Programmierung Paralleler Systeme

5.2 Vektorisierung auf VPP300/700

Nach einem Skript von M. Brehm und T. Störtkuhl, LRZ-München, denen mein unbegrenztes fachliches Vertrauen gehört.
http://www.lrz-muenchen.de/Services/compute/vpp/optvecpar

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

5.2 Vektorisierung

- Unter **Vektorisierung**
  versteht man den Vorgang, ein Programm in eine solche Form zum bringen, daß die Ausführung so weit wie möglich durch Vektoroperationen realisiert werden kann.
- **Compiler der Vektorrechner**
  versuchen im allgemeinen diese Aufgabe automatisch zu lösen. Das gelingt natürlich dann am besten, wenn der Programmierer beim Programmentwurf die Rechnerarchitektur im "Hinterkopf" hat und ein Basiswissen über die Hinderungsgründe einer Vektorisierung vorhanden ist. Zusätzlich sollte man den Compilern mit Compilerdirektiven helfen.
- **Compilerdirektiven**
  sind Zusicherungen über das dynamische Verhalten des Programms. Sie bringen den Compiler dazu, Programmteile zu vektorisieren, obwohl sein internes Regelwerk bzw. sein Wissen um den Programmkontext das u. U. verbieten würden. Die Verantwortung trägt natürlich der Programmierer.
- **Analysedaten**
  teilen dem Programmierer mit, wie und wo erfolgreich vektorisiert wurde.

5.2.1 Einflußgrößen auf Vektorperformance

- Vektorlänge
- Anzahl der arithmetischen Operationen in einer Schleife
- Verhältnis von Load/Store-Operationen zu arithmetischen Operationen
- Art der Operationen
- Verteilung der Vektorelemente im Speicher
- Einige Zahlen:

Vektorlängen	Geeigneter Rechnerklasse
1 bis 8	PC
1 bis 32	Workstation (leistungsfähige)
32 bis 64	Workstation und Vektorrechner, insbesondere Cray
64 bis 128	CRAY T90 besser als Workstation
128	Vektorregisterlänge der Cray
128 bis 256	VPP besser als Workstation
256 bis 2048	Vektorrechner deutlich besser als Workstation
2048	Max. Registerlänge der VPP
> 2048	Keine Chance für Workstations

5.2.2 Vektorisierung von Schleifen

- Geordnete Mengen von Daten gleichen Typs auf die ein- und dieselbe Operation ausgeführt wird, sind immer dann “problemlos” vektorisierbar, wenn keine
  - Datenabhängigkeiten
  - Rekursionen
  - Reduktionen
  - indirekte Adressierung
  - IF-Statements
  - GOTO-Statements
  - Unterprogrammaufrufe oder
  - I/O

  dies verhindern.
- Obwohl real bei den Vektoroperationen nur überlappend gearbeitet wird, gehen wir bei unseren Betrachtungen von einer echt parallelen Abarbeitungsweise - bezogen auf einen Vektor (Vektorlänge) - aus.

Vektorisierung von Schleifen

- Betrachten wir folgende Schleife;

```
DO i = 1,n
   a(i) = b(i) + c(i) * s
   d(i) = b(i) * c(i)
ENDDO
```

- Die Schleife wird “vektoriell” wie folgt abgearbeitet:

```
1. a(i) = b(i) + c(i) * s (parallel) für alle i
2. d(i) = b(i) * c(i) (parallel) für alle i
3. i = n + 1
```

- Die uns bekannte Array-Notation von Fortran90 spiegelt diesen Sachverhalt korrekt wider:

```
a(1:n) = b(1:n) + c(1:n) *s
d(1:n) = b(1:n) * c(1:n)
```

- Unter der Voraussetzung das n = Länge des Feldes ist:

```
a = b + c * s;
d = b * c
```


Vektorisierung von Schleifen

- Bei der skalaren Abarbeitung der Schleife werden bekanntlich alle Statements der Schleife nacheinander mit jeweils einem festen Index pro Iteration abgearbeitet.
- Da die semantische Definition von Fortran die skalare Abarbeitung von Schleifen zugrunde legt, ist eine Vektorisierung nur dann zulässig, wenn beide Lösungswege exakt zum gleichen Ergebnis führen.
  - Tatsächlich untersuchen die Compiler
    - die Semantik,
    - die Durchführbarkeit, aufgrund des Instruktionssatzes des Rechners und
    - ob es aus Performancegründen überhaupt sinnvoll ist, zu vektorisieren.
  - wobei nicht nur innere Schleifen vektorisiert werden, sondern auch
    - Umordnungen, Zusammenfassungen (Linearisierung) und/oder
    - Aufrollen von Schleifen oder Teilschleifen (unrolling)

  zur besseren Vektorisierbarkeit durchgeführt werden.

Vektorisierung von Schleifen

- Der vektorisierte Code ist (selbstverständlich) hochgradig hardware-abhängig, d. h. gut vektorisierbarer Code für die Cray muß nicht auch in gleicher Weise gut vektorisierbar auf der Fujitsu VPP sein. Insoweit gelten im folgenden die Empfehlungen nur für die VPP; wobei natürlich die grundsätzlichen Probleme die gleichen sind. Ich werde gelegentlich auf Unterschiede hinweisen. (Siehe auch Skript: Brehm/Störtkuhl)
- Generierung eines *Analyse Listings* auf der VPP

`Compileroptionen:`

`-Post -Px -Wv, -m3, -qm`

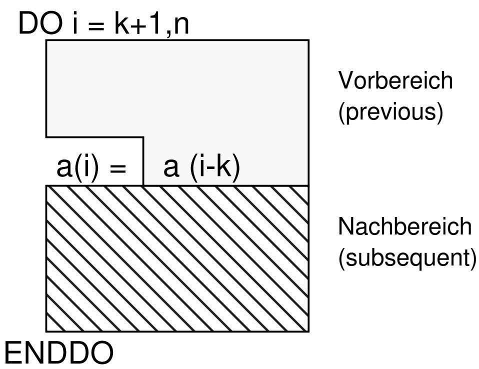
man erhält Information über

- welche Schleifen wurden vektorisiert
- wie wurde vektorisiert (**`-Wv, -m3, -qm`**)
- *Cross Reference Listings* (**`-Px`**)

`frt -o` ***`objectname`*** **`-Post -Px -Wv, -m3, -qm`** ***`source.f90`***

Vektorisierung von Schleifen

- Bei Datenabhängigkeiten werden vier Typen unterschieden
  - previous minus
  - previous plus
  - subsequent minus
  - subsequent plus

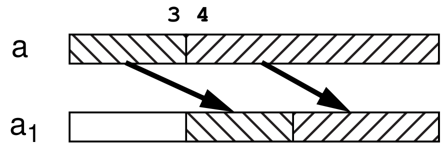


Vektorisierung von Schleifen

- previous minus: bedingt vektorisierbar

```
DO i = k+1,n
   a(i) = a(i-k)
ENDDO
```

sequentiell:




```
Parallel: k = 3; n = 10;
          i = 4,10

a(4:10) = a(1:7)

in a(4:10) - alte Werte

für k >= n/2
pot. vektorisierbar
```

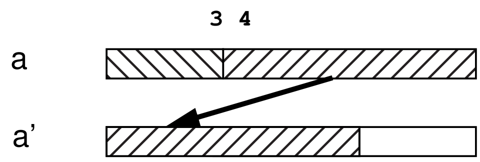
```
sequentiell:

a(4) = a(1)   alt
a(5) = a(2)   alt
a(6) = a(3)   alt
a(7) = a(4)   neu
a(8) = a(5)   neu
...
a(10) = a(7)  neu
```


Vektorisierung von Schleifen

- previous plus: vektorisierbar

```
DO i = 1,n-k
   a(i) = a(i+k)
ENDDO
```




```
Parallel: k = 3; n = 10;
          i = 1,7

a(1:7) = a(4:10)

in a(1:7) - alte Werte
```

```
sequentiell:

a(1) = a(4)   alt
a(2) = a(5)   alt
a(3) = a(6)   alt
a(4) = a(7)   alt
a(5) = a(8)   alt
...
a(7) = a(10)  alt
```

Vektorisierung von Schleifen

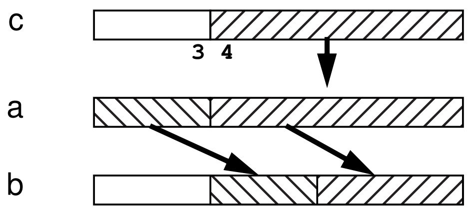
- subsequent minus: vektorisierbar

```
DO i = k+1,n
   a(i) = c(i)
   b(i) = a(i-k)
ENDDO

Parallel: k = 3; n = 10;
          i = 4,10

a(4:10) = c(4:10)
b(4:10) = a(1:7)

in b(4:6) - alte Werte
in b(7:10)- neue Werte
```




```
sequentiell:

a(4)  = c(4)
b(4)  = a(1)   alt
...
b(6)  = a(3)   alt
a(7)  = c(7)
b(7)  = a(4)   neu
...
a(10) = c(10)
b(10) = a(7)   neu
```


Vektorisierung von Schleifen

- subsequent plus: bedingt vektorisierbar

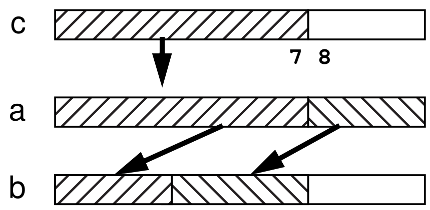
```
DO i = 1,n-k
   a(i) = c(i)
   b(i) = a(i+k)
ENDDO

Parallel: k = 3; n = 10;
          i = 1,7

a(1:7) = c(1:7)
b(1:7) = a(4:10)

in b(1:3) - neue Werte
in b(4:7) - alte Werte
für k >= n/2
pot. vektorisierbar
```

parallel:




```
sequentiell:

a(1) = c(1)
b(1) = a(4)    alt
a(2) = c(2)
b(2) = a(5)    alt
a(3) = c(3)
b(3) = a(6)    alt
...
a(7) = c(7)
b(7) = a(10)   alt
```

Vektorisierung von Schleifen

- An folgendem Beispiel soll verdeutlicht werden, wie zwei verschiedene Rechnerarchitekturen dasselbe Problem verschiedenartig lösen. Gegeben ist folgende Schleife:

```
DO i = 1,n
   DO j = 2,m
      a(i,j) = a(i,j-1) + s
   ENDDO
ENDDO
```

Problem: Für Index j kann die Schleife nicht vektorisiert werden, da sie eine sogenannte Rekursion enthält.

Lösung VPP: Die äußere Schleife wird vektorisiert; die innere Schleife zweifach entrollt; es entsteht folgende (äquivalente) Schleifenkonstruktion:

```
DO j = 2,m,2         ! m durch 2 teilbar
   DO i = 1,n
      a(i,j) = a(i,j-1) + s
      a(i,j+1) = a(i,j) + s
   ENDDO
ENDDO
```


Vektorisierung von Schleifen

- Das Beispiel liegt in der directory ~iwi412/tmp unter dem Namen: loop10.f90 Bitte das Beispiel ausprobieren und das Analyse-Listing diskutieren. Wieviel MFlops werden erreicht?
- Lösung T90: Der Cray-Compiler nutzt sein Wissen, daß n > 128 ist und vektorisiert die äußere Schleife derart, daß das zweidimensionale Feld in ein eindimensionales transformiert wird; es entsteht folgende (äquivalente) Schleifenkonstruktion:

```
! fuer n > 128 (Vektorregisterlänge der T90)

DO i = n+1,n*m
   a(i,1) = a(i-n,1) + s
ENDDO
```

Vektorisierung von Schleifen

- Ein weiterers Beispiel: Source: ~iwi412/tmp/loop20.f90

```
DO i = 2,n
   a(i) = SQRT(a(i))
   b(i) = b(i-1)
ENDDO
```

Hier ist aufgrund der rekursiven Referenz im 2. Statement eine Vektorisierung nicht möglich. Die Schleife wird zerlegt in zwei Schleifen, eine wird vektorisiert, die andere sequentiell abgearbeitet, wobei die zweite x-fach entrollt wird. Das Entrollen dient nur dem Zweck, das Ergebnis im Vektorregister der vorhergehenden Instruktion direkt nutzen zu können.

Bitte ausprobieren!

Vektorisierung von Schleifen

- Registerlängen
sind in Vektorrechnern begrenzt. Sind die Vektoren im Programm länger als die Registerlänge, dann werden die Vektoren gemäß der Möglichkeiten der Hardware in Teilvektoren aufgespalten (*stripmining*).
  - Registerlänge der T90: 128 Elemente (= DOUBLE PRECISION)
  - Registerlänge der VPP: variabel
    - von 8 Registern zu 2048 Elemente bis zu
    - 256 Registern mit 64 Elementen

  jeweils in 2er Potenzen konfigurierbar.
- bei der VPP sollte man - wenn möglich - dem Compiler die maximale Länge einer Schleife mit einer Compilerdirektive mitteilen. Dadurch kann der Code von unnötigen Abfragestatements befreit werden:

```
!OCL REPEAT(maxlength)
```

Beispielprogramm: ~iwi412/tmp/loop30.f90

Vektorisierung von Schleifen

- Conditional Vector Loop
Wie wir bereits am Beispiel Folie 11 gesehen haben, kommt es vor, daß die Vektorisierbarkeit von Kontextbedingungen abhängig sind, die erst zur Laufzeit bekannt sind. Der Compiler kann in solchen Fällen keine Entscheidung treffen, also nicht vektorisieren. Beispiel:

```
DO i = 1,n-k
   a(i) = a(i+k)
ENDDO
```

ist vektorisierbar, falls k >= 0 ist. Ist das der Fall, kann dem Compiler mit der Direktive

```
!OCL NOVREC(a)     ! NO Vector RECursion
DO i = 1,n-k
   a(i) = a(i+k)
ENDDO
```

mitgeteilt werden, daß die Schleife vektorisiert werden kann.

Beispiel: ~iwi412/tmp/loop35.f90

Vektorisierung von Schleifen

- Conditional Vector Loop(cont)
Steht der Wert von k tatsächlich erst zur Laufzeit fest, können keine Direktiven angesetzt werden. Die Schleife ist dann vektorisierbar für
  - k <= 0 oder k < -n

und nicht oder nur teilweise vektorisierbar für
  - -n <= k <= -1.

Eine solche Konstruktion wird als *Conditional Vector Loop* bezeichnet. Bei der Schleife (Feld a mit 1000 Elementen; ebenfalls loop35.f90)

```
Do k = -128,128
   DO i =129,872
       a(i) = a(i+k)
   ENDDO
ENDO
```

verhält sich der VPP-Compiler wie folgt:

Vektorisierung von Schleifen

- Conditional Vector Loop(cont)
  Selbst der höchste Optimierungslevel “-Of” bringt den Compiler nicht dazu, die Schleife geeignet zu vektorisieren.

  Mit der Option **-Wv -Ov** oder mit der Compilerdirektiven **!OCL VDOPT** kann man jedoch den Compiler veranlassen, sowohl vektorisierten als auch sequentiellen Code zu generieren. Zur Laufzeit wird in Abhängigkeit von k entschieden, welcher Code abgearbeitet wird.
  (Overhead durch Abfragestatements!)

- Falls der Programmierer sicher weiß, daß k immer > 0 ist, ist es sinnvoller, mit

```
!OCL NOVREC(a)
```

  für eine Vektorisierung ohne dynamische Abfragen zu sorgen.

5.2.3 Umordnung von Schleifen

- In manchen Fällen führt eine einfache Umordnung der Statements innerhalb einer Schleife zur Vektorisierbarkeit. Dieser Fall soll anhand eines Beispiels (~iwi412/tmp/loop50.f90) diskutiert werden:

```
DO i = 2,n
   b(i) = a(i-1)
   a(i) = 2.0 * a(i)
ENDDO
```

- Diese Schleife ist aufgrund der *previous minus* Datenabhängigkeit so nicht vektorisierbar. Der Compiler erkennt jedoch die Möglichkeit der Umordnung und bringt folgende Analyse:

```
0000016 v      LOOP50:     DO i = 2,n
0000017 v                       b(i) = a(i-1)
0000018 v                       a(i) = 2.0 * a(i)
0000019 v                  ENDDO LOOP50

jpc2217i-i     “loop50.f90”, line 17-18:Recursive reference (a)
jpc1001i-i     “loop50.f90”, line 17-19:Vectorized by DO var i
```

Umordnung von Schleifen

- Hat der Compiler tatsächlich die Statements umgeordnet wie vermutet? Zur Bestätigung ordnen wir selbst um und erhalten das gleiche Ergebnis, wie zuvor.
- Als Gegenbeweis zwingen wir den Compiler mit Compilerdirektiven die ursprüngliche Schleifenkonstruktion zu vektorisieren:

```
!OCL NOVREC(a)
DO i = 2,n
   b(i) = a(i-1)
   a(i) = 2.0 * a(i)
ENDDO
```

Die Ergebnisse sind gegenüber den vorangegangenen Beispielen andere (falsche!!); also muß der Compiler ein Umordnung vorgenommen haben.

Umordnung von Schleifen

- Vorsicht bei *f90-array-syntax:*

b(2:n) = a(1:(n-1))

a = 2.0 * a

Diese Konstruktion wird (natürlich) vektorisiert. Falls das Ergebnis nicht der Intension des Programmierers entspricht, ist ihm ein Fehler unterlaufen, da er vergessen hat, daß die Semantik einer Schleife und die der Array-Syntax verschieden sind.

5.2.4 Aufspalten von Schleifen

- Bei der Lösung partieller Differentialgleichungen sind folgende Schleifenkonstruktionen typisch (siehe ~iwi412/tmp/poisson.f90):

```
Gauss-Seidel-Relaxation
DO j = 2,n
   DO i = 2,n
      u(i,j) = (f(i,j) + u(i,j-1) + u(i,j+1) &
                       + u(i-1,j) + u(i+1,j)) * 0.25
   ENDDO
ENDO
```

Die *privious minus* Datenabhängig erlaubt es nicht, diese Schleifen zu vektorisieren.

Aufspalten von Schleifen

Der Mittelwertbildung jeweils aus den Nachbarpunkten (5 Punkte-Stern) ließe sich die Datenabhängigkeit nehmen, wenn man zunächst die geraden und dann die ungeraden Punktindizes berechnet, man also das Feld schachbrettartig aufteilt:

```
! RED-BLACK GAUSS SEIDEL
DO j = 2,n
   DO i = 2,n,2
      u(i,j) = (f(i,j) + u(i,j-1) + u(i,j+1) &
                       + u(i-1j) + u(i+1,j)) * 0.25
   END DO
   DO i = 3,m,2
      u(i,j) =(f(i,j) + u(i,j-1) + u(i,j+1) &
                       + u(i-1,j) + u(i+1,j)) * 0.25
   ENDDO
ENDO
```

Diese Lösung ist nicht äquivalent der 1. Konstruktion, was zu einem langsameren Konvergenzverhalten führt.

Aufspalten von Schleifen

- Eine weitere Variante ist das Jacobi-Verfahren:

```
! Jacobi in Array-Syntax
u(2:n,2:n) = (f(2:n,2:n) + u(2:n,1:(n-1)) + u(2:n,3:(n+1)) &
             + u(1:(n-1),2:n) + u(3:(n+1),2:n)) * 0.25
```

was folgender Schleifenkonstruktion entspricht:

```
DO j = 2,n
   DO i = 2,n
      temp(i,j) = (f(i,j) + u(i,j-1) + u(i,j+1) &
                       + u(i-1,j) + u(i+1,j)) * 0.25
   ENDDO
ENDO
   DO j = 2,n
      DO i = 2,n
            u(i,j) = temp(i,j)
      ENDDO
   ENDDO
```


Aufspalten von Schleifen

- Die Messung dieses Beispiels wird zeigen, daß die Laufzeiten vergleichbar sind; es handelt sich also (vermutlich) um die gleiche Lösung.
- **Fazit:**
  Mit dem Aufspalten von Schleifen gewinnt man in diesen Fällen zwar vektorisierbaren Code, nimmt dafür aber schlechteres Konvergenzverhalten - was gleichbedeutend mit höherem Rechenaufwand ist - inkauf.
  Im allgemeinen kann man jedoch davon ausgehen, daß der Gewinn durch die Vektorisierung den Verlust durch schlechtes Konvergenzverhalten überwiegt.
- Besser wäre natürlich, einen Löser zu finden, der ein gutes Konvergenzverhalten zeigt und vektorisierbar ist. Das ist bei den sogenannten Multigridverfahren der Fall.

5.2.5 Reduktion und Rekursion

- Als Reduktionen werden Zuweisungen bezeichnet bei denen sich der Rang der Variablen reduziert; z. B. ein Vektor wird einem Skalar zugeordnet. Typische Beispiele sind:
  - Vektorsumme
  - Produkt der Vektorelemente
  - Skalarprodukt
  - Maximum/Minimum eines Vektors

  Im allgemeinen handelt es sich bei Reduktionen um spezielle Rekursionen vom Typ *previous minus*, sind also eigentlich nicht vektorisierbar. Mit einem “Trick” ist es jedoch wenigstens stückweise möglich, wie das folgende Beispiel zeigt.
  Gegeben ist folgende Schleifenkonstruktion zur Bildung der Vektorsumme:

```
s = 0.0
DO i = 1,n
   s = s + a(i)
ENDDO
```


Reduktion und Rekursion

- Ein Compiler geht dabei etwa wie folgt vor:

```
DO i = 1, i_teilsumme
   temp(i) = 0.0
ENDDO
DO j = 1,n,i_teilsumme
   DO i = 1,128                          ! z. Beispiel
      temp(i) = temp(i) + a(j+i-1)       ! vektoriell
   ENDDO
ENDDO
s = 0.0
DO j = 1,i_teilsumme
   s = s + temp(j)
ENDDO
```

  Tatsächlich wird man in diesen Fällen auf Bibliotheksfunktionen zurückgreifen, die - in der Regel sogar in Assemblersprache programmiert - ein Optimum an Performance herauszuholen gestatten.

Lineare Rekursion

- Lineare Rekursion 1. Ordnung

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

läßt sich umformen in

```
temp = a(1)
DO i = 2,n
   a(i) = temp + (i-1)*s
ENDDO
```


Lineare Rekursion

- Lineare Rekursion 2. Ordnung

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

lassen sich durch wiederholtes Einsetzen in eine Form bringen, die Schleifen mit der Vektorlänge k vektorisierbar macht:

```
DO i = k+1,n
   a(i) = f(i) * a(i-k) + y(i)
ENDDO
```

5.2.6 Indirekte Adressierung

- Bisher haben wir nur Beispiele behandelt, die eine Datenabhängigkeit zwischen Elementen eines Vektors aufwiesen. Ein weiterer wichtiger Fall, der eine Vektorisierung erschwert oder sogar verhindert, ist die indirekte Adressierung und die nichtlineare Adreßfortschaltung.
  Beispiele: ~iwi412/tmp/index.f90

```
DO i = 1,m                      ! Indirekte Adressierung
   b(i) = a(index1(i)) + c(i)
ENDDO
DO i = 1,m                      ! Indirekte Adressierung
   b(index2(i)) = a(index1(i))
ENDDO
DO i = 1,k                      ! Nichtlineare Adress.
   b(i**2) = a(i) + c(i)
ENDDO
```

Diese Schleifen werden auf der VPP vektorisiert, die Maschine verfügt über indirekte LOAD/STORE Operationen.

Indirekte Adressierung

- Erfolgt die indirekte Adressierung nicht über einen Indexvektor, wie in den ersten beiden Beispielen, sondern als Indexfunktion, so erfolgt eine Vektorisierung u. U. dann, wenn eine *intrinsic_function* verwendet wird. Handelt es sich um eine *external_function* kann der Compiler keine Vektorisierung vornehmen.
  Eine Lösung ist dann, die Funktion als *inline_code* zu ersetzen. Dies kann man an der VPP mit einer Compileroption (nicht Direktive!) veranlassen (~iwi412/tmp/loop60.f90):

```
DO i=1,m
   b(i) = a(index1(i)) * s
ENDDO
! mit der index - Funktion:
INTEGER FUNCTION index1(i)
   INTEGER i
   IF (i<100) THEN
      index1 = i*i
   ELSE
      index1 = 2*i
   ENDIF
END FUNCTION
```

Indirekte Adressierung

- Compileroption zum *“inlinen”* der Funktion index1:

```
–Ne, index1
```

- Man erhält dann in etwa folgende Analyse:

```
0000020 v       LOOP60:     DO i = 1,m
0000021 vi                        b(i) = a(index1(i)) * s
0000022 v                   ENDDO LOOP60
```

vi: zeigt an, diese Schleife wurde vektorisiert und es wurde ein *inlining* vorgenommen.

Indirekte Adressierung

- Wenn auf beiden Seiten einer Zuweisung der gleiche Indexvektor vorkommt

```
DO i,1,m
   a(index1(i)) = a(index1(i)) + b(i)
ENDDO
```

dann wird diese Schleife nicht automatisch vektorisiert, da nicht auszuschließen ist, daß auf gleiche Elemente zugegriffen wird.
Kann man das als Programmierer ausschließen, so kann man eine Vektorisierung mit

```
!OCL   NOVREC(a)
```

- erzwingen oder aber mit einer Zusicherung der Art

```
0000026! OCL    DISJOINT(a)
0000028 v       DO i = 1,m
0000029 v             a(index1(i)) = a(index1(i)) + b(i)
0000030 v       ENDDO
```

das gleiche erreichen und zusätzlich noch etwas für die Dokumentation tun.

5.2.7 IF-Statements in Schleifen

- IF-Statements in Schleifen sind immer ein Vektorisierungshemmnis und wirklich nur dann zu verwenden, wenn dies unabdingbar ist. Für die Vektorisierung von Schleifen mit IF-Statements gibt es 3 Methoden
  - die *List Vector* Methode,
  - die *Compress/Expand* Methode und
  - die *Mask* Methode

  Die Verfahren sind bei den verschiedenen Vektorrechnern ähnlich, die Namen der Methoden jedoch mit Sicherheit verschieden! Wir wollen anhand des folgenden Beispiels die verschiedenen Methoden vergleichen:

```
DO i = 1,m
   IF (a(i) < 1.0) THEN
      b(i) = c(i) + d(i)
   ENDIF
ENDDO
```


IF-Statements in Schleifen

- Die *List Vector* Methode

```
i                     1 2 3 4 5 6 7 8 9
LOAD a(i) nach VR1    1 0 1 2 3 0 0 0 1
MASK (MR1)            1 0 1 1 1 0 0 0 1   1=true; 0 = false
INDEX (VR2)           1 3 4 5 9
c(i)                  1 2 2 2 3 4 6 0 1
d(i)                  3 4 4 5 1 2 3 4 6
LOAD c(INDEX) to VR3  1 2 2 3 1
LOAD d(INDEX) to VR4  3 4 5 1 6
ADD VR1, VR2 to VR5   4 6 7 4 7
STORE VR5 to b(INDEX) 4 . 6 7 4 . . . 7
```

Die VPP setzt diese Methode dann ein, wenn die *true-ratio* des *IF-Statments* gering ist und sich der Overhead für das indirekte Laden rentiert.

IF-Statements in Schleifen

- Die *Compress/Expand* Methode

```
i                          1 2 3 4 5 6 7 8 9
LOAD a(i) nach VR1         1 0 1 2 3 0 0 0 1
MASK (MR1)                 1 0 1 1 1 0 0 0 1  1=true; 0 = false
c(i)                       1 2 2 2 3 4 6 0 1
d(i)                       3 4 4 5 1 2 3 4 6
LOAD c to VR2
COMPR VR2,MASK to VR3      1 2 2 3 1
LOAD to VR4
COMPR VR4,MASK to VR5      3 4 5 1 6
ADD VR3, VR5 to VR6        4 6 7 4 7
STORE VR6, MASK to b       4 . 6 7 4 . . . 7
```

Die VPP setzt diese Methode im Gegensatz zur Cray T90(*mit HW-gather/ scatter*) sehr selten ein. Die Methode ist bei der VPP für geringe bis moderat hohe *true-ratio* geeignet.

IF-Statements in Schleifen

- Die *Mask* Methode

```
i                          1 2 3 4 5 6 7 8 9
LOAD a(i) nach VR1         1 0 1 2 3 0 0 0 1
MASK (MR1)                 1 0 1 1 1 0 0 0 1   1=true; 0 = false
LOAD c to VR2              1 2 2 2 3 4 6 0 1
LOAD d to VR3              3 4 4 5 1 2 3 4 6
ADD VR2, VR3 to VR4        4 6 6 7 4 6 9 4 7
STORE VR4, MASK to b       4 . 6 7 4 . . . 7
```

Es ist offensichtlich, daß diese Methode nur bei hoher *true-ratio* sinnvoll ist. Diese Methode funktioniert natürlich nur, wenn die “ansich überflüssigen Operationen” zu keiner *“exception”* führen, bzw. diese abgefangen wird; wie z. B. Division durch 0.

```
DO i = 1,n
   IF (a(i) > 0.0) a(i) = 1.0/a(i)
ENDDO
```

IF-Statements in Schleifen

- Steuerung der zu wählenden Methode
  - durch Compileroptionen

```
frt -Wv, -I[m|l|c]
```

  - besser jedoch durch Compilerdirektiven

```
!OCL   VCT(LIST)
oder
!OCL   VCT(CEX)
oder
!OCL   VCT(MASK)
```

  - oder am besten, es dem Compiler zu überlassen und ihm die *true-ratio* anzugeben:

```
! OCL  IF(62)
```


5.2.8 Basisoperationen der Linearen Algebra

- Typischerweise verwendet man für diese Operationen die Unterprogramme aus den Programmbibliotheken, wie IMSL, NAG, LAPACK oder SCALAPACK in der die auf die Rechnerarchitektur optimierten BLAS-Routinen enthalten sind. Man unterscheidet die BLAS - Routinen (= Basic Linear Algebraic Subroutines), gemäß ihrer Ordnung
  - in BLAS - Level1 O(N)
    z. B. Vektorsumme, Skalarprodukt
  - in BLAS - Level 2 O(N^2)
    z. B. Matrix-Vektor-Multiplikation
  - und in BLAS - Level 3 O(N^3).
    z. B. Martix-Multiplikation
- Achtung Cray T90 - Nutzer:
  Bei der VPP ist REAL = 4 BYTE; man muß also bei der VPP die DOUBLE PRECISION - Routinen für 8 Byte - Floating-Point-Zahlen verwenden.

5.2.9 Optimierung von Speicherzugriffen

- Bankkonflikte
  Wie bereits im Kapitel 5.1 (Fujitsu-Hardware) beschrieben, verwendet die VPP (im Gegensatz zur Cray) Standard-Speicherbausteine (SDRAMS) mit relativ langen Zugriffszeiten. Zwei aufeinander folgende Speicherzyklen kosten dabei etwa 20 Taktzyklen, in sehr ungünstigen Situationen sogar noch mehr.
- Abhilfe schafft dabei ein hohes Interleaving in 512 Speicherbänke, wobei aufeinanderfolgende Speicheradressen über die Speicherbänke verteilt werden. Jedoch zeigt sich ein Erfolg nur, wenn der Zugriff auf die Daten ebenfalls in geeigneter Weise auf die Speicherbänke verstreut ist.

Optimierung von Speicherzugriffen

- Mit dem folgenden Testbeispiel kann man die Effekte untersuchen:

```
DO i = 1, m-1,memstride
   a(i) = b(i) * s + c(i)
ENDDO
```

Bei der VPP ist zu beachten, daß die Maschine nur einen Load- und einen Storepfad zum Memory hat (im Gegensatz dazu die Cray T90: 2 x Load/1 x Store), so daß möglichst nur in Einer-Schrittweiten auf das Memory zugegriffen werden sollte.

Optimierung von Speicherzugriffen

- Bei mehrdimensionalen Feldern ist bei der Bearbeitung von Mehrfachschleifen darauf zu achten, daß bei Fortran die Matrizen spaltenweise organisiert sind (C = zeilenweise); d. h. daß darauf zu achten ist, daß der 1. Index immer in der innersten Schleife bearbeitet wird.
- Wenn es aus irgendwelchen Gründen nicht anders geht und die äußere Schleife vektorisiert werden muß, dann entspricht die Adreßfortschaltung der Länge des Feldes in der 1. Dimension. Wenn man dann auch noch eine 2er-Potenz als Feldlänge vereinbart hat, kommt es leicht zu Speicherbankkonflikten. Bei der VPP müßten die ungünstigsten Fälle bei Vielfachen von 64 * DOUBLE PRECISION eintreten. Durch Überdimensionierung läßt sich in solchen Fällen diesen Effekt vermeiden, wie folgendes Beispiel demonstrieren soll:

Optimierung von Speicherzugriffen

- Beispiel:

```
REAL a(128,n), b(128,n)
DO i = 1,m                  ! nach i nicht vektorisierbar
   DO j= 1,n
      a(i,j) = a(i-1,j)
   ENDDO
ENDDO


REAL a(129,n), b(129,n) ! ueberdimensioniert
DO i = 1,m                  ! nach i nicht vektorisierbar
   DO j= 1,n
      a(i,j) = a(i-1,j)
   ENDDO
ENDDO
```

5.2.10 Compilerdirektiven

- Compilerdirektiven sind Zusicherungen an den Compiler, insbesondere über das dynamische Verhalten des Programms. Es handelt sich also um Informationen für den Compiler, die dieser aus dem Quellcode nicht unmittelbar entnehmen kann.
- Compilerdirektiven sind als Kommentarzeilen angelegt und beginnen am Zeilenanfang mit !OCL (=Optimization Contol Line).
- Compilerdirektiven können sich auf Programmeinheiten (P), DO-Loops (D), Array-Expressions(A) und Statements(A) beziehen:

Beispiele:

```
!OCL    NOVREC(a), mt .GT. 5             ...Programm Unit
        SUBROUTINE sub1 (a,b)
!OCL    REPEAT (100)                     ... DO Loop
        DO i = 1,m
!OCL    IF (95)                          ... Statement
        IF (w(i) > 0) THEN
!OCL    SCALAR                           ... Array-Expression
        a(1:m) = b(1:m:2)
```


Compilerdirektiven

Computerdirektive	Bedeutung	möglicher Bereich
Variable.relop. Variable Variable.relop.Const	Beschreibt die Größenbezeichnung von Variablen. Beispiel: !OCL m .GT. 0 DO i = 1,n a(i) = a(i+m) + s ENDDO Diese Schleife wird vektorisiert, da der Programmierer sicherstellt, daß m immer größer Null ist.	PDA
NOVREC[(a1[,a2])]	Beschreibt, daß keine (Vektor)rekursion vorliegt. Falls Arrays A1, A2 usw. angegeben werden, liegt keine Rekursion für diese Arrays vor. Beispiel: !OCL NOVREC(a) DO i = 1,n a(i) = a(i+m) + s ENDDO Diese Schleife wird vektorisiert, da der Programmierer die Verantwortung übernimmt, daß keine Rekursion vorliegt.	PDA

DISJOINT[(a1[,a2])]	Zeigt an, daß eine Arrayreferenz sich nicht über mehrere Schleifendurchläufe bezieht.
	!OCL DISJOINT(b)
	DO i = 1,n
	IF (w(i)) THEN
	a(i+2) = b(I(i))
	ELSE
	b(I(i))=a(i) + 1.
	ENDDO
		PDA

Compilerdirektiven

Computerdirektive	Bedeutung	möglicher Bereich
REPEAT(num)	Definiert die maximale Iterationsanzahl einer Schleife	PDA
SCALAR(num) VECTOR	Schleifen mit einer Iterationsanzahl kleiner num, werden nicht vektorisiert, wenn num fehlt, werden die Schleifen skalar ausgeführt. VECTOR hebt die Wirkung von SCALAR auf.	PDA
VI(i)	Gibt die Schleifenvariable an, über die innerhalb von geschachtelten Schleifen vektorisiert werden soll.	PD
EVAL/NOEVAL	Erlaubt bzw. verbietet die Umordnung bei der Auswertung von Ausdrücken	PD
TEMP(s1[,s2])	Gibt die Variablen an, die nur als temporäre Variablen in einer Schleife zu betrachten sind.	PD
UNROLL (level UNROLL("FULL") NOUNROLL	Gibt die Tiefe des Unrolling einer Schleife an.	PDA
VDOPT NOVDOPT	Es wird zur Laufzeit entschieden, ob eine Schleife skalar oder vektoriell ausgeführt wird. Der Compiler erzeugt sowohl skalaren als auch vektoriellen Code. NOVDOPT schaltet diese Möglichkeit aus.	PDA

Compilerdirektiven

Computerdirektive	Bedeutung	möglicher Bereich

IF(percentage)
WHERE(percentage)
CASE(percentage)
	Gibt den (ungefähren) Prozentsatz der "wahren" Fälle eines IF-, WHERE- oder CASE Konstruktes an. Der Compiler wählt danach eine optimale Vektorisierungsstrategie aus.
	Beispiel:
	`       DO i = 1,n`
	`!OCL   IF(95)`
	`         IF (w(i)) THEN`
	`           (a(i) = 2.`
	`         ELSE`
	`           a(i) = a(i)+ 1.`
	`       ENDDO`
		A

VCT(MASK) VCT(LIST) VCT(CEX)	Die Vektorisierung einer Schleife mit IF soll nach folgenden Methoden erfolgen: - MASK: Masked Operations (besser geeignet bei hohen TRUE-Werten) - LIST: List Vector Operations (besser geeignet bei geringen TRUE-Werten) -CEX: Compress/Expand Operations	PDA
VCT(VFORCE) VCT(SFORCE) VCT	Eine Schleife, die nicht vektorisierbare Konstrukte enthält, wird immer partiell vektorisiert (VFORCE) bzw. vollständig skalar (SFORCE) ausgeführt. Wird nur VCT angegeben, entscheidet der Compiler wie eine solche Schleife vektorisiert wird.	PDA
VCT(VOL) VCT(NOVOL)	Geschachtelte Schleifen werden zu einer einzigen zusammengefaßt bzw. dieses Zusammenfassen wird unterbunden.	PDA

5.2.11 Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Auswahl eines geeigneten Algorithmus unter Berücksichtigung von Bibliotheks-Routinen und anderen Quellen wie z. B.
  http: //gams.nist.gov
  http: //www.netlib.org
  http: //www.nhse.org

- Übersichtliche, wohlstrukturierte (goto-freie), lesbare Programme, frei von "sinnwidrigen" Konstrukten.
  "Tricksen" ist nur erlaubt, wenn dadurch bessere Vektorisierbarkeit erreicht wird. Wenn der Trick nicht unmittelbar einsehbar ist, sollte er dokumentiert sein. Das heißt aber keineswegs, daß man den Compiler "aus"-tricksen soll! Dem Compiler geht es wie den meisten Menschen, wenn es ums Verstehen geht.

Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Klare Indizierung; keine unnötigen Zwischenrechnungen, z. B.

```
a(i+4,j-3,k*2) = ...
! anstatt
ia = i + 4
ja = j - 3
ka = k * 2
a(ia,ja,ka) = ....
```

- Anordnung von Schleifen
  - bei der innersten Schleife den Indexraum möglichst groß wählen, um lange Vektoren zu gewinnen,
  - in geschachtelten Schleifen sollte die innere Schleife auf dem 1. Index arbeiten,
  - auch die weiteren Indizes von links nach rechts abarbeiten, wenn man von der innersten zur äußersten Schleife fortschreitet.

Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Vertauschen von Schleifen
  Obwohl dies in der Regel von den Compilern erkannt wird, ist es doch ratsam die Umordnung von Schleifen selbst vorzunehmen. Es dient der Lesbarkeit des Codes und macht das Analysefile einfacher.

- Konstanten statt Variablen
  Wenn man Konstante verwendet, soll man sie auch mit PARAMETER... vereinbaren, anstatt DATA oder Zuweisung an eine Variable. Dies kann den Compilern die Vektorisierung erheblich vereinfachen.

- Dynamische Speicherverwaltung (dynamische arrays) sind für Vektorisierungsentscheidungen hinderlich und deshalb zu vermeiden.

- Loop Unrolling
  Ziel von *Unrolling* ist es, redundante Load-Store-Operationen zu vermeiden. Überlassen Sie das dem Compiler, der kann das besser!

Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- *Loop Unwinding*
  nennt man die Maßnahme, eine kleine innere Schleife in eine größere äußere Schleife zu integrieren um so längere Vektoren zu erreichen. Das ist immer von Vorteil!

- *Loop Fusion*
  nennt man die Vereinigung mehrerer unabhängiger Schleifen in eine Schleife, wenn es gelingt datenabhängigkeitsfrei zu bleiben und der Indexraum der gleiche ist.

  Bei der VPP kann man dies auch vom Compiler erledigen lassen mit den Optionen

```
frt -Wv, -fus
```

  zur Analyse;
  danach sollte man den Code entsprechend umformulieren, wenn es ein Produktionscode werden soll.

Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Unterprogrammaufrufe
  Scheitert die Vektorisierung am Aufruf eines Unterprogramms, dann sollte man versuchen die Schleife in das Unterprogramm hereinzuziehen oder aber das Unterprogramm als *In-Line-Code* in die Schleife zu ziehen.

- IF-Statements
  - sollte man in Schleifen vermeiden.
  - Ist das unvermeidbar dann sollte man
    - die IF-Blöcke in der Reihenfolge der *true-ratio* ordnen und
    - dem Compiler durch Compiler-Direktiven die *true-ratio* mitteilen.

- Modularisierung
  Modularisierung sollte nicht übertrieben werden; Data, Save und Common vermeiden, Module verwenden.

Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- *Mixed Mode Expressions* vermeiden!

- DOUBLE PRECISION
  Genauigkeit beachten: Cray Standard 64 Bit; VPP Standard: 32 Bit
  REAL/COMPLEX(kind=16) werden von der VPP nicht vektorisiert.

- INTEGER
  kind = 1,2,8 werden nicht vektorisiert

- *Strengh Reduction*
  Zeitaufwendige Operation durch einfachere ersetzen z. B.

```
a * 2 durch a + a
a ** 2 durch a * a
a ** 0.5 durch SQRT(a)
a/2 durch a * 0.5
a/5 durch a * 0.2
LOG(a) + LOG(b) durch LOG(a * b)
```

Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Alternierende Vorzeichen
  schlecht:

```
DO i = 1,n
   s = s + (-1)**i * a(i)
ENDDO
besser:
DO i = 1,n,2
   s = s - a(i)
ENDDO
DO i = 2,n,2
   s =s + a(i)
ENDDO
```

- Minimum, Maximum, Absolutbetrag, Reduktionen aller Art, etc.
  Bibliotheksroutinen verwenden

Strategien zur Vektorisierung und Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Compilerdirektiven
  sind zur Unterstuetzung des Compilers da; sie liefern Kontextinformationen, die dem Compiler nicht zur Verfügung stehen.
  Man hüte sich davor zu versuchen den Compiler auszutricksen; daß kann in der nächsten Compiler-Release zu völlig anderen Ergebnissen führen.

5.2.12 Strategien zur Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Mit den bisher aufgelisteten Regeln müßte sich bereits ein gut vektorisierbares Programm implementieren lassen.
  Der nächste Schritt wäre mit Hilfe des Compilers und der Analyse Tools/ Files das Programm weiter zu optimieren.
  - Untersuchen ob Schleifen noch nicht vektorisiert wurden, warum nicht, gibt es Abhilfe?

  ➜ Analyselistings
- Liste ausgewählter Vektorisierungskürzel
  - v vektorisiert
  - vi vektorisiert mit inling code
  - vv vektorisiert array-option
  - s skalar
  - m mixed vektorisiert

Strategien zur Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- Gesamtzeitverhalten messen, ein Programm beginnt ab 400 MFlops und 200 Mips gut zu werden.

```
FJPEPA=ON           oder  csh:  setenv FJPEPA ON
export FJPEPA
a.out
Ergebnis: Performanceanalyse
```

Strategien zur Optimierung von Fortran90-Programm-systemen auf den VPP-Systemen

- Pepa-Output des Codes

```
! OCL UNROLL(4)
   DO j = 1,n
       DO k = 1,n
               DO i = 1,n
                     c(i,j) = c(i,j) + a(i,k) * b(k,j)
               ENDDO
       ENDDO
   ENDDO
```

bringt etwa folgenden Output:

```
....
VU busy ratio                                 [%]          9.939848e+01
PE average Performance                        [MFlops]     2.144664e+03
Operation count of Scalar and Vector UNIT     [times*1e6]  2.000000e+03
Operation count of Vektor UNIT                [times*1e6]  2.000000e+03
Measurement time                              [ sec ]      9.325469e-01
```

Das ist fast die theoretische Höchstleistung, mehr geht nicht!!

Strategien zur Optimierung von Fortran90-Programm-systemen auf den VPP-Systemen

- Sollte man beim Entwurf seines eigenen Programms mehr Performance erwartet haben, dann ist der nächste Schritt festzustellen, in welchen Teilen des Programms die meiste Rechenzeit verbraucht wird. Der SAMPLER ist ein dynamischer Profiler, der in einem vorgegebenen Meßtakt feststellt auf welche Adreßbereiche zugegriffen wird.

```
FJSAMP="file:vppsampler,interval:10,type:vtime,pe:on"
export FJSAMP
a.out
fjsamp a.out
```

Im Output erhält man Information über absolute und prozentuale “Treffer”

- in Unterprogrammen,
- in Schleifen und
- Array-Konstrukten,
- sowie Aussagen über Vektorlängen.

Strategien zur Optimierung von Fortran90-Programm-systemen auf den VPP-Systemen

- Operationscounts analysieren

```
frt -Of -Wv, -m3, -qm -Wc ... -lfjcnt

FJCNT=file:counts
export FJCNT
a.out
fjsamp *.ainf
```


Strategien zur Optimierung von Fortran90-Programm-systemen auf den VPP-Systemen

- Empfohlene Compilereinstellungen
- In der Testphase

```
frt -On -g -sc -Wv,-m3 -NI -A2 -AR -AT -Dasux \
-Eclmeipu -Pidaoxt quelle.f90 ....
```

- In der Optimierungsphase
  Fortschreitend höhere Optimierungsstufen einsetzen:

```
frt -Ob -NI -Wv,-m3,-qs -AR -AT -Eleipu -Post ...
frt -Oe -NI -Wv,-m3,-qm -AR -Eleipu -Post ...
frt -Of -NI -Wv,-m3,-qm -Sw -Eleipu -Pos ...
```

- Einsatz von Optimierungstools wie FHPEPA, FJSAMP, COUNT, etc
- In der Produktionsphase

```
frt -Kfast -Of [-Nsubroutine -No,file] -Wv,-m3,-q[m|v],\
[-Wv,-Gf,-preload,-noalias,-I[c|l|m],-fusion,-ilfunc]...
```

© Claus-Uwe Linster – pps5.2_vektorisierung: 23/3/98

Strategien zur Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- ausgewählte Compileroptionen
  Siehe auch :http://www.uni-erlangen.de/RRZE/dienste/vpp/uxpv-doku/C/fvp/index.htm
  - -On; -Ob; -Oe; -Of: Optimierung: no; basic; extended?: auch Umordnungen; full
  - -sc: keine Vektorisierung
  - -g: es wird fdb Debugging-Information generiert
  - -A2: Statementnummer bei Fehlern
  - -AR: Allocates constant data in write-protected area
  - -Dxxxx: Diverse Debug-Optionen
  -

Strategien zur Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- ausgewählte Compileroptionen(cont)
  - Wv: Vektorisierung ",-xxx" weitere Optionen" werden an den Vektorisierer weitergegeben, wie:
    - ,-m3: Ausgabe aller Vektorisierungsmaßnahmen und Begründung warum Teile nicht vektorisiert wurden
    - ,-qm: partielle Vektorisierung erlaubt
    - ,-noalias: Vektorisierung Pointervar.
    - ,-fusion: Verschmelzung von Schleifen
    - ,-ilfunc: Inlining von Funktionen

Strategien zur Optimierung von Fortran90-Programmsystemen auf den VPP-Systemen

- ausgewählte Compileroptionen(cont)
  - -Exxxx: Diverse Optionen zu Programmanalyse-Informationen
  - -Pxxxx: Diverse Output-Optionen
    - o Optionsliste
    - s Source Programm Liste
    - t: Statistik
    - x: Cross Reference Liste
    - d: Include File-Name-List
  - -Z filename: Output wird umgelengt in Datei filename
  - -o filename: Dateiname des Objekts
  - -c: nur Objectfile “.o”