Technical University of Denmark

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

High Performance Computing

FORTRAN, OpenMP and MPI

41391

Content

- Day 5:
  - Types of files in HPC.
  - Data transfer.
  - Operations on external files.
  - Recommendations.
  - Submitting jobs (batch systems).
  - Memory and caches.
  - Revision control systems (git)
  - Fortran 2003/2008 extensions.

Types of files in HPC

- Setup file: small ASCII file.
- Input files: large binary/ASCII files.
- Post processing files: single precision snapshots in time/iterations (FORTRAN binary, NetCDF, HDF5, VTK, PLOT3D, …)
- Diagnostic files (small ASCII files; maybe binary + small utility program to convert to ASCII).
- Restart files: full precision image of the memory/state of the problem.
- Auxiliary files.

Data transfer

Conversion

- Data transfer to external files (data outside the main memory) requires a conversion of the internal machine representation (e.g., hexadecimal) to a human readable (ASCII) format.
- The conversion is performed according to an *edit descriptor* contained in a *format description*.

  Example:

  Convert (from hexidecimal) and write the number with the value 0.0045 use the edit descriptor: F10.4

  (10 is the total width of the field, and 4 the length of the fractional part).

Conversion

- The format description is a list of edit descriptors enclosed in parentheses.

  Example:

  ```
  WRITE(*,'(I10,F10.3,A10)') …
  ```

  The format description is a character string

  or in a separate format statement:

  ```
  WRITE(*,10) …
  10 FORMAT(I10,F10.3,A10)
  ```

- Use READ and PRINT/WRITE for input and output.

PRINT only produces output to the screen;
WRITE also allows IO to external files

I/O lists and format definition

- The quantities to be read or written by a program are specified in an *I/O list*. For output they may be expressions, but for input they must be variables.

  Example: to print the integer *i*, the real *f*, and the character *c* use the PRINT statement:

```
      PRINT '(I10,F10.3,A10)',i,f,c

      PRINT 10,i,f,c
10    FORMAT(I10,F10.3,A10)
```

Advice: use PRINT for easy/short debugging print; and/but use WRITE for permanent IO

I/O lists and format definition

Example: INPUT

```
    INTEGER :: j(3)
    CHARACTER(LEN=256) :: form = '(3I10)'
    READ '(3I10)', j
    READ form,j
    READ 10, j
    READ*,j !     list-directed I/O
10  FORMAT(3I10)
```

Advice: always use list-directed I/O (the '*' syntax) for ASCII files and terminal IO.

I/O lists and format definition

Example: OUTPUT

```
INTEGER :: i,ilen
REAL, DIMENSION(10) :: a
CHARACTER(LEN=20)     :: word
PRINT‘(I10)’,i
PRINT ‘(F10.3)’, 3.0+5.0+SQRT(34.)
PRINT ‘(10F10.3)’, a
PRINT ‘(3F10.3)’, a(1:3) ! or (a(i), i=1,3)
PRINT ‘(A,A10)’,‘Here is a word: ‘,word
ilen = LEN_TRIM(word)
PRINT ‘(2A)’,’Here is a word: ‘,word(1:ilen)
```

Unit numbers

- I/O is associated with a UNIT number, a default scalar integer in the range 1 – 99:
  - READ(*u*,*fmt*) *list*
  - WRITE(*u*,*fmt*) *list*

STDERR: unit = 0
STDIN: unit = 5
STDOUT: unit = 6

```
Example:              ! Unit number:
READ(4,*) q           ! Scalar default integer constant
READ(nunit,*) q       ! Scalar default integer variable
READ(i*4+j,*) q       ! Scalar default integer expression
READ(*,*) q           ! The first asterisk (*) implies IO to
                      ! the STDIN/STDOUT
```

Internal files

- *Internal files* allow format conversion between various representations to be carried out by the program in a storage area defined within the program itself.

  Example:

  CHARACTER(LEN=256) :: filename

  INTEGER :: istep

  WRITE(filename,'(A,I6.6,A)') 'post',istep,'.dat'

  OPEN(10,FILE=filename)

  WRITE(10,*) …

I6.6: set aside 6 digits for the number with leading zeros: fx.: post000100.dat

Internal files

Example (*scanning* a file):

```
CHARACTER(LEN=256) :: buffer
INTEGER :: freq
READ(10,’(A)’) buffer
IF (buffer(1:1).NE.’#’) THEN ! Skip comments
    READ(buffer,*) freq ! Read/convert from character to integer
ENDIF
```

Formatted input

- The READ statement:

```
READ([UNIT=]u,[FMT=]fmt[,IOSTAT=ios] &
[,ERR=error-label][,END=end-label]) list
```

Example:

```
   i = 1
   DO
      READ(10,*,IOSTAT=ios,ERR=100,END=200) buffer(i)
      i = i + 1
   ENDDO
100 CONTINUE ! Error label
   WRITE(*,'(2(A,I5))') 'Error reading file in line: ',i,' IOSTAT = ',ios
200 CONTINUE ! End of file label
   WRITE(*,'(A,I6,A)') 'Read ',i-1,' lines'
```

Formatted output

- The WRITE statement:

  WRITE([UNIT=]u,[FMT=]fmt[,IOSTAT=ios] &

  [,ERR=error-label][,END=end-label]) list

  – The components are similar to the READ.
  – If the WRITE uses a unit number that is not opened (using the OPEN(…)) a default file name will be used (fx. ftn<u> or fort.<u>).

List-directed I/O

- List-directed I/O (asterisk format) will use a format defined by the computer system.
- WRITE of long lines may break the line. Hence a corresponding READ of the same line (with a different compiler) might fail!

List-directed I/O

Example:

```
READ(*,*) cbuf1
READ(*,'(A)') cbuf2
```

Will read the two lines:

“FORTRAN 90”

FORTRAN 90

As:

cbuf1 = FORTRAN 90	cbuf2 = “FORTRAN 90”
cbuf1 = FORTRAN	**cbuf2 = FORTRAN 90**

List-directed I/O

- Separators: blank, comma or slash (/).
- If the length of the variable is longer than the string, blanks are padded at the end.

  Example:

  CHARACTER(LEN=256) :: cbuf = ‘test’

  i = LEN_TRIM(cbuf)

  WRITE(*,’(A)’) cbuf ! will write 256 characters

  WRITE(*,’(A)’) cbuf(1:i) ! will write 4 characters

Namelist I/O

- *Namelist* is an annotated I/O list.

  Example:

  ```
  INTEGER :: Nx = 20,Ny = 30
  REAL :: diff, array(5)
  CHARACTER(LEN=20) :: name = ‘string’
  NAMELIST /list/ Nx,Ny,diff,array,name

  WRITE(*,nml=list)
  READ(*,nml=list)
  ```

Namelist I/O

will write/read the file:

```
&LIST
NX = 20
NY = 30
DIFF = 0.0
ARRAY = 0.0E+0  0.0E+0  0.0E+0  0.0E+0  0.0E+0
NAME = string
/
```

- Any element can be left out !
- In any multiple occurrence of an object for namelist input, the final value is taken.
- Lines beginning with ! are comment lines.

Non-advancing I/O

- Non-advancing input and output through the ADVANCE='no' (or 'yes')

  Example:

  WRITE(*,'(A)',ADVANCE='NO') 'Enter dt '

  READ(*,*) dt

Edit descriptors

- Three classes:
  - Data (formatting/conversion).
  - Control (control of leading sign, blank separators).
  - Character-string.
- The data edit descriptors may have repeat counters.

  Examples:

```
WRITE(*,'(4(I5,F8.2))') (i(j),a(j),j=1,4)
```

Edit descriptors

- INTEGER: i-descriptor:
  - iw.m: w = width of the field and m is the least number of digits to be output.

  Example:

  The number: 99 printed with: i5.3 returns: bb099

- For the *i* and all the other numerical edit descriptors, if the output field is too narrow to contain the number to output, it is filled with asterisks (******).

Edit descriptors

- REAL: e,en,es or f-descriptors:
  - fw.d: *w* = width of the field and *d* is the number of decimal points after the comma.
  - ew.d and ew.dee (latter for large exponents).
  - enw.d: (engineering) As the *e*-format, but the decimal exponent is divisible with three (3).
  - esw.d: (scientific): as the *e*-format, but the nonzero significant is greater or equal to one and less than 10.
  - Example: 100*pi will be printed as:

F12.4	314.1593
E12.4	0.3142E+03
E12.4E3	0.3142E+003
EN12.4	314.1593E+00
ES12.4	3.1416E+02

Edit descriptors

- COMPLEX: edited using pairs of f, e, en, og es descriptors.
- LOGICAL: lw-descriptor.
- CHARACTER: a-descriptor:
  - A: width determined by the input/output.
  - Aw: width

  Example: string: STATEMENT (9 chars)

  A9: STATEMENT, A10: STATEMENTb, A8: STATEMEN

Edit descriptors

- General: g-descriptor may be used for any intrinsic data type.
  - gw.d
  - gw.dee
- When used for reals it is identical to the e-descriptor.
- The g-descriptor automatically chooses between the e and f descriptor.
- Derived types are edited by the appropriate sequence of edit descriptors corresponding to the intrinsic types of the type.

Edit descriptors

- Minimal field width editing:
  - If the width is set to zero (i0, f0.3 etc.) the minimal width will be used.
- Control edit descriptors:
  - Embedded blanks may be controlled by the bn (blanks null) or bz (blanks zero) descriptor.

    Example: bb1b4 with bn,i5 returns 14 and

    bz,i5 returns 104.

Edit descriptors

- Control edit descriptors:
  - Leading signs may be controlled by the ss (sign suppress) and sp (sign plus)

    Example: 1000 with sp,en9.2
    returns: +1.00E+03
  - Scaling factor: kp-editor, where k is a default integer literal constant.
  - Example: 1000 with 3p,e9.2 returns: 100.0E+01
  - Colon editing: will terminate format control if there are no further items in an I/O list.

```
program main
integer :: n
integer, dimension(5) :: l

n = size(l)
do i=1,n
   l(i) = 3*i
enddo
n = 3
print '(" l1 = ",i4, : , &
        " l2 = ",i4, : , &
        " l3 = ",i4, : , &
        " l4 = ",i4, : , &
        " l5 = ",i4)',(l(i),i=1,n)
end program main
```

Unformatted I/O

- Formatted I/O has a large overhead for the conversion and the file size is significantly larger than the corresponding binary representation of the data.
- Unformatted I/O:

  Example:

  ```
  REAL(MK), DIMENSION(100,100) :: array
  OPEN(10,FILE='binary.dat',FORM='UNFORMATTED')
  WRITE(10) array
  or
  READ(10) array
  ```

Direct-access files

- The default file access in FORTRAN is SEQUENTIAL – that is data can only be inserted at the end of the file.
- Direct-access files allows to insert or read an arbitrary record.

Operations on external files

External files

- FORTRAN is connected to a file through the *UNIT* number. A unit number must not be connected to more than one file at once, and a file must not be connected to more than one unit at once.
- Notice also:
  - The set of allowed names for a file is processor dependent.
  - A file never contains both formatted and unformatted records. (FORTRAN 2003 allows ACCESS=‘STREAM’)

File positioning

- The backspace statement:
  - BACKSPACE([unit=]u[,iostat=ios][,err=error-label])

  Position the file before the current record.

  This statement is often very costly in computer resources and should be used as little as possible.
- The REWIND statement
  - REWIND([unit=]u[,iostat=ios][,err=error-label])

  Will rewind the file to the beginning of the file.

File positioning

- The ENDFILE statement
  - ENDFILE([unit=]u[,iostat=ios][,err=error-label])

  Will write an End Of File (EOF) to the unit (thus effectively truncate the file).
- Data transfer statements:
  - READ
  - WRITE or PRINT

The OPEN statement

- The OPEN statement is used to connect an external file to a unit, create a file that is pre-connected, create a file and connect it to a unit, or change certain properties of a connection:

  OPEN([unit=]u[,olist])

  – iostat=ios, ios is zero on success.
  – err=err-label: label of the statement in the same scoping unit to which control is transferred if an error occurs.

The OPEN statement

- status=st, where st is a character expression that provides the value: old, new, replace, scratch, or unknown. With “scratch” the file= is illegal and the file is deleted by the matching CLOSE([unit=]u).
- access=SEQUENTIAL or DIRECT.
- form=UNFORMATTED or FORMATTED (default).
- recl=rl the record length.
- blank=NULL or ZERO.
- position=ASIS, REWIND, or APPEND.
- action=READ, WRITE or READWRITE.
- delim=QUOTE, APOSTROPHE, or NONE (default).
- pad=YES (default) or NO.

The OPEN statement

Example:

```
OPEN(10,FILE=‘bin.dat’,FORM=‘UNFORMATTED')
WRITE(10) array
OPEN(11,FILE=‘ascii.dat’)
WRITE(11,*) array
```

The CLOSE statement

- The CLOSE statement is used to disconnect an external file to a unit:

  CLOSE([UNIT=]u[,iostat=ios][,err=err-label][,status=st])
  - Status=’delete’ or ‘keep’ (default)

  Example:

  How to remove an external file in FORTRAN:

  OPEN(10,FILE=‘ABORT’)

  CLOSE(10,STATUS=‘DELETE’)

The INQUIRE statement

- The INQUIRE statement is used to inquire about the status and attributes of a file.
  - INQUIRE([unit=]u,ilist) or
  - INQUIRE(file=fln, ilist)

  Where u refers to the unit number and fln to the

  filename. The ilist options include:
  - iostat
  - exist=.TRUE. If the file exist – otherwise .FALSE.

The INQUIRE statement

- opened=.TRUE. If the unit or filename are opened.
- number=the unit number assigned to the file. It returns -1 if it is not assigned.
- named=true if the file has a name.
- name=nam the name of the file.
- access=SEQUENTIAL/DIRECT (char).
- sequential=YES or NO
- form=FORMATTED or UNFORMATTED
- formatted=YES or NO (char).

The INQUIRE statement

- recl=record length
- nextrec=the number of the last record read or written.
- blank=NULL or ZERO.
- position=REWIND, APPEND or ASIS.
- action=READ, WRITE or READWRITE.
- read=YES, NO or UNKNOWN.
- write=YES, NO or UNKNOWN.
- readwrite=YES, NO or UNKNOWN.
- delim=QUOTE, APOSTROPHE or NONE.
- pad=YES or NO.

Recommendations

Use STANDARD tools

- Use STANDARD FORTRAN 90/95 (move to FORTRAN 2003 if/when needed).
- Use STANDARD MPI/OpenMP.
- Use STANDARD cpp for source preprocessing.
- Use STANDARD make for compilation.
- USE STANDARD editors (vi, emacs, nano, gedit).
- Use STANDARD revision control systems (rcs,cvs/svn,git).

Compilation

- Read the *man* pages for the compiler.
- Initially compile with –C to check for array bound violation.
- When –C passes switch to –O3 (or more).
- Profile the code using compiler instructions (f90 -p … or similar) or through call to CPU_TIME().
- Compile on different platforms and with different compilers during the development of the code.

Style

- Write all FORTRAN keywords in capital letters.
- Indent 3 spaces after branches, loops etc..
- Do NOT use the tab – use the spacebar !
- Limit the length of lines to maximum 80 characters.
- A name should not exceed 31 characters.
- Keep variables names short, but meaningful.
- Variable names of length one should ONLY be used for loop counters (fx.: i,j,k)
- Add comments in the source code to describe variables.
- RCS/CVS/SVN: consider using Log in the header.

Coding

- Use IMPLICIT NONE
- Define the precision in an include file ‘param.h’ or module

```
INTEGER, PARAMETER :: MK = KIND(1.0E0)
```

- Declare the floats using the MK:

```
INCLUDE ‘param.h’
REAL(MK) :: value
REAL(MK), DIMENSION(:), POINTER :: array
value = 1.0_MK
```

Coding – Implicit interface

- Use ONE source file for ONE subroutine.
- Encapsulate each subroutine in a MODULE.
- Will automatically build an interface – so you NEVER need to type things twice = you NEVER need to type INTERFACE blocks.
- Subroutines in modules can only be accessed by USE – so you are forced to use the (implicit) interface; with explicit interfaces you might forget.
- Declare ALL arrays local to a subroutine in the MODULE which encapsulates the subroutine.
- Arrays in MODULEs will be allocated on the HEAP – automatic/local arrays can/will be allocated on the STACK !

Coding – explicit interface

- Use ONE source file for ONE subroutine.
- Write an interface file for each routine.
- Use separate module files for each subroutine to hold local data or write the module in the FORTRAN file but without including the subroutine in the module (no CONTAINS)

```
MODULE m_sub
        REAL, DIMENSION(:), POINTER :: work
END MODULE m_sub
SUBROUTINE sub(x)
USE m_sub
REAL, DIMENSION(:), POINTER :: x
```

Coding

- Do NOT use AUTOMATIC arrays!
- Do NOT allocate LOCAL arrays – but put them in a MODULE.
- Use the ONLY and PRIVATE module options.
- ALWAYS check the status of ALLOCATE !
- Remember to DEALLOCATE.

Coding

- Use cpp to encapsulate hardware/library specific source:

```
#ifdef __LAPACK
  CALL LAPACK_ROUTINE(…)
#else
  CALL  MY_BEST_VERSION(…)
#endif
```

- All cpp directives should start with # in column 1.
- Do not break FORTRAN character strings (this is usually incompatible with cpp).

Coding

- Never use quotes (‘ or “) unless you are defining a FORTRAN string (also not in comments); quotes will disturb cpp.
- Perform check of routine arguments.
- Define and check a return status from all internal and external routines.
- All routines should have “info” as the last non-OPTIONAL argument
- Every routine should have a jump-out label 9999 CONTINUE at its end.

Efficiency

- Do NOT use small arrays (vectors) – use registers:
- **BAD** performance:

```
com(1:2) = 0.0
DO i=1,N
    com(:) = com(:) + xp(:,i)
ENDDO
```

- **GOOD** performance:

```
comx = 0.0; comy = 0.0
DO i=1,N
    comx = comx + xp(1,i)
    comy = comy + xp(2,i)
ENDDO
```

Efficiency

- Use DO loops (do NOT use array notation).
- **BAD** performance:

  A = A + B ! Where A and and B are arrays

- **GOOD** performance:

```
DO j=1,N
    DO i=1,M
        A(i,j) = A(i,j) + B(i,j)
    ENDDO
 ENDDO
 OR:
 A(1:M,1:N) = A(1:M,1:N) + B(1:M,1:N)
```

Efficiency

- For maximum efficiency use UNFORMATTED I/O.
- Write the entire array (do NOT use DO loops or other array subsections)

  Example of fast IO:

  REAL, DIMENSION(10,11,12,13) :: array

  OPEN(10,FILE='data',FORM='UNFORMATTED')

  WRITE(10) KIND(array)

  WRITE(10) SHAPE(array)

  WRITE(10) array

  CLOSE(10)

Extras

- Use iargc() and getarg() to read the command line arguments. This is NOT standard but virtually it is.

  Example:

  nargc= iargc()

  DO i=0,nargc

  CALL getarg(i,string)

  PRINT*,'argument: ',string

  ENDDO

  returns for ./a.out –a –b 23 input.file

  ./a.out

  -a

  -b

  23

  input.file

Extras

- In Fortran 2003/2008 use the COMMAND_ARGUMENT_COUNT() and the GET_COMMAND_ARGUMENT(number[,value][,length][,status])

  Example:

```
program main
character(len=256) :: arg
integer :: i,ilen,info,narg

narg = command_argument_count()
print*,'command_argument_count() = ', narg
do i=1,narg
   call get_command_argument(i,arg,ilen,info)
   print*,'arg = ',arg(1:ilen)
enddo
end program main
```

Submitting jobs

In HPC, simulations are usually submitting as a job to a queue:

The PBS queuing system format: qsub submit.sh:

```
#!/bin/sh
#PBS -N my_job_name
#PBS -l walltime=00:02:00
#PBS -l nodes=1:ppn=16
#PBS -m bea

# change directory to the directory from where you submitted the job from
cd $PBS_O_WORKDIR

# or change explicitly to a specific directory
cd /home/jhwa/runs/case001/

# run the executable:
./a.out > stdout
```

Submitting jobs

The SLURM queuing system: sbatch submit.sh:

```
#!/bin/bash
#SBATCH --job-name=case022
#SBATCH --no-requeue
#SBATCH --mail-type=ALL
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=24
#SBATCH --time=40:00:00
#SBATCH --output=mpi_job_slurm.log
#SBATCH --partition=xeon24
#-----------------------------------------------------------------------------
# change to local directory
#-----------------------------------------------------------------------------
cd $SLURM_SUBMIT_DIR

# or change explicitly to a specific directory
cd /home/jhwa/runs/case001/

# run the executable:
./a.out > stdout
```