

# Mathematical Programming with Fortran

Shimon Panfil, Ph. D.

Industrial Physics and Simulations  
<http://industrialphys.com/>

January 14, 2010

# Contents

## Mathematical Programming

Problem

Solution

## Modern Fortran

Fortran History

F03 Features

State of the Art

## Optimization

Optimization Myths

Compiler Optimization

Memory Optimization

Algorithm Optimization

## Fortran in Mathematical Programming

Why Fortran?

Expressive Power

example

# Problem

Mathematical Programming (MP) means the solution of large and difficult scientific and engineering problems with computers.

"Large and difficult" are problems that cannot be solved without programming, by using Matlab, Excel and the like, e.g. solving 3D PDE.

MP is the oldest field of programming, actually Fortran (which stands for *formula translation*) - the first ever created programming language was designed with such kind of problems in mind. It was developed over a period from 1954 to 1957 by an IBM team, lead by John Backus. Both MP and Fortran are still under active development mainly in large computing centers belonging to both universities and industry.

# Problem

Intensive growth of business, multimedia, internet and other non-mathematical applications has overshadowed MP, so most of modern computer professionals and users know nothing about it. More importantly, most of the people teaching programming and writing books on the subject do not know about MP either. So, their recommendations on programming style and methodology are in many cases counterproductive when applied to Mathematical Programming.

However, the most important fact is that modern programming languages like C, C++, Java, . . . were designed primarily for non-mathematical applications. So we have a dilemma: use Fortran which is convenient for MP, but does not exploit modern computer architecture or use a modern programming language, one less suitable for MP.

# Solution

There are a number of solutions for the problem described above:

1. Use supercomputer with High Performance Fortran. Not discussed here.
2. Write mathematics in C. Needs proficiency both in mathematics and programming. Not discussed here.
3. Use modern Fortran.

# Fortran History

**Fortran 66** First standard, reasonably machine independent programming language with efficient implementation.

**Fortran 77** Most of existing code complies to this standard.

**Fortran 90** Adds: pointers, modules, recursion, dynamic storage allocation.

**Fortran 95** Marks a number of F77 features as obsolete.

**Fortran 2003** Adds object-oriented programming support and C compatibility.

**Fortran 2008** Adds Co-arrays and BIT data type.

## F03 Features

- ▶ Derived type enhancements
- ▶ Object-oriented programming support: type extension and inheritance, polymorphism, dynamic type allocation, and type-bound procedures.
- ▶ Data manipulation enhancements: VOLATILE attribute, pointer enhancements, extended initialization expressions, and enhanced intrinsic procedures.
- ▶ Input/output enhancements: asynchronous transfer, stream access ...
- ▶ Procedure pointers.
- ▶ Support for IEEE floating-point arithmetic and floating point exception handling
- ▶ Interoperability with the C programming language.
- ▶ Support for international usage
- ▶ Enhanced integration with the host operating system

## State of the Art

Language standard is not enough for practical programming. One also needs implementation: compiler, runtime system, libraries . . . . Before 2005, the only good available Fortran for PC was the g77 from gnu compiler collection (gcc), which implements F77 standard with some extensions. Development of the replacement was started in 2000 (g95 project). In 2003 gfortran forked from g95 and became a part of gcc from version 4.0. It includes support for the Fortran 95 language and is compatible with most language extensions supported by g77, allowing it to serve as a drop-in replacement in many cases. Parts of Fortran 2003 and Fortran 2008 have also been implemented. A number of commercial compilers also exists. Here, F77 will refer to g77 and F03 to gfortran. Fortran will stand for both F77 and F03.

## Optimization Myths

There are a lot of myths about programming languages' efficiency and optimization. Anybody can do simple experiment: write a program which multiplies 2 matrices in C, Fortran, and Matlab. Matrices should be large enough, otherwise the effect will be too small. You will see that C and Fortran code have approximately the same speed, Matlab script is much slower, but Matlab's built-in function is much faster. So, in this case Fortran is not essentially more effective than C and is less effective than properly used Matlab. However, if you use intrinsic matrix multiplication in Fortran you'll get speed similar to Matlab's built-in function. For a problem like 3D PDE solution Fortran (and C) will outperform Matlab. The answer is that there are at least 3 levels of optimization.

```
program mm3
  integer :: n,cac,i,j,k
  real,allocatable, dimension(:,:) :: a,b,c1,c2
  CHARACTER(len=32) :: arg
  real::start_time,stop_time
  real, parameter:: eps=1.0e-15
  cac=command_argument_count()
  if(cac/=1) then
    print *,"need 1 argument: dimension"
    stop
  end if
  CALL get_command_argument(1, arg)
  read (arg,*) n
  print *,"dimension=",n
```

```
allocate(a(n,n),b(n,n),c1(n,n),c2(n,n))
  call random_seed()
  call random_number(a)
  call random_number(b)
  call cpu_time(stop_time)
print *, "matrix preparation", stop_time-start_time, "s"
  call cpu_time(start_time)
  c1=matmul(a,b)
  call cpu_time(stop_time)
print *, "matmul", stop_time-start_time, "s"
```

```
call cpu_time(start_time)
  c1=0.0
  do k=1,n
    do i=1,n
      do j=1,n
        c1(i,j)=c1(i,j)+a(i,k)*b(k,j)
      end do
    end do
  end do
call cpu_time(stop_time)
print *,"loop kij",stop_time-start_time,"s"
if(any(abs(c1-c2)>eps)) then
  print *,"different values, stop"
  stop
end if
end program mm3
```

```
dimension=          500
matrix preparation  9.99899999999999921E-003 s
matmul  0.129991000000000000          s
loop ijk  0.29664799999999997          s
loop ikj   3.00647000000000002          s
loop jik  0.303313999999999986          s
loop jki  0.153322999999999988          s
loop kji  0.189988000000000005          s
loop kij   3.0031369999999997          s
```

```
dimension=          1000
matrix preparation  4.33300000000000005E-002 s
matmul    1.7032219999999998          s
loop ijk   11.222601000000001          s
loop ikj   31.081305999999998          s
loop jik   11.659240000000004          s
loop jki   1.93987399999999961         s
loop kji   2.74315399999999970         s
loop kij   31.221298000000004          s
```

```
dimension=          1500
matrix preparation  9.66589999999999949E-002 s
matmul    5.70962800000000004          s
loop ijk   45.2737140000000005          s
loop ikj   109.266209000000000          s
loop jik   45.9536709999999986          s
loop jki   6.53290699999999945          s
loop kji   9.669369999999999865          s
loop kij   108.819570999999997          s
```

```
dimension=          2000
matrix preparation  0.169988000000000000      s
matmul    13.4624550000000000      s
loop ijk   114.159224000000001      s
loop ikj   280.438380999999999      s
loop jik   116.785718999999997      s
loop jki   15.60898300000000080      s
loop kji   22.7018529999999915      s
loop kij   280.771692999999991      s
```

## Dynamic vs. Static

Here the results of two similar programs are compared for  $n = 1000$ . The only difference is that one use allocatable arrays and another static ones.

algorithm	dynamic	static
mm	1.70	1.70
ijk	11.22	11.54
ikj	31.16	31.29
jik	11.66	11.78
jki	1.94	1.56
kji	2.79	2.24
kij	32.25	31.60

# Compiler Optimization

Large and difficult topic. All modern compilers comprise of 3 main parts

1. Language dependent front-end.
2. Intermediate representation "middle-end".
3. Platform dependent back-end.

Optimization can occur during any phase of compilation, however, the bulk of optimizations are performed in the "middle-end" — after the syntax and semantic analysis of the front-end and before the code generation of the back-end. So optimization is mostly language and architecture independent. Fortran front-end passes more information to middle-end but the compiler is smart enough to infer the necessary information in relatively simple cases.

# Memory Optimization

Compiler performs well when restricted to processor optimization. Modern computers have memory hierarchy: fast and expensive (hence small) memory near processor core (L1 cache), slower and larger L2 cache etc. up to main memory which is the slowest. Optimization of cache usage is generally beyond compiler abilities. Actually, too aggressive compiler optimization may spoil cache usage optimization. BTW, it is just this optimization that is responsible for results of our experiment: both Matlab built-in and Fortran intrinsic functions may be optimized for cache usage.

# Algorithm Optimization

This is the bread and butter of mathematical programming: the choice of algorithm or combination of algorithms which works best for problem at hand. Simple example: you want to solve linear system  $Ax = b$ . The theoretical answer is  $x = A^{-1}b$  and you can use highly optimized algorithms for matrix inversion. However, a much better solution is LU decomposition of matrix A with subsequent back-substitution.

The program may comprise of more than one algorithm. Even if all algorithms are the most optimal, their combination may not be if, for example, large amount of data must be rearranged between subsequent parts.

# Why Fortran?

So why Fortran is good for MP (say better than C)?

**compiler optimization?** No, practically no difference.

**cache optimization?** No, it's language neutral.

**algorithms?** Yes, modern Fortran (F03) allows writing mathematical algorithms in the most natural, and so less error prone, way!

# Expressive Power

Fortran now has all the features that one expects in a modern programming language:

- ▶ Arrays can be used in expressions, as arguments to intrinsic functions, and in assignment statements; there is also a concise array-section notation.
- ▶ Dynamic storage is fully supported, with both automatic and allocatable arrays; pointers can be used to handle more complex objects such as linked-lists and trees.
- ▶ Derived data types (data structures) are fully supported.
- ▶ The MODULE is a program unit which fully supports encapsulation of data and procedures and greatly facilitates code re-use.
- ▶ Procedures can be recursive or generic, they can have optional or keyword arguments, arrays can be passed complete with size/shape information and functions can return arrays or data structures.

# Expressive Power

- ▶ New operators can be defined, or existing operators can be overloaded for use on objects of derived type.
- ▶ Modules and data structures may declare their contents `PRIVATE` to enhance encapsulation and avoid name-space pollution.

So Fortran is as expressive as C++, but arrays and complex numbers are first-class objects, not derived ones - they are part of the language. Highly optimized and well tested intrinsic procedures exist to work with these objects. In a sense modern Fortran combines the power of Matlab to express mathematical algorithm with the efficiency of C.

## Example

This Fortran 2003 program calculates 2d heat transfer over rectangular plate using relaxation method.

```
program heat_transfer
  implicit none
  integer :: m,n,c
  CHARACTER(len=32) :: arg
  real :: coeff
  real, allocatable, dimension(:,,:), target :: plate
  real, allocatable, dimension(:,,:) :: temp
  real, pointer, dimension(:,,:) :: north, east, south, west
  real, parameter :: tolerance =1.0e-4
  real :: diff
  integer :: i,j,niter
```

## Example

```
c=command_argument_count()
if(c/=2) then
    print *, "need 2 arguments"
    stop
end if
CALL get_command_argument(1, arg)
read (arg, '(I10)') m
CALL get_command_argument(2, arg)
read (arg, '(I10)') n
allocate(plate(m,n))
allocate(temp(m-2,n-2))
```

## Example

```
!initial conditions
plate=0
!boundary conditions
plate(1:m,1)=1.0
coeff=1.0/n
plate(1,1:n)=[(coeff*j,j=n,1,-1)]
```

## Example

Pointers to different parts of the same array. Fortran pointers are not merely addresses, they have full information on index triplet start:stop:stride.

```
inside=>plate(2:m-1,2:n-1)
north=>plate(1:m-2,2:n-1)
south=>plate(3:m,2:n-1)
east=>plate(2:m-1,1:n-2)
west=>plate(2:m-1,3:n)
```

## Example

Main loop (relaxation), no indices at all!

```
niter=0
do
    temp=0.25*(north+east+south+west)
    diff=maxval(abs(temp-inside))
    niter=niter+1
    inside=temp
    if(diff<tolerance) then
        exit
    endif
end do
print *,plate(m/2,:)
end program heat_transfer
```