

The art of Astro- Programming

Introduction to Fortran and C
(Some thing about F90 and C++)

Arman Khalatyan

The summer school 2006 on "Computational Cosmology"
Astrophysical Institute Potsdam (AIP).



Index

- How were Fortran and C born?
- Basic syntax
- Most useful functions
- Files I/O
- Advanced programming
 - n F90 modules
 - n C++ classes
- Code Optimizations and bottlenecks
- What next?

Birth of FORTRAN

- It was developed over a three year period (1954-1957) by a team at IBM lead by John Backus.
- FORMula TRANslation and was used mainly by people with a scientific background for solving problems with a significant arithmetic content.
- By 1966 and the first standard it was widely used, easy to teach, had demonstrated the benefits of subroutines and independent compilation, was relatively machine independent and often had very efficient implementations.
 - n This standard were F66.
- Under the pressure of X3J3 committee was standardized in 1978 (even though the next version was called Fortran 77)





FORTRAN

...ed over a three year period
...am at IBM lead by John L.



In mean while:

- Bill Joy produces the computer operating system called **BSD** (Berkeley Software Distribution). Bell Laboratories permitted Berkeley and other universities to *use* and *extend* the source code to their UNIX operating system in its infancy.
- **AWK** programming language released by Alfred Aho (A), Peter Weinberger (W), and *Brian Kernighan* (K).
- Apple introduces the Apple II.
- The TRS-80 Model I (**Trash-80**) begins selling. The operating systems that ran on the TRS-80 were TRSDOS and CP/M
- The Atari 2600 game console is selling for \$199.95.
- Public cell phone testing begins. The first cell phones are similar in size to a brick.

2nd Birth of FORTRAN

- Fortran was next standardized in 1991 (yet again out by one) and called **Fortran 90**
 - n There are a lot of major features free source form, modern control structures
 - n There where added structures, modules, matrix/vector operators.
- **Fortran 95** revision was finalized in 1996
 - n Minor corrections and clarifications
 - n **FORALL** statement and construct
 - n Implicit initialization of derived type objects
- For **FORTTRAN 2003** in September 2003 *draft* standard came up, which contains several extensions to Fortran 95
 - n Interoperability with the C programming language
 - n Object oriented programming support

Birth of C



- C came into being in the years 1969-1973, in parallel with the early development of the Unix operating system; the most creative period occurred during 1972
- Its parent *B* and grandparent *BCPL*
 - n B-Combined Programming Language
 - n Type less language
 - n Most UNIX code was written in B and then rewritten in *NB* then *C*
- 1978-1979 – “*The C programming language*” by Brian Kernighan and Dennis Ritchie is published. (“*white book*”) *K&R standard*
- ANSI (American National Standards Institute) C became dominant

Birth of C++

- Bjarne Stroustrup at Bell Labs initially developed C++ during the early 1980's
- It was designed to support:
 - n features of C:
 - in such as efficiency
 - low-level support for system level coding
 - n **New** features such as *classes* with *inheritance* (in other words *Polymorphism*, *encapsulation*) and *virtual functions* (derived from *Simula67*)
 - n operator overloading (derived from *Algol68*)



A little bit Philosophy.....

Which language is preferable?

What we require for **Astro-Programming?**

1. Easy and fast implementation. (*strong*)
2. Possibility to make parallel with minimum effort (ex: using compilers auto options)
3. OpenMP support. (or other SMP paral.)
4. MPI support.
5. Portability of the code in the different supercomputing centers.
6. Understanding code without documentation, or code simplicity (*poor statement*)
7. Easy to share to other colleagues/groups

Choose your language.

	F77	F90	C	C++
1 Easy	good	very good	very good	bad
2 auto	very good	good	bad	bad
3 OpenMP	very good	good	poor	bad
4 MPI	very good	very good	very good	good and bad
5 Port.	very good	good	very good	good and bad
6 Simpli.	poor	very good	very good	good and bad
7 Share	good	good	good	good and bad

What kind of tasks we are solving?

1. Data production:

- n N-Body (Leap frog, direct summation)
- n N-Body + Hydro (GRID,SPH)

2. Data analysis:

- n Finding "*clumps*" (FOF, MST, BDM)
- n Trace objects (hash table)

3. Visualizations

- n Presentation of results. (**Our goal!!!**)
- n 2D or 3D (slice, mock catalogs, movies)

Choose your language by task.

	F77	F90	C	C++
1a Prod	very good	very good	very good	bad
1b	very good	good	very good	bad
2a Anal	very good	good	poor	poor
2b	very good	very good	very good	good and bad
3a Visual	-	good	very good	very good
3b	good	-	good	very good
total	good	good	good	bad

Basic Syntax

FORTRAN 77

- A program consists of one or more program units
- A program unit is a sequence of statements, terminated by an **END**.
- **Standard fixed format** and **Tab format**
 - n The first **72** columns of each line are scanned
 - n The first **five** columns must be blank or contain a numeric label
 - n Short lines are padded to **72** characters
 - n Long lines are truncated
 - n **A tab in any of columns 1 through 6, or an ampersand in column 1**
 - n **Continuation lines are identified by an ampersand 1 row (&)**
 - n You can format lines both ways in one program unit, but not in the **same line**.

ANSI C

- Same as in FORTRAN
- Program unit is a sequence of statements, each statement bracketed by "**{ }**" brackets.
- Free format:
 - n Each "**sentence**" ending with "**;**" symbol
- The code is case sensitive!!!
 - n The **upper/lower case** chars are regarded as being different in **C**.

Minimum code

```
_____ program Test1  
_____ include "some.h"  
_____ write(*,*) "Hello."  
_____ STOP  
_____ end
```

```
Program Test1  
  use somemodule  
  print*, "Hello."  
end program Test1
```

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    printf("Hello.\n");  
    return 0;  
}
```

```
#include <iostream>  
using namespace std;  
int main(int argc, char *argv[])  
{ ...  
    cout<<"Hello."<<endl;  
    return 0;  
}
```

Basic Syntax: comments

FORTRAN 77

Lines starting with "C" character will be ignored

C This is an example of comments in F77

C all lines will

C be ignored during the compilation.

ANSI C

Text between "/*" "*/" will be ignored.

/* one line comment */

/* multi-

Line

comments

*/

NOTE: In C recursive comments are **not allowed**.

C++ => "///" , F90 => "!"

Basic Syntax: an Example

FORTRAN 77

C "Check and Cool" program

```
Program ControlRoomTemp
parameter(rapid_cool_flag=1)
integer rapid_cool_flag
real GetRoomTemp
Temp= GetRoomTemp(1)
12345 write(*,*) "Hello there."
write(*,*) "The temperature in room is about: ",
* Temp," degree."
IF(Temp .GT. 35.0) then
  call COOL_ROOM(rapid_cool_flag)
  Temp= GetRoomTemp(1)
  write(*,*) "Room was cooled down."
  GOTO 12345
ENDIF

END
```

Basic Syntax an Example, ANSI C

ANSI C

```
/* "Check and Cool" program */
#include <stdio.h>
#include "CoolRoomModule.h"
int main(int *argc, char **argv)
{
    const int rapid_cool_flag=1;
    float Temp=0.0;
    Temp= GetRoomTemp(1);
Label0: printf( "The temperature in room is about: %6.4f degree.\n",
               Temp);
    if(Temp > 35.0)
    {
        COOL_ROOM(rapid_cool_flag);
        Temp= GetRoomTemp(1);
        printf("Room was cooled down.\n");
        GOTO Label0;
    }
    return 0;
}
```

Data Types

FORTRAN 77	ANSI C/C++	SIZE(bytes)	
		32bit	64bit
character	char	1	1
	short int/short unsigned int	2	2
logical	bool (C++)	1	1
integer	int	4	4
integer*8	long/long long	4/8	8/8
Real*4	float	4	4
Real*8/ double precision	double	8	8
	long double	12	16
COMPLEX (sys.)		4*2	8/16*2

Data Types

FORTRAN 77	ANSI C/C++	SIZE(bytes)	
<p>To get the size of variable you can use in C/C++: <code>sizeof(varname)</code> or <code>sizeof(typeName)</code>.</p> <p>BUT no such a function in FORTRAN!!!</p> <p>Exercise: Write an Fortran90 code to get the size of types. Note: <i>do not use mixed programming!!!</i> Estimated code size 2-3 lines.</p>			
Real*8/ double precision	double	8	8
	long double	12	16
COMPLEX (sys.)		4*2	8/16*2

32-bit to 64-bit Migration Considerations

Interlanguage Calls with Fortran

C/C++ type	32-bit	64-bit
int	INTEGER	INTEGER
unsigned int	LOGICAL	LOGICAL
signed long	INTEGER	INTEGER*8
unsigned long	LOGICAL	LOGICAL*8
pointer	INTEGER	INTEGER*8

Execution control

- Branch statements
- The CALL statement
- The CASE construct
- The CONTINUE statement
- The DO construct
- The END statement
- The IF construct
- The PAUSE statement
- The RETURN statement
- The STOP statement

Condition construction

FORTRAN 77

```
IF( a .lt. b) then  
  call DoSomething  
end if
```

.le. .lt.

.ge. .gt.

.eq. .ne.

```
if( .NOT.(a .eq. 10) ) write(*,*) "Ok"
```

ANSI C

```
if(a < b)  
{  
  DoSomething();  
}
```

<= <

>= >

== !=

```
if( !(a == 10) ) printf( "Ok\n");
```

Construct	Flow of Control
<pre> SELECT CASE (TEST 1) CASE (1) block 1 CASE (2) block 2 END SELECT </pre>	<pre> graph TD Start(()) --> Eval[Evaluate Test 1] Eval --> D1{Matches CASE (1)} D1 -- Yes --> E1[Execute block 1] D1 -- No --> D2{Matches CASE (2)} D2 -- Yes --> E2[Execute block 2] D2 -- No --> Exit(()) E1 --> Exit E2 --> Exit Exit --> End(()) </pre>
<pre> SELECT CASE (TEST 2) CASE (1) block 1 CASE (2) block 2 CASE (3) block 3 CASE DEFAULT block 4 END SELECT </pre>	<pre> graph TD Start(()) --> D1{Matches CASE (1)} D1 -- Yes --> E1[Execute block 1] D1 -- No --> D2{Matches CASE (2)} D2 -- Yes --> E2[Execute block 2] D2 -- No --> D3{Matches CASE (3)} D3 -- Yes --> E3[Execute block 3] D3 -- No --> E4[Execute block 4] E1 --> Exit(()) E2 --> Exit E3 --> Exit E4 --> Exit Exit --> End(()) </pre>

C/C++: program flow control

- break
- else
- `__if_exists`
- `__try`
- case
- `__except`
- `__if_not_exists`
- try
- catch
- for
- `__leave`
- while
- continue
- goto
- return
- default
- `__finally`
- switch
- do
- if
- throw

Example

```
// do_while_statement.cpp
#include <stdio.h>
int main() {
    int i = 0;
    do {
        printf("\n%d", i++);
    } while (i < 3);
    return 0;
}
```

More complex example in C++

```
#include <iostream>
using namespace std;
int main() {
char *buf;
try {
    buf = new char[512];
    if( buf == 0 )
        throw "Memory allocation failure!";
} catch( char * str )
{
    cout << "Exception raised: " << str << '\n';
}
return 0;
}
```

Output:

Exception raised: Memory
allocation failure!

Files I/O

- Formatted Sequential
- Formatted Direct
- Unformatted Sequential
- Unformatted Direct
- Binary Sequential
- Binary Direct

File Formats

- Formatted Files

- n You create a formatted file by opening it with the `FORM='FORMATTED'` option

- Unformatted Files

- n You create an unformatted file by opening it with the `FORM='UNFORMATTED'`

- Binary Files (F90/95)

- n You create a binary file by specifying `FORM='BINARY'`. Binary files are the most compact, and good for storing large amounts of data.

File Formats

- Sequential-Access Files

- n Data in sequential files must be accessed in order
(unless you change your position in the file with the `REWIND` or `BACKSPACE` statements)

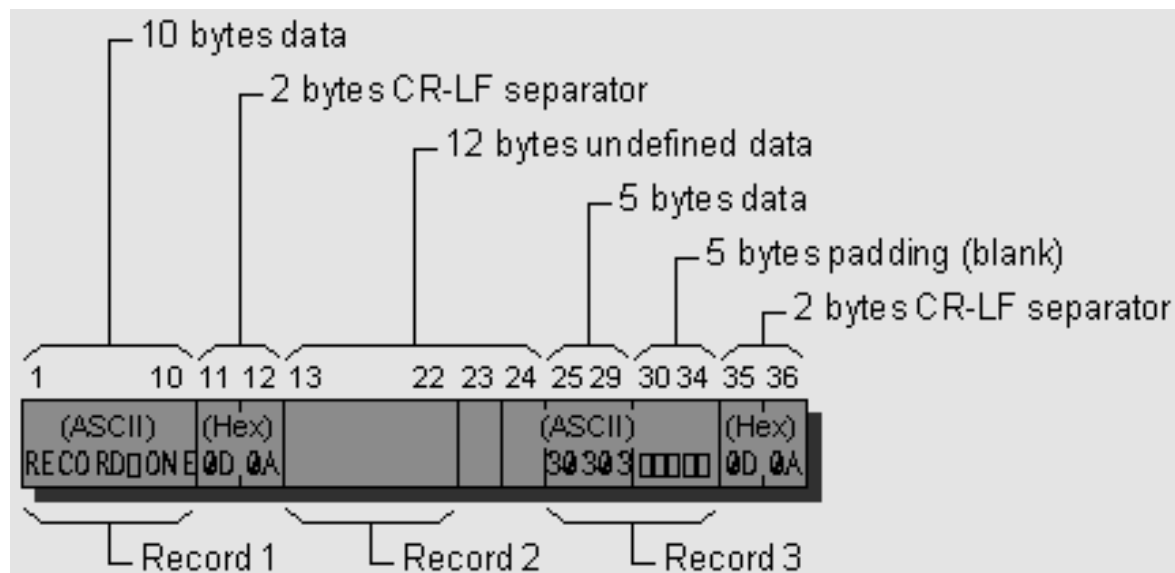
- Direct-Access Files

- n Data in direct-access files can be read or written to in any order.
- n Records are numbered sequentially, starting with record number 1.
- n All records have the length specified by the `RECL=` option in the `OPEN` statement.
- n Data in direct files is accessed by specifying the record you want within the file.
- n If you need random access I/O, use direct-access files.

For C programmers

```
OPEN (3,FILE='FDIR', FORM='FORMATTED', ACCESS='DIRECT',RECL=10)
WRITE (3, '(A10)', REC=1) 'RECORD ONE'
WRITE (3, '(I5)', REC=3) 30303
CLOSE (3)
END
```

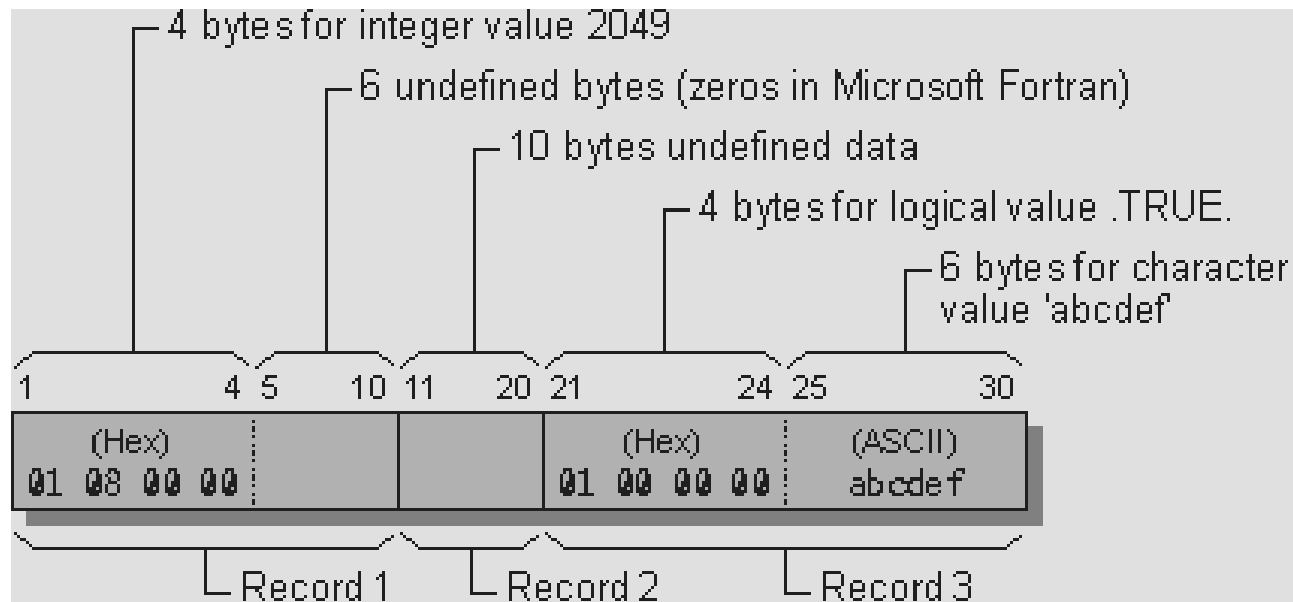
Formatted Direct File



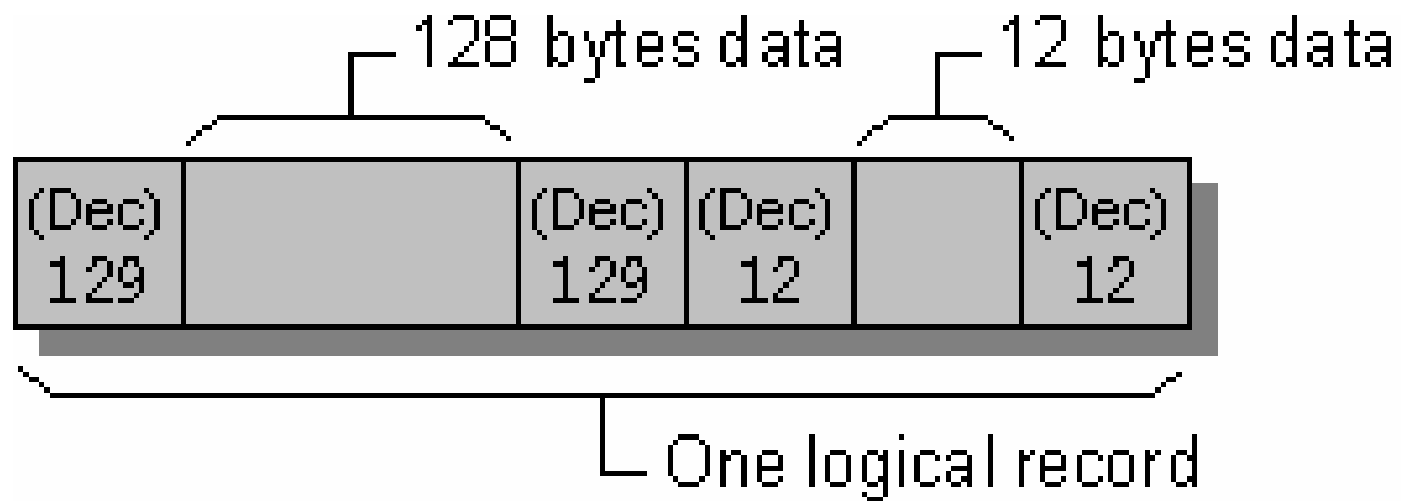
For C programmers

```
OPEN (3, FILE='UFDIR', RECL=10,&  
      FORM = 'UNFORMATTED', ACCESS = 'DIRECT')  
WRITE (3, REC=3) .TRUE., 'abcdef'  
WRITE (3, REC=1) 2049  
CLOSE (3)  
END
```

Unformatted Direct Files



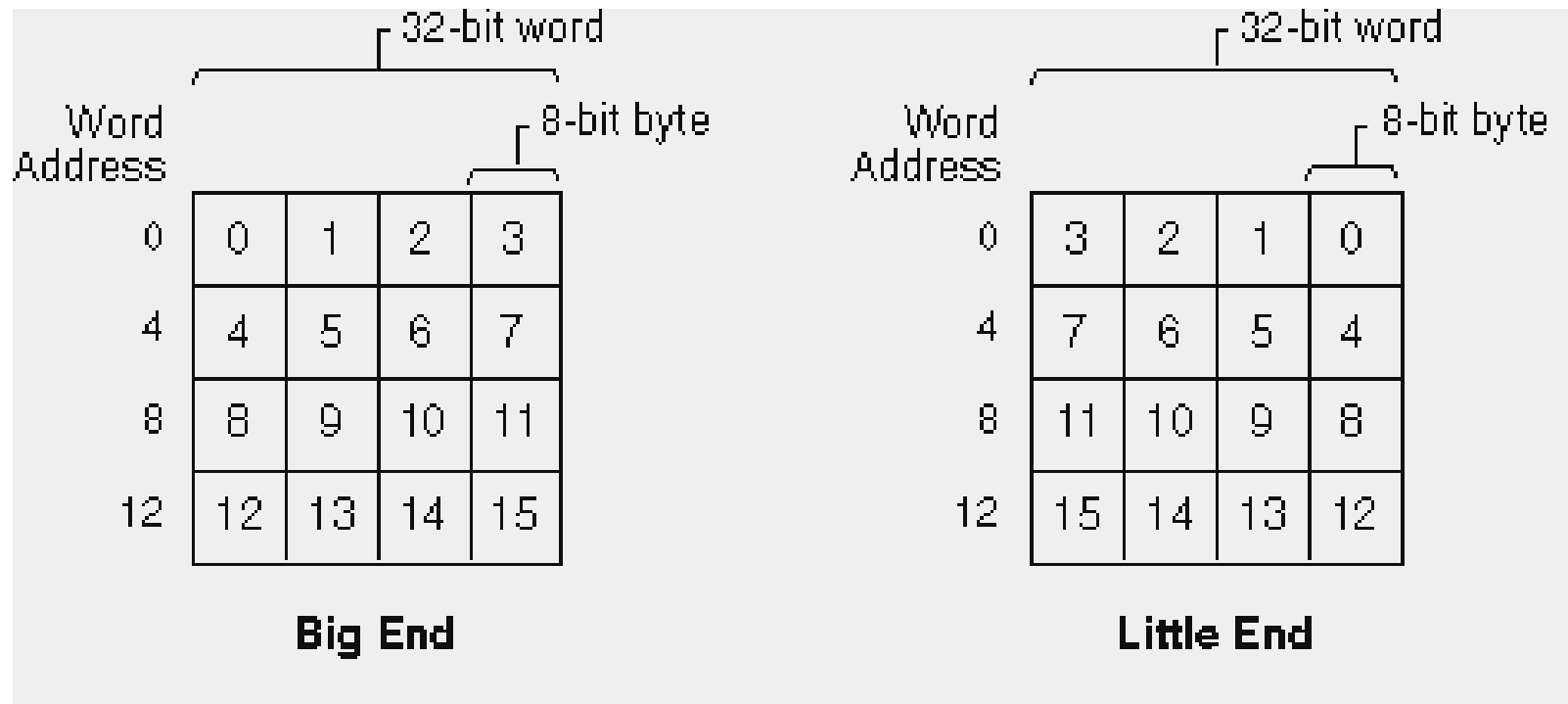
Once more...



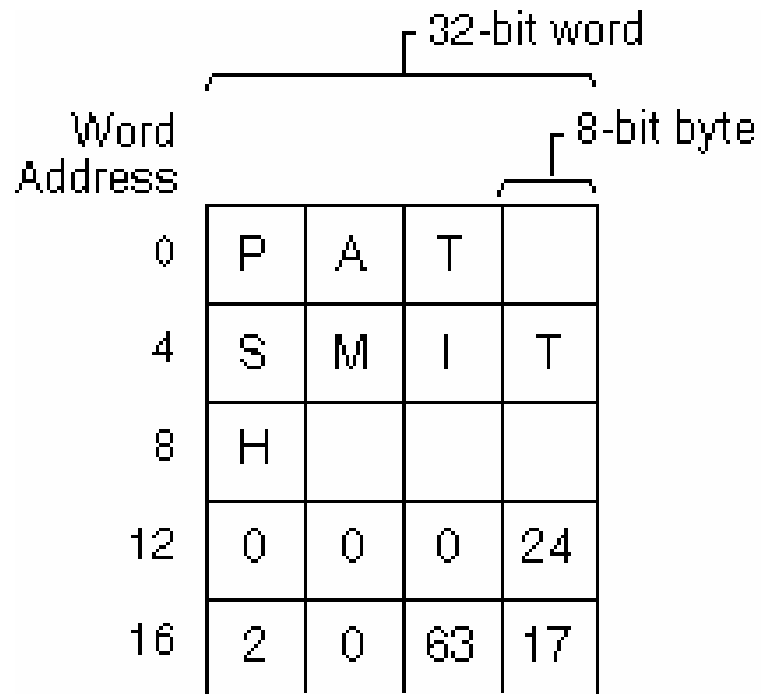
Big End or Little End Ordering

- Computer memory is a linear sequence of bits organized into a hierarchical structure of bytes and words.
- One system is the "**Big End**," where bits and bytes are numbered starting at the most significant bit (MSB, "left," or high end).
- Another system is the "**Little End**," where bits and bytes start at the least significant bit (LSB, "right," or low end).

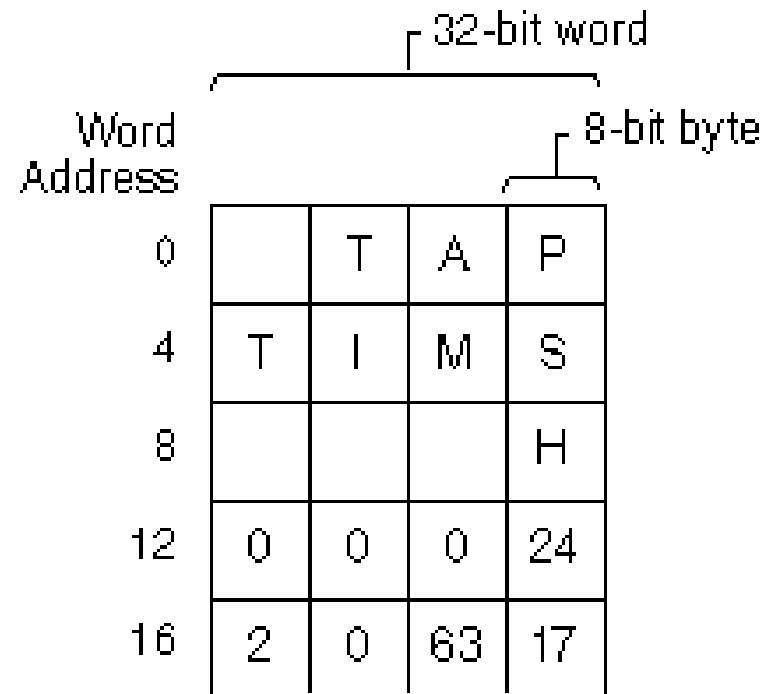
LSB and MSB



LSB and MSB



Big End



Little End

Ordering Nature of Processors

Processor	Byte Order	Bit Order
Intel, Pentium, Altix	Little	Little
DIGITAL Alpha, VAX	Little	Little
IBM® Mainframes	Big	Big

Storage endian.

- Little endian storage occurs when:
 - n The least significant bit (LSB) value is in the byte with the lowest address.
 - n The most significant bit (MSB) value is in the byte with the highest address.
 - n The address of the numeric value is the byte containing the LSB. Subsequent bytes with higher addresses contain more significant bits.
- Big endian storage occurs when:
 - n The least significant bit (LSB) value is in the byte with the highest address.
 - n The most significant bit (MSB) value is in the byte with the lowest address.
 - n The address of the numeric value is the byte containing the MSB. Subsequent bytes with higher addresses contain less significant bits.

Converting

- In FORTRAN
 - n by compiler option.
 - ifort -convert big_endian/little_endian
 - pgf77 -Mbyteswapio
- In C/C++ you need to do by your self:

```

#define SWAP_DOUBLE(Var) \
    Var = *(double*) SwapEndian((void*)&Var, sizeof(double))
void *SwapEndian(void* Addr, const int Nb)
{
    static char Swapped[16];
    switch (Nb)
case 2: Swapped[0]=*((char*)Addr+1);
        Swapped[1]=*((char*)Addr );
        break;
case 4: Swapped[0]=*((char*)Addr+3);
        Swapped[1]=*((char*)Addr+2);
        Swapped[2]=*((char*)Addr+1);
        Swapped[3]=*((char*)Addr );
        break;
case 8: Swapped[0]=*((char*)Addr+7);
        Swapped[1]=*((char*)Addr+6);
        Swapped[2]=*((char*)Addr+5);
        Swapped[3]=*((char*)Addr+4);
        Swapped[4]=*((char*)Addr+3);
        Swapped[5]=*((char*)Addr+2);
        Swapped[6]=*((char*)Addr+1);
        Swapped[7]=*((char*)Addr );
        break;
.....
return (void*)Swapped;
}

```

f=SWAP_ENDIANAN(f);

Advanced Programming

Highlight of Fortran 90 Features

- Free-form (line delimiter ";")
- Array handling
- Comments start with "!", not "c"
- Continuation is "&" and should appear at the end of the line to be continued
- Relational Operators
 - n .EQ. or ==
 - n .NE. or /=
 - n .LT. or <
 - n .LE. or <=
 - n .GT. or >
 - n .GE. or >=
- Recursion function/subroutine
- IMPLICIT NONE
- Automatic array -- you can now declare adjustable arrays inside a subroutine without having to pass them through the argument list
- Arrays can now be allocated/deallocated dynamically
- Derived data type
- Pointer
- Module

Fortran 90 Constructs

- **ALLOCATABLE** -- declare an assumed-shape array
- **ALLOCATE** -- request size for an assumed-shape array
- **CASE** -- element of SELECT construct
- **CASE DEFAULT** -- default case of SELECT construct
- **CONTAINS** -- Indicates procedures within subprogram/module
- **CYCLE** -- element of DO statement; continue to next iteration under certain condition
- **DEALLOCATE** -- return memory allocation to system
- **ELSE WHERE** -- else branch of WHERE
- **END DO** -- ends a DO loop
- **END FUNCTION** -- the end of a function subprogram
- **END INTERFACE** -- ends an INTERFACE block
- **END MODULE** -- the end of a module
- **END PROGRAM** -- the end of a (main) program
- **END SELECT** -- ends the SELECT construct
- **END SUBROUTINE** -- the end of a subroutine
- **END TYPE** -- ends TYPE
- **END WHERE** -- ends WHERE
- **EXIT** -- element of DO statement; exits DO loop under certain condition
- **INCLUDE** -- insert external (source) file
- **INTENT** -- declare intention
- **INTERFACE** -- define procedure interface
- **MODULE** -- a form of subprogram block whose data are made available by USE
- **OPTIONAL** subprogram arguments classification
- **POINTER** -- data type qualifier
- **PRESENT** -- determine whether an argument of a subroutine is present
- **PRIVATE** -- data access control
- **PUBLIC** -- general data access
- **SELECT CASE** -- decision/branch construct
- **SEQUENCE** -- data alignment; used in conjunction with derived type
- **TARGET** pointer target
- **TYPE** -- derived data type defined by user
- **USE** -- data access construct
- **WHERE** -- array "if" construct

Fortran 90 Array Intrinsic

- **ALL**(MASK,*dim*) -- true if all values are true
- **ANY**(MASK,*dim*) -- true if any value is true
- **CSHIFT**(ARRAY,SHIFT,DIM) -- circular shift
- **COUNT**(MASK,*dim*) -- number of true elements in an array
- **DOT_PRODUCT**(A,B) -- dot product of two rank-one arrays
- **EOSHIFT**(ARRAY,SHIFT,*boundary, dim*) -- end-off shift
- **MATMUL**(A,B) -- pre-multiply matrix B by matrix A; columns of A must equal rows of B
- **MAXLOC**(ARRAY,*mask*) -- location of element with maximum value
- **MAXVAL**(ARRAY,*dim, mask*) -- maximum value of ARRAY
- **MERGE**(TSOURCE,FSOURCE,MASK) -- combining two arrays using a mask
- **MINLOC**(ARRAY,*mask*) -- location of element with minimum value
- **MINVAL**(ARRAY,*dim, mask*) -- minimum value of array
- **PACK**(ARRAY,MASK,*vector*) -- pack an array into a vector under a mask
- **PRODUCT**(ARRAY,*dim, mask*) -- product of array elements
- **RESHAPE**(SOURCE,SHAPE,*pad, order*) -- reshape an array
- **SHAPE**(SOURCE) -- shape of an array or scalar
- **SELECTED_INT_KIND**(*n*) -- Returns integer kind with range $(-10n, 10n)$
- **SELECTED_REAL_KIND**(*d, n*) -- Returns real kind
- **SIZE**(ARRAY,*dim*) -- number of elements in an array
- **SPREAD**(SOURCE,DIM,NCOPIES) -- replicate an array by adding a dimension
- **SUM**(ARRAY,*dim, mask*) -- sum array elements
- **TRANSPOSE**(MATRIX) -- transpose array of rank two
- **UNPACK**(VECTOR,MASK,FIELD) -- unpack rank-one array into a multidimensional array under a mask

Function	Return Value
MAXVAL(A)	Maximum value in A
MINVAL(A)	Minimum value in A
SUM(A)	Sum of elements of A
PRODUCT(A)	Product of elements of A
MAXLOC(ARRAY)	Indices of maximum value in A
MINLOC(ARRAY)	Indices of minimum value in A
MATMUL(A,B)	Matrix multiplication A*B
DOT_PRODUCT(A,B)	Vector dot product A.B
TRANSPOSE(A)	Transpose of A
CSHIFT(A,SHIFT,DIM)	Rotation of elements of A

```
real X(0:99), B(0:99)
do i = 0,99
  B(i) = ( X(mod(i+99,100) + X(mod(i+1,100)) )/2
enddo
```

BUT F90

```
real X(100), B(100), L(100), R(100)
L = CSHIFT(X,+1)
R = CSHIFT(X,-1)
B = ( L + R )/2
```

OR

```
real X(100), B(100)
B = ( CSHIFT(X,+1) + CSHIFT(X,-1) )/2
```

Another example

```
real s,xmin,xmax X(100)
integer maxind
s = SUM(X)
xmin=minval(X); xmax=maxval(X);maxind=maxloc(x)
```

C++ language

- This is **not procedural** language!!!
 - n This is known as **top-down design**.
 - n C++ is an object-oriented language (OOP).
 - n To solve a problem with C++ the first step is to design classes that are abstractions of physical objects.
 - n These classes contain both:
 - the state of the object, its *members*
 - the capabilities of the object, its *methods*

Language Support for OOP Features

```
class Particle {
private:
    double mass;
    double position[3], velocity[3];
public:
    Particle(double imass=0.0) {
        mass = imass;
        for (int i=0; i<3; i++) {
            position[i] = 0.0;
            velocity[i] = 0.0;
        }
    }
    virtual ~Particle() { }
    virtual double Position(int i) const { return position[i]; }
    virtual double Velocity(int i) const { return velocity[i]; }
    virtual double Kinetic_Energy() const {
        double ke = 0.0;
        for (int i=0; i<3; i++) ke += velocity[i]*velocity[i];
        return mass*ke;
    }
    virtual double Charge() const { return 0.0; }
};
```

```
class Particle {
private:
    ...
    double position[3], momentum[3];
public:
    ...
    virtual double Velocity(int i) const {
        return momentum[i]/mass;
    }
    virtual double Kinetic_Energy() const {
        double ke = 0.0;
        for (int i=0; i<3; i++) ke += momentum[i]*momentum[i];
        return ke/mass;
    }
    ...
};
```

Particle p1, p2;

double p1_ke = p1.Kinetic_Energy();

F90: TYPES

```
TYPE Particle
```

```
    REAL mass
```

```
    REAL, DIMENSION(0:2) :: position, velocity
```

```
END TYPE Particle
```

```
TYPE(Particle) :: p
```

```
REAL :: pmass = p%mass
```

```

MODULE ParticleModule
  TYPE Particle
    PRIVATE
    REAL mass
    REAL, DIMENSION(0:2) :: position, velocity
  END TYPE Particle
CONTAINS
  SUBROUTINE Initialize(p,imass)
    TYPE(Particle), INTENT(INOUT) :: p
    REAL, OPTIONAL :: imass
    INTEGER i
    IF (PRESENT(imass)) THEN
      p%mass = imass
    ELSE
      p%mass = 0.0
    ENDIF
    DO i=0,2
      p%position(i) = 0.0
      p%velocity(i) = 0.0
    END DO
  END SUBROUTINE Initialize
  REAL FUNCTION Position(p,i)
    TYPE(Particle), INTENT(IN) :: p
    INTEGER, INTENT(IN) :: i
    Position = p%position(i)
    RETURN
  END FUNCTION Position
  REAL FUNCTION Velocity(p,i)
    TYPE(Particle), INTENT(IN) :: p
    INTEGER, INTENT(IN) :: i
    Velocity = p%velocity(i)
    RETURN
  END FUNCTION Velocity
  REAL FUNCTION KineticEnergy(p)
    TYPE(Particle), INTENT(IN) :: p
    INTEGER i
    REAL :: ke = 0.0

```

Usage of Code:

USE ParticleModule

TYPE(Particle) :: p1, p2

Initialize(p1)

Initialize(p2)

call DoSomething(p1)

END

Somewhere in another project...

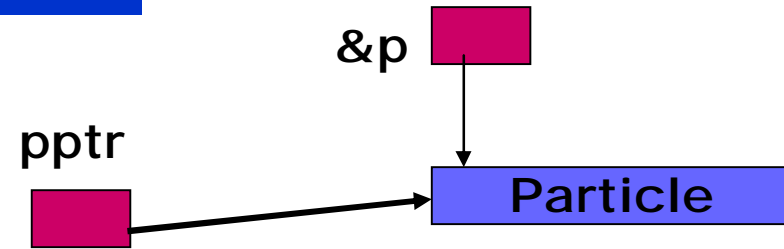
```
class Nucleus : public Particle {
private:
    int numProtons, numNeutrons;
    static double elemCharge;
public:
    Nucleus(int inumProtons, int inumNeutrons, double
        imass=0.0)
    : Particle(imass) {
        numProtons = inumProtons;
        numNeutrons = inumNeutrons;
    }
    ~Nucleus() { }
    double Charge() const {
        return numProtons*elemCharge;
    }
};
```

Particle p;
Nucleus n;

.....

Particle *pptr;

pptr = &p;



```
(*pptr).Charge();  
pptr->Charge();
```

```
pptr = &n;
```

```
pptr->Charge();
```

User defined TYPE

```
real x(N),y(N),z(N)
common /PART/ x,y,z
```

**AVOID to USE COMMON
BLOCKS**

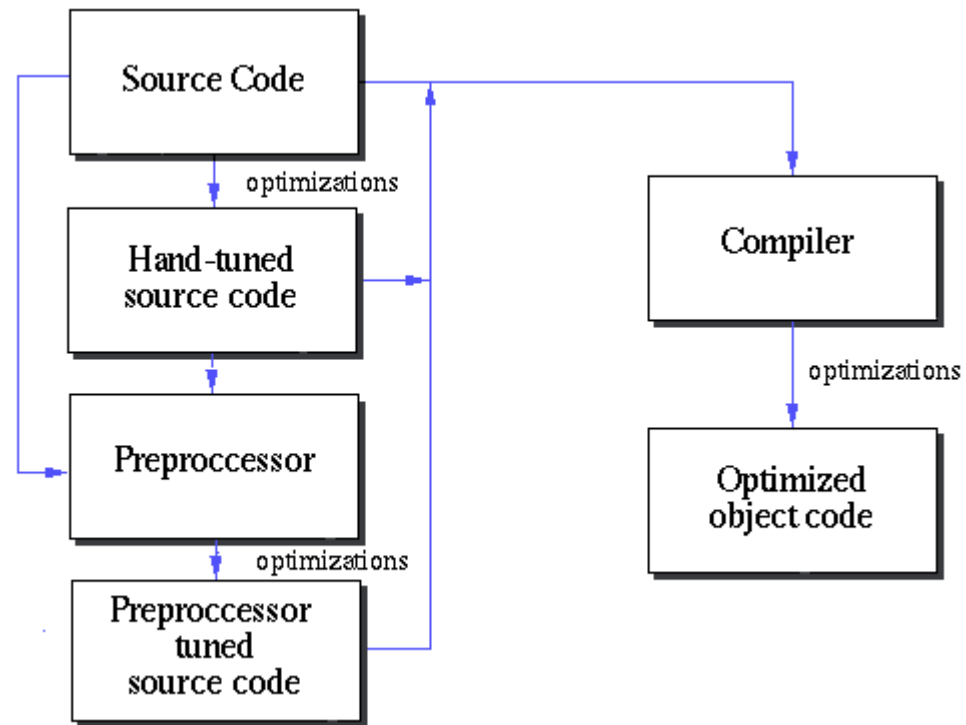
```
TYPE (PART)
  real::x,y,z
END TYPE
TYPE (PART),allocatable :: Posv(:)
TYPE(PART):: Pos(1:N)
allocate(Posv(1:N))
! Do calculation
deallocate(Posv)
```

```
structure tagParticle
{
  float x,y,z;
}*Posv,Pos[N];
```

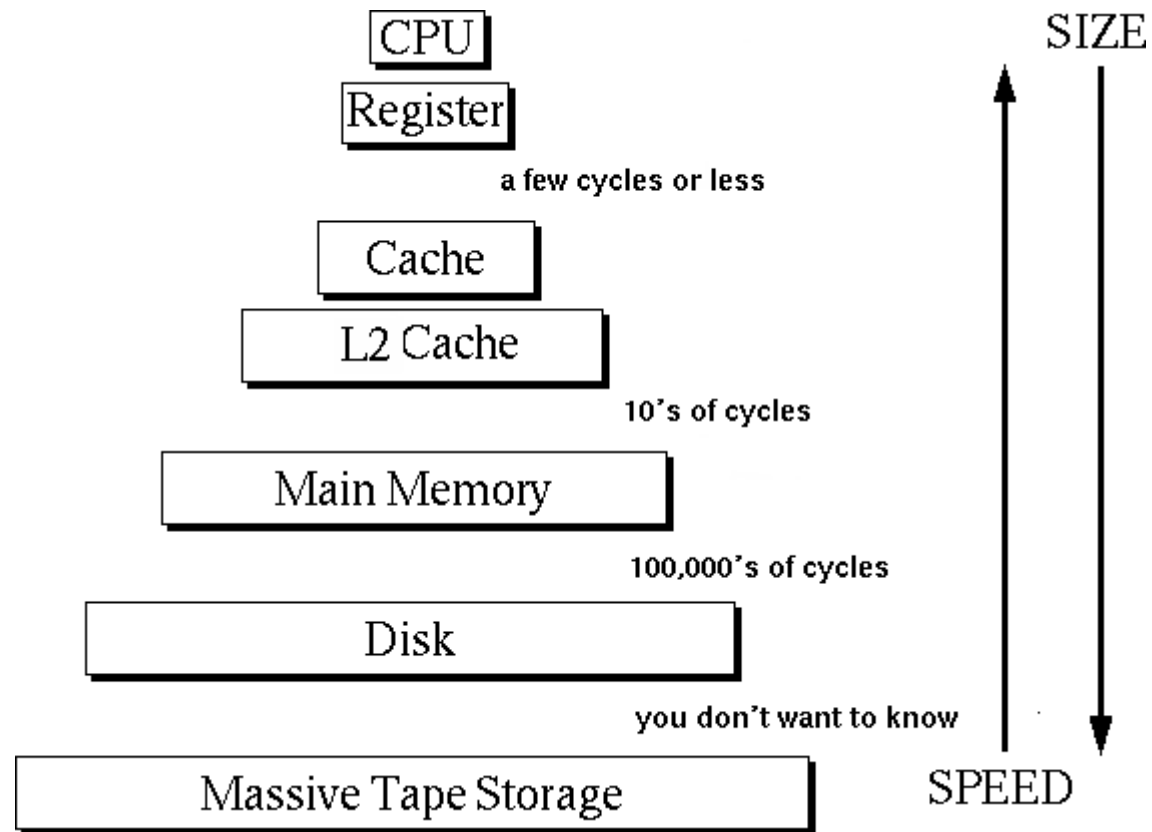
```
class Part
{
  public:
    void GetPos(float *);
  private:
    float x,y,z;
  virtual:
    void DrawParticle();
}
```

Code Optimizations and bottlenecks

The Optimization Process



Memory Considerations



Array and Memory Management Optimizations

Array Allocation in C and Fortran

Example 4 by 4 matrix with data values

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C: Row-Major Order: rows of the matrix are stored contiguously

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

Fortran: Column-Major Order: columns of the matrix are stored contiguously

1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16
---	---	---	----	---	---	----	----	---	---	----	----	---	---	----	----

In a loop, stride is defined as the distance between successively accessed elements of a matrix in successive iterations of the loop.

stride 1

```
DO I=1,1000
  DO J=1,1000
    C(J,I) = A(J,I) * B(J,I)
  END DO
END DO
```

stride 1000

```
-----
DO J=1,1000
  DO I=1,1000
    C(J,I) = A(J,I) * B(J,I)
  END DO
END DO
```

```
for(i=0;i<1000;i++)
  for(j=0;j<1000;j++)
    c[i][j] = a[i][j] * b[i][j]
```

```
-----
for(j=0;j<1000;j++)
  for(i=0;i<1000;i++)
    c[i][j] = a[i][j] * b[i][j]
```

Array Padding

- Array Padding is a technique that can be used to reduce cache misses due to set associativity in arrays that are sized by a power of two.
- To pad an array, simply increase the leading array dimension in Fortran, or the last dimension in C/C++.

The optimal amount to pad varies, but should be at least the number of elements that fit in one line of cache.

Array Padding

Untuned

```
REAL*8 A(256,256)
```

```
DO I=1,256
```

```
  DO J=1,256
```

```
    DO K=1,256
```

```
      A(I,J)=A(I,J)+A(J,K)-A(K,I)
```

```
    END DO
```

```
  END DO
```

```
END DO
```

Array Padding

Tuned

```
REAL*8 A(257,256)

DO I=1,256
  DO J=1,256
    DO K=1,256
      A(I,J)=A(I,J)+A(J,K)-A(K,I)
    END DO
  END DO
END DO
```

Loop Optimizations

Untuned

```
for(i=0;i<100000;i++)
```

```
    x = x * a[i] + b[i];
```

```
for(i=0;i<100000;i++)
```

```
    y = y * a[i] + c[i];
```

Tuned

```
for(i=0;i<100000;i++) {  
    x = x * a[i] + b[i];  
    y = y * a[i] + c[i];  
}
```

Invariant IF Code Floating

Untuned

```
for(i=0;i<1000;i++)
{
  for(j=0;j<1000;j++)
  {
    if (a[i] > 100)
      b[i] = a[i] - 3.7;
    x = x + a[j] + b[i];
  }
}
```

Tuned

```
for(i=0;i<1000;i++)
{
    if (a[i] > 100)
        b[i] = a[i] - 3.7;
    for(j=0;j<1000;j++)
        x = x + a[j] + b[i];
}
```

Loop Defactorizing

Factorized

```
DO I=1, ARRAY_SIZE
  A(I) = 0.0D0
  DO J = 1, ARRAY_SIZE
    A(I) = A(I) + B(J) * D(J) * C(I)
  ENDDO
ENDDO
```

Defactorized

Up to 30 time speedup

```
DO I=1, ARRAY_SIZE
  A(I) = 0.0D0
  DO J = 1, ARRAY_SIZE
    A(I) = A(I) + B(J) * D(J)
  ENDDO
  A(I) = A(I) * C(I)
ENDDO
```

Loop Unrolling and Sum Reductions

Untuned

```
a = 0.0;
for(i=0;i < ARRAY_SIZE;i++)
    for(j=0;j < ARRAY_SIZE;j++)
        a = a + b[j] * c[i];
```

Tuned

```
a1 = a2 = a3 = a4 = 0.0;
for(i=0;i < ARRAY_SIZE;i++)
  for(j=0;j < ARRAY_SIZE;j+=4)
  {
    a1 = a1 + b[j] * c[i];
    a2 = a2 + b[j+1] * c[i];
    a3 = a3 + b[j+2] * c[i];
    a4 = a4 + b[j+3] * c[i];
  }
aa = a1 + a2 + a3 + a4;
```

Right shift replacement for integer division by a power of 2

Untuned

```
IL = 0
```

```
DO I=1,ARRAY_SIZE
```

```
  DO J=1,ARRAY_SIZE
```

```
    IL = IL + A(J)/2
```

```
  ENDDO
```

```
ILL(I) = IL
```

```
ENDDO
```

Tuned

```
IL = 0
ILL = 0
DO I=1,ARRAY_SIZE
  DO J=1,ARRAY_SIZE
    IL = IL + ISHFT(A(J),-1)
  ENDDO
ILL(I) = IL
ENDDO
```

Input/Output Optimizations

- I/O is orders of magnitude slower than internal memory accesses

Suggestions:

- Eliminate all unnecessary I/O.
- Move I/O statements outside of loops if possible.
- Use unformatted (binary) I/O whenever possible
- Formatted I/O requires that library calls be made to convert the binary representation to human readable format and then converted back again to binary format when the processor must process the data.
- Formatted I/O can also result in lost precision and rounding errors.
- Binary data is smaller - requires less physical I/O time to process and less disk space to store. For example, to store the number "1" as a double precision formatted value, requires 21 bytes (1.00000000000000000000 plus newline). Storing it as a binary value requires only 8 bytes (if it is part of a large record).
- Use large record sizes. Each record (result of one write operation) of unformatted I/O includes a 4-byte header and 4-byte trailer. Files with short records will contain a higher percentage of header and trailer records - thus requiring more I/O and space.

Compilers:

- Intel

- n ifort

- -O3 -parallel -openmp -Ftz *-fast -ipo*

- PGI

- n pgf90 -O3 -mp -Mconcour -tp cputype
-fastsse -fast

- IBM AIX

- n xlf90_r *-O5*(-O4) -qsmp -qsmp=omp
-qstrict -qarch=pwr5 -qtune=pwr5 *-qipa*
-Pk

**Happy programming J
and
Thank YOU for attention.**

References

- www.google.com J
- F77 vs F90 by 1995 by *Ian Foster*
- The history of C and C++, their uses and differences by *John Kopp*
- Comparison of C++ and Fortran 90 for Object-Oriented Scientific Programming *John R. Cary and Svetlana G. Shasharina*
- Fortran 90, 95, 2003, 77 Information Resources *Ian Chivers and Jane Sleightholme*
- DIGITAL Visual Fortran documentation
- MSDN
- SP Parallel Programming Workshop.