Wissenschaftliches Programmieren

Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi

10. Optimierung, Benutzung externer Bibliotheken

Quellen:

- LAPACK und BLAS auf netlib
  http://www.netlib.org/lapack/
  http://www.netlib.org/blas/
- LAPACK95:
  http://www.netlib.org/lapack95

Prioritäten

Prioritäten bei der Entwicklung numerischer Software:

- **Korrektheit**
  (Wahl des richtigen numerischen Algorithmus, korrekte Implementierung)
- **Numerische Stabilität**
  (z.B. Numerischer Fehler schaukelt sich nicht hoch)
- **Genaue Diskretisierung**
  (Diskretisierung muss entsprechend der maximalen Fehlertoleranz gewählt werden)
- **Flexibilität**
  (das Programm und auch Teile des Programms sind wiederverwendbar – dazu gehört auch richtige Dokumentation!)
- **Effizienz**
  Sowohl Zeit- als auch Speicherbedarf sollte nach Möglichkeit minimiert werden
  (meistens Kompromiss nötig, da Zeit- und Speicherbedarf sich oft reziprokal verhalten)

Profiling (Laufzeitoptimierung)

Die Laufzeit des Programmes läßt sich grundsätzlich auf zwei Arten verkürzen:

- Optimierung durch entsprechende Compileroptionen
- **Optimierung durch Codeveränderung** (Umschreiben langsamer Codeteile)

⟶ Es muss zuerst ermittelt werden, welcher Codeteil so langsam ist! (**Profiling**)

Laufzeitarten:

Wahre Zeit (*real time*, *wall clock time*): Zeit, die vergeht bis der Code abgelaufen ist

CPU-Zeit (*cpu time, user time*): Zeit, die der Prozessor insgesammt mit der Bearbeitung der Codebefehle verbringt

```
time ./gauss_elim
:
real    0m2.691s
user    0m0.624s
sys     0m0.112s
```

Wahre Zeit =
CPU-Zeit für das gemessene Programm
+ Prozessortotzeit (Prozessor wartet auf Speicher, I/O, etc.)
+ Zeit, die der Prozessor mit paralleler Ausführung anderer Prozesse (Programme) verbracht hat

Beim Profiling müssen immer CPU-Zeiten verglichen werden.
(Außer Prozess ist I/O-dominiert.)

Teillaufzeitmessung

- CPU-Zeit Messung durch **cpu_time()**

```
# t1 and t2 are real (!) variables
call cpu_time(t1)
call mysubroutine(...)
call cpu_time(t2)
write(*,*) "CPU-secs needed:", t2 - t1
```

Nachteil: Kompliziert, wenn es viele Unterfunktionen gibt, oder die Funktionen sehr oft aufgerufen werden.

Profilinginformationen durch Compileroptionen:

- Mit entsprechenden Compiler- und Linkeroptionen wird während der Laufzeit CPU-Zeit-Information gesammelt.
- Die gesammelten Informationen werden automatisch in die Datei **gmon.out** geschrieben.
- Die Informationen können (nach Ausführung) mit einem Profiler (**gprof**) dargstellt werden.

```
gfortran -p -c accuracy.f90
:
gfortran -p -c gauss_elim.f90
groftran -p -o gauss_elim accuracy.o eq_solver.o ...
./gauss_elim
```

Entsprechende Option beim Compilieren **und** beim Linken setzen

- Programm muss lange genug (über 10s) laufen, damit verlässliche Informationen gesammelt werden können!

Auswertung der Laufzeitinformationen

```
gprof ./gauss_elim

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.96      9.52     9.52        1     9.52     9.52  __eqsolver_MOD_touppertriangle
  0.11      9.53     0.01        1     0.01     0.01  __eqsolver_MOD_substituteback
  0.00      9.53     0.00        1     0.00     9.53  MAIN__
  0.00      9.53     0.00        1     0.00     9.53  __eqsolver_MOD_solvelinearsystem
  0.00      9.53     0.00        1     0.00     0.00  __input_MOD_readinput
  0.00      9.53     0.00        1     0.00     0.00  __output_MOD_writetofile
```

Zeit (%)

Kumulative Zeit

Zeit im Unterprogramm (ohne aufgerufene Unterprogramme)

Aufrufe

Zeit per Aufruf (ohne aufgerufene Unterprogramme)

Zeit per Aufruf (insgesamt)

Name des Unterprogrammes

Module (Intel-Compiler): *modul*_MOD_*unterprogramm*

Auswertung der Laufzeitinformationen (#2)

```
granularity: each sample hit covers 2 byte(s) for 0.10% of 9.53 seconds

index % time    self  children    called     name
                0.00    9.53       1/1           main [2]
[1]    100.0    0.00    9.53       1         MAIN__ [1]
                0.00    9.53       1/1           __eqsolver_MOD_solvelinearsystem [3]
                0.00    0.00       1/1           __input_MOD_readinput [10]
                0.00    0.00       1/1           __output_MOD_writetofile [11]
-----------------------------------------------
                                               <spontaneous>
[2]    100.0    0.00    9.53                  main [2]
                0.00    9.53       1/1           MAIN__ [1]
-----------------------------------------------
                0.00    9.53       1/1           MAIN__ [1]
[3]    100.0    0.00    9.53       1         __eqsolver_MOD_solvelinearsystem [3]
                9.52    0.00       1/1           __eqsolver_MOD_touppertriangle [4]
                0.01    0.00       1/1           __eqsolver_MOD_substituteback [5]
-----------------------------------------------
                9.52    0.00       1/1           __eqsolver_MOD_solvelinearsystem [3]
[4]     99.9    9.52    0.00       1         __eqsolver_MOD_touppertriangle [4]
-----------------------------------------------
```

Funktionsindex

Zeit im Unterprogramm (ohne Unterprogramme)

Zeit in aufgerufenen Unterprogrammen

Anzahl der Aufrufe

Zeit im Unterprogramm

Externe Bibliotheken

- Es gibt zahlreiche mathematische Problemen, für die es bereits fertige Programme oder Bibliotheken (Unterprogrammsammlungen) gibt.
- Man sollte die öffentlich zugänglichen, von vielen Anwendern bereits getesteten Bibliotheken gegenüber die eigens codierten Algorithmen vorziehen. (Sind nicht nur besser getestet, aber oft auch wesentlich schneller.)
- Die Bibliotheken sollten unabhängig vom Projekt sein, Projekt sollte nur durch die öffentlichen Schnittstellen mit der Bibliothek kommunizieren.
- Quellcode der Bibliotheken steht nicht immer zur Verfügung (z.B. kommerzielle Bibliotheken)

Einbettung von externen Bibliotheken (Fortran 95)

- Die Bibliothek ist in Modulen aufgeteilt, via „use“ importiert
- Da die Bibliotheksroutinen in Modulen eingebettet sind, hat der Compiler entsprechende Informationen über
  - Name der aufrufbaren Unterprogramme
  - Anzahl und Typ der Unterprogrammargumente
- Es stehen entweder Fortran 95 Quellcodedateien zur Verfügung, oder mod-Dateien mit Modulinformationen (Compilerabhängig)!

Projekt:

```
program TestExtLibrary
  use MyModule1
  use MyModule2
  use ExtModule
  implicit none
  :
  call extSub(ii, jj)
```

Externe Bibliothek:

```
module ExtModule

contains

  subroutine extSub(arg1, arg2)
  :
```

Einbindung von externen Bibliotheken (F77/C) (#1)

- Name des Unterprogrammes in der externen Bibliothek und Anzahl und Typ dessen Argumente sollten in einem „interface“ deklariert werden, damit diese beim Compilieren der übeprüft werden können. (keine mod-Dateien)

```
program testlibrary
  implicit none
  :
  interface
    subroutine extsub(arg1, arg2)
      integer, intent(in) :: arg1
      integer, intent(out) :: arg2
    end subroutine extsub

    integer function extfunc(arg1)
      integer, intent(in) :: arg1
    end function extfunc
  end interface

  integer :: ii, jj
  call extSub(ii, jj)
```

Externe F77-Bibliothek:

```
subroutine extsub(m1, m2)
  integer :: m1, m2
  :
end subroutine extsub

integer function extfunc(inp)
  integer :: inp
  :
end function extfunc
```

- Name der Argumente ist egal.
- Explizite Schnittstelle kann *sinngemäß* „intent“ Attribute definieren, auch wenn diese in der externen Bibliothek nicht explizit definiert sind.

Einbindung von externen Bibliotheken (F77/C) (#2)

- Am besten werden die expliziten Schnittstellen für eine Bibliothek in einem Modul deklariert und das Modul überall eingebunden, wo Bibliotheksunterprogramme aufgerufen werden sollten.

```
module extbib
  implicit none

  interface
    subroutine extsub(arg1, arg2)
      integer, intent(in) :: arg1
      integer, intent(out) :: arg1
    end subroutine extsub

    integer function extfunc(inp)
     :
  end interface

end module extbib
```

```
module my,odul1
  use extbib
  implicit none

contains

  subroutine test()

    integer :: ii
    call extSub(1, ii)

  end subroutine test
   :
end module MyModul1
```

- Bei C-Bibliotheken sollte in der Interface-Deklaration das Fortran 2003 Konstrukt **bind(c)** verwendet werden. (S. beliebiges Fortran 2003 Handbuch)

Basic Linear Algebra Subprograms (BLAS)

- Grundlegende Operationen für lineare Algebra
- Routinen im **BLAS** bilden drei große Gruppen:
  - **Level 1:** Skalar-, Vektor- und Vektor-Vektor-Operationen
    z.B.: *daxpy()*: alpha * x + y
  - **Level 2:** Matrix-Vektor-Operationen
    z.B. *sgemv()*: matrix multiplied by vector
  - **Level 3:** Matrix-Matrix-Operationen
    z.B. *zherk()*: a*A*transpose(A) + b*C
- Erste Buchstabe gibt den Datentyp an:
  - **s**: single precision (4 Byte)
  - **d**: double precision (8 Byte)
  - **c**: comlex single precision (2*4 Byte)
  - **z**: complex double precision (2*8 Byte)
- Dann folgt (BLAS2, BLAS3) die Beschreibung des Matrixtypes:
  - **GE**: general
  - **SY**: symmetric
  - :

Linear Algebra Package (LAPACK)

- Komplexere Algorithmen für lineare Algebra
  - Beispiele:

    dsteqr – Diagonalisierung reeller, symmetrischer, tridiagonaler Matrizen

    dgesv – Lösung von linearem Gleichungssystem
- Nützt BLAS-Routinen sehr intensiv.
- Namenskonvention für Zahlen und Matrixtypen ähnlich wie bei BLAS
- Die meisten LAPACK-Aufgaben können auf zwei Arten ausgeführt werden:
  - **Vereinfachte Schnittstelle** (*driver routines*)

    Verrichten komplexe Aufgaben, verstecken die Teilschritte vom Benutzer
  - **Berechnungsroutinen** (*computational routines*)

    Teilschritte von komplexen Aufgaben, viele Parameter

    Praktisch wenn nur gewisseTeilaufgaben wiederholt werden sollten.

- **LAPACK-Routinen allokieren keinen Speicher!** Alle übergebene Felder müssen vor dem Aufruf allokiert werden (und eventuell nach dem Aufruf wieder deallokiert werden)
- Es müssen evtl. auch zusätzliche Felder (work-arrays) allokiert und übergeben werden.
- Bei der Verwendung von **LAPACK95** entfällt die Allokierung, da LAPACK95 diese automatisch ausführt.

LAPACK95 – Einbindungsbeispiel

- Enthält bequeme Fortran95 Wrappers für die Lapack-Routinen

```
module eqsolver
  use accuracy
  use f95_lapack
  implicit none

contains

  subroutine solvelinearsystem(aa, bb)
    real(dp), intent(inout) :: aa(:,:), bb(:,:)

    call la_gesv(aa, bb)

  end subroutine solvelinearsystem

end module eqsolver
```

- Ist meistens auf den Unix-Plattformen nicht vorinstalliert (i*m Poolraum und auf LiveCD vorhanden*)
- Da mod-Dateien **compilerabhängig** sind, muss die Bibliothek für jeden Compiler separat installiert werden!

Dokumentation zur LAPACK-Bibliothek, Alternativen

- LAPACK User Manual (gute Übersicht): http://www.netlib.org/lapack/lug/
- LAPACK Man-Seiten: http://www.netlib.org/lapack/manpages.tgz
- LAPACK95-Manual: http://www.netlib.org/lapack95/lug95/
  - LA_GESV (Lösung lin. Gleichungssysteme, analog zu dgesv)
    http://www.netlib.org/lapack95/lug95/node73.html
  - LA_STEV (Diagonalisierung von sym. tridiag. Matrizen, analog zu dsteqr)
    http://www.netlib.org/lapack95/lug95/node233.html

Alternative BLAS/LAPACK-Implementationen

- ATLAS
- gotoBLAS
- praktisch jeder Compilerhersteller bietet eigene optimierte BLAS/LAPACK Bibliothek an
  - Intel (ifort-Compiler): MKL

GNU Scientific Library

- C++, C wrapper vorhanden
- Umfangreicher als BLAS/LAPACK
- Weniger getestet als BLAS/LAPACK
- Lizenz: GPL, kann also nur in GPL Programmen eingesetzt werden

Linken von externen Bibliotheken

- Die Bibliothek muss beim Linken angegeben werden:
- Reihenfolge der Bibliotheken beim Linken ist wichtig:
  Die Objektdateien, die ein Unterprogramm aus einer Bibliothek verwenden, müssen in der Linkreihenfolge vor der Bibliothek sein.
- Analog auch für Bibliotheken, die aufeinander aufbauen (z.B. BLAS muss nach LAPACK kommen!)

```
gfortran -o project mod1.o mod2.o mod3.o -llapack -lblas
```

Linkt das LAPACK library (/usr/lib/**lib**lapack.so.3gf)

Linkt das BLAS library (/usr/lib/**lib**blas.so.3gf)

- Dynamische Bibliotheken (Endung .so) müssen auch zur Laufzeit präsent sein! (dynamisches Linken)
- Statische Bibliotheken (Endung .a) werden ans Programm angefügt, und vergrößern die ausführbare Datei (statisches Linken)
- Ob dynamische oder statische Version einer Bibliothek gelinkt wird, kann durch Linkeroptionen gesteuert werden (z.B. *-static*)

Compilieren und Linken mit LAPACK95

Wenn LAPACK95 per Hand installiert wurde (ist nicht Teil des Standard-Betriebssystems), müssen extra Optionen spezifiziert werden, damit der Compiler sie findet:

- Beim Compilieren der einzelnen Dateien muss das Verzeichnis angegeben werden, wo sich die Mod-Dateien für LAPACK95 befinden:

```
gfortran -c -I/home/pool01/aradi/public/lapack95/include eq_solver.f90
```

.mod-Dateien werden auch unter /home/pool01/aradi/public/lapack95/include gesucht

- Beim Linken muss der Pfad angegeben werden, wo sich die LAPACK95-Bibliothek befindet

```
gfortran -o project -L/home/pool01/aradi/public/lapack95/lib mod1.o
                    mod2.o mod3.o -llapack95 -llapack -lblas
```

Bibliotheken werden auch unter /home/pool01/aradi/public/lapack95/lib/ gesucht

Einbinden von LAPACK95 (/home/pool01/aradi/public/lapack95/lib/liblapack95-gfortran.a)

Einbinden von LAPACK und BLAS (/usr/lib/lapack.a, /usr/lib/libblas.a)

Erzeugung von Zufallszahlen

- Zufallszahlen sind Pseudozufallszahlen, die Zahlenfolge ist deterministisch!
- Bei gleicher Initialisierung (interner Zustand) des Zufallsgenerators kommen immer die selben Zahlen heraus!
- Zufallsgenerator sollte (außer beim Debuggen) „zufällig“ (mit Systemuhr) initialisiert werden.

```
real(dp) :: a1
real(dp) :: array(10)

# (quasi random) initialisation
call random_seed()

# Random number from [0.0, 1.0)
call random_number(a1)
# Convert to range [-5.0, 5.0)
a1 = 10.0_dp * a1 - 5.0

# Filling an array with random numbers [0.0, 100.0)
call random_number(array)
array = array * 100.0_dp
```

Aufgabe

- Schreiben Sie ein Programm ('inputgen'), das Input für das Gaussprogramm erstellt.
- Das Programm soll zwei Zahlen (Anzahl der Gleichungen, Anzahl der b-Vektoren) einlesen, und eine entsprechende Inputdatei ('gauss.inp') mit zufälligen Matrix-/Vektorelementen erzeugen.
- Das Programm soll im selben Verzeichnis wie das Gaussprogramm erstellt werden, und soll dessen Module (z.B. Accuracy) verwenden.
- Makefile soll so modifizert werden, dass mit 'make inputgen' das inputgen-Programm kompiliert wird.

Aufgabe (#2)

- Modifizieren Sie das Gauss-Programm so, dass nur das nötigste am Speicher allokiert wird. Die LR-zerlegte Matrix (sowohl L als auch R) soll in der ursprünglichen Matrix A gespeichert werden, die Lösungen in den entsprechenden b-Vektoren.
- Versuchen Sie das Programm auch ohne Optimierung (-O0) bzw. mit agressiver Optimierung (-O3) compilieren. Machen Sie mal Laufzeitmessungen für die Gesammtlaufzeit (z.B. mit „time“). Gibt es Veränderungen in den Endergebnissen?
- Machen Sie Laufzeitmessungen mit gprof. Wieviel verbrauchen die einzelnen Subroutinen?
- Versuchen Sie Matrizen unterschiedlicher Größe einzugeben. (Erzeugen Sie die Matrizen mit inputgen.)

Aufgabe (#3)

- Schreiben Sie den Gleichungslöser so um, dass er nicht mehr die handcodierte Gausselimination, sondern die LAPACK95-Routine `la_gesv` zur Lösung des linearen Gleichungssystems verwendet.
- Verändern Sie die Makefile, damit die Bibliotheken LAPACK95-, LAPACK und BLAS-Bibliotheken beim Linken mitgelinkt werden. (Checken Sie das Ergebnis ins Repository ein.)
- Machen Sie Tests mit einem Gleichungssystem der Größe 500 x 500. Wie groß ist der Laufzeitunterschied zwischen der LAPACK- und der handcodierten Version?
- Optional: Machen Sie Zeitskalierungstests für beide Programmversionen (z.B. mit Matrixgrößen 500x500, 1000x1000, 2000x2000, ...). Wie skalieren die beiden Programmversionen?