Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.2 HPF - High Performance Fortran

Credits: Adam Marshall, University of Liverpool

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

3.2 HPF - High-Performance-Fortran

- 3.2.1 Geschichte
- 3.2.2 Konzeption
- 3.2.3 Daten-Verteilung (Data-Distribution)
- 3.2.4 Parallel-Verarbeitung (Data Parallel Execution)
- 3.2.5 Input/Output
- 3.2.6 Proceduren, Extrinsics, Intrinsics, Library

3.2.1 Geschichte

- Ausgang war die Gründung einer informalen Gruppe von interessierten Parteien aus Hochschulen und aus der Hardware- und Softwareindustrie:.
  High Performance Fortran Forum (HPFF)
- 1. Meeting auf der Supercompunting ‘91 in Houston, Tx.
- 1. Draft-Standard zur SUPERCOMPUTING ‘92
- 1. *de facto* Standard: HPF v1.0 im May 1993
- Derzeitiger Standard: HPF v2.0

3.2.2 Konzeption

- HPF ist eine Erweiterung von Fortran90/Fortran95, wobei das Ziel ist, sämtliche Kontrollstatements von HPF in den jeweiligen Fortranstandard zu integrieren.
  Z. B. ist ab Fortran95 das *‘Forall- Statement’* im Fortran-Standard enthalten.
- Programmiermodell
  - ▲ Single-Programm-Multiple-Data (SPMD); Daten-Parallelität; Message-Passing
  - ▲ Es gibt keine expliziten Message-Passing- und Synchronisationskonstrukte; dadurch u. U. weniger effizient als eine explizite Message-Passing-Programmierung, jedoch weniger fehlerbehaftet, “Kompakter” Code.

Konzeption

- HPF ist grundsätzlich sowohl für MIMD- als auch für SIMD-Architekturen geeignet.
- HPF ist ursprünglich für NORMA-Systeme konzipiert worden; ist aber auch auf UMA- und NUMA-Systemen einsetzbar, wobei bei UMA eine reine Fortran90(95)-Compilierung genügen sollte; bei NUMA könnten jedoch die Datenverteilungsfunktionen interessant sein.
- HPF-Compiler sollten so konstruiert sein, daß sie alle im *"draft"* enthaltenen Konstrukte wenigstens *"parsen"* können. Siehe jeweils Manuals!

Konzeption

- In einem HPF-Programm
  - wird ein (konzeptionelles) Netz von Prozessoren definiert, welches unabhängig von der tatsächlichen physikalischen Verbindungsstruktur sein kann: Virtuelle HPF-Maschine.
  - werden die Daten auf das Prozessornetz verteilt.
  - wird der parallele Programmfluß strukturiert.
- In einem HPF-Programm sind häufig syntaktisch verschiedene Schreibweisen möglich (Kompatibilität mit Fortran77). Im Rahmen dieser Vorlesung werden nur die Fortran90/95-Schreibweisen verwendet!

Konzeption

- Der HPF-Compiler parallelisiert
  - Anweisung in Fortran90/Fortran95-Array-Syntax
  - `WHERE`
  - `FORALL` (Fortran95)
    - “Paralleles DO” und mehr
  - `INDIPENDENT` Do Loops; `NEW(i)`
    - die Reihenfolge der Abarbeitung der Anweisungen innerhalb der Schleifen ist unbedeutend
  - `PURE` Procedures
    - Seiteneffektfreie Prozeduren
  - EXTRINSICS
  - HPF-LIBRARY-Module, HPF-INTRISICS
  - Keine Zeiger!!

Konzeption

- HPF Derektiven
  - spezifizieren eine virtuelle HPF-Maschine,
    `PROCESSORS`
  - verteilen die Daten auf dieser Maschine
    `DISTRIBUTE; ALIGN`
  - und verarbeiten die Daten parallel.
    `FORALL; INDIPENDENT; NEW(i); PURE;`
    `(WHERE)`
    `(F90/F95-Array-Syntax)`
-

Konzeption

- HPF-Direktiven
  - ermöglichen die Einbettung von HPF-Sprachkonstrukten in Standard-Fortran90/95.
    Syntax:

```
!HPF$ < hpf-direktive>
```

  - Ein Fortran90/95-Compiler interpretiert diese Direktiven als Kommentare ("!"); erzeugt also sequentiellen oder *"shared memory-parallel"* - Code.
  - Grundsätzlich ist es also möglich mit einem Quellcode sowohl sequentielle als auch parallele Problemlösungen zu implementieren, letztere wiederum ablauffähig auf UMA, NUMA und NORMA-Architekturen.

Konzeption

- Man unterscheidet
  - Deklarative Direktiven, zum Beispiel:

```
!HPF$ PROCESSORS, DIMENSION(4)      :: P1
!HPF$ DISTRIBUTE (BLOCK) ONTO P1    :: a
```

  - Ausführbare Direktiven, zum Beispiel

```
!HPF$ INDIPENDENT
      FORALL  (i = 1:j) a(i,j) = b(i)
```

© Claus-Uwe Linster – pps3.2_spmd_hpf: 19/3/98

3.2.3 Datenverteilung (Data-Distribution)

- Oberstes Ziel bei der Datenverteilung ist
  - die Minimisierung
    - der Kommunikation
    - und Synchronisation
  - sowie die möglichst gleichmäßige Auslastung der Prozessoren (Loadbalancing).
- Im Konfliktfall ist
  - bei NORMA-Systemen die Minimisierung von Kommunikation und Synchronisation,
  - bei UMA/NUMA-Systemen? eher der gleichmäßigen Auslastung, der Vorrang einzuräumen.

3.2.3.1 Processors Direktive

- Processors Directive

```
!HPF$ PROCESSORS, DIMENSION(4)     :: P1
```

(4 Prozessoren werden 1-dimensional angeordnet)

```
!HPF$ PROCESSORS, DIMENSION(2,2)   :: P2
```

(4 Prozessoren werden 2-dimensional angeordnet)

```
!HPF$ PROCESSORS, DIMENSION(2,2,2) :: P3
```

(8 Prozessoren werden 3-dimensional angeordnet)

- Alle Netze von Prozessoren in einem Programm **müssen** die gleiche Anzahl von Prozessoren enthalten. (Resource-Management)

Processors Direktive

- Sind zwei Prozessoranordnungen im Sinne der F90/95-Array-Syntax konform, dann sind die korrespondierenden Prozessoren auch die gleichen physikalischen Prozessoren.
- Definition eines skalaren Prozessors:

```
!HPF$ PROCESSORS :: P
```


3.2.3.2 Distribute (Mapping) Direktive

- Verteilung von Daten über das Prozessornetz
  - blockweise
  - zyklisch
  - nicht verteilen
- Beispiele

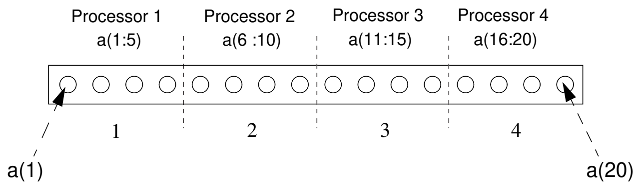
```
      REAL, DIMENSION(50)                          :: a
      REAL, DIMENSION (10,10)                      :: b,c,d
!HPF$ PROCESSORS, DIMENSION(4)                     :: P1
!HPF$ PROCESSORS, DIMENSION(2,2)                   :: P2
!HPF$ DISTRIBUTE (BLOCK) ONTO P1                   :: a !1-dim
!HPF$ DISTRIBUTE (CYCLIC, CYCLIC) ONTO P2:: b,c!2-dim
!HPF$ DISTRIBUTE (BLOCK,*) ONTO P2                 :: d !1-dim
```

3.2.3.3 Blockweise Verteilung

- Die blockweise Verteilung ist insbesondere dann gut geeignet, wenn die Bearbeitung “nachbarschaftsbezogen” strukturiert ist.
  Die Datenblöcke werden auf das Prozessornetz gleichverteilt, wobei der “letzte” Prozessor dann einen kleineren Teil der Daten bekommt, wenn die Datenumfang nicht ohne Rest durch die Anzahl der Prozessoren teilbar ist.

- Beispiel:

```
      REAL DIMENSION (20)                  :: a
!HPF$ PROCESSORS, DIMENSION(4)             :: P1
!HPF$ DISTRIBUTE  (BLOCK) ONTO P1          :: a  !1-dim
```

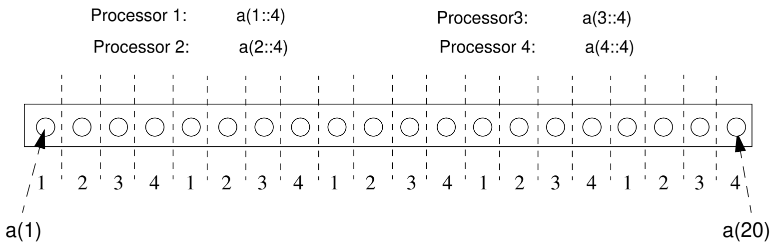


3.2.3.4 Zyklische Verteilung

- Die zyklische Verteilung bietet häufig die Möglichkeit das “Loadbalancing” günstig zu beeinflussen.
  Die Daten werden über die Prozessoren “round-robin” verteilt

- Beispiel:

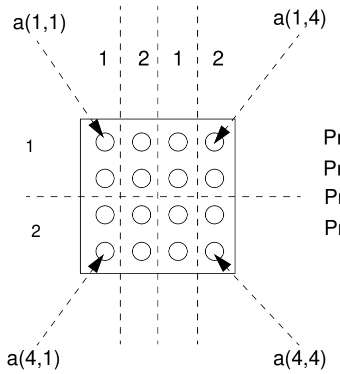
```
      REAL DIMENSION (20)         :: a
!HPF$ PROCESSORS, DIMENSION(4)    :: P1
!HPF$ DISTRIBUTE (CYCLIC) ONTO P1 :: a  !1-dim
```



3.2.3.5 2-D Verteilung block/zyklisch

- Beispiel

```
       PROGRAM block_cycle
       IMPLICIT NONE
       REAL DIMENSION (4,4)                   :: a,b,c
!HPF$  PROCESSORS, DIMENSION(2,2)             :: P2
!HPF$  DISTRIBUTE (BLOCK,CYCLE) ONTO P2       :: a,b,c
       b = 1; c = 1
       a = b + c
       END PROGRAM block_cycle
```



Processor P(1,1): a(1:2,1::2)
Processor P(2,1): a(3:4,1::2)
Processor P(1,2): a(1:2,2::2)
Processor P(2,2): a(3:4,2::2)

3.2.3.6 “*” - Verteilung

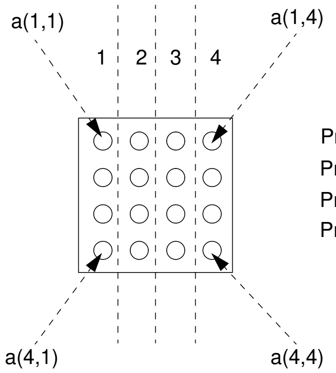
- Wird ein ”*” statt einer Verteilungsmethode angegeben, so bedeutet das, daß diese Dimension nicht verteilt werden soll. So hat man z. B. die Möglichkeit spalten- oder zeilenweise die Daten einer Matrix zu verteilen.

- Beispiel

```
       PROGRAM star
       IMPLICIT NONE
       REAL DIMENSION (4,4)                   :: a,b,c
!HPF$  PROCESSORS, DIMENSION(4)               :: P1
!HPF$  DISTRIBUTE (*,BLOCK) ONTO P1           :: a,b,c
       b = 1; c = 1
       a = b + c
       END PROGRAM star
```

“*” - Verteilung

- Beispiel(cont)



Processor 1: a(: ,1)
Processor 2: a(: ,2)
Processor 3: a(: ,3)
Processor 4: a(: ,4)

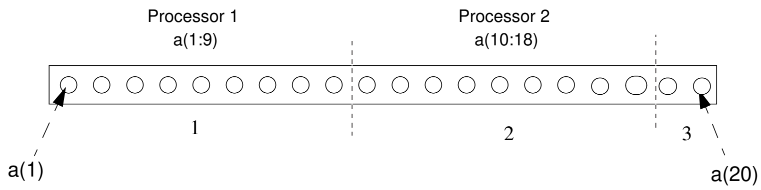
3.2.3.7 Weitere Verteilungsvarianten

- BLOCK(m)
  - Es wird blockweise verteilt; jedoch ist die Länge des Blocks durch m festgelegt.
    Diese Methode ist nützlich, wenn man nur einen Teil des definierten Netzes von Prozessoren verwenden will.
- CYCLIC(m)
  - Es werden zyklisch “Blöcke” der Länge m verteilt.
    Diese Methode führt zu “internem Programmoverhead”.
- Beispiel:

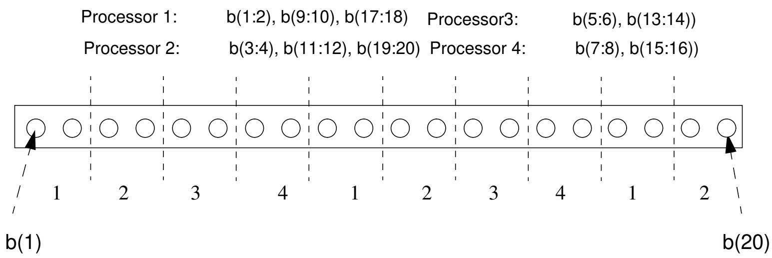
```
      REAL, DIMENSION(20)                  :: a,b
!HPF$ PROCESSORS, DIMENSION(4)             :: P1
!HPF$ DISTRIBUTE (BLOCK(9)) ONTO P1        :: a
!HPF$ DISTRIBUTE (CYCLIC(2)) ONTO P1       :: b
```

Weitere Verteilungsvarianten

BLOCK(9)



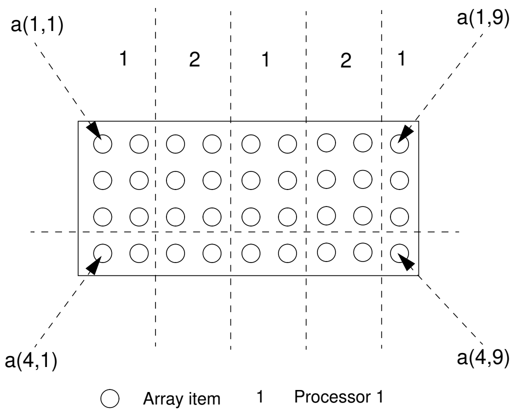
CYCLIC(2)



2D-Beispiel

```
      REAL, DIMENSION(4,9)                         :: a
!HPF$ PROCESSORS, DIMENSION(2)                     :: P
!HPF$ DISTRIBUTE (BLOCK(3),(CYCLIC(2)) ONTO P1 :: a
```

P(1,1):a(1:3,1:2),a(1:3,5:6),a(1:3),9:9).
P(2,1):a(4:4,1:2),a(4:4,5:6),a(4:4,9:9).
P(1,2):a(1:3,3:4),a(1:3,7:8).
P(2,2):a(4:4,3:4),a(4:4,7:8).



Distribute (Mapping) Direktive

- Allgemeines
  - fehlt das Schlüsselwort `ONTO` werden die Objekte auf ein “default” Prozessorgitter verteilt.

Distribute (Mapping) Direktive

- Skalare Variable werden in Kopien auf allen Prozessoren gehalten. Die Kohärenz ist gesichert;
  d. h. Kommunikationsaufwand und Synchronisationsverlust!
  Beispiel:

```
      REAL, DIMENSION(100,100)     :: x
      REAL                         :: scalar
!HPF$ DISTRIBUTE  (BLOCK,BLOCK)    :: x
      ...
      scalar = x(i,j)
```

Der Prozessor, auf dem x(i,j) plaziert ist, führt die Berechnung aus und schickt an alle anderen Prozessoren das Ergebnis (broadcast). Alle Prozessoren synchronisieren sich bis zum Abschluß der Operation. Will man das vermeiden, sollte man einen Vektor anlegen und die Elemente verteilen. Die Kohärenz ist dann Angelegenheit des Programmierers!

Distribute (Mapping) Direktive

- Beispiel für eine gute Verteilung:

```
!HPF$ DISTRIBUTE (BLOCK) ONTO P    :: a
!HPF$ DISTRIBUTE (CYCLIC) ONTO P   :: b
!HPF$ DISTRIBUTE (BLOCK) ONTO P    :: c !oder (CYCLIC)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO P  :: d !oder (*,CYCLIC)
      ...
      a(2:99) = a(:98) + a(3:))/2
      b(22:56) = 4.0*ATAN(1.0)
      c(:) = SUM(d, DIM=1)
```

- Weil
  - das 1. Statement eine typische “Nachbarschaftsstruktur” besitzt,
  - das 2. Statement nur einen Abschnitt von b enthält und
  - das 3. Statement jeweils eine Spalte von d zur Berechnung benötigt.

3.2.3.8 Verteilung und dynamische Speicherverwaltung

- Grundsätzlich wird genauso verfahren, wie bei der statischen Verteilung; jedoch werden die Daten erst dynamisch nach dem Aufruf des `ALLOCATE` - Statements verteilt.

- Beispiel

```
      REAL, ALLOCATABLE, DIMENSION (:,:)  :: a
!HPF$ PROCESSORS, DIMENSION(10,10)        :: P2
!HPF$ DISTRIBUTE (BLOCK,CYCLE) ONTO P2    :: a
      ...
      ALLOCATE(a(100,20), STAT = status)
      ...
      DEALOCATE (a)
      ...
```

Verteilung und dyn. Speicherverw. - Beispiel 1

```
   PROGRAM summe
   IMPLICIT NONE
   INTEGER :: i, j, n, ierr
   REAL, DIMENSION(:,:), ALLOCATABLE :: a
   REAL, DIMENSION(:), ALLOCATABLE   :: s
!HPF$ PROCESSORS DIMENSION(4)        :: P
!HPF$ DISTRIBUTE (*, CYCLIC) ONTO P  :: a
!HPF$ DISTRIBUTE (CYCLIC) ONTO P     :: s
   PRINT *, "Type in array size"
   READ *, n
   ALLOCATE(a(n,n),STAT=ierr)
   IF (ierr <= 0) THEN
     ALLOCATE(s(n),STAT=ierr)
     IF (ierr <= 0) THEN
       s = 0
       CALL RANDOM_NUMBER(a)
       DO j=1,n
         DO i=1,n
           s(j) = s(j) + a(i,j)
         END DO
       END DO
       PRINT*, s
       DEALLOCATE(s)
     END IF
     DEALLOCATE(a)
   END IF
   END PROGRAM summe
```


Verteilung und dyn. Speicherverw. - Beispiel 2

```
     PROGRAM produkt
     IMPLICIT NONE
     INTEGER :: i, j, n, ierr
     REAL, DIMENSION(:,:), ALLOCATABLE   ::a
     REAL, DIMENSION(:), ALLOCATABLE     :: pr
!HPF$PROCESSORS, DIMENSION(2,2)          :: Q
!HPF$DISTRIBUTE (CYCLIC,*) ONTO Q        :: a
!HPF$DISTRIBUTE (CYCLIC) ONTO Q          :: pr
     PRINT*, "Type in array size"
     READ*, n
     ALLOCATE(a(n,n),STAT=ierr)
     IF (ierr <= 0) THEN
       ALLOCATE(pr(n),STAT=ierr)
       IF (ierr <= 0) THEN
         pr = 1
         CALL RANDOM_NUMBER(a)
         DO i=1,n
           DO j=1,n
             pr(i) = pr(i) * a(i,j)
           END DO
         END DO
         PRINT*, pr
         DEALLOCATE(pr)
       END IF
         DEALLOCATE(a)
     END IF
     END PROGRAM produkt
```

3.2.3.9 Owner-Compute-Rule

- Owner-Compute-Rule
  Der Prozessor, dem das linke Element einer Zuweisung zugeordnet ist, führt die Rechenoperation aus. Dazu werden ggf. Elemente der rechten Seite vorher transferiert.

- Diese Regel wird von HPF-Compilern häufig angewendet, sie ist jedoch nicht Teil des Standards. Mit anderen Worten: aus Optimierungsgründen kann der Compiler auch andere Entscheidungen treffen.
  Beispiel: Alle Elemente der rechten Seite sind auf einem Prozessor plaziert, dann ist es sinnvoller erst auf diesem Prozessor die Berechnung vorzunehmen und das Ergebnis an den Prozessor der “linken Seite” zu senden.

3.2.3.10 Zuordnung (Alignment)

- Zuordnung heißt, Datenbereiche relativ zueinander zu verteilen. Dadurch kann
  - die Datenlokalität verstärkt,
  - die Kommunikation minimiert und
  - die Arbeitslast besser gleichverteilt werden,

  und zwar in einer übersichtlichen Weise. Darüberhinaus besteht die Möglichkeit
  - den Rang von Feldern zu erhöhen bzw. zu vermindern. (replicating or collapsing dimensions)

Zuordnung (Alignment)

- Syntax am Beispiel:

```
!HPF$ ALIGN (:,:) WITH t(:,:):: a,b,c
```

Alternativ

```
!HPF$ ALIGN (i,j) WITH t(i,j):: a,b,c
```

“:” im Align-Statement entsprechen dem Verständnis nach der F90/95-Array-Syntax.
Es handelt sich in diesen Beispielen um 2-dim Felder mit der Eigenschaft, daß jeweils die Elemente `a(i,j)`, `b(i,j)`, `c(i,j)` und `t(i,j)` dem gleichen Prozessoren zugeordnet werden.

- Zur Erinnerung, Array-Syntax:
`von:bis:incr`; default: `1:max:1`
`Anzahl “,” + 1 ist der Rang (Dimension)`

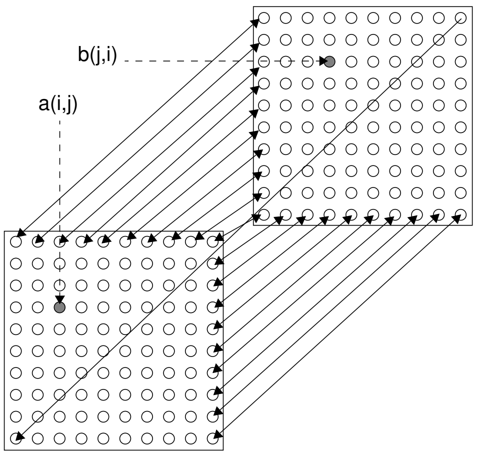
Zuordnung (Alignment)

- Transponierte Zuordnung
Zuordnung von Zeilen und Spalten, im 2-dim-Fall:

```
      REAL, DIMENSION(10,10)       :: a,b
!HPF$ ALIGN (i,:) WITH b(:,i)      :: a

! oder andere Schreibweise:

!HPF$ ALIGN a(i,:) WITH b(:,i)
!Zuordnung: Spalte a zu Zeile b
!HPF$ ALIGN a(:,j) WITH b(j,:)
!Zuordnung: Zeile a zu Spalte b
! oder auch
!HPF$ ALIGN a(i,j) WITH b(j,i)

!Das Ergebnis ist in allen Varianten das gleiche.
```

Zuordnung (Alignment)

- Die zugeordneten Teilfelder müssen konform (von gleicher Größe) sein.

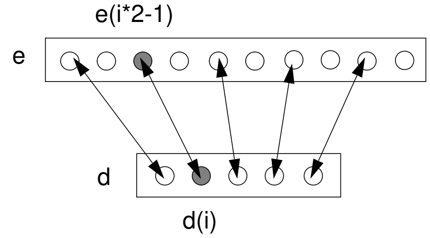


Zuordnung (Alignment)

- Strided Aligment (Beispiel):

```
        REAL, DIMENSION(5)      :: d
        REAL, DIMENSION(10)     :: e
!HPF$   ALIGN d(:) WITH e(1::2)
```

Die Zuordnung entspricht der folgenden Abbildung:



Alternative Schreibweise:

```
!HPF$ ALIGN d(i) WITH e(i*2-1)
```

Ideal für folgende Berechnung:

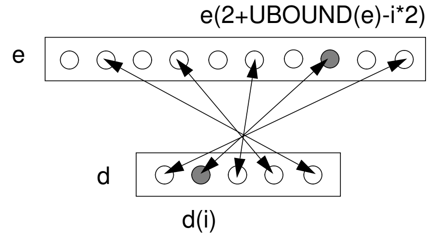
```
        d = d + e(::2)     ! alle Berechnungen lokal
```

Zuordnung (Alignment)

- Reversed Strided Aligment (Beispiel):

```
      REAL, DIMENSION(5)            :: d
      REAL, DIMENSION(10)           :: e
!HPF$ ALIGN d(:) WITH e(UBOUND(e)::-2)
```

Die Zuordnung entspricht der folgenden Abbildung:



Alternative Schreibweise:

```
!HPF$ ALIGN d(i) WITH e(2+UBOUND(e)-i*2)
```

Ideal für folgende Berechnung:

```
      d = d + e(10:1:-2)! alle Berechnungen lokal
```


Zuordnung (Alignment)

- Gegeben ist folgendes Problem:

```
   PROGRAM aufgabe_1
   IMPLICIT NONE
   REAL, DIMENSION(4)    :: c
   REAL, DIMENSION(8)    :: d
   REAL, DIMENSION(2)    :: e
   c= 1; d = 2
   e = d(::4) + c(::2)
   END PROGRAM aufgabe_1
```

Finden Sie eine Verteilung und Zuordnung derart, daß keine Kommunikation stattfinden muß.

3.2.3.11 Alignment und dynamische Speicherverwaltung

- Dynamisch angelegte Felder können - analog zur Verteilung - auch zugeordnet werden, wobei
  - die Zuordnung zum Zeitpunkt der Allocation erfolgt und
  - nur existierende Objekte einander zugeordnet werden können - Reihenfolge der Allocate-Statements bzw. der Parameter ist wichtig!
- Beispiele:

```
      REAL, DIMENSION(:), ALLOCATABLE:: a,b
!HPF$ ALIGN a(:) WITH b(:)
      ...
      ALLOCATE (b(100), STAT=status)
      ALLOCATE (a(100), STAT=status)      !RICHTIG
```


Alignment und dynamische Speicherverwaltung

- Beispiele:

```
      REAL, DIMENSION(:), ALLOCATABLE:: a,b
!HPF$ ALIGN a(:) WITH b(:)
      ...
      ALLOCATE (a(100), STAT=status)
      ALLOCATE (b(100), STAT=status)           !FALSCH
oder
      ALLOCATE (b(100),a(100), STAT=status)    !RICHTIG
oder
      ALLOCATE (a(100,b(100), STAT=status)     !FALSCH
```

Alignment und dynamische Speicherverwaltung

- Beispiele:

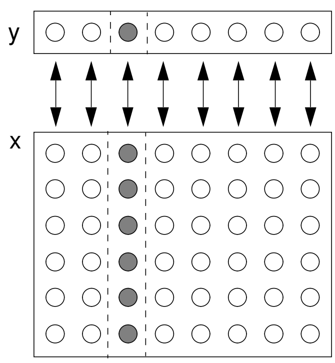
```
       REAL, DIMENSION(:)                      :: x
       REAL, DIMENSION(:), ALLOCATABLE   :: a
!HPF$ ALIGN x(:) WITH a(:)                    ! FALSCH
       REAL, DIMENSION(:), ALLOCATABLE:: a,b
!HPF$ ALIGN a(:) WITH b(:)
       ALLOCATE (b(100), STAT=status)
       ALLOCATE (a(50), STAT=status)           ! FALSCH
       REAL, DIMENSION(:), ALLOCATABLE:: a,b
!HPF$ ALIGN a(i) WITH b(i)
       ALLOCATE (b(100), STAT=status)
       ALLOCATE (a(50), STAT=status)           ! RICHTIG
```


3.2.3.12 Collapsing/Replicating Dimensions

- Collapsing Dimensions

```
!HPF$ ALIGN (*,:) WITH y(:)  :: x
```

Die Zuordnung entspricht der folgenden Abbildung:

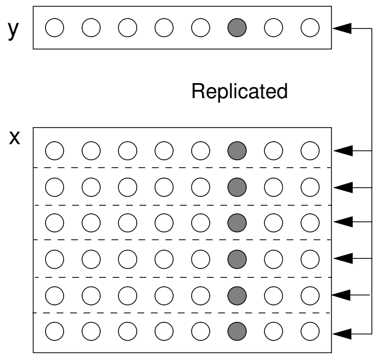


Collapsing/Replicating Dimension

- Replicating Dimensions

```
!HPF$ ALIGN (:) WITH x(*,:)   :: y
```

Die Zuordnung entspricht der folgenden Abbildung:



Übungsaufgabe 3.2 -1

- Schreiben Sie ein Programm, daß ein Matrix-Vektor-Produkt der Form
  y = matrix * x
  berechnet.
  - Die Größe der Matrix soll vom Benutzer zur Laufzeit bestimmt werden
  - Verwenden Sie den Zufallszahlengenerator RANDOM_NUMBER zur Initialisierung von A und x
  - Zur Berechnung von y(i) verwenden Sie die Intrinsic Funktion y(i) = DOT_PRODUCT(Spalte von a, x)
  - Das Ergebnis y soll auf dem Bildschirm ausgegeben werden.
  - Bilden Sie die Datenstrukturen einmal auf ein eindimensionales Prozessorfeld und einmal auf ein zweidimensionales ab. Vergleichen Sie die Laufzeitstatistiken!
  - Prozessoranzahl: 4

Zugang zum System

- Unsere virtuelle “HPF Maschine” werden wir aus den Maschinen faui01a bis faui01k (Sun -Ultra1) bilden. In der Regel sollte jeder jeweils 4 Maschinen zur Lösung seiner Aufgaben verwenden.
- Es gibt drei Möglichkeiten die Virtuelle Maschine aufzubauen
  - auf PVM
  - auf MPI
  - auf RPM der “Portland Group”; am effizientesten; kein Standard (wir verwenden RPM = “Real Parallel Maschine”)
- Environment: Laufzeitparameter “überschreiben” env-param.

```
setenv PGHPF_HOST fauі01a,faui01b,faui01c,faui01d, ...
setenv PGHPF_NP 4
setenv PGHPF_STAT alls
/local/pgi/bin
```


Zugang zum System

- Compilieren und Starten (compile_and_run_script)

```
pghpf -o $1 $1.hpf  gm.hpf -Mfreeform -Mstats
./$1 -pghpf -np 4

oder

pghpf -o matrix matrix.hpf -Mfreeform

./matrix -np 6 -stat cpus
```

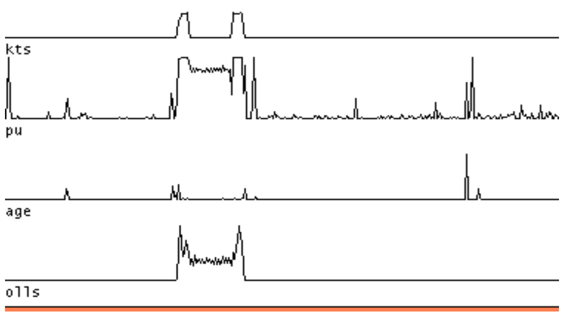
Zugang zum System

cpu	real	user	sys	ratio	node
0*	89.74	52.26	17.07	77%	faui45n
1	89.73	48.19	3.63	58%	faui40h
2	89.73	54.89	3.62	65%	faui40i
3	89.73	43.65	3.13	52%	faui40j
min	89.73	43.65	3.13		
avg	89.73	49.75	6.86		
max	89.74	54.89	17.07		
total	89.74	198.99	27.45	2.52x	

memory	heap used	pag flts no i/o	pag flts i/o	signals received	voluntary switches	involunt switches	res size (pages)
0*	192KB	0	0	0	0	0	0
1	168KB	0	0	0	0	0	0
2	168KB	0	0	0	0	0	0
3	168KB	0	0	0	0	0	0
total	696KB	0	0	0	0	0	0

Zugang zum System

messages	send cnt	send total	send avg	recv cnt	recv total	recv avg	copy cnt	copy total	copy avg
0*	311012	2MB	5B	9806	194KB	20B	460604	4MB	8B
1	19606	270KB	14B	120008	741KB	6B	470404	4MB	8B
2	19606	270KB	14B	120008	741KB	6B	470404	4MB	8B
3	9806	194KB	20B	110208	665KB	6B	460604	4MB	8B
total	360030	2MB	7B	360030	2MB	7B	1862016	15MB	8B



3.2.3.13 Nützliche HPF-Intrinsics

- NUMBER_OF_PROCESSORS
  - gibt die physikalische Anzahl der Prozessoren “zurück”.
- PROZESSORS_SHAPE
  - gibt die Kommunikationsstruktur zwischen den Prozessoren zurück; Anzahl der Verbindungen pro Dimension.
- Diese Intrinsics sind nur bei Deklarationen anzuwenden, sie sind nicht geeignet Konstanten zu vereinbaren oder als Wert einer Variablen zugewiesen zu werden.

Nützliche HPF-Intrinsics

- Beispiele:

```
!HPF$ PROCESSORS (NUMBER_OF_PROCESSORS)          :: P1
!HPF$ PROZESSORS (4,NUMBER_OF_PROCESSORS/16)     :: P2
```

NUMBER_OF_PROCESSORS hat einen optionalen Parameter, der angibt, daß man nur Prozessoren der angegebenen (physikalischen) Dimension wünscht:

```
!HPF$ PROCESSORS (0 : NUMBER_OF_PROCESSORS(1)-1, &
      0 : NUMBER_OF_PROCESSORS(2)-1)
```

Eine Hypercubemaschine mit 8 Prozessoren würde bei

```
   PRINT *, NUMBER_OF_PROCESSORS    ! 8 ausgeben
   PRINT *, NUMBER_OF_PROCESSORS(1) ! 2 ausgeben
   PRINT *, PROCESSOR_SHAPE         ! 2 2 2 ausgeben
```

3.2.3.14 Templates

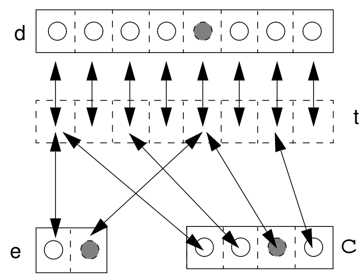
- Templates sind “konzeptionelle” Objekte; sie belegen kein Speicherplatz. Die Intension bei der Verwendung ist die, der besseren Lesbarkeit des Programmcodes.
- Beispiel:

```
      PROGRAM templ
      IMPLICIT NONE
      REAL, DIMENSION(4)      :: c
      REAL, DIMENSION(8)      :: d
      REAL, DIMENSION(2)      :: e
!HPF$ TEMPLATE, DIMENSION(8) :: t
!HPF$ ALIGN d(:) WITH t(:)
!HPF$ ALIGN c(:) WITH t(::2)
!HPF$ ALIGN e(:) WITH t(::4)
!HPF$ DISTRIBUTE (BLOCK)     :: t
      c= 1; d = 2
      e = d(::4) + c(::2)
      END PROGRAM templ
```


Templates

- Beispiel(cont)

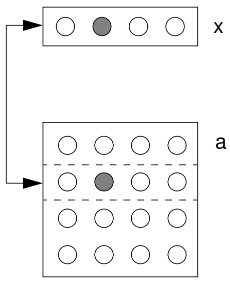
Die Zuordnung entspricht der folgenden Abbildung:



Templates

- Aligning auf ein Template-Bereich am Beispiel:

```
      INTEGER                          :: i = 2
      REAL, DIMENSION(4)               :: x
      REAL, DIMENSION(4,4)             :: a
!HPF$ TEMPLATE, DIMENSION(4,4)         :: t
!HPF$ ALIGN(:,:) WITH t(:,:)           :: a
!HPF$ ALIGN(:) WITH t(i,:)             :: x
```

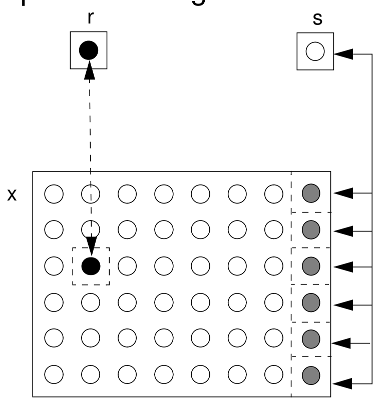


3.2.3.15 Aligning Scalars

- an einem Beispiel:

```
      REAL, DIMENSION(6,8)    :: x
      REAL                    :: r,s
!HPF$ ALIGN r WITH x(3,2)
!HPF$ ALIGN s WITH x(*,8)
```

Die Zuordnung entspricht der folgenden Abbildung:



3.2.4 Parallel-Verarbeitung *(Data Parallel Execution)*

- `FORALL`; Teil des F95-Standards
- `PURE`; Teil des F95-Standards
- `INDIPENDENT`
- `INDIPENDENT NEW`
- F90/95-ARRAY-SYNTAX
- Indrinsics

3.2.4.1 Forall-Statement

- Forall-Statement - Syntax:

```
FORALL(<forall-triplet-list>[,<scalar-mask>]) &
       < assignment-statement>
```

Beispiel:

```
FORALL(i=1:n, j=1:m, a(i,j) /= 0) &
       a(i,j) = 1/a(i,j)
```

Die Berechnung wird (parallel) für alle Elemente von a durchgeführt die ungleich 0 sind. i, j sind **keine** Laufindizes, die Zuweisung wird **“einmalig”** ausgeführt.

- Alle Zuweisungen in ARRAY-Syntax lassen sich auch mit einem Forall-Statement realisieren. Die Umkehrung gilt nicht!

Forall-Statement

- Der Maskenanteil des Statements entspricht dem WHERE-Konstrukt in F90/95.
  Das Beispiel in F90/95-Syntax:

```
WHERE(a /= 0) a = 1/a
```

- Faustformel:
  Einfache Zusammenhänge lassen sich “lesbarer” in ARRAY-Syntax formulieren, komplexere häufig nur mit den Forall-Konstrukten.

- Aufgabe: Transponiere eine n x n - Matrix unter Verwendung eines Forall-Statements.

Forall-Statement

- Weitere Beispiele

```
      FORALL (i=1:n)    a(i,i) = b(i)
!  Vektor b wird Diagonale von matrix a

      DO j = 1,n
       FORALL (i=1:j) a(i,j) = b(i)
      END DO
! matrix a wird triangular (inkl. Dg) mit b besetzt
! DO-loop ist sequentiell

      FORALL (i=1:n, j=1,n, i /= j) a(i,j) = REAL(i+j)
! Diagonale bleibt erhalten; REAL ist “PURE”

      FORALL (i=1:n:3, j=1,n:5) a(i,j) = SIN(a(j,i)
! SIN ist “PURE”-Procedure
```

Forall-Statement

- Weitere Beispiele

```
      FORALL (j = 1:n)
       FORALL (i=1:j) a(i,j) = b(i)
      END FORALL
! matrix a wird triangular (inkl. Dg) mit b besetzt

! Parallel!
   FORALL(i=1:n,j=1:n) a(vs(i),j) = i + vs(j)
! indirekte Adressierung (vector subscribing) ist
! möglich
```


Forall-Statement

- Ausführungsreihenfolge Forall-Statement:
  - Auswertung der “forall-triplet-list”
  - Auswertung der Maske
  - Berechnung der rechten Seite für alle “TRUE-Elemente”
  - Zuweisung der rechten Seiten (RHSs) an die korrespondierenden linken Seiten (LHSs)

  Nach jedem Schritt ist eine Synchronisation über alle Prozessoren erforderlich!

- Forall-Semantik ist verschieden von Do-Loop-Semantik
  Forall ist eine “Feldzuweisung”;
  Do hat eine sequentielle Ordnung; Forall nicht!

Forall-Statement

- Beispiel:

```
        DO i = 2,n-1
                a(i) = a(i-1) + a(i) + a(i+1)
        END DO

! ist verschieden von

        FORALL (i=2:n-1) &
                 a(i) = a(i-1) + a(i) + a(i+1)

! und (in array-syntax)

        a(2:n-1) = a(1:n-2) + a(2:n-1) + a(3:n)
```


3.2.4.2 Forall-Konstrukt

- FORALL-Konstrukt
  Ist dem Forall-Statement ähnlich mit der Ausnahme, daß es einen Ausführungsrumpf gibt. Die einzelnen Anweisung in einem Rumpf werden nacheinander abgearbeitet.
  Beispiele:

```
    FORALL (i=1:n:2, j=n:1:-2, a(i,j) /= 0)
        a(i,j) = 1/a(i,j)
        a(i,i) = b(i)
    END FORALL
! auch geschachtelt ist möglich:
    FORALL (1=1:3, j=1:3, i>j)
        WHERE (ABS(a(i,i,j,j) < 0.1) a(i,i,j,j) = 0.0)
        FORALL (k=1:3, l=1:j, k+l>i) a(i,j,k,l) = j*k+l
    END FORALL
```

3.2.4.3 PURE-Prozeduren

- PURE-Prozeduren
  - Prozeduren ohne Seiteneffekte, d.h.
    - es gibt keine Auswirkungen auf den Zustand globaler Daten,
    - keine externe E/A,
    - keine dynamische Speicherverwaltung (ALLOCATE),
    - keine statischen Variablen (SAVE),
    - Kein “PAUSE” oder “STOP,”
    - Form. Parameter müssen (sollten) das Atribut INTEND(IN) besitzen.
    - Form. Parameter können kein “Alignment” zu globalen Daten haben.
  - alle F90/95-Intrinsics sind “PURE”
  - alle F95 “ELEMENTAL” Funktionen sind “PURE”

PURE-Prozeduren

- Zulässig in
  - FORALL-Konstrukten
  - wiederum in PURE-Prozeduren
- Beispiele:

```
PURE REAL FUNCTION f_name(x,y)
PURE SUBROUTINE s_name(a,b,c)
```

PURE-Prozeduren

- Definition

```
PURE REAL FUNCTION f_name (x,y)
   REAL, INTENT(IN):: x,y
   f_name=x*x + y*y + 2*x*y + ASIN(MIN(x/y,y/x))
END FUNCTION f_name
```

- Aufruf

```
FORALL (i=1:n,j=1:n) &
   a(i,j) = b(i) + f_name(1.0*i, 1.0*j)
```


PURE-Prozeduren

- Es ist guter Stil, dem Compiler durch Bereitstellung eines “INTERFACES” die Möglichkeit zu geben, die “PURE” -Eigenschaft einer aufzurufenden Prozedur innerhalb einer PURE-Prozedure zu überprüfen.
  Interfaces sind Teil des Deklarationsteils einer Prozedure bzw. eines Mosuls.

- Beispiel:

```
INTERFACE
 REAL FUNCTION f_name(x,y)
 REAL, INTENT(in) :: x,y
END FUNCTION f_name
```

PURE-Prozeduren

- Subroutine-Beispiel

```
PURE SUBROUTINE s_name(x,y,z)
    REAL INTENT(OUT), DIMENSION(:)   :: z
    REAL INTENT(IN), DIMENSION (:)   :: x,y
    INTEGER :: i
    INTERFACE
            REAL FUNCTION f_name(x,y)
                  REAL, INTENT(in)     :: x,y
            END FUNCTION f_name
    END INTERFACE
!       ...
    FORALL (i=1:SIZE(z)) z(i) = f_name(x(i), y(i))
END SUBROUTINE s_name

Aufruf: CALL s_name (x, y, result)
```


PURE-Prozeduren

- “MIMD”-Beispiel

```
PURE REAL FUNCTION f(x,i)
IMPLICIT NONE
   REAL, INTENT(IN)      :: x ! element
   INTEGER, INTENT(IN)   :: i ! index

       IF (x > 0.0) THEN
               f = x*x
       ELSEIF (i==1 .OR. i==n) THEN
               f = 0.0
       ELSE
               f = x
       END IF
END FUNCTION F
```

3.2.4.4 INDIPENDENT-Direktive

- INDIPENDENT-Direktive
  - Die INDIPENDENT-Direktive als Prefix einer Do-Schleife sagt aus, daß die Reihenfolge der Iterationen beliegig ist, d.h. der Rumpf ist parallel ausführbar. Beispiel:

```
!HPF$ INDIPENDENT
   DO i = 1,n
       x(i) = i ** 2
   END DO
```

  - Die INDIPENDENT-Direktive als Prefix eines Forall-Statements oder - Konstrukts sagt aus, daß keine Synchronisation zwischen den einzelnen (internen) Abarbeitungsschritten des Foralls notwendig sind. Beispiel:

```
!HPF$ INDIPENDENT
   FORALL (i = 1:n) x(i) = i**2
```


INDIPENDENT-Direktive

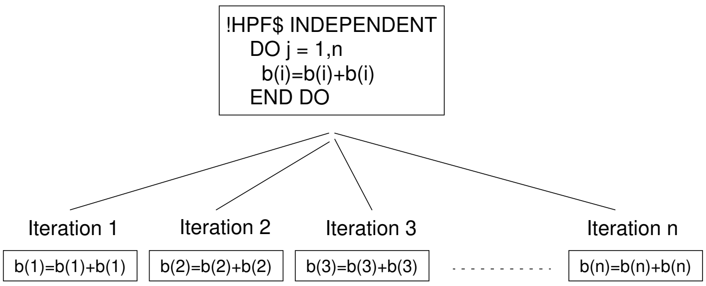
- In INDIPENDENT-Loops darf
  - keiner Variablen mehrfach etwas zugewiesen werden,
  - keine EXIT-, PAUSE- oder STOP - Anweisung vorkommen,
  - kein expliziter “Aussprung” enthalten sein und
  - keine externe E/A erfolgen.
- INDIPENDENT-Regel:
  Wenn bei einer Schleife die Ergebnisse eines Ablaufs identisch sind mit denen, die entstehen, wenn obere und untere Grenze vertauscht und negativ inkrementiert wird, dann ist die Schleife “INDIPENDENT”.

INDIPENDENT-Direktive

INDIPENDENT-Beispiele

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

“indipendent”; Reihenfolge der Iterationen beliebig



INDIPENDENT-Direktive

INDIPENDENT-Beispiele(cont)

```
   DO i = 1, n
      b(i) = b(i+1) + b(i)
   END DO
```

nicht “indipendent”; b(i+1) muß bekannt sein, bevor b(i) berechnet werden kann; “forward reference”

```
   DO i = 1, n
      b(i) = b(i-1) + b(i)
   END DO
```

nicht “indipendent”; b(i-1) muß bekannt sein, bevor b(i) berechnet werden kann; “backward reference”

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

INDIPENDENT-Direktive

- INDIPENDENT- Beispiele (cont)

```
!HPF$ INDEPENDENT
      DO i = 1, n
             a(i) = b(i-1) + b(i) + b(i+1)
      END DO
!HPF$ INDEPENDENT
      FORALL (i=1:n) &
             a(i) = b(i-1) + b(i) + b(i+1)
!HPF$ INDEPENDENT
      DO i = 1, n
             a(index(i)) = b(i-1) + b(i) + b(i+1)
      END DO
```

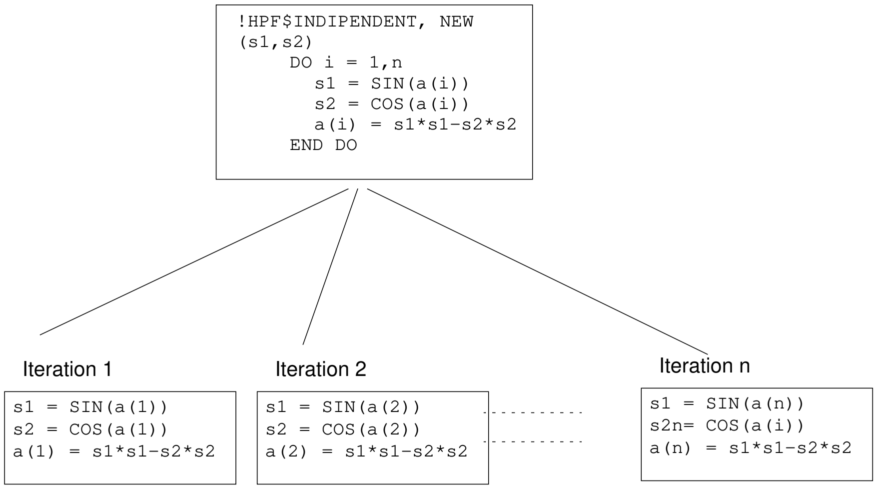

3.2.4.5 INDIPENDENT- NEW -Schleifen

- INDIPENDENT- NEW -Schleifen
  Mehrfachzuweisungen innerhalb einer Indipendent - Do - Schleife sind nicht zulässig; d. h. es dürfen keine scalaren Variablen auf der linken Seite von Zuweisungen vorkommen - es sei denn, für jede Iteration wird eine lokale Variable gehalten.
  NEW gibt an, für welche Variablen dies der Fall sein soll.
  Beispiel:

```
!HPF$ INDIPENDENT, NEW (s1,s2)
      DO i = 1,n
             s1 = SIN(a(i))
             s2 = COS(a(i))
             a(i) = s1*s1-s2*s2
      END DO
```

INDIPENDENT- NEW -Schleifen

- INDIPENDENT- NEW -Schleifen(cont)



INDIPENDENT- NEW -Schleifen

- Regeln zur Verwendung der “NEW-Variablen”:
  - Ihre Werte sind ausserhalb der Schleife undefiniert, eine Verwendung ist also erst nach einen Zuweisung möglich.
  - Für Forall ist eine Verwendung nicht möglich.
  - Sie können keine formalen Parameter oder Zeiger sein.
  - Sie können nicht statisch (SAVE) sein.

INDIPENDENT- NEW -Schleifen

- INDIPENDENT- NEW -Beispiele
  Analyse der Schleifen von innen nach außen:

```
        DO i = 1, n
         DO j = 1, m
               x = a(j)
               y = b(j)
               c(i,j) = x+y
         END DO
        END DO
!HPF$ INDEPENDENT, NEW (j)
        DO i = 1, n
!HPF$ INDEPENDENT (x,y)
         DO j = 1, m
               x = a(j)
               y = b(j)
               c(i,j) = x+y)
         END DO
        END DO
```


INDIPENDENT- NEW -Schleifen

- INDIPENDENT- NEW -Beispiele

```
        DO j = i+1, n
           alpha = a(j,i) / swap(i)
           a(j, i+1:n) = a(j, i+1:n) - alpha &
                    * SWAP(i+1:n)11
           y(j) = y(j) - alpha * TEMP
           a(j,i) = alpha
        END DO
!HPF$ INDEPENDENT, NEW(alpha)
        DO j = i+1, n
           alpha = a(j,i) / swap(i)
           a(j, i+1:n) = a(j, i+1:n) - alpha &
                    * SWAP(i+1:n)12
           y(j) = y(j) - alpha * TEMP
           a(j,i) = alpha
        END DO
```

3.2.5 Input/Output

- Paralle - Ein/Ausgabe ist nicht Teil des Sprachumfangs von HPF. Der Hauptgrund ist der, daß darunter die Portabilität der Programme sehr leiden würde.
  Der Stand der Technik ist z. Z. der, daß die E/A von einem ausgezeichneten Knoten ausgeführt wird; in der Regel von dem, von dem aus das Programmsystem gestartet wurde.
  Hierzu Manual der lokalen Installation lesen!

- Hinweis: Die obige Lösung impliziert, daß Ein-/Ausgabe globale Kommunikationsoperationen nach sich ziehen. Das heißt E/A innerhalb von Schleifen tötet sicherlich jegliche Performance!

3.2.6 HPF-Prozeduren

- Der z. Zt. am IMMD verfügbare HPF-Compiler akzeptiert keine internen Unterprogramme im Hauptprogramm-Modul. Unterprogramme können nur als “Methoden” in einem Modul formuliert werden.

- Ein Problem bei HPF - Prozeduren ist bei Parameterübergabe auch die Informationen über Ausrichtung und Verteilung der Daten mit zu übergeben. Insbesondere bei den Möglichkeiten in Fortran, den Rang von Feldern zu verändern, macht eine automatische “Anpassung” große Probleme, müßte doch für jede mögliche dynamische Variante der zugehörige Code “vorgehalten” werden.

- Sinnvoll sind zwei Methoden
  - eine prescriptive, die exakt so aussieht wie bisher besprochen und
  - eine descriptive, die eher eine “Zusicherung” des globalen Zustands des Programms ist.

3.2.7 Prescriptive Methode

- Grundsätzlich sollte man ein “remapping” beim Aufruf einer Prozedur vermeiden, weil damit immer Kopiervorgänge verbunden sind und zwar sowohl beim Einsprung als auch beim Return; es sei denn ein Intent-Attribut zeigt an, daß dies beim Rücksprung nicht notwendig ist.
  Preskriptive Methode am Beispiel:

```
      SUBROUTINE Subby(a,b,res)
      IMPLICIT NONE
      REAL, DIMENSION(:,:), INTENT(IN)    :: a, b
      REAL, DIMENSION(:,:), INTENT(OUT)   :: res
!HPF$ PROCESSORS, DIMENSION(2,2)           :: P
!HPF$ TEMPLATE, DIMENSION(4,6)             :: t
!HPF$ ALIGN (:,:) WITH t(:,:)              :: a, b, res
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P      :: t
      ...
      END SUBROUTINE Subby
!Sind die aktuellen Parameter nicht gleichmaßen gemappt
!kommt es zum Remapping!
```


3.2.8 Descriptive Methode

- Descriptive Methode
  Der “*” sagt aus, daß die Spezifikation dem momentanen Zustand enspricht.
  Beispiel:

```
      SUBROUTINE Subby(a,b,res)
      IMPLICIT NONE
      REAL, DIMENSION(:,:), INTENT(IN)    :: a, b
      REAL, DIMENSION(:,:), INTENT(OUT)   :: res
!HPF$ PROCESSORS, DIMENSION(2,2)           :: P
!HPF$ TEMPLATE, DIMENSION(4,6)             :: t
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) ONTO *P    :: t
!HPF$ ALIGN (:,:) WITH *t(:,:)             :: a,b,res
      ...
      END SUBROUTINE Subby
```

- Es ist immer guter Stil ein Interface mit Mappinginformationen bereitzustellen.

3.2.9 Templates und Module

- Eine empfehlenswerte Methode:
  Globale Daten, Aligning- und Distribute - Informationen sollte man in einem speziellen Modul bereitstellen, COMMON - Bereiche sollte man vermeiden.
  Beispiel:

```
      MODULE Global_Mapping_Info
!HPF$ PROCESSORS, DIMENSION(2,2)          :: P
!HPF$ TEMPLATE, DIMENSION(4,6)            :: t
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P     :: t
      END MODULE Global_Mapping_Info

      SUBROUTINE Subby(a,b,res)
      USE Global_Mapping_Info
      IMPLICIT NONE
      REAL DIMENSION(:,:), INTENT(IN)     :: a, b
      REAL DIMENSION(:,:), INTENT(OUT)    :: res
!HPF$ ALIGN WITH *t                       :: a, b, res
      ...
      END SUBROUTINE Subby
```


3.2.10 Funktionen

- Mapping von Ergebnissen einer Funktion am Beispiel:

```
      MODULE Block_Dist_1D_Template_Onto_P
!HPF$ PROCESSORS, DIMENSION(2)            :: P
!HPF$ TEMPLATE, DIMENSION(4)              :: t
!HPF$ DISTRIBUTE (BLOCK) ONTO P           :: t
      END MODULE Block_Dist_1D_Template_Onto_P

      FUNCTION ar_f(a,b)
      USE Block_Dist_1D_Template_Onto_P
      IMPLICIT NONE
      REAL, INTENT(IN)                    :: a(:), b(:)
      REAL, DIMENSION(SIZE(a))            :: ar_f
!HPF$ ALIGN (:) WITH *t(:)                :: a
!HPF$ ALIGN (:) WITH *t(:)                :: b
!HPF$ ALIGN (:) WITH t(:)                 :: ar_f
      ...
      END FUNCTION ar_f
```

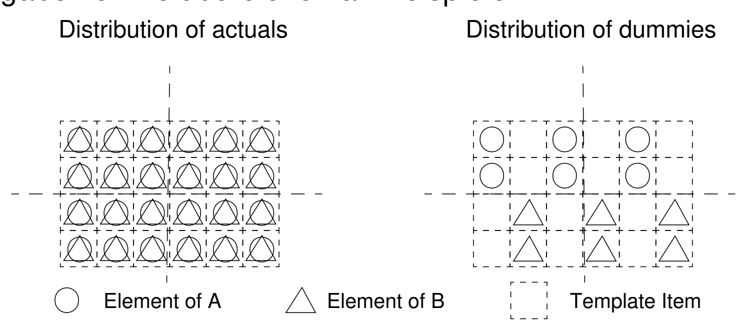
3.2.11 Weiteres

- Übergabe von Feldbereichen an Beispielen:

```
      REAL, DIMENSION(4,6)                :: a, b
      REAL, DIMENSION(2,3)                :: res
!HPF$ PROCESSORS, DIMENSION(2,2)          :: P
!HPF$ ALIGN (:,:) WITH a(:,:)             :: b
!HPF$ ALIGN (:,:) WITH a(::2,::2)         :: res
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P      :: a, b, res
      ...
      CALL Zubbie(a(1:2,1::2),b(3:4,2::2),res)
```


Weiteres

- Übergabe von Feldbereichen an Beispielen:




```
      SUBROUTINE Zubbie(x,y,z)
!HPF$ TEMPLATE, DIMENSION(4,6)            :: t
!HPF$ PROCESSORS, DIMENSION(2,2)          :: P
      REAL, INTENT(INOUT), DIMENSION(:,:) :: x, y
      REAL, INTENT(INOUT), DIMENSION(:,:) :: z
!HPF$ ALIGN (:,:) WITH *t(:,::2)          :: x, z
!HPF$ ALIGN (:,:) WITH *t(3:,::2)         :: y
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) ONTO *P   :: t
      ...
```

Weiteres

- Übergabe von Feldbereichen an Beispielen

```
      SUBROUTINE Zubbie(x,y,z)
!HPF$ TEMPLATE, DIMENSION(4,6)           :: t
!HPF$ PROCESSORS, DIMENSION(2,2)         :: P
      REAL, INTENT(INOUT), DIMENSION(:,:):: x, y
      REAL, INTENT(INOUT), DIMENSION(:,:):: z
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P    :: x, y, z
      ...
```

- Collapsing Dimensions am Beispiel:

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK)      :: A, B
      ...
      CALL Xubbie(A(i,:),b(i,:))
      ...
      SUBROUTINE Xubbie(x,y)
      REAL, DIMENSION(:)            :: x, y
!HPF$ DISTRIBUTE (BLOCK)            :: x, y
```


Weiteres

- Scalare Argumente am Beispiel:

```
      REAL, DIMENSION(100,100)      :: a, b
      REAL                          :: z
!HPF$ DISTRIBUTE (BLOCK,BLOCK)      :: a, b
      INTERFACE
      SUBROUTINE Schmubbie(r,t,x)21
             REAL, INTENT(OUT)      :: r
             REAL, INTENT(IN)       :: t
             REAL, INTENT(IN)       :: x(:,:)
!HPF$ DISTRIBUTE *(BLOCK,BLOCK)     :: x
      END SUBROUTINE Schmubbie
      END INTERFACE
      ...
      CALL Schmubbie(a(1,1),z,b)
```

3.2.12 Extrinsics, Intrinsics, HPF-Library

- Extrinsics: Siehe manual und WWW-Kurs
- Intrinsics: Siehe Manual und WWW-Kurs
- HPF-Library: Siehe Manual und WWW-Kurs
  - Einbindung der Library (Modul) durch

```
USE HPF_LIBRARY
```


3.2.13 Allgemeine Programmierrichtlinien

- Man muß die richtige Balance zwischen Parallelität und Kommunikationsaufwand finden (Granularität des Programms). Die zugrundeliegenden Hardwarearchitektureigenschaften sind dabei von zentraler Bedeutung.
  - Mehr Prozessoren bedeuten bei gleichem Programm in der Regel mehr Kommunikation, da Granularität feiner.
  - Ausgewogene “Load” ist wichtig.
  - Inhärente Datenlokalität sollte genutzt werden.
  - Benutzung der F90/95-Array-Syntax ist von Vorteil!
- Es ist relativ leicht, ein korrektes HPF-Fortran-Programm zu schreiben, jedoch außerordentlich schwierig ein auch effizientes! Tools????

Allgemeine Programmierrichtlinien

- Empfohlene Standardvorgehensweise
  - Schreibe zunächst den “parallelen” Code als sequentielles Programm.
  - Teste und Debugge.
  - Füge dann die Verteilungsdirektiven ein, mit dem Ziel, die Kommunikation zu minimieren (bei NORMA immer richtig).
  - Lokalität ist der Schlüssel! Richtige Verteilung und Ausrichtung (Alignment) hat signifikante Auswirkungen auf die Performance.
  - Wo irgend möglich dem Compiler mit Direktiven “helfen”. z. B. `INDIPENDENCE`
  - Dem Compiler geht es wie uns: Je klarer ein Programm formuliert ist, desto leichter ist es für ihn, effizienten, parallelen Code zu generieren.
  - Traue nie einem HPF-Compiler! Untersuche die Laufzeitstatistik auf Plausibilität.

3.2.14 Übungen

- Aufgabe 2

Eine Mandelbrotmenge ist eine Menge von Zahlen, die das Ergebnis von wiederholten Iterationen der Funktion

$$X = Z^2 + C$$

sind.
Diese Funktion kann man in Realteil und Imaginärteil zerlegen:

$$z_i = 2.0 * z_r * z_i + c_i \text{ und}$$
$$z_r = z_r^2 - z_i^2 + c_r$$

Definitionsbereich der Funktion: [-1.0, 1.0],[-1.0, 1.0]

In der folgenden Aufgabe wollen wir uns jedoch auf den 1. Quadranten beschränken, also: [0.0, 1.0],[0.0, 1.0].

Was üblicherweise grafisch dargestellt wird, ist die Anzahl der Iteration, die notwendig ist sich einen geeigneten Grenzwert zu nähern:

$$z_r^2 + z_i^2 > 4.0$$

Um Ihnen den “Kampf” mit Randproblemen zu ersparen, geben wir einen Programmrahmen vor, den Sie sich aus der Datei ~uelinste/tmp/mandel_rahmen.hpf kopieren können.

Übungen

```
    PROGRAM mandel
!       Credits to Edinburgh Parallel Computing Center
!       Dr. Adam Marshall, Univ. of Liverpool
!       Colouring by Michael Schroeder, Univ. of Erlangen
    IMPLICIT NONE
    INTEGER, PARAMETER :: n = 400, iteration = 100
    INTEGER :: i, j, start, stop
    INTEGER, DIMENSION(3,0:16*7-1) :: palette
    INTEGER, DIMENSION(3,0:7) :: pcolor = (/&
                    0,0,0, 0,255,255, 0,255,0, 255,255,0, 255,0,0,&
                    255,0,255, 0,0,255, 0,0,0/)
```

! Vereinbaren Sie die Felder fuer colour, zr, zi, cr, ci, zrs, zis:

! Legen Sie das Prozessorgrid fest und verteilen Sie die Daten:

Ubungen

```
!   Iinitialisiere Palette
        FORALL (i=0:6, j=0:15)
            palette(1:3,i*16+j) = ((16-j)*pcolor(1:3,i) + j*pcolor(1:3,i+1))/16
        END FORALL
```

! Initialisieren Sie die Felder zr und zi mit einem FORALL-Konstrukt,
! derart, dass zr zeilenweise und zi spaltenweise aequidistant
! im Intervall [0.0,1.0] vorbesetzt wird.
! Also z. B. bei einem Feld mit 6 Zeilen:
! 0.0, 0.2, 0.4, 0.6, 0.8, 1.0, jeweils die ganze Zeile bzw. Spalte:

! Initialisieren Sie weiterhin die Felder cr wie zr und ci wie zi;
! und die Felder zrs mit zr*zr und zis mit zi*zi.
! Das Feld color wird mit 0 vorbesetzt:

```
!   Hier wird die Systemuhr gestartet
    CALL SYSTEM_CLOCK(start)
    WRITE(*,*) start
```

© Claus-Uwe Linster – pps3.2_spmd_hpf: 19/3/98

Ubungen

```
! Hier parallelisieren Sie bitte den folgenden sequentiellen Code, und zwar
! unter Verwendung des where-Konstrukts und der f90-Array-Syntax:
!
      DO i = 0, iteration
      DO k = 1,n
          DO j = 1,n
            IF (zrs(j,k) + zis(j,k) <= 4.0) THEN
              zrs(j,k) = zr(j,k) * zr(j,k)
              zis(j,k) = zi(j,k) * zi(j,k)
              zi(j,k) = 2.0 * zr(j,k) - zi(j,k) + ci(j,k)
              zr(j,k) = zrs(j,k) - zis(j,k) + cr(j,k)
              colour(j,k) = iteration - i
            END IF
          END DO
      END DO
```


Ubungen

```
  ! Ab hier lassen Sie bitte den Code unveraendert

          CALL SYSTEM_CLOCK(stop)

          WRITE(*,*) stop-start

  !   Open output file

          OPEN(UNIT=10, FILE='mandel.pgm')
          WRITE(10, FMT='(''P3'',/,i3,2x,i3,/,i3)')n, n, 255
          DO i = 1,n
           DO j = 1,n
              WRITE(10,*) palette(:, MOD(colour(j, i), 7*16))
           END DO
          END DO
          CLOSE(UNIT=10)

      END PROGRAM mandel
```

Ubungen

- Aufgabe 3

Parallelisieren Sie den Ihnen bereits bekannten
“Flächenglättungsalgorithmus”

a.) “total” parallel und

b.) “partiell” parallel.

Können Sie bezüglich des Konvergenzverhaltens Unterschiede feststellen?