

Lecture 14: Code optimization

1. Introduction: compiler optimization and code optimization
2. Timing programs
3. Optimization of do-loops
4. Optimal array usage

Introduction

- Aim of this lecture:
 - simple recipes to obtain a better (faster) performing code
 - good programming practice (no “hacking” required or desired)
- Optimization during compilation: `f90 -fast`; `f90 -O3`
 - Loop unrolling (example on next page)
 - Inline function calls
 - Should be used once a program is tested and known to work correctly
 - works best on programs with a simple and transparent structure
- Code optimization:
 - Design code with view on good performance

Loop unrolling

Original Loop:

```
DO I=1,20000
  X(I) = X(I) + Y(I)*A(I)
END DO
```

Unrolled by 4 *compiles as*:

```
DO I=1, 19997, 4
  TEMP1 = X(I) + Y(I)*A(I)
  TEMP2 = X(I+1) + Y(I+1)*A(I+1)
  TEMP3 = X(I+2) + Y(I+2)*A(I+2)
  X(I+3) = X(I+3) + Y(I+3)*A(I+3)
  X(I) = TEMP1
  X(I+1) = TEMP2
  X(I+2) = TEMP3
END DO
```

Timing

- Crudest approach: time execution via command “time a.out”

```
moe:/u06/tpapenbr/programs/hw7 $ time a.out
```

```
real    0m1.140s
```

```
user    0m0.010s
```

```
sys     0m0.040s
```

```
moe:/u06/tpapenbr/programs/hw7 $
```

- More detailed approach: time sections of the program with Fortran90
subroutine date_and_time

subroutine date_and_time

```
SUBROUTINE timer(xtime) ! returns time in milliseconds
  INTEGER :: values1(8)
  real(4)  :: xtime

  call date_and_time(values=values1)

  xtime = values1(8) + values1(7)*1000.0 +
values1(6)*1000.0*60.0 &
          + values1(5)*1000.0*60.0*60.0

! values(8): ms
! values(7): s
! values(6): min
! values(5): h
end subroutine timer
```

In the fortran program use the following:

```
call timer (t1)
  ...
  ...
call timer(t2)
write(*,*) 'execution time:', t2-t1
```

Good programming practice

1. Experiment with different compiler optimization levels $-O[n]$ and measure time of performance
2. Examine Fortran source code for further tuning
 - Replace handwritten procedures with calls to equivalent optimized libraries or intrinsic functions
 - Remove I/O, calls, and unnecessary conditional operations from key loops
 - Rationalize tangled, spaghetti-like code to use block IF
 - Examine nested do-loops for order of nesting and correct array usage
 - Check changes by using subroutine `DATE_AND_TIME`

Replace handwritten procedures with calls to equivalent optimized libraries or intrinsic functions

```
real(4):: a(100,200), b(200,50), c(100,50)
```

```
c=matmul(a,b)
```

better than

```
do i=1, 50
  do j=1, 100
    hlp=0.0
    do k=1, 200
      hlp = hlp + a(j,k)*b(k,i)
    enddo
    c(j,i)=hlp
  enddo
enddo
```

Likewise: dot_product

Remove I/O, calls, and unnecessary conditional operations from key loops

If possible, use only assignments in do-loops. This is particularly important for the innermost loop in a nested loop construct. Avoid I/O, function/subroutine calls, IF operations in innermost loops

```
do i=1, 100
  do j=1, 2000
    a=prod(i,j)    !slow
  enddo
enddo
```

```
function prod(i,j)
  integer:: i, j, prod
  prod=i*j
end function
```

```
do i=1, 100
  do j=1, 2000
    a=i*j          !fast
  enddo
enddo
```

Examine nested do-loops for order of nesting and correct array usage

order nested do-loops such that the innermost loop has the highest number of iterations

```
do i=1, 1000      !wrong order of loops
  do j=1, 3       ! 1000 x (1 IF + 1 increment +
    .....       !           3 x (1 IF + 1 increment + N assignments))
  enddo          ! = 1000(8 + 3N) operations
enddo
```

```
do j=1, 3        !correct order of loops
  do i=1, 1000   ! 3 x (1 IF + 1 increment +
    .....       !           1000 x (1 IF + 1 increment + N assignments))
  enddo          ! = 3(2 + 1000(N+2)) operations
enddo
```

Efficient I/O

Write arrays in one write statement.

```
real(4):: a(100)
```

```
write(10,*) a    ! best
```

```
write(10,*) (a(i),i=1,100)
```

```
do i=1, 100
```

```
    write(10,F8.3) a(i)    ! worst: formatted and in loop
```

```
enddo
```

Use unformatted read/write if disk is used as temporary storage

```
open(unit=12, file='temp_storage', form='unformatted')
```

```
write(10,*) bigarray
```

arrays

- **Arrays with fixed shape are faster than assumed-shape arrays**

```
subroutine arrays(a)
  real(4):: a(*,*)
  integer:: n, m
  n=size(a,1)
  m=size(a,2)
```

```
subroutine arrays(a,n,m) !faster
  integer:: n, m
  real(4):: a(n,m)
```

- **Allocation of memory for arrays takes time**
- **Array assignments fastest for arrays with fixed size**

```
real(4):: a(10), b(10)
a=0.0
b=a
```

compare with

```
real(4), allocatable, dimension(:):: a, b
n=k+j
allocate(a(n),b(n))
a(1:n)=0.0
b(1:n)=a(1:n)
```

which is slower

arrays cont'd

arrays in loops: let leftmost array index run in innermost loop

```
do i=1, 1000    ! right order of loops
  do j=1, 1000
    a(j,i) = b(j)+c(j-i)
  enddo
enddo
```

```
do i=j, 1000    ! wrong order of loops
  do i=1, 1000
    a(j,i) = b(j)+c(j-i)
  enddo
enddo
```

Summary

- Keep programs transparent and simple (that's the art)
- use compiler optimizations
- clock your programs
- check order of do-loops, array usage, I/O
- Keep programs portable