Weitere Extras

Datentypen selbstgemacht

Neben Standarddatentypen `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, `CHARACTER` kann man (in Fortran 90) eigene Datentypen definieren („Derived Types")

```
TYPE typname
  !  Vereinbarung von Komponenten
END TYPE typname
```

Vereinbarung von Variablen dieses Typs:

```
TYPE(typname) [,attribute ::]  varname [, ...]
```

Beispiel 1:

```
type coord
  real ::  x,y
end type coord
type place
  character*20 ::  name
  type(coord) ::  location
end type place
```

```
type(coord) c
type(place) p
p%name = 'hier'
c%x = 50.8173
c%y = 12.9310
p%location = c
```

Weitere Extras

Datentypen selbstgemacht

Neben Standarddatentypen `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, `CHARACTER` kann man (in Fortran 90) eigene Datentypen definieren („Derived Types“)

```
TYPE typname
  !  Vereinbarung von Komponenten
END TYPE typname
```

Vereinbarung von Variablen dieses Typs:

```
TYPE(typname) [,attribute ::]  varname [, ...]
```

Beispiel 2:

```
type nne
  double precision v
  integer i,j
end type nne
```

```
type(nne),dimension(nNNE) ::  A
double precision,dimension(N) ::  x,y,b
integer ::  k
y=b
do k=1,nNNE
  y(A(k)%i) = y(A(k)%i) + A(k)%v * x(A(k)%j)
end do
```

Weitere Extras

Zufallszahlen ...

... bilden üblicherweise eine deterministische Zahlenfolge, gleichverteilt in $[0, 1)$, die allerdings von einem Startwert (*seed*) abhängt,

d. h. bei gleichem Startwert ist die Zufallszahlenfolge reproduzierbar (wichtig für numerische Tests)

Initialisierung einer Folge: `call RANDOM_SEED(size,put,get)`

`size`	gibt Anzahl der für *seed* verwendeten Integer-Werte zurück, z. B. 8
`put`	Integer, Dimension(*size*) – Festlegen der *seed*-Werte
`get`	Integer, Dimension(*size*) – Rückgabe der aktuellen *seed*-Werte

Abrufen der nächsten „Zufallszahl" aus der aktuellen Folge:

```
call RANDOM_NUMBER(x)
```

Für eine skalare `REAL`-Variable `x` wird ein Wert $0. \leq x < 1.$ zurückgegeben.
Für ein Array `DIMENSION x(n)` wird jedem Element eine „Zufallszahl" zugewiesen.

Zufallszahlen . . .

. . . bilden üblicherweise eine deterministische Zahlenfolge, gleichverteilt in $[0, 1)$, die allerdings von einem Startwert (*seed*) abhängt,

d. h. bei gleichem Startwert ist die Zufallszahlenfolge reproduzierbar (wichtig für numerische Tests)

Initialisierung einer Folge: `call RANDOM_SEED(size,put,get)`

`size` gibt Anzahl der

`put` Integer, Dimens

`get` Integer, Dimens

Mögliche Initialisierungs-Aufrufe (optionale Parameter!)

```
call RANDOM_SEED ! Initialisierung durch Prozessor
call RANDOM_SEED(size=k) ! k=Anzahl seed-Werte
call RANDOM_SEED(get=o(1:k)) ! alte Werte merken
call RANDOM_SEED(put=s(1:k)) ! Werte selbst festlegen
```

Abrufen der nächsten „Z

```
call RANDOM_NUMBER(x)
```

Für eine skalare `REAL`-Variable `x` wird ein Wert $0. \leq x < 1.$ zurückgegeben.
Für ein Array `DIMENSION x(n)` wird jedem Element eine „Zufallszahl“ zugewiesen.

Weitere Extras

Eigene Unterprogrammbibliotheken

Bibliothek = Archiv von (compilierten) Unterprogrammen (Dateien `*.o`)

- Verwaltung von Bibliotheken mittels „Archivar"(Linux-Befehl: `ar`)
  z. B. `ar sru libVektor.a *.o`
  alle Objektfiles aus dem Verzeichnis in der Bibliothek aktualisieren
- jeweils eine der Operationen: r (replace), d (delete), x (extract), t (test)
  Zusätze: c (create), u (update), v (verbose), s (symbol index)
- Einbinden von Bibliotheken beim Linken:
  `gfortran -o myexec hauptprog.o [-Lverzeichnis] -lVektor`
- Beachte: Dateinamen und Unterprogrammnamen können verschieden sein
  (dafür gibt es den Symbolindex als Inhaltsverzeichnis der Bibliothek)
  Archivar verwaltet **Dateien**, Linker sucht nach **Unterprogrammnamen**.
- optimal: pro Datei `*.o` genau ein Unterprogramm

Weitere Extras

Eigene Unterprogr

Bibliothek = Archiv v

- Verwaltung von
  z. B. `ar sru li`
  alle Objektfiles a
- jeweils eine der
  Zusätze: c (crea
- Einbinden von B
  `gfortran -o m`
- Beachte: Datein ... in
  (dafür gibt es de
  Archivar verwalt ... .
- optimal: pro Da

Bibliotheksaktualisierung via Makefile

(vorher im Makefile definiert: `FC`, `FFLAGS`, `AR`, `RM` u.a.)

```
ARFLAGS=sru
FSOURCE = x.f y.f up1.f up2.f
LIBDIR = /pfad/
LIBNAM = Vektor
LIB=$(LIBDIR)lib$(LIBNAM).a   → /pfad/libVektor.a
...
$(PROGRAM):  $(MAINOBJ) $(LIB) ...
    $(FC) $(MAINOBJ) -L$(LIBDIR) -l$(LIBNAM) -o $@
$(LIB):  $(LIB)($(FSOURCE:.f=.o))
...
.f.o:
    $(FC) $(FFLAGS) -c $*.f
.f.a:
    $(FC) $(FFLAGS) -c $<
    $(AR) $(ARFLAGS) $@ $%
    $(RM) $%
```

Wirkung:
wenn eine Quelle `x.f` aktueller ist als die in der Bibliothek archivierte Datei `x.o`, wird `x.f` neu übersetzt zu `x.o`, dann wird `x.o` im Archiv aktualisiert und im Arbeitsverzeichnis wieder gelöscht. Mit der aktualisierten Bibliothek wird neu gelinkt.

Weitere Extras

Eigene Unterprogr

Bibliothek = Archiv v

- Verwaltung von
  z. B. `ar sru li`
  alle Objektfiles a
- jeweils eine der
  Zusätze: c (crea
- Einbinden von B
  `gfortran -o m`
- Beachte: Datein ... in
  (dafür gibt es de
  Archivar verwalt
- optimal: pro Da

Bibliotheksaktualisierung via Makefile

(vorher im Makefile definiert: `FC`, `FFLAGS`, `AR`, `RM` u.a.)

```
ARFLAGS=sru
FSOURCE = x.f y.f up1.f up2.f
LIBDIR = /pfad/
LIBNAM = Vektor
LIB=$(LIBDIR)lib$(LIBNAM).a   → /pfad/libVektor.a
...
$(PROGRAM):  $(MAINOBJ) $(LIB) ...
    $(FC) $(MAINOBJ) -L$(LIBDIR) -l$(LIBNAM) -o $@
$(LIB):  $(LIB)($(FSOURCE:.f=.o))
...
.f.o:
    $(FC) $(FFLAGS) -c $*.f
.f.a:
    $(FC) $(FFLAGS) -c $<
    $(AR) $(ARFLAGS) $@ $%
    $(RM) $%
```

Wirkung:
wenn eine Quelle `x.f` aktueller ist als die in der Bibliothek archivierte Datei `x.o`, wird `x.f` neu übersetzt zu `x.o`, dann wird `x.o` im Archiv aktualisiert und im Arbeitsverzeichnis wieder gelöscht. Mit der aktualisierten Bibliothek wird neu gelinkt.

Weitere Extras

Parallelisierung (distributed memory)

- Mehrere Programme (üblicherweise Kopien einunddesselben Programms) laufen gleichzeitig auf verschiedenen Prozessoren (vernetzten Computern).
- Das Programm entspricht einer Funktion, die von 2 Umgebungsparametern abhängt: Anzahl der parallelen Prozesse und
  Identifikationsnummer des eigenen Prozesses
- Kein direkter Zugriff auf Daten der anderen Prozessoren, sondern expliziter Austausch durch Kommunikationsroutinen (`SEND / RECV`).
- Standards: MPI (Message Passing Interface),
  PVM (Parallel Virtual Machines), Spezialvarianten (PARIX, NCUBE, ...)
- Synchronisation des Prozessablaufs erfolgt durch entsprechende Kommunikationsroutinen (point-to-point oder global);
  Vermeidung von „deadlocks“ ist Aufgabe des Programmierers!

Weitere Extras

Parallelisierung (shared memory) OpenMP

- Speziell für Multicore-Architekturen ist shared-memory interessant; mehrere parallele Prozesse nutzen denselben Speicher.
- Es ist undefiniert, mit welcher Geschwindigkeit die einzelnen Prozesse laufen und zu welchem Zeitpunkt sie auf gemeinsame Daten zugreifen.
- Zeitweise exklusiver Zugriff auf bestimmten Speicher ist erforderlich, um ungewollte Zufallsergebnisse zu vermeiden.
- Sperrung durch exklusiven Zugriff sollte so kurz wie möglich sein, z.B. gesteuert durch „Semaphoren“
(spezieller Datentyp, auf den nur mit 2 unteilbaren Operationen zugegriffen werden kann, wie etwa „Resource anfordern“ / „Resource freigeben“).
- Unterstützung durch OpenMP: spezielle Kommentarzeilen im Quelltext erlauben dem Compiler, bestimmte Anweisungen auf mehrere parallele Prozesse (threads) aufzuteilen, falls auf dieser Hardware möglich.

Weitere Extras

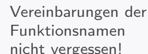
Nutzung von OpenMP

Direktiven als Kommentar im Quelltext (werden nur beachtet, wenn das Programm mit der Option `-fopenmp` compiliert wird)

```
!$omp parallel [...]        !  auch:  *$omp, C$omp
    Fortran-Anweisungen (zu parallelisierender Block)
!$omp end parallel
```

Runtime Library wird benötigt, um parallele Prozesse zu starten und stellt einige Unterprogramme zur Verfügung (dazu muss beim Linken ebenfalls `-fopenmp` als Option angegeben sein), z.B.

```
nproc = omp_get_num_procs()
n_thr = omp_get_num_threads()
n_max = omp_get_max_threads()
IF ( omp_in_parallel() ) ...
myID = omp_get_thread_num()
```



Environment-Variablen (`setenv VAR wert` bzw. `export VAR=wert`)
z. B.: `OMP_THREAD_LIMIT`, `OMP_NUM_THREADS`, `OMP_SCHEDULE`

Nutzung von OpenMP — Beispiel

Beispiel: Skalarprodukt zweier Vektoren

```
      s1=0d0



      DO i=1,n
        s1=s1+x(i)*y(i)
      END DO
```

Beispiel: Linearkombination zweier Vektoren $z = x + \alpha y$

```
!$omp parallel default(shared) private(i)
!$omp do schedule(guided)
      DO i=1,n
        z(i)=x(i)+alpha*y(i)
      END DO
!$omp end do
!$omp end parallel
```

Nutzung von OpenMP — Beispiel

Beispiel: Skalarprodukt zweier Vektoren

```
      s1=0d0
!$omp parallel do default(shared) schedule(GUIDED)
!$omp& private(i) reduction(+:s1)   !  OMP-Fortsetzungszeile
      DO i=1,n
        s1=s1+x(i)*y(i)
      END DO
!$omp end parallel do
```

Beispiel: Linearkombination zweier Vektoren $z = x + \alpha y$

```
!$omp parallel default(shared) private(i)
!$omp do schedule(guided)
      DO i=1,n
        z(i)=x(i)+alpha*y(i)
      END DO
!$omp end do
!$omp end parallel
```

Weitere Extras

Nutzung von OpenMP — Beispiel

Beispiel: Skalarprodukt zweier Vektoren

```
      s1=0d0
!$omp parallel do default(shared) schedule(GUIDED)
!$omp& private(i) reduction(+:s1)   !  OMP-Fortsetzungszeile
      DO i=1,n
        s1=s1+x(i)*y(i)
      END DO
!$omp end parallel do
```

Beispiel: Linearkombination zweier Vektoren $z = x + \alpha y$

```
!$omp parallel default(shared) private(i)
!$omp do schedule(guided)
      DO i=1,n
        z(i)=x(i)+alpha*y(i)
      END DO
!$omp end do
!$omp end parallel
```

Nutzung von OpenMP — Beispiel

Bei eher „ungeordneten“ Zugriffen auf Datenfelder kann Parallelisierung schwierig werden, wie im obigen Bsp. mit dem definierten Typ `nne`:

```
y=b

do k=1,nNNE
  y(A(k)%i) = y(A(k)%i) + A(k)%v * x(A(k)%j)
end do
```

Weitere Extras

Nutzung von OpenMP — Beispiel

Bei eher „ungeordneten“ Zugriffen auf Datenfelder kann Parallelisierung schwierig werden, wie im obigen Bsp. mit dem definierten Typ `nne`:

```
!$omp parallel default(shared)
!$omp workshare
      y=b
!$omp end workshare
!$omp do private(k)
      do k=1,nNNE
        y(A(k)%i) = y(A(k)%i) + A(k)%v * x(A(k)%j)
      end do
!$omp end do




!$omp end parallel
```

Ergebnisse auf `y` sind falsch (zufällig durch Mehrfachzugriff auf `y(i)`)

Weitere Extras

Nutzung von OpenMP — Beispiel

Bei eher „ungeordneten“ Zugriffen auf Datenfelder kann Parallelisierung schwierig werden, wie im obigen Bsp. mit dem definierten Typ `nne`:

```
!$omp parallel default(shared)
!$omp workshare
      y=0
!$omp end workshare
!$omp do private(k) reduce(+:y)
      do k=1,nNNE
        y(A(k)%i) = y(A(k)%i) + A(k)%v * x(A(k)%j)
      end do
!$omp end do
!$omp workshare
      y=y+b
!$omp end workshare
!$omp end parallel
```

Vektor y als private Kopie, anschließend aufsummiert über alle Threads;
Aufwand – Nutzen ?

Weitere Extras

Beipiele für OpenMP-Direktiven

`*$omp workshare`	Anweisungsblock zwischen Threads aufteilen
`*$omp master`	Anweisungsblock nur durch Masterprozess bearbeitet
`*$omp single`	Anweisungsblock nur durch **einen** (beliebigen) Prozess bearbeitet
`*$omp atomic`	exklusiver Speicherzugriff für nächste Anweisung (z.B. `X=X+1`)
`*$omp critical`	Anw.-Block zu jeder Zeit nur von einem Thread bearbeitet
`*$omp sections`	mehrere Abschnitte, die ggf. in verschiedenen Threads laufen
`*$omp flush`	Synchronisationspunkt, alle Threads „sehen“ gleiche Daten

Beispiele für Zusatzangaben bei Direktiven

`private(liste)`	Liste privat genutzter Variablen (kein Anfangswert)
`firstprivate(liste)`	Liste privat genutzter Variablen (mit Anfangswert)
`schedule(typ[,size])`	`typ = static | dynamic | guided | auto | runtime` Art und Größe der an die Threads zugewiesenen Einheiten
`if(log_Ausdruck)`, z.B.	`!$omp parallel if(omp_get_max_threads().GE.4)`
`num_threads(anzahl)`	Anzahl paralleler Threads ($\leq max$) für Anw.-Block

Weitere Extras

. . . und nochmal **Hello World!**

```
      Program HelloOMP
      INTEGER id,n
      INTEGER omp_get_num_threads,omp_get_max_threads
      INTEGER omp_get_thread_num
      write(*,100) omp_get_max_threads()
!$omp parallel private(id) shared(n)
      n=omp_get_num_threads()
      id=omp_get_thread_num()
      write (*,200) id,n
!$omp end parallel
100   Format('Hello World - max.  threads:  ',I2)
200   Format('Hi! - from thread ',I2,' (of ',I2,')')
      END
```

Modifikation der Thread-Anzahl vor Programmstart:
`setenv OMP_NUM_THREADS 10` (csh), bzw. `export OMP_NUM_THREADS=10` (bash)