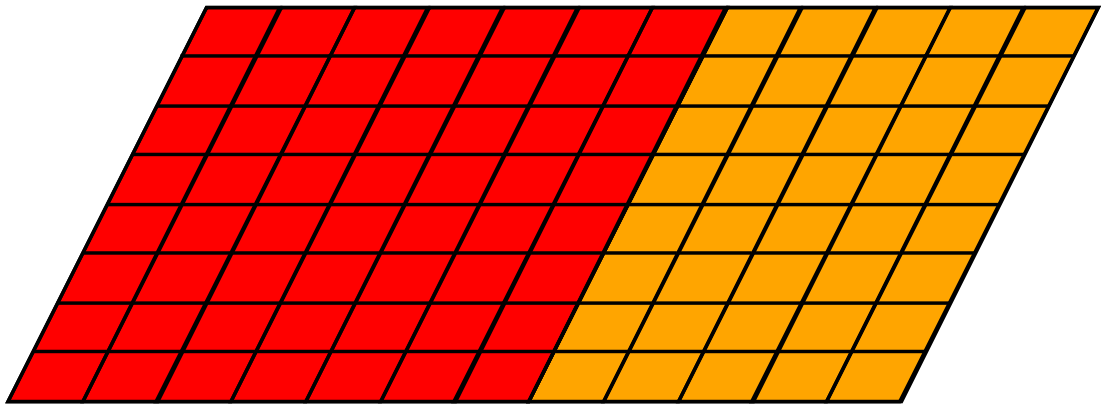
```
distribute T(...,...) onto P
```

Block distribution



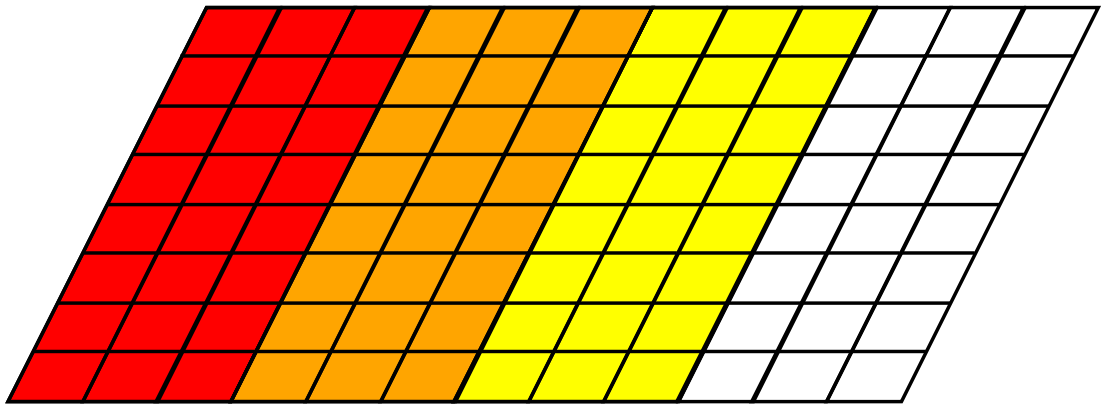
```
distribute T(block,*) onto P
```

Block(n) distribution



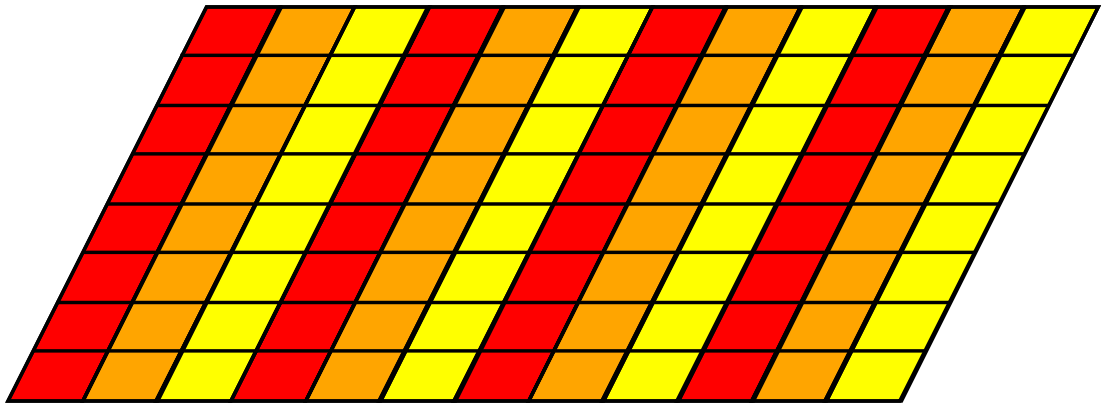
```
distribute T(block(7),*) onto P
```

Block(n) distribution



```
distribute T(block(3),*) onto P  ! non
```

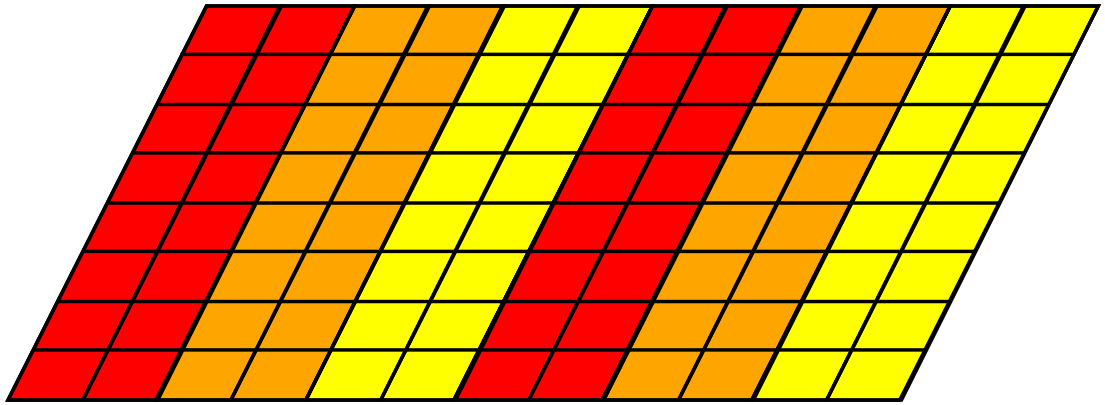
Cyclic distribution



n

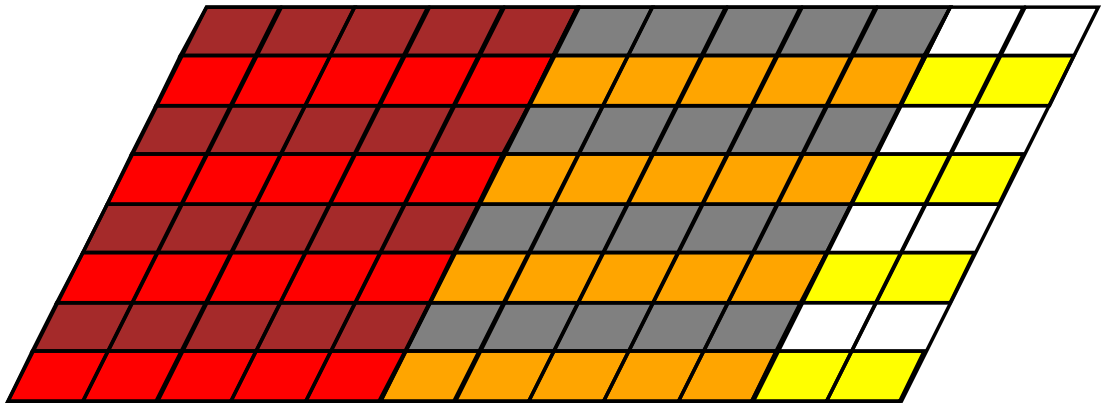
```
distribute T(cyclic,*) onto P
```

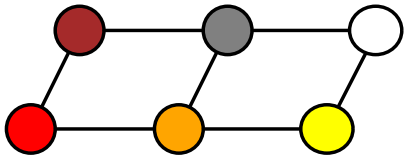
Cyclic(n) distribution



```
distribute T(cyclic(2),*) onto P
```

Multi dimensional distributions





```
distribute T(block(5),cyclic(1)) onto P
```

The template debate...

Heavy discussions on the mailing list

- Hans Zima, Vienna
  - Vienna Fortran does not include templates
  - templates do not improve expressivity
  - one more non user friendly concept
- Ken Kennedy, Rice
  - Fortran D includes similar decompositions
  - they can be larger than any array
  - easier port, clearer for the user

Other mapping directives

REALIGN executable, dynamic alignment

REDISTRIBUTE executable, dynamic distribution

DYNAMIC attribute to an object which can be RE...

Moved as approved extensions...

More about HPF mapping directives

- Fortran 90 syntactic sugar
- use of intrinsics in directives
- default mapping
- some strange mappings
- HPF mapping limits

Syntactic sugar

```
!hpf$ processors, dimension(10) :: PROC
!hpf$ template, &
!hpf$   distribute(block) onto PROC, &
!hpf$   dimension(100), &
!hpf$   dynamic :: T1,T2
!hpf$ dynamic, align (:) with T1(:) :: A, B
```

Use of intrinsics

- `NUMBER_OF_PROCESSORS()`

  number of physical processors available

- `PROCESSORS_SHAPE()`

  processor number of dimension

```
!hpf$ PROCESSORS P(NUMBER_OF_PROCESSORS()), &
!hpf$   AP(NUMBER_OF_PROCESSORS()/10,10)
```

Default mappings

- direct alignement
- 1D vector of physical processors
- block distribution

```
!hpf$ align A with B
!hpf$ distribute B
```

HPF mapping limits

No arbitrary irregular mapping

```
! From Vienna Fortran
      ALIGN A(I) WITH B(f(I))
```

- f is an arbitrary function
- compiler optimizations?

Mapping at subprogram interfaces

prescriptive mapping to inforce (with a remapping)

descriptive specify an expected mapping

transcriptive unspecified mapping

Prescriptive mapping

- just like a static mapping

```
      subroutine CALCUL(A)
      real A(100,100)
!hpf$ distribute A(block,block)
      ...
```

- remapping on entry and exit (?)
- runtime cost

Descriptive mapping

- similar to prescriptive
- assertion to the compiler
- favor optimisations
- all callers must conform, otherwise errors!

```
!hpf$ align A with * T
!hpf$ distribute * T(block)
```

Transcriptive mapping

- no information for optimisations?
- template inheritance concept
- syntaxes

```
      subroutine CALCUL(A)
      real A(100,100)
!hpf$ distribute A *
!hpf$ inherit A
```

Conclusion to HPF mapping

- close to Fortran D
- replication
- suits arrays
- should favor optimizations
- directives = advice!

So what?

- array distributed on processors
- the data are shared between them...
- we have now to share the computation
- without changing the semantics
- let's look for parallelism!
  - array expressions
  - parallel loops
  - dataparallel loops
  - and sequential loops...

3 loops in HPF

- Group of Charles Koelbel, Rice University
- sequential do loop: DO
- parallel loop: DO + INDEPENDENT

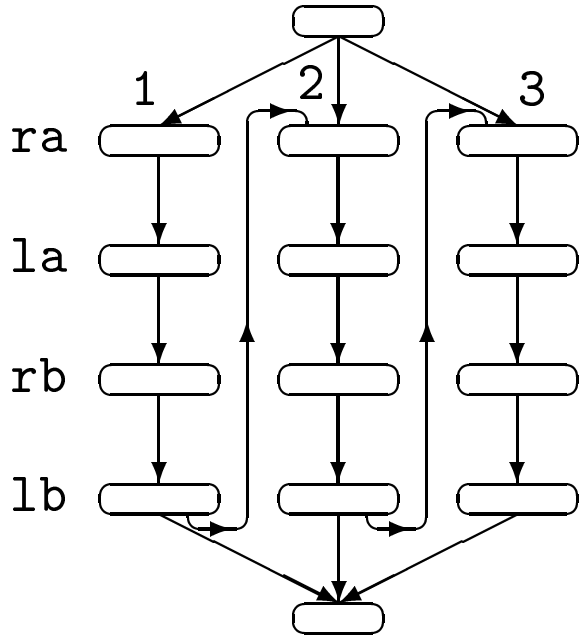
  iterations to not dependent of the other
- dataparallel loop: FORALL

3 loops in HPF

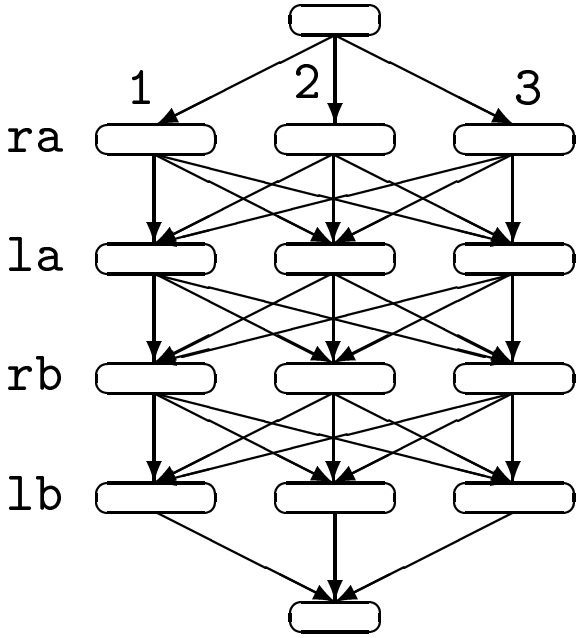
```
      do i=1, 10
         a(i) = a(i-1) + a(i) + a(i+1)
      enddo

!hpf$ independent
      do i=1, 10
         a(i) = 1.0
      enddo

      forall (i=1:10)
        a(i) = a(i-1) + a(i) + a(i+1)
      end forall
```

DO et FORALL

```
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```

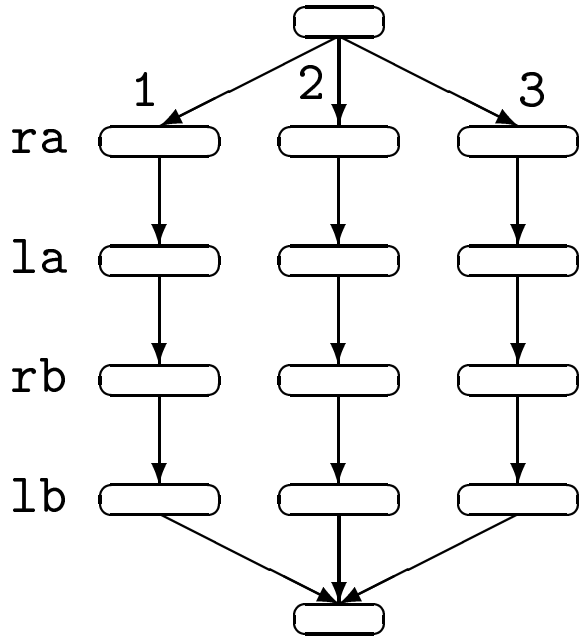



```
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

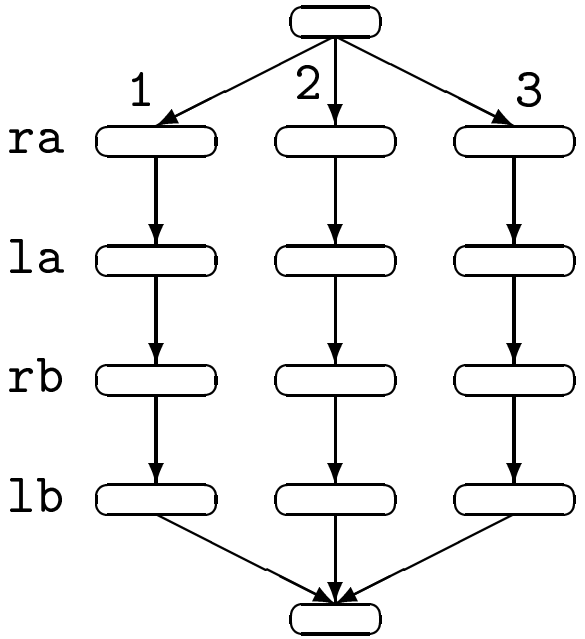


INDEPENDENT

```
!HPF$ INDEPENDENT
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```




```
!HPF$ INDEPENDENT
FORALL( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```



FORALL: 3 dataparallel loops

- implicit with array sections/expressions

```
A = B + C           ! matrices
D(1:20) = D(2:21) ! shift
```

- generalization: FORALL statement

  index explicitely named

```
FORALL (i=10:n, j=10:m) a(i,j) = 0.0
```

- extension: FORALL construct

FORALL statement

- generalize array sections
- dataparallel semantics (SIMD)
- only one assignment
- optional mask
- can call PURE functions

```
FORALL(i=1:n, ..., mask) &
  assignment
```

```
! transpose
forall(i=1:n, j=1:n) x(i,j) = y(j,i)

forall(i=1:n, y(i).ne.0.0) &
  x(i) = 1.0 / y(i)

! relaxation
forall(i=2:n-1) &
  x(i) = 0.5 * x(i) + 0.25 *(x(i-1)+x(i+1))

forall(i=1:n, y(i).ge.0.0) &
  x(i) = SQRT(y(i))
```

FORALL construct

- extension of FORALL
- includes
  - assignements
  - WHEREs or FORALLs

```
FORALL(i=1:n,...,mask)
  WHERE or FORALL or assigns...
ENDFORALL
```

```
FORALL(i=1:n)
  A(i,i) = SQRT(A(i,i))
  B(i) = LOG(B(i,i)) + i
ENDFORALL

FORALL(i=1:n,j=1:n)
  WHERE (b(i,j)>0.0)
    a(i,j) = 1.0
  ELSEWHERE
    a(i,j) = -1.0
  ENDWHERE
ENDFORALL
```

PURE attribute to functions

- user functions which behaves like intrinsics
  - no I/Os, no STOP, no PAUSE...
  - no access to commons or save data...
- must be declared as such
  - in the function interface
  - in the function itself

```
pure function f(x)
  real :: f
  real, intent(in):: x
  f = (4.3*x+2.1)*x+1.7
end function
```

INDEPENDENT directive

- a parallel loop!
- iterations can be executed in any order
- can specify private variables

  variables private to iterations
- HPF 2.0: can specify reductions
- can apply to DO and FORALL

```
!hpf$ INDEPENDENT, NEW(var-list), REDUCTION(...)
      DO i=1,n ... / FORALL(i=1:n)
```

Examples of INDEPENDENT

```
!hpf$ INDEPENDENT
      DO i=1, n
        A(P(i)) = B(i) ! P is a permutation
      ENDDO

!hpf$ INDEPENDENT, NEW(x)
      DO i=1, n
        x = A(i)**3
        B(i) = x*x + x
      ENDDO
```

```
! squere matrix multiplication
!hpf$ INDEPENDENT, NEW(j)
      DO i=1, n
!hpf$   INDEPENDENT, NEW(k,t)
        DO j=1, n
          t = 0.0
!hpf$     INDEPENDENT, REDUCTION(t)
          DO k=1, n
            t = t + A(i,k)*B(k,j)
          ENDDO
          C(i,j) = t
        ENDDO
      ENDDO
```

Efficient parallel loop?

- assertion about parallel semantics
- compiler may not know how to exploit it!
- additionnal constraints helps:
  - block distributions
  - all data accessed in an iteration are aligned
  - access to neighbor processors
  - remote data stored in overlap areas

Parallel library functions

- reductions applied on arrays
  - Fortran 90: SUM PROD ALL MAXVAL...
  - HPF/F95: ALL PARITY IALL MAXLOC...
  - include options (DIM, MASK...)
- scans...
  - `_SCATTER`
  - `_PREFIX`
  - `_SUFFIX`
- others: `SORT_UP SORT_DOWN GRADE_DOWN GRADE_UP`

Reductions

associative/commutative operations

parallelism induce numerical issues...

- Fortran 90
  - arithmetic: SUM (+) PROD (*)
  - logical: ALL (.and.) ANY (.or.) COUNT (.true.)
  - order: MAXVAL (max) MINVAL (min)
  - location: MAXLOC MINLOC
- HPF
  - logical: PARITY (.xor.)
  - bitwise: IALL IANY IPARITY

SCATTER functions

- reordering in Fortran 90
- based on permutations
- extended with _SCATTER
- how to combine indexed elements...

```
! A   = [ 10 20 30 ]
! V   = [ 1 2 3 4 ]
! IDX = [ 3 2 1 ] ! IDX is a permutation
      V[IDX] = A
! V   = [ 30 20 10 4 ]
```

SCATTER functions...

- base array to be modified
- contributing array
- integer index arrays
  - as many as DIM(base)
  - conformant to contributing array
- optionnal mask (logical array)

```
! A    = [ 10. 20. 30. 40. ]
! IDX  = [ 1 2 1 2 ]
! BASE = [ 1. 2. 3. ]
      X = SUM_SCATTER(A, BASE, IDX)
! X    = [ 41. 62. 3. ]
```

PREFIX and SUFFIX functions

- accumulation of elements
- forward or backward
- optional MASK, SEGMENT, DIM, EXCLUSIVE

```
! A = [ 1. 2. 3. 4. ]
      S = SUM_PREFIX(A)
! S = [ 1. 3. 6. 10. ]
      P = PROD_SUFFIX(A)
! P = [ 24. 24. 12. 4. ]
```

PREFIX and DIM option

- DIM: operation performed on it
- else array is linearized...

```
! A = [ 1 2 3 ]
!     [ 4 5 6 ]
  S = SUM_PREFIX(A, DIM=2)   ! S = [ 1 3  6 ]
                             !     [ 4 9 15 ]
  P = PROD_SUFFIX(A, DIM=1) ! P = [ 1 10 18 ]
                             !     [ 4  5  6 ]
  T = SUM_PREFIX(A)          ! T = [ 1  7 15 ]
                             !     [ 5 12 21 ]
```

PREFIX options

- MASK: element contributes if true
- EXCLUSIVE: current element does not contribute
- SEGMENT: grouping on linearized array

```
! S = [ T T F F F T T F ]
! A = [ 1 2 3 4 5 6 7 8 ]
      B = SUM_PREFIX(A,SEGMENT=S)
! B = [ 1 3 3 7 12 6 13 8 ]
      C = SUM_PREFIX(A,MASK=S)
! C = [ 1 3 3 3 3 9 16 16 ]
      D = SUM_PREFIX(A,MASK=S,EXCLUSIVE=.true.)
! D = [ 0 1 0 3 7 0 6 0 ]
```

options and array expressions

- MASK, SEGMENT, indexes

  can be the result of array op.

```
! A = [ 0 1 8 4 7 9 ]
!     [ 2 5 6 3 1 7 ]
      S = SUM_PREFIX(A, MASK=A.gt.4)
! AM = [ 0 0 8 0 7 9 ]
!      [ 0 5 6 0 0 7 ]
! S  = [ 0 0 13 19 26 35 ]
!      [ 0 5 19 19 26 42 ]
```

Rationale for scans

- SCATTER, PREFIX, SUFFIX
- quite specialized operations
- usefull to some applications
- parallel version can be developped
- but not in HPF!
- thus made available in the library

Sort functions

efficient parallel implementations

- compute permutations
  - `GRADE_UP`
  - `GRADE_DOWN`
- performs the sort
  - `SORT_UP`
  - `SORT_DOWN`

GRADE example

```
! A = [ -1 5 -3 4 ]
!     [  2 1  0 9 ]
! GRADE_UP(A) = [ 1 3 ]
!               [ 1 1 ]
!               [ 2 3 ]
!               [ 2 2 ]
!               [ 2 1 ]
!               [ 1 4 ]
!               ...
```

Miscelaneous scalar functions

- ILEN: number of bits needed
- LEADZ: number leading zero bits
- POPCNT: number of bits to 1
- POPPAR: 1 if POPCNT is even, 0 if odd

HPF inquiry functions

- a set of functions to query about array mappings
- non portable wrt Fortran 90
- examples :
  - `HPF_ALIGNMENT`
  - `HPF_DISTRIBUTION`
  - `HPF_TEMPLATE`

Conclusion about the library

- not portable wrt Fortran 90
- performance depends on implementation
- use them if appropriate!

Extrinsic procedures

- ability to call routine in
  - another language (C, F77, Fortran, HPF-lite)
  - another parallel paradigm (MPI, ...)
- routine called with different conventions:
  - on one pe with all data (serial)
  - on pes... (global, standard HPF!)
  - on every pe with local data (local)
  - ...

Syntax for extrinsics

- explicit interface in caller
- special declaration in the callee
- with fortran 90 keyword: EXTRINSIC

```
INTERFACE
  EXTRINSIC(HPF_LOCAL) SUBROUTINE FOO(A)
    REAL, DIMENSION(:), INTENT(INOUT):: A
  END SUBROUTINE
END INTERFACE

!hpf$ distribute X(BLOCK)
  CALL FOO(X)
```

function definition

```
 EXTRINSIC(HPF_LOCAL) SUBROUTINE FOO(A)
   REAL, DIMENSION(:):: A
   INTEGER I
! query about A LOCAL declaration...
   DO I=LBOUND(A), UBOUND(A)
     ...
   ENDDO
 END SUBROUTINE
```

F77 serial model

- call a serial routine in Fortran 77
- for instance, a special I/O routine for X display
- available in ADAPTOR

HPF serial model

- call a serial routine in restricted HPF
- needed data collected before the call
- use Fortran 90 intent declarations!

HPF local model

- routine coded in restricted HPF!
- no data distribution!
- can use message passing such as MPI
- standard inquiry about local data
- inquery about data mapping (HPF 2.0 Ap. Ext.)

  `MY_PROCESSOR()`

  `GLOBAL_TO_LOCAL()`

HPF 2.0 approved extensions

Many proposal which are not incorporated

- data mapping
- data and task parallelism
- asynchroneous I/Os
- library extensions
- more extrinsics...

Approved extensions for data mapping

- REALIGN, REDISTRIBUTE, DYNAMIC...
- mappings on processor subsets
  `DISTRIBUTE T(BLOCK) ONTO P(2:4)`
- RANGE: list of mappings a subroutine might accept...
- SHADOW: overlap width declaration
  `DISTRIBUTE(BLOCK)` SHADOW(1:2):: A,
- ...

Approved extensions for parallelism

- ON directive to locate computations

```
!hpf$ on home(a(1)), new(x) begin
      x = ...
      a(1) = x*x
!hpf$ end on
```

- TASK_REGION + ON directive
- active processors
- ...

Port of ONDE24 to HPF

- 2D acoustic wave propagation
- from IFP: French Oil Institute
- 1000 lines of Fortran 77
- target: IBM SP2
- stencil-based application!
- direct resolution over time
- order 2 in time, order 4 in space

Should be a dream!

ONDE24 initial performance

- HPF = High Performance!
- code profile: gprof/tprof...
- can use RS6000 performance counters
- result: 40 Mflop/s
- processor peak performance is 250 Mflop/s
- what is the trouble?

Improving the kernel loop

- cache behavior is ok.
- expression structure:
  - processor needs 4 independent fma: a*b+c
  - more fma can be extracted
  - balanced tree ⇒ more parallelism
- result: 80 Mflop/s!
- with unrolling and scheduling: 110 Mflop/s

  but will not suit HPF...

ONDE24 declaration cleaning

- larger declaration than used...
- impact on distribution and load balance:
  - few processors used if BLOCK
  - or more communications if CYCLIC(n)
- U and V are the big arrays
- dynamic allocation? not F77...

Initial declarations

```
INTEGER    NPMAX, NTMAX, NBTRAC
PARAMETER (NPMAX     = 809)
PARAMETER (NTMAX     = 4000)
PARAMETER (NBTRAC  = 50)
REAL*8    U      (NPMAX+1,NPMAX,2)
REAL*8    V      (NPMAX+1,NPMAX)
REAL*8    B      (NPMAX+1,3,3)
REAL*8    UB     (NPMAX+1,3)
REAL*8    UINT   (NPMAX)
REAL*8    SISMO (NBTRAC,NTMAX)
INTEGER   NP, NT
```

Array declarations

- static allocation
- bounds moved as parameters
- runtime check to match problem
- actual bounds needed in directives
- hence MUST be compile-time constants

Simplified declarations

```
INTEGER NP, NT, NBTRAC
PARAMETER (NP = 800)
PARAMETER (NT = 1000)
PARAMETER (NBTRAC = 50)
REAL*8    U(NP,NP,2)
REAL*8    V(NP,NP)
REAL*8    B(NP,3,3)
REAL*8    UB(NP,3)
REAL*8    UINT(NP)
REAL*8    SISMO(NBTRAC,NT)
```

ONDE24 Data Mapping

- 1D, 2D and 3D arrays are used
- align data used together...
- must examine actual uses!
- then only ONE distribution

```
!hpf$ TEMPLATE DOMAIN(NP,NP)
```

U and V arrays

- scan all uses
- U(I,J,*) and neighbors used with V(I,J)
- suggest direct alignement!

```
!hpf$ ALIGN U(I,J,*) WITH DOMAIN(I,J)
!hpf$ ALIGN V(I,J) WITH DOMAIN(I,J)
!
! or
!
!hpf$ ALIGN U, V WITH DOMAIN
```

UINT vector first use

```
      DO 92 I = 3,NP-2
         UINT(I) = B(I,1,1)*(U(I,NP,KP)+U(I,NP-1,KP))
         [...]
     &    + B(I,1,3)*(U(I+1,NP-1,KP)+U(I+1,NP,KP)+
     &                U(I-1,NP-1,KP)+U(I-1,NP,KP))
92    CONTINUE
      DO 93 I = 3,NP-2
        U(I,NP,KP) = UINT(I)
93    CONTINUE

!hpf$ ALIGN UINT(I) WITH DOMAIN(I,NP)
```

Second use

```
      DO 94 I = 3,NP-2
         UINT(I) = B(I,2,1)*(U(NP,I,KP)+U(NP-1,I,KP))
         [...]
     &     + B(I,2,3)*(U(NP-1,I+1,KP)+U(NP,I+1,KP) +
     &                 U(NP-1,I-1,KP)+U(NP,I-1,KP))
94    CONTINUE
      DO 95 I = 3,NP-2
        U(NP,I,KP) = UINT(I)
95    CONTINUE

!hpf$ ALIGN UINT(I) WITH DOMAIN(NP,I)
```

Third use

```
      DO 96 I = 3,NP-2
         UINT(I) = B(I,3,1)*(U(I,1,KP)+U(I,2,KP))
         [...]
     &    + B(I,3,3)*(U(I+1,2,KP)+U(I+1,1,KP)+
     &                U(I-1,2,KP)+U(I-1,1,KP))
96    CONTINUE
      DO 97 I = 3,NP-2
        U(I,1,KP) = UINT(I)
97    CONTINUE

!hpf$ ALIGN UINT(I) WITH DOMAIN(I,1)
```

UINT Mapping?

- not used elsewhere
- 3 different alignements?
- values directly reused!
- REALIGN?
  - usually not implemented
  - no need to move the data!
- this suggest 3 different arrays!

  UINTW UINTS UINTE (West, South, East)

UINT* Mapping

```
      REAL*8 UINTW(NP), UINTS(NP), UINTE(NP)
!hpf$ ALIGN UINTE(I) WITH DOMAIN(I,NP)
!hpf$ ALIGN UINTS(I) WITH DOMAIN(NP,I)
!hpf$ ALIGN UINTW(I) WITH DOMAIN(I,1)
```

UB use

```
    DO 71 I = 2,NP-1
     UB(I,1) = U(I,NP-1,KP)
     UB(I,2) = U(NP-1,I,KP)
 71  UB(I,3) = U(I,2,KP)

    DO 92 I = 3,NP-2
     UINTE(I) = U(I,NP,KP) .. U(I,NP-1,KP) .. UB(I,1)

!hpf$ ALIGN UB(I,1) WITH DOMAIN(I,NP-1) ???
!hpf$ ALIGN UB(I,2) WITH DOMAIN(NP-1,I) ???
!hpf$ ALIGN UB(I,3) WITH DOMAIN(I,2)    ???
```

UB Mapping

- dimension (NP,3)
- temporary store for borders in time loop
- cannot map dimensions separatly in HPF
- split UB: UBW, UBS, UBE

UB* Mapping

```
      REAL*8 UBE(NP), UBS(NP), UBW(NP)

!hpf$ ALIGN UBE(I) WITH DOMAIN(I,NP-1)
!hpf$ ALIGN UBS(I) WITH DOMAIN(NP-1,I)
!hpf$ ALIGN UBW(I) WITH DOMAIN(I,2)

      DO I = 2,NP-1
        UBE(I) = U(I,NP-1,KP)
        UBS(I) = U(NP-1,I,KP)
        UBW(I) = U(I,2,KP)
      ENDDO
```

B Mapping? first use

```
      REAL*8 B(NP,3,3)

      DO 41 I = 2,NP-2
          V(I,NP)   = 2/((1/V(I,NP))+(1/V(I,NP-1)))
          W         = V(I,NP)*DELTAT
          B(I,1,1) = 1 - (1/(W+H)) * ( 2*H + W*W/H)
          B(I,1,2) = 2*H/(W+H)
 41       B(I,1,3) = W*W/(2*H*(W+H))

!hpf$ ALIGN B(I,1,*) WITH DOMAIN(I,NP)
```

2nd and 3rd uses

```
      DO 42 I = 3,NP-2
          V(NP,I)   = 2/((1/V(NP,I))+(1/V(NP-1,I)))
          W         = V(NP,I)*DELTAT
          B(I,2,1) = 1 - (1/(W+H)) * ( 2*H + W*W/H)
          B(I,2,2) = 2*H/(W+H)
42        B(I,2,3) = W*W/(2*H*(W+H))

      DO 43 I = 2,NP-2
          V(I,1)   = 2/((1/V(I,1))+(1/V(I,2)))
          W         = V(I,1)*DELTAT
          B(I,3,1) = 1 - (1/(W+H)) * ( 2*H + W*W/H)
          B(I,3,2) = 2*H/(W+H)
43        B(I,3,3) = W*W/(2*H*(W+H))
```

B Mapping

- similar to UB!
- split in UBW, UBS, UBW!

```
      REAL*8 BE(NP,3), BS(NP,3), BW(NP,3)

!hpf$ ALIGN BE(I,*) WITH DOMAIN(I,NP)
!hpf$ ALIGN BS(I,*) WITH DOMAIN(NP,I)
!hpf$ ALIGN BW(I,*) WITH DOMAIN(I,1)
```

SISMO Mapping

- stores received signal
- sismograph placed on the field

```
      REAL*8 SISMO(NBTRAC,NT)

      DO 84 I = 1,NBTRAC
84       SISMO(I,1) = 0.0

      DO 81 I = 1,NBTRAC
81       SISMO(I,N) = U(IS, NP - NBTRAC + I, KP)

!hpf$ ALIGN SISMO(I,*) WITH DOMAIN(IS,(NP-NBTRAC)+I) ???
```

SISMO Mapping

- array shift is legal!
- but IS is a runtime constant...
- depth of source of in field

```
!hpf$ ALIGN SISMO(I,*) WITH DOMAIN(1,(NP-NBTRAC)+I)
```

1D or 2D distribution on 4 processors?

(BLOCK,*)

- shorter inner loop
- 4*3*n communications per iteration

(*,BLOCK)

- longer inner loop
- 4*3*n communications per iteration

(BLOCK,BLOCK)

- smaller local array
- 4*2*n communications per iteration
- 50% less communications!

ONDE24 HPF Mapping

- alignment and distribution done!
- now let's find some parallelism!
- must scan all computations and loops...
- parallel if independent of execution order
- Berstein's conditions:
  - no data conflicts (R/W, W/R, W/W)
  - between DISTINCTS iterations
- requires dependence analyses

ONDE24 Kernel computation

Potential conflicts

- Write of U(I,J,KP)
- Reads of U array elements:
  - U(I,J,KP): no, same iteration
  - U(I,J,KM): idem
  - U(I[+-][12],J,KM): no if KM and KP differs
  - U(I,J[+-][12],KM): no if KM and KP differs

KM and KP flip-flop

```
      INTEGER   KM,KP
      KM = 1
      KP = 2
      DO 70 N = 2,NT
         ...
         KM = KP
         KP = 3 - KP
70    CONTINUE
```

Parallel kernel

```
!hpf$ INDEPENDENT, NEW(I)
      DO J = 3,NP-2
!hpf$  INDEPENDENT
       DO I = 3, NP-2
        U(I,J,KP) =
     $  (2.*U(I,J,KM)-U(I,J,KP))-V(I,J)*
     $  (60.*U(I,J,KM)+(((U(I+2,J,KM) + U(I-2,J,KM))
     $                  + (U(I,J-2,KM) + U(I,J+2,KM)))
     $           - 16. * ((U(I+1,J,KM) + U(I-1,J,KM))
     $                  + (U(I,J-1,KM) + U(I,J+1,KM)))))
        ENDDO
       ENDDO
```

Parallel loops in ONDE24

- 2 parallel loops found!
- time loop is not parallel (obvious)
- still 33 other loops to consider! really!
- initializations...
- other computations...
- I/O and statistics...

Obviously parallel loops

```
      DO 10 K = 1,2
        DO 10 J = 1,NP
          DO 10 I = 1,NP
10          U(I,J,K) = 0.0

      DO 84 I = 1,NBTRAC
84       SISMO(I,1) = 0.0

      DO 51 I = 3,NP-1
51       V(I,2) = (V(I,2)*DELTAT/H)**2
```

Other loops (1)

many such loop nests:

```
      DO 140 I = 3,NP-1
          U(I,2,KP) = 2*U(I,2,KM)-U(I,2,KP)-V(I,2) *
     &     (4*U(I,2,KM) - (U(I+1,2,KM)+U(I-1,2,KM)
     &                   + U(I,1,KM)+U(I,3,KM)))
 140  CONTINUE
```

Parallel version

- border stencils similar to kernel stencil

```
!hpf$ INDEPENDENT
      DO 140 I = 3,NP-1
          U(I,2,KP) = 2*U(I,2,KM)-U(I,2,KP)-V(I,2) *
     &     (4*U(I,2,KM) - (U(I+1,2,KM)+U(I-1,2,KM)
     &                   + U(I,1,KM)+U(I,3,KM)))
 140  CONTINUE
```

Other loops (2)

3 such loops:

```
      DO 41 I = 2,NP-2
         V(I,NP)   = 2/((1/V(I,NP))+(1/V(I,NP-1)))
         W         = V(I,NP)*DELTAT
         BE(I,1) = 1 - (1/(W+H)) * ( 2*H + W*W/H)
         BE(I,2) = 2*H/(W+H)
         BE(I,3) = W*W/(2*H*(W+H))
41    CONTINUE
```

Parallel version

- arrays are aligned
- W is private (defined and not used AFTER)

```
!hpf$ INDEPENDENT, NEW(W)
      DO 41 I = 2,NP-2
         V(I,NP)   = 2/((1/V(I,NP))+(1/V(I,NP-1)))
         W         = V(I,NP)*DELTAT
         BE(I,1) = 1 - (1/(W+H)) * ( 2*H + W*W/H)
         BE(I,2) = 2*H/(W+H)
         BE(I,3) = W*W/(2*H*(W+H))
41    CONTINUE
```

Other loops (3)

- fully parallel... but data not aligned!
- hence it is serialized!
- can be distributed

```
!hpf$ INDEPENDENT
      DO 71 I = 2,NP-1
         UBE(I) = U(I,NP-1,KP)
         UBS(I) = U(NP-1,I,KP)
         UBW(I) = U(I,2,KP)
71    CONTINUE
```

Efficient parallel version

```
!hpf$ INDEPENDENT
      DO 71 I = 2,NP-1
 71      UBE(I) = U(I,NP-1,KP)

!hpf$ INDEPENDENT
      DO 72 I = 2,NP-1
 72      UBS(I) = U(NP-1,I,KP)

!hpf$ INDEPENDENT
      DO 73 I = 2,NP-1
 73      UBW(I) = U(I,2,KP)
```

I/O Loops

- not parallel

```
      DO 21 I = 1,NP
        DO 22 J = 1,NP
          READ (3,*) V(I,J)
22      CONTINUE
21    CONTINUE
```

Reduction loops

```
      VMAX = V(1,1)
      VMIN = V(1,1)
      DO 112 I = 1,NP
        DO 122 J = 1,NP
          IF (V(I,J).LT.VMIN) THEN
            VMIN = V(I,J)
          ENDIF
          IF (V(I,J).GT.VMAX) THEN
            VMAX = V(I,J)
          ENDIF
122     CONTINUE
112   CONTINUE
```

Reduction loops

- REDUCTION directive?
  - operation must appear: MAX/MIN
  - not always available in compilers (HPF 2.0)
- Fortran 90 array intrinsics! MAXVAL/MINVAL

```
      VMIN = V(1,1)
!hpf$ INDEPENDENT, REDUCTION(VMIN)
      DO 112 J=1, NP
!hpf$   INDEPENDENT
        DO 112 I=1, NP ! fixed loop order
 112      VMIN = MIN(VMIN, V(I,J))


      VMIN = MINVAL(V)
```

Any other array accesses?

- many individual array accesses
  - signal source
  - initializations and computations in corners
- may lead to many small communications!
- may imply private scalars...
- hints:
  - ON HOME HPF 2.0 directive
  - 1 iteration parallel loop?

Signal source

```
W = F * (N * DELTAT - 1.0/F)
IF (W**2.LT.TMAX) THEN
  W = (W*PI)**2
  G0 = (1.0 - 2.0*W) * EXP (-W)
  U(IS,JS,KM) = U(IS,JS,KM) + G0 * 12*V(IS,JS) * H**2
ENDIF
```

Efficient version with HPF 2.0

```
!hpf$ ON HOME(V(IS,JS)), NEW(W, GO), BEGIN
      W = F * (N * DELTAT - 1.0/F)
      IF (W**2.LT.TMAX) THEN
        W = (W*PI)**2
        GO = (1.0 - 2.0*W) * EXP (-W)
        U(IS,JS,KM) = U(IS,JS,KM) + GO * 12*V(IS,JS) * H**2
      ENDIF
!hpf$ END ON
```

Efficient version with HPF 1.1?

```
!hpf$ INDEPENDENT, NEW(W,GO)
      DO I=1,1
      W = F * (N * DELTAT - 1.0/F)
      IF (W**2.LT.TMAX) THEN
        W = (W*PI)**2
        GO = (1.0 - 2.0*W) * EXP (-W)
        U(IS,JS,KM) = U(IS,JS,KM) + GO * 12*V(IS,JS) * H**2
      ENDIF
      ENDDO
```

Corner computation

```
!hpf$ ON HOME(V(NP-1,NP)), NEW(C1,W), BEGIN
      C1 = 4/((1/V(NP-1,NP))+(1/V(NP-1,NP-1)) +
     &         (1/V(NP-2,NP))+(1/V(NP-2,NP-1)) )
      W  = C1*DELTAT
      BE(NP-1,1) = 2*W/(4*W+3*H)
      BE(NP-1,2) = 3*H/(4*W+3*H)
      V(NP-1,NP) = C1
!hpf$ END ON
```

Summary

- all arrays are mapped
- all parallel loops are marked as such
- is that sufficient?
- no, I/O: sequentialized with elementary messages...

```
      DO 21 I = 1,NP
        DO 21 J = 1,NP
          READ (3,*) V(I,J)
21    CONTINUE
```

Efficient I/O Loops

```
! implied DO
      READ (3,*) ((V(I,J), J=1,NP), I=1, NP)

! array I/O
      READ (3,*) V(1:NP,1:NP)
      V = TRANSPOSE(V)
```

Performance

- with 3 compilers:
  - HPFC (my prototype!)
  - ADAPTOR (Thomas Brandes, GMD)
  - xlhpf (IBM)
- with three different versions...
- 250 Mflop/s on 4 processors

ONDE24 Conclusion

- alignment suits the parallelisation process
  - focus on relative accesses
  - some data need spliting
- HPF parallelism ok for stencils
  - cannot apply on unrolled version
  - limited to 80 Mflop/s/node
- details:
  - array scalar accesses...
  - I/O...

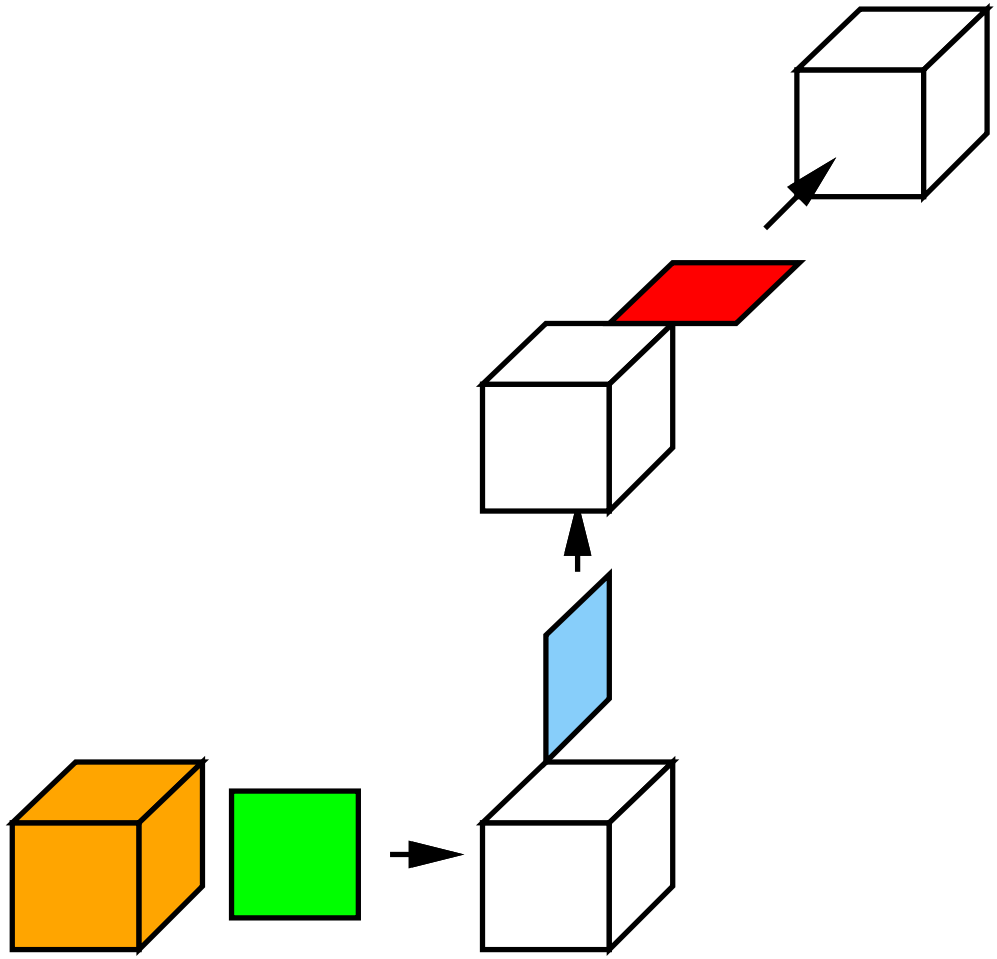
Tensrus

- solver for elliptic differential equations
- for Poisson, Helmoltz equations
- code features:
  - 3D domain, non uniform cart. grid
  - B-spline collocation method of odd order
  - Multipole expansion for boundary conditions
  - cost: N^4 (FFT is $N^3 \ln N$)
- Authors: Jean-Yves Berthou and Laurent Plagne
- CEA: French Nuclear Research Institute

Computation kernel

- basically matrix products!
- large 3D arrays
  - double float type
  - N*N*N, N=64, 128... 512
  - up to 1GB per matrix!

Tensrus: 3D tensor products at CEA



Initial code

```
SUBROUTINE tensrus(N, MX, MY, MZ, V, T)

REAL, DIMENSION(N,N)  :: MX, MY, MZ
REAL, DIMENSION(N,N,N):: T, V
REAL, DIMENSION(N,N,N):: G, F

DO i = 1, N
  DO j = 1, N
    DO k = 1, N
      F(i,j,k) = V(i,k,j)
    ENDDO
  ENDDO
ENDDO
```

```
DO i = 1, N
  DO j = 1, N
    DO c = 1, N
      G(i, j, c) = 0.0
      DO k = 1, N
        G(i,j,c) = G(i,j,c) + MZ(k,c) * F(i,j,k)
      ENDDO
    ENDDO
  ENDDO
ENDDO

! and 2 other products with MY and MZ
! final result stored in T
```

Improved sequential code

- reorder all matrix dimensions before product
- loop order take care of cache
- otherwise basically the same

Better sequential code

```
SUBROUTINE tensrus(N, MX, MY, MZ, V, T)

REAL, DIMENSION(N,N)  :: MX, MY, MZ
REAL, DIMENSION(N,N,N):: T, V
REAL, DIMENSION(N,N,N):: G, F

DO i = 1, N
  DO j = 1, N
    DO k = 1, N
      F(k,j,i) = V(i,j,k)
    ENDDO
  ENDDO
ENDDO
```

```
DO i = 1, N
  DO j = 1, N
    DO k = 1, N
      G(k,j,i) = 0.0
      DO c = 1, N
        G(k,j,i) = G(k,j,i) + MZ(c,k)*F(c,j,i)
      ENDDO
    ENDDO
  ENDDO
ENDDO

! and 2 other products with MY and MZ
! final result stored in T
```

HPF version with remappings (hpfc)

- descriptive mappings + interface in caller
- (BLOCK,BLOCK,BLOCK) distribution (better comms)
- redistributions AND reorderings...

  no commercial HPF compiler implements RE*...
- partial replication!
- no reductions: in local memory only

Parallel code

```
      SUBROUTINE tensrus(MX, MY, MZ, V, T)

      REAL, DIMENSION(N,N)  :: MX, MY, MZ
      REAL, DIMENSION(N,N,N):: T, V
! prescriptive/descriptive mappings:
!hpf$ TEMPLATE D(N,N,N)
!hpf$ PROCESSORS P(4,4,4)
!hpf$ DISTRIBUTE D(BLOCK,BLOCK,BLOCK) ONTO P

!hpf$ ALIGN MX(*,I) WITH D(I,*,*)
!hpf$ ALIGN MY(*,J) WITH D(*,J,*)
!hpf$ ALIGN MZ(*,K) WITH D(*,*,K)
!hpf$ ALIGN WITH D:: V, T
```

```
      REAL, DIMENSION(N,N,N):: G, F
!hpf$ ALIGN F(k,j,i), G(k,j,i) WITH D(i,j,k)
!hpf$ DYNAMIC F, G
! reordering
!hpf$ INDEPENDENT
      DO i = 1, N
!hpf$   INDEPENDENT
        DO j = 1, N
!hpf$   INDEPENDENT
          DO k = 1, N
            F(k,j,i) = V(i,j,k)
          ENDDO
        ENDDO
      ENDDO
```

```
! redistribute with partial replication
!hpf$ REALIGN F(*,j,i) WITH D(i,j,*)
!hpf$ INDEPENDENT
      DO i = 1, N
!hpf$   INDEPENDENT
        DO j = 1, N
!hpf$     INDEPENDENT
          DO k = 1, N
            G(k,j,i) = 0.0
            DO c = 1, N
              G(k,j,i) = G(k,j,i) + MZ(c,k)*F(c,j,i)
            ENDDO
          ENDDO
        ENDDO
      ENDDO
```

CGS

- Conjugate Gradient with Neuman Preconditionning
- generated by some applications
- large sparse matrices involved...
- does it suits HPF?
- Catherine Gaudart and Jean-Yves Berthou, CEA

Computation kernel

```
! Y = D * X
      SUBROUTINE matvect(n, ndiag, irf, d, x, y)
      INTEGER n, ndiag, irf(n, ndiag)
      REAL*8 d(n, ndiag), x(n), y(n)

      DO 10 i = 1, n
 10     y(i) = 0.0

 20   DO 20 i = 1, n
        DO 20 j = 1, ndiag
 20       y(i) = y(i)+d(i,j)*x(irf(i,j))

      END
```

Comments about matvect

- n = 216,000, ndiag = 7
- optimization:
  - n and ndiag compile time constants!
  - loop unrolling?
  - array transposition?
- parallel on the first large dimension

Improved matvect

```
 SUBROUTINE matvect(irf, d, x, y)
 parameter (n=216000)
 INTEGER irf(7,n)
 REAL*8 d(7,n), x(n), y(n)
 DO i = 1, n
    y(i) = d(1,i)*x(irf(1,i))
$         + d(2,i)*x(irf(2,i))
$         + d(3,i)*x(irf(3,i))
$         + d(4,i)*x(irf(4,i))
$         + d(5,i)*x(irf(5,i))
$         + d(6,i)*x(irf(6,i))
$         + d(7,i)*x(irf(7,i))
 ENDDO
```

HPF version

- from call site and loops
  - Y and X BLOCK distributed
  - pointer passed, switch to F90 array sections?
- in matvect: prescriptive mappings
- indirect access to X: full replication

HPF version

```
      SUBROUTINE matvect(irf, d, x, y)
      parameter (n=216000)
      INTEGER irf(7,n)
      REAL*8 d(7,n), x(n), y(n)

!hpf$ TEMPLATE T(n)
!hpf$ DISTRIBUTE T(BLOCK)
!hpf$ ALIGN WITH T(i):: IRF(*,i), D(*,i), Y(i)
!hpf$ ALIGN WITH T(*):: X(*)

!hpf$ INDEPENDENT
      DO 10 i = 1, n
 10      y(i) = d(1,i)*x(irf(1,i)) + ...
```

Communication analyses

Let us assume p processors

- 216000/p elements per processor from X and Y
- 13*216000/p ops per processor in matvect
- thus at most 13*216000/p elements of X used
- remapping: ((p-1)/p)*216000 data transfered!
- p grows: constant communications, reduced ops!!
- useless elements transfered!!!
- p=4: 702 Kops/p, 162 Ktransfers/p
- p=32: 87 Kops/p, 209 Ktransfers/p

How to improve the situation?

- IRF does not change over matvect calls...
- inspect once to know which data to transfer
- execute the communications many times!
- a lot of reuse: really reduced communications!
- smaller X can be allocated
- implemented by hand in MPI, called from HPF

Conclusion about HPF

- efficiency?
  - for some applications
  - fine tunning useful
- portability?
  - if same compiler used, yes!
  - otherwise, different versions often needed...
- simplicity?
  - mappings require geometric training;-)
  - YES if compared to MPI!

HPF vs MPI

- Jean-Yves Berthou:
  - high degree of technicity in both
  - few hours vs few days
  - few weeks vs few months
- HPF always make sense as a first try
  - very good performance on some codes
  - helps understanding the application

HPF Future ?

- processor performance increases a lot
- shared memory architecture bottleneck: 8 processors
- some applications/institution requires large machines
- vendor support? only PGI?

Pointers on the web

- HPF: `http://dacnet.rice.edu/Depts/CRPC/HPFF/`
- ADAPTOR:
  `http://www.gmd.de/SCAI/lab/adaptor/`
- PGI: `http://www.pgroup.com/`
- HPFC: `http://www.cri.ensmp.fr/~coelho/hpfc`
- a tutorial: `http://qqq.npac.syr.edu/hpfa/`