IRPF90 : a Fortran code generator for HPC

Anthony Scemama[1] <scemama@irsamc.ups-tlse.fr>
François Colonna[2]

[1] Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)
[2] Laboratoire de Chimie Théorique (Paris VI)

Introduction

- Scientific codes need *speed* -> Fortran/C
- Low level language -> difficult to maintain
- High-level features of Fortran 95 or C++ can kill the efficiency (pointers, array syntax, objects, STL, etc) -> not a good solution for HPC

We need to hide the code complexity and keep the code efficient :

1. Implicit Reference to Parameters programming strategy
2. IRPF90 : Facilitates programming with IRP in Fortran

What is a scientific code?

A program is a function of its input data:

```
output = program (input)
```

A program can be represented as a **production tree** where

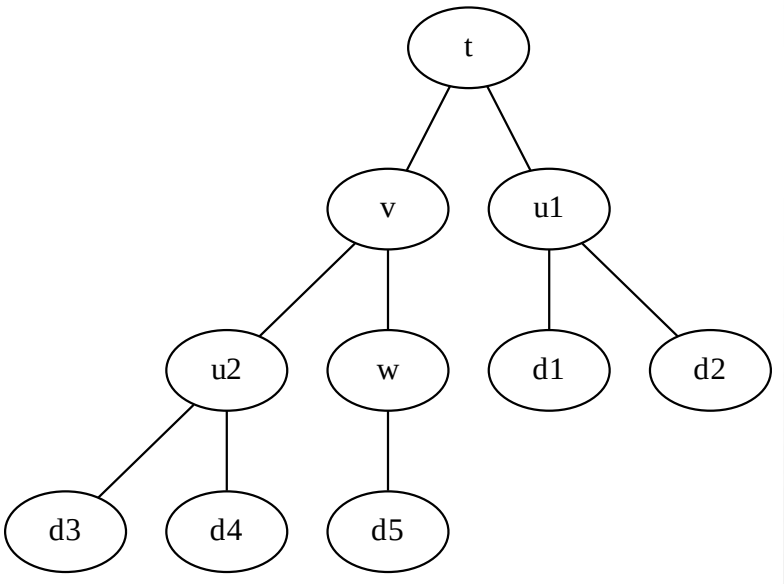
- The root is the output
- The leaves are the input data
- The nodes are the intermediate variables
- The edges represent the relation *needs/needed by*

Example:

```
u(x,y) = x + y + 1
v(x,y) = x + y + 2
  w(x) = x + 3
t(x,y) = x + y + 4
```

What is the production tree of t(u(d_1, d_2), v(u(d_3, d_4), w(d_5)))?

```
u(x,y) = x + y + 1
v(x,y) = x + y + 2
  w(x) = x + 3
t(x,y) = x + y + 4
```



Traditional Fortran implementation

```
program compute_t
  implicit none
  integer :: d1, d2, d3, d4 d5
  integer :: u, u, v, w, t

  call read_data(d1,d2,d3,d4,d5)
  call compute_u(d1,d2,u1)
  call compute_u(d3,d4,u2)
  call compute_w(d5,w)
  call compute_v(u2,w,v)
  call compute_t(u1,v,t)

  write(*,*), "t=", t
end program
```

```
!            t
!          /   \
!       u1      v
!     /  |      | \
! d1   d2    u2   w
!            /  \   \
!          d3   d4   d5
```

Difficulties

The subroutines need to be called in **the correct order**:

- The programmers needs have the **global knowledge** of the production tree : Production trees are usually too complex to be handled by humans
- Programmers may not be sure that their modification did not break some other part
- Collaborative work is difficult : any user can alter the production tree

Using the functional paradigm

```
program compute_t                          !           t
  implicit none                            !         /   \
  integer :: d1, d2, d3, d4 d5             !       u1     v
  integer :: u1, u2, v, w, t               !     /  |     | \
                                           ! d1   d2    u2   w
  call read_data(d1,d2,d3,d4,d5)           !          /   \   \
                                           !        d3    d4   d5


  write(*,*), "t=", t( u(d1,d2), v( u(d3,d4), w(d5) ) )

end program
```

- Instead of telling the machine **what to do**, we express **what we want**
- The production tree is now explored from the root to the leaves.
- The programmer doesn't handle the execution sequence

From global to local knowledge

For each node, we can express the needed entities

```
* t  -- needs --> u1 and v
* u1 -- needs --> d1 and d2
* v  -- needs --> u2 and w
* u2 -- needs --> d3 and d4
* w  -- needs --> d5
```

In this way, all the knowledge is **local**, and much easier to handle by the programmer.

Let's write our program in this way:

```
program compute_t                    |  integer function t()
    implicit none                    |      implicit none
    integer, external :: t           |      integer, external :: u1, v
    write(*,*), "t=", t()            |      t = u1() + v() + 4
end program                          |  end
```

```
integer function v()
    implicit none
    integer, external :: u2, w
    v = u2() + w() + 2
end

integer function w()
    implicit none
    integer :: d1,d2,d3,d4,d5
    call read_data(d1,d2,d3,d4,d5)
    w = d5+3
end

integer function f_u(x,y)
    implicit none
    integer, intent(in)  :: x,y
    f_u = x+y+1
end
```

```
integer function u1()
    implicit none
    integer :: d1,d2,d3,d4,d5
    integer, external :: f_u
    call read_data(d1,d2,d3,d4,d5)
    u1 = f_u(d1,d2)
end

integer function u2()
    implicit none
    integer :: d1,d2,d3,d4,d5
    integer, external :: f_u
    call read_data(d1,d2,d3,d4,d5)
    u2 = f_u(d3,d4)
end
```

- Problem : The same data will be recomputed multiple times.
- Solution : memo functions

Implicit Reference to Parameters programming strategy

1. Each entity has only only one builder : a subroutine that builds a *valid* value of an entity

```
subroutine build_t(x,y,result)
    implicit none
    integer, intent(in)  :: x, y
    integer, intent(out) :: result
    result = x + y + 4
end subroutine build_t

subroutine build_w(x,result)
    implicit none
    integer, intent(in)  :: x
    integer, intent(out) :: result
```

```
    result = x + 3
end subroutine build_w

subroutine build_v(x,y,result)
    implicit none
    integer, intent(in)  :: x, y
    integer, intent(out) :: result
    result = x + y + 2
end subroutine build_v

subroutine build_u(x,y,result)
    implicit none
    integer, intent(in)  :: x, y
    integer, intent(out) :: result
    result = x + y + 1
end subroutine build_u

subroutine build_d(d1,d2,d3,d4,d5)
```

```
    implicit none
    integer, intent(out) :: d1,d2,d3,d4,d5
    read(*,*) d1,d2,d3,d4,d5
end
```

2. Each entity has only one provider : a subroutine with no input arguments whose role is to prepare a *valid* value of an entity.

```
module nodes

  ! Nodes
  integer :: u1
  logical :: u1_is_built = .False.

  integer :: u2
  logical :: u2_is_built = .False.

  integer :: v
```

```
  logical :: v_is_built  = .False.

  integer :: w
  logical :: w_is_built  = .False.

  integer :: t
  logical :: t_is_built  = .False.

  ! Leaves
  integer :: d1, d2, d3, d4, d5
  logical :: d_is_built  = .False.

end module
```

```
subroutine provide_t
    use nodes
    implicit none
    if (.not.t_is_built) then
        call provide_u1
        call provide_v
        call build_t(u1,v,t)
        t_is_built = .True.
    endif
end subroutine provide_t

subroutine provide_w
    use nodes
    implicit none
    if (.not. w_is_built) then
        call provide_d
        call build_w(d5,w)
        w_is_built = .True.
```

```
    endif
end subroutine provide_w

subroutine provide_v
    use nodes
    implicit none
    if (.not. v_is_built) then
        call provide_u2
        call provide_w
        call build_v(u2,w,v)
        v_is_built = .True.
    endif
end subroutine provide_v

subroutine provide_u1
    use nodes
    implicit none
    if (.not. u1_is_built) then
```

```
        call provide_d
        call build_u(d1,d2,u1)
        u1_is_built = .True.
    endif
end subroutine provide_u1

subroutine provide_u2
    use nodes
    implicit none
    if (.not. u2_is_built) then
        call provide_d
        call build_u(d3,d4,u2)
    endif
end subroutine provide_u2

subroutine provide_d
    use nodes
    implicit none
```

```
    if (.not. d_is_built) then
        call build_d(d1,d2,d3,d4,d5)
        d_is_built = .True.
    endif
end
```

3. Calling a provider always *guarantees* that the entity of interest is **valid** after the provider has been called

The main program is simply:

```
program test
    use nodes
    implicit none
    call provide_t
    print *,  "t=", t
end program
```

Summary

With the IRP method:

- Code is easy to develop for a new developer : Adding a new feature only requires to know the names of the needed entities
- If one developer changes the dependence tree, the others will not be affected : collaborative work is simple
- Forces to write clear code : one builder builds only one thing
- Forces to write efficient code : temporal locality is good, as in cache oblivious algorithms

But in real life:

- A lot of typing is required
- Programmers are lazy

IRPF90

- Code generator that will write all the IRP glue code for you
- Fortran with additional keywords
- Extends fortran to add very useful features :
  - Automatic makefile generation
  - Text editor integration
  - Some Introspection
  - Meta programming
  - Many more interesting things

```
BEGIN_PROVIDER [ integer, t ]
  t = u1+v+4
END_PROVIDER

BEGIN_PROVIDER [ integer,w ]
  w = d5+3
END_PROVIDER

BEGIN_PROVIDER [ integer, v ]
  v = u2+w+2
END_PROVIDER

BEGIN_PROVIDER [ integer, u1 ]
  integer :: fu
  u1 = fu(d1,d2)
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
```

```
  integer :: fu
  u2 = fu(d3,d4)
END_PROVIDER

integer function fu(x,y)
  integer, intent(in) :: x,y
  fu = x+y+1
end function
```

```
program irp_example
  print *, 't=', t
end
```

When you write a provider for *x*, you **only** have to focus on

- How do I build *x*?
- What are the variables that I need to build *x*?
- Am I sure that *x* is built correctly when I exit the provider?

Features

Arrays

```
BEGIN_PROVIDER [ double precision, A, (dim1, 3) ]
  ...
END_PROVIDER
```

- Allocation of IRP arrays done automatically
- Dimensioning variables can be IRP entities, provided before the memory allocation
- FREE keyword to force to free memory. Invalidates the entity.

Documentation

Every subroutine/function/provider should have a documentation section:

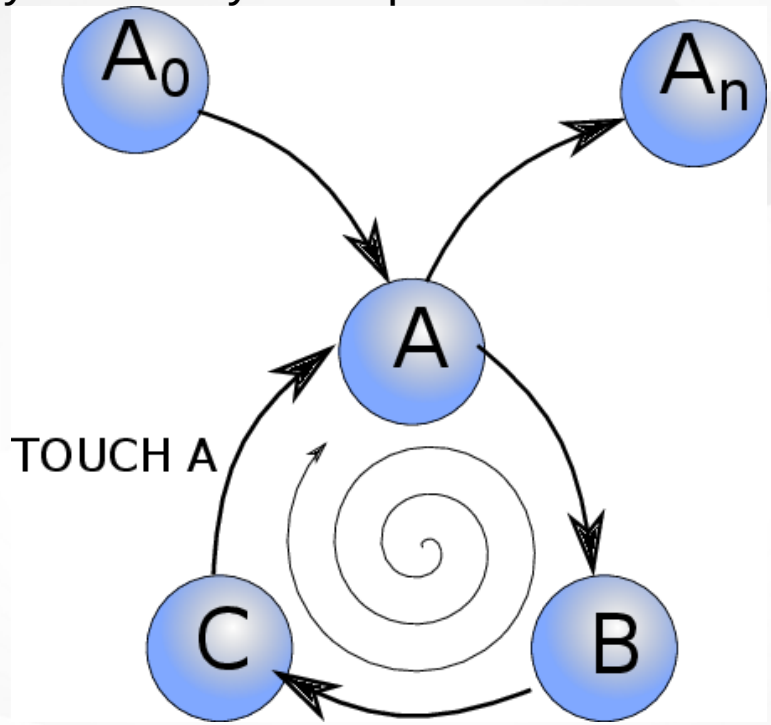
```
BEGIN_PROVIDER [ double precision, Fock_matrix_beta_mo, (mo_tot_num_align,mo_tot_num) ]
 implicit none
 BEGIN_DOC
 ! Fock matrix on the MO basis
 END_DOC
 ...
END_PROVIDER
```

```
$ irpman fock_matrix_beta_mo
```

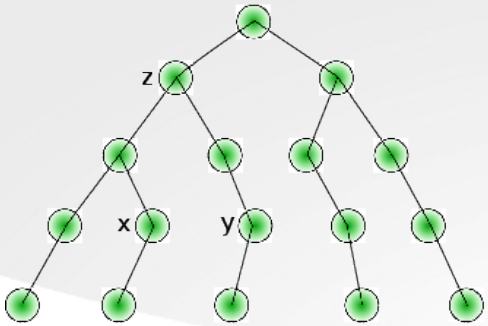
```
IRPF90 entities(l)                    fock_matrix_beta_mo                    IRPF90 entities(l)

Declaration
       double precision, allocatable :: fock_matrix_beta_mo   (mo_tot_num_align,mo_tot_num)

Description
       Fock matrix on the MO basis

File
       Fock_matrix.irp.f

Needs
       ao_num
       fock_matrix_alpha_ao
       mo_coef
       mo_tot_num
       mo_tot_num_align

Needed by
       fock_matrix_mo

IRPF90 entities                       fock_matrix_beta_mo                    IRPF90 entities(l)
```

Iterative processes

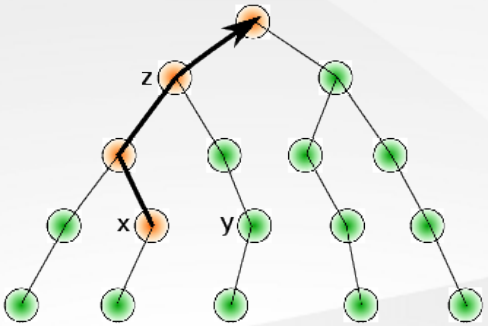
Iterative processes may involve cyclic dependencies:



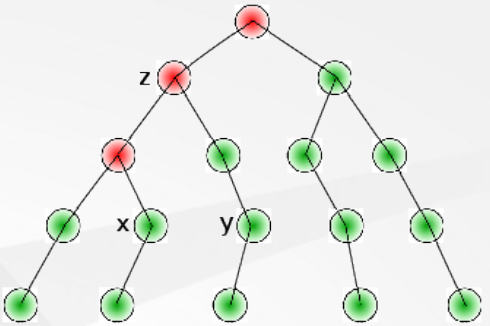
`TOUCH A` : `A` is valid, but everything that needs `A` is invalidated



(a)



(b)



(c)

Enbedding scripts

- Info at compile time
- Specific formulas (see fast power functions later...)

```
BEGIN_SHELL [ /bin/bash ]
  echo print *, \'Compiled by `whoami` on `date`\'
END_SHELL
```

```
BEGIN_SHELL [ /usr/bin/python ]
for i in range(100):
  print """
     double precision function times_%d(x)
       double precision, intent(in) :: x
       times_%d = x*%d
     end
  """%locals()
END_SHELL
```

Other features

- Assert keyword
- Templates
- Syntax highlighting in Vi
- Generation of tags to navigate in the code
- Variables can be declared *anywhere*
- Dependencies are known by IRPF90 -> Makefiles are built automatically
- No problem using external libraries
- etc...

IRPF90 for HPC

In this section, it is recommended to use the Intel Fortran compiler (ifort).

Array alignment

- Vector instructions (ADD/MUL/LOAD/STORE/ etc) operate on **aligned** data.
- SSE : 16 bytes, AVX/AVX2 : 32 bytes, AVX512 : 64 bytes.
- If we can easily align data -> performance gain
  - Array : `!DIR$ ATTRIBUTES ALIGN : 32 :: A`
  - Loop : `!DIR$ VECTOR ALIGNED`
- For an *aligned* multi-dimensional array, all columns are aligned *if* the LDA is a multiple of the alignment

Using the `--align <n>` option, IRPF90 can introduce compiler directives for ifort such that *all* the IRP arrays are *n*-byte aligned. The `$IRP_ALIGN` variable corresponds *n*.

```
integer function align_double(i)
  integer, intent(in) :: i
  integer :: j
  j = mod(i,max($IRP_ALIGN,4)/4)
  if (j==0) then
    align_double = i
  else
    align_double = i+4-j
  endif
end

 BEGIN_PROVIDER [ integer, n ]
&BEGIN_PROVIDER [ integer, n_aligned ]
  integer :: align_double
  n = 19
  n_aligned = align_double(19)
END_PROVIDER
```

```
BEGIN_PROVIDER [ double precision, Matrix, (n_aligned,n) ]
  Matrix = 0.d0
END_PROVIDER
```

- All IRP entities are aligned
- All columns of array `Matrix` are aligned
- -> We can happily use `!DIR$ Vector aligned`

Variable substitutions

Create a binary targeted for a given input :

```
if (choice1) then
  !DIR$ VECTOR ALIGNED
  do i=1,lmax
    call do_stuff
  enddo
else
  !DIR$ VECTOR ALIGNED
  do i=1,nmax
    call do_something_else
  enddo
endif
```

```
irpf90 --align=32 -s lmax:100 -s nmax:48 -s choice1:.True.
```

```
if (.True.) then
  !DIR$ VECTOR ALIGNED        !
  do i=1,100                  ! Compiler knows
    call do_stuff             ! what is the best
  enddo                       ! optimization
else                          !
  !DIR$ VECTOR ALIGNED        !
  do i=1,48                   ! Dead code
    call do_something_else    ! removed by
  enddo                       ! the compiler
endif
```

Other features

- Profiler based on `rdtsc (--profile)`
- Codelet generation for code optimization
- No problem using external libraries (MKL, MPI, etc)
- No problem using OpenMP (`--openmp`)
- Support for Coarray Fortran (`--coarray`)
- Generated code is **very** efficient : sustained 960 Tflops/s on Curie in 2011 with QMC=Chem (12 GFlops/s / core)

Interested?

Quantum Package : *Quantum Chemistry (OpenMP)*

https://github.com/LCPQ/quantum_package

QMC=Chem : *Quantum Monte Carlo (ZeroMQ)*

http://qmcchem.ups-tlse.fr

EPLF : *Electron pair localization function (MPI)*

http://eplf.sourceforge.net

EZFIO : *Easy Fortran I/O library generator*

https://github.com/scemama/ezfio

Source on GitHub

https://github.com/scemama/irpf90

GitBook (not finished)

http://scemama.gitbooks.io/irpf90/

Web page

http://irpf90.ups-tlse.fr