IRPF90: A Fortran code generator for HPC

Anthony Scemama

13/07/2017

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse, France

Issue

- Scientific codes need speed $\Longrightarrow$: Fortran / C
- Low-level languages : difficult to maintain
- Too high-level features of modern Fortran (`matmul`, array syntax, derived types, ...) or C++ (objects, STL) can kill the efficiency

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

Issue

A simple solution : use multiple languages.

- High-level : text parsing, global code architecture, ...
- Low-level : computation
- Meta-programming : generate low-level code with a higher-level language

Problem addressed here

Make code in the low-level language easy to write and maintain

Outline

Programming with Implicit Reference to Parameters (IRP)

Outline

What is a scientific code?

A program is a function of its input data:

$$\text{output} = \text{program}(\text{input})$$

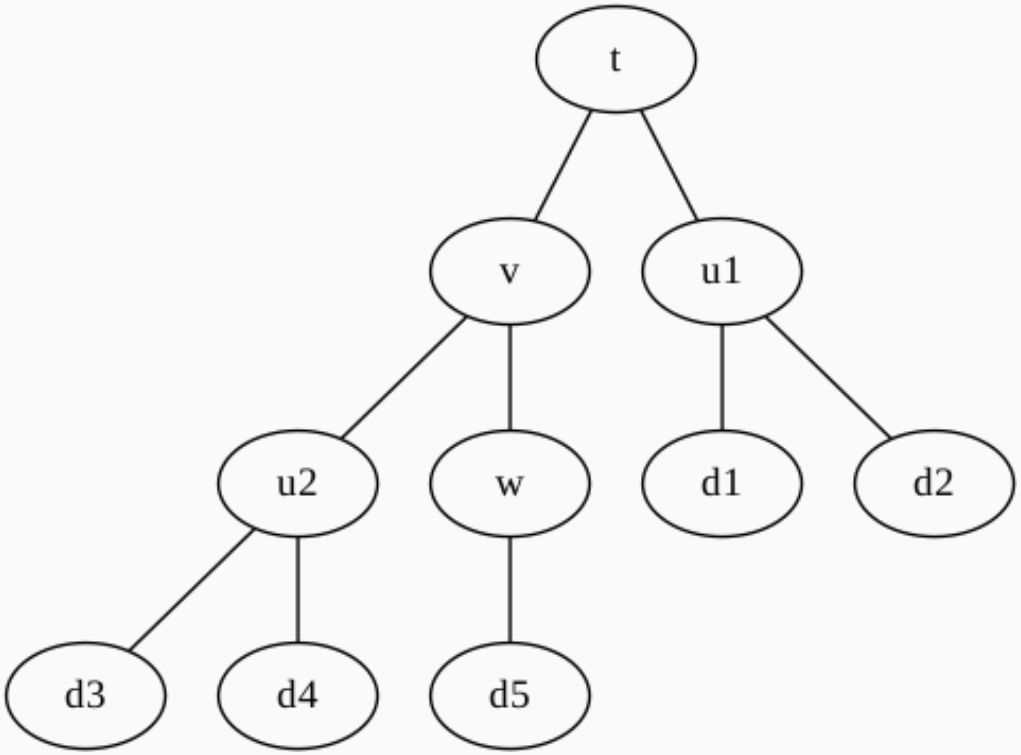
A program can be represented as a production tree where

- 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

What is a scientific code?

Example: Production tree of $t(u(d_1, d_2), v(u(d_3, d_4), w(d_5)))$

$$
\begin{aligned}
u(x,y) &= x+y+1 \\
v(x,y) &= x+y+2 \\
w(x) &= x+3 \\
t(x,y) &= x+y+4
\end{aligned}
$$



Traditional Fortran implementation

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

  call read_data(d1,d2,d3,d4,d5)       !           t
                                       !         /    \
  call compute_u(d3,d4,u)              !      u         v
  call compute_w(d5,w)                 !    /  |       |  \
  call compute_v(u,w,v)                ! d1    d2     u     w
  call compute_u(d1,d2,u)              !           /    \    \
  call compute_t(u,v,t)                !         d3     d4    d5

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

Difficulties

Imperative programming (wikipedia)

[. . .] programming paradigm that uses statements that change a program's state.

- The code expresses the exploration of the production tree
- The routines have to be called in the correct order
- The values of variables are time-dependent

Consequences

Sources of complexity

1. Time-dependence of the data (*mutable data*)
2. Handling the complexity of the production tree

1. Time-dependence

Functional programming (wikipedia)

[. . .] programming paradigm [. . .] that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

No time-dependence (*immutable data*) $\Longrightarrow$ reduced complexity

"Functional-like" implementation in Fortran

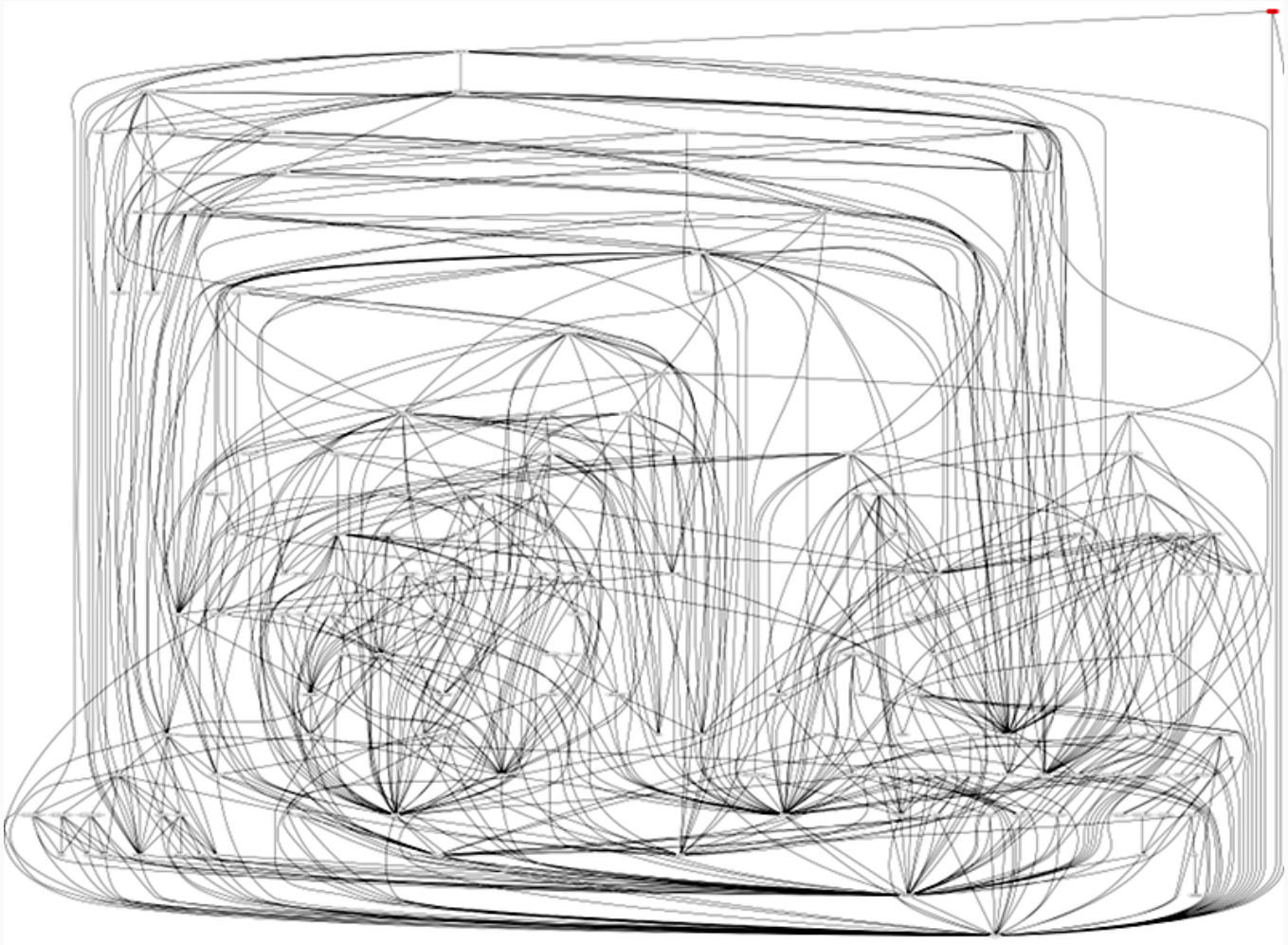
```
program compute_t                          !           t
  implicit none                            !         /   \
  integer :: d1, d2, d3, d4 d5             !       u       v
  integer :: u, v, w, t                    !     /  |      |  \
                                           ! d1   d2    u    w
  call read_data(d1,d2,d3,d4,d5)           !           /   \   \
                                           !         d3    d4   d5
  ! Functional starts here
  write(*,*), "t=", t( u(d1,d2), v( u(d3,d4), w(d5) ) )
end program
```

- Instead of telling *what to do*, we express *what we want*
- The programmer doesn't handle the execution sequence

No time-dependence left

2. Complexity of the production tree

Production tree of Ψ in QMC=Chem: 149 nodes / 689 edges



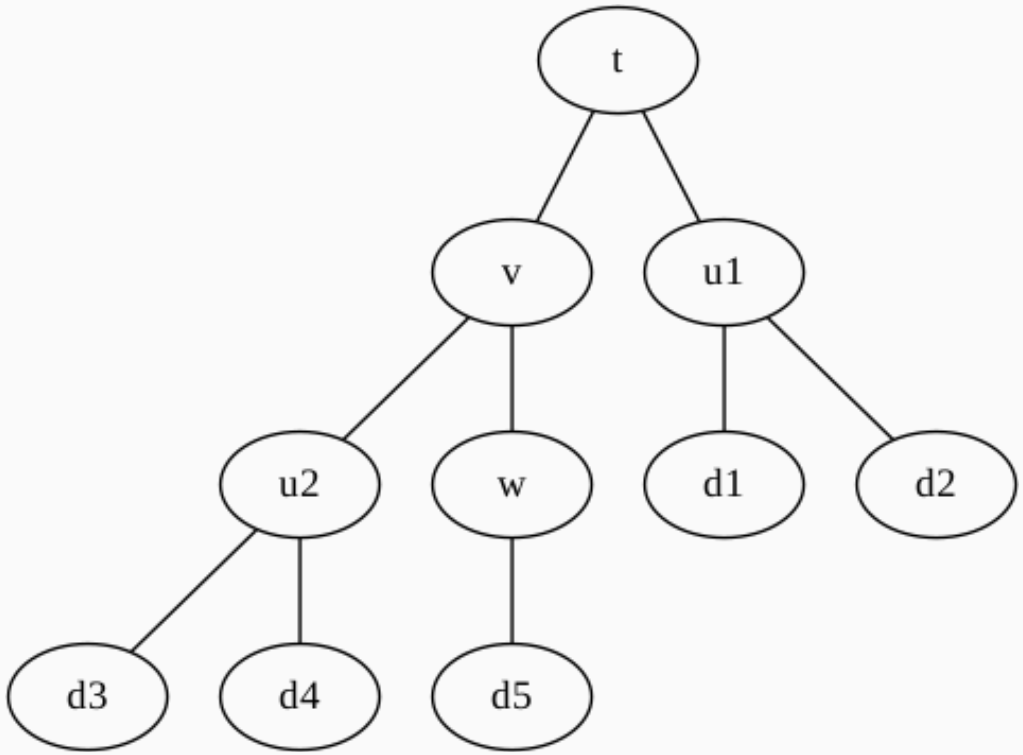
2. Complexity of the production tree

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

From global to local knowledge

Express the needed entities for each node:

- $t \rightarrow u_1$ and v
- $u_1 \rightarrow d_1$ and d_2
- $v \rightarrow u_2$ and w
- $u_2 \rightarrow d_3$ and d_4
- $w \rightarrow d_5$



The information is now *local* and easy to handle.

Localize information

```
program compute_t
  integer, external :: t
  write(*,*), "t=", t()
end program

integer function t()
  integer, external :: u1, v
  t = u1() + v() + 4
end

integer function v()
  integer, external :: u2, w
  v = u2() + w() + 2
end

integer function w()
  integer :: d1,d2,d3,d4,d5
  call read_data(d1,d2,d3,d4,d5)
  w = d5+3
end
```

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

integer function u1()
  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()
  integer :: d1,d2,d3,d4,d5
  integer, external :: f_u
  call read_data(d1,d2,d3,d4,d5)
  u2 = f_u(d3,d4)
end
```

Consequences

- The global production tree is not known by the programmer
- The program is easy to write
- Any change of dependencies will be handled properly automatically

But: The same data will be recomputed multiple times.

Lazy evaluation

Simple solution : Lazy evaluation using memo functions.

Lazy Evaluation (Wikipedia)

> *In programming language theory, lazy evaluation, or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing). The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name.*

Outline

Glossary

Entity Node of the production tree

Builder Subroutine that builds a valid value of an entity from its dependencies

Valid Fully initialized with meaningful values

Provider Subroutine with no argument which guarantees to return a valid value of an entity

Rules of IRP[1]

1. Each entity has only one provider
2. Before using an entity, its provider has to be called

[1] François Colonna : "IRP programming : an efficient way to reduce inter-module coupling ", DOI: 10.13140/RG.2.1.3833.0406

IRP example in standard Fortran

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

module entities
  ! Entities
  integer :: u1, u2, v, w, t
  logical :: u1_is_built = .False.
  logical :: u2_is_built = .False.
  logical :: v_is_built  = .False.
  logical :: w_is_built  = .False.
  logical :: t_is_built  = .False.

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

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

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
```

Summary

With the IRP method:

1. Code is easy to develop for a new developer : Adding a new feature only requires to know the *names* of the needed entities
2. If one developer changes the dependence tree, the others will not be affected : collaborative work is simple
3. Forces to write clear code : one builder builds only one thing
4. Forces to write efficient code (spatial and temporal localities are good)

But in real life:

1. A lot more typing is required
2. Programmers are lazy

Outline

IRPF90

- Extends Fortran with additional keywords
- Fortran code generator (source-to-source compiler)
- Writes all the mechanical IRP code

Useful features:

- Automatic Makefile generation
- Automatic Documentation
- Text editor integration
- Some Introspection
- Meta programming
- Some features targeted for HPC

`http://irpf90.ups-tlse.fr`

`https://github.com/scemama/irpf90`

`https://www.gitbook.com/book/scemama/irpf90`

Questions

3 questions that the programmer should ask before writing a provider for x:

- How to I build x?
- What are the names of the entities that I need?
- Am I sure that when I exit x will be built correctly?

That's it.

IRPF90 example

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

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
```

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.

Features : Documentation

```
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
```

Features : Documentation

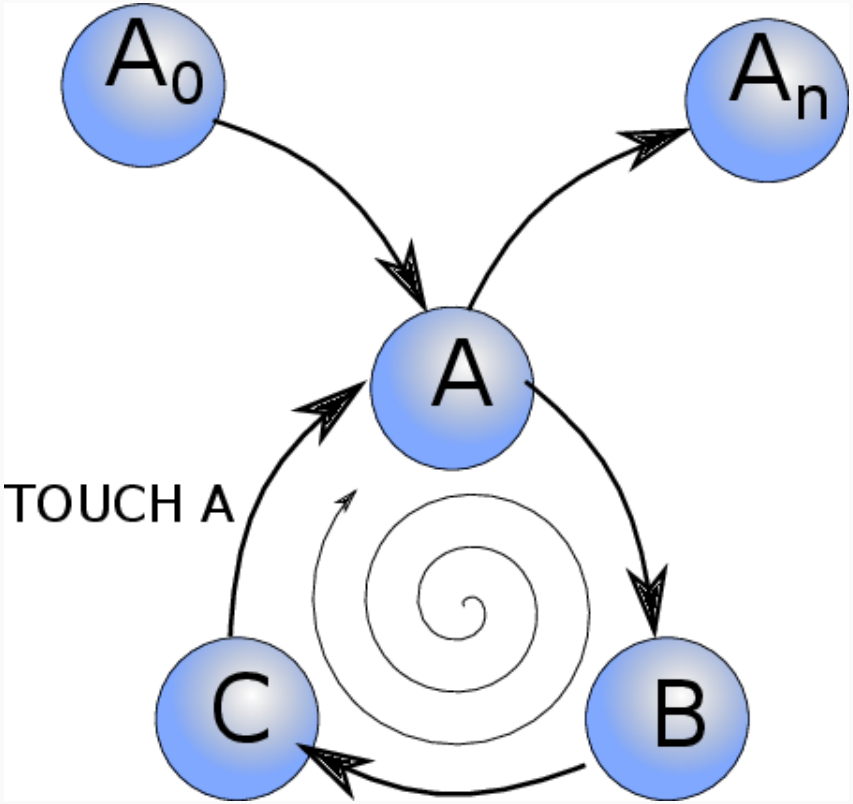
```
IRPF90 entities(1)                    fock_matrix_beta_mo                    IRPF90 entities(1)

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(1)
```

IRPF90 demo

MOVIE

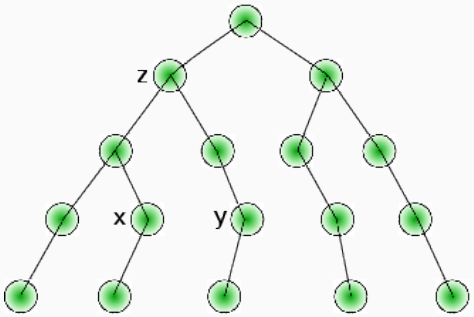
Iterative processes

Iterative processes involve cyclic dependencies

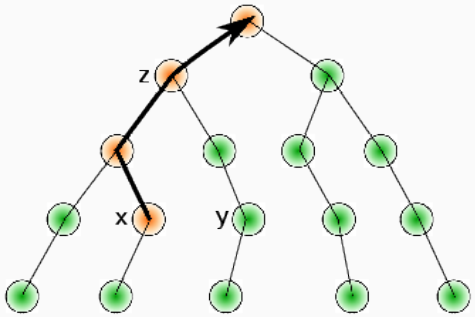


`TOUCH A` : `A` is valid, but everything that needs `A` is invalidated.

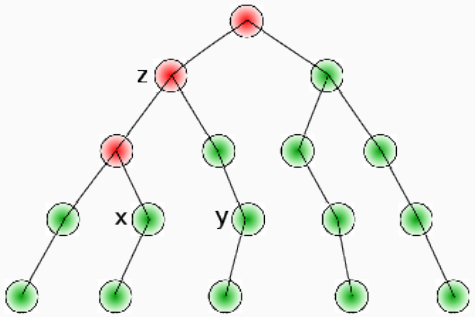
Iterative processes



(a)



(b)



(c)

(a) Everything is valid

(b) x is modified

(c) x `TOUCH`ed

Many other features

- Assert keyword, Templates
- Variables can be declared *anywhere*
- +=, -=, *= operators
- Dependencies are known by IRPF90 $\rightarrow$ Makefiles are built *automatically*
- Syntax highlighting in Vim
- Generation of tags to navigate in the code
- No problem mixing with external Fortran files
- No problem using external libraries (MKL, MPI, etc)
- ...

HPC features of IRPF90

Array alignment

- Vector instructions require *aligned* data
- Different alignment for SSE4.2, AVX, AVX512
- Compiler directives / OpenMP

```
!DIR$ ATTRIBUTE ALIGN : 64 ::  A
!DIR$ VECTOR ALIGNED
```

- The program should parameterized by the alignment
- For an aligned matrix, all the columns are aligned *iff* the 1st dimension is a multiple of the alignment
- The `--align=<n>` option tells IRPF90 to align all the provided arrays
- The `$IRP_ALIGN` variable is available everywhere in the code

Array alignment

Write a function to provide the closest multiple of the alignment:

```
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
  end if
end
```

Array alignment

```
BEGIN_PROVIDER [ integer, n ]
  n = 19  ! or whatever dimension
          ! read from input / computed ...
END_PROVIDER

BEGIN_PROVIDER [ integer, n_aligned ]
  BEGIN_DOC
! Provider for the leading dimension of the array
  END_DOC
  integer, external :: align_double
  n_aligned = align_double(n)
END_PROVIDER
```

Array alignment

```
BEGIN_PROVIDER [double precision, MyMatrix, (n_aligned,n)]
  BEGIN_DOC
! n.n matrix with padding
  END_DOC
  MyMatrix = 0.d0
END_PROVIDER
```

If `MyMatrix` is aligned by `--align=32`

- `n=19` ; `n_align` = 20
- Every column of the array has the proper alignment
- We can happily use `!DIR$ VECTOR ALIGNED`

Variable substitution

Create an executable for specific input data

```
if (choice1) then
  !DIR$ VECTOR ALIGNED
  do i=1,lmax
    ! Do some work
  end do
else
  !DIR$ VECTOR ALIGNED
  do i=1,nmax
    ! Do something else
  end do
end if
```

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

Variable substitution

Generated code:

```
if (.True.) then
  !DIR£ VECTOR ALIGNED
  do i=1,100                ! The Fortran compiler
    ! Do some work          ! knows wht is the best
  end do                    ! optimization strategy
else
  !DIR£ VECTOR ALIGNED
  do i=1,48                   ! This dead code is
    ! Do something else       ! automatically removed
  end do                      ! by the compiler
end if
```

Embedding scripts

Get information at compile-time :

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

Meta-programming (specific formulas, etc) :

```
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"""%(i,i,i)
END_SHELL
```

Profiling

`--profile` gives a summary for every provider

	N.Calls	Tot Cycles	Avg Cycles	Tot Secs	Avg Secs
...					
ci_energy	6.	1662765.	277127.+/- 59043.	0.00072451	0.00012+/- 0.00003
coef_hf_selector	7.	13009101.	1858443.+/- 605660.	0.00566841	0.00081+/- 0.00026
davidson_criterion	1.	1736.	1736.+/- 0.	0.00000076	0.00000+/- 0.00000
davidson_sze_max	1.	18.	18.+/- 0.	0.00000001	0.00000+/- 0.00000
det_connections	1.	6945057.	6945057.+/- 0.	0.00302614	0.00303+/- 0.00000
diag_algorithm	6.	15253.	2542.+/- 246.	0.00000665	0.00000+/- 0.00000
do_pt2_end	1.	233928.	233928.+/- 0.	0.00010193	0.00010+/- 0.00000
elec_alpha_num	1.	751170.	751170.+/- 0.	0.00032730	0.00033+/- 0.00000
exc_degree_per_selectors	7.	209402.	29915.+/- 10827.	0.00009124	0.00001+/- 0.00000
expected_s2	1.	240961.	240961.+/- 0.	0.00010499	0.00010+/- 0.00000
ezfio_filename	1.	386883.	386883.+/- 0.	0.00016858	0.00017+/- 0.00000

Codelet generation

`--codelet` command line option creates a new program to time a specific provider

```
$ irpf90 --codelet v:t:100000
```

Creates a codelet to time `v` by looping 100 000 times.
`t` is required to be provided before.

Other HPC features

- `--memory`: traces memory alloc/dealloc
- No problem using external libraries (MKL, MPI, `etc`)
- Using OpenMP requires `--openmp` for thread-safety of providers
- Experimental support for coarray Fortran

Real world examples

Quantum Package : simplicity of development

IRPF90 library for post-HF quantum chemistry

- Goal : easy for the programmer
- Long term objective : Massively parallel quantum chemistry
- Open Source (GPL)
- Hosted on GitHub : `https://github.com/LCPQ/quantum_package`
- Multi-site development (Toulouse, Paris, ANL) with easy integration

QMC=Chem : High performance

`https://github.com/scemama/qmcchem`

- Quantum Monte Carlo for Chemistry
- Mixed single/double precision
- Linear Algebra on small (sparse) matrices ($< 300 \times 300$)
- Highly optimized for Sandy Bridge (Codelets)
- Sustained 0.96 PFlops/s in 2011: Curie (GENCI/France)
- Without touching the code, AVX2 $\rightarrow$ AVX-512 on Skylake : 30% perf gain

QMC=Chem : 1st hot spot

Dense matrix $\times$ sparse vector:

```
do kao=1,kmax2,4
  ...
  do k=0,LDA-1,$IRP_ALIGN/4
      !DIR$ VECTOR ALIGNED
      do j=1,$IRP_ALIGN/4
        C1(j+k) = C1(j+k) + A(j+k,k_vec(1))*d11 + A(j+k,k_vec(2))*d21  &
                          + A(j+k,k_vec(3))*d31 + A(j+k,k_vec(4))*d41
      end do

      !DIR$ VECTOR ALIGNED
      do j=1,$IRP_ALIGN/4
        C2(j+k) = C2(j+k) + A(j+k,k_vec(1))*d12 + A(j+k,k_vec(2))*d22  &
                          + A(j+k,k_vec(3))*d32 + A(j+k,k_vec(4))*d42
        C3(j+k) = C3(j+k) + A(j+k,k_vec(1))*d13 + A(j+k,k_vec(2))*d23  &
                          + A(j+k,k_vec(3))*d33 + A(j+k,k_vec(4))*d43
      end do

      !DIR$ VECTOR ALIGNED
      do j=1,$IRP_ALIGN/4
        C4(j+k) = C4(j+k) + A(j+k,k_vec(1))*d14 + A(j+k,k_vec(2))*d24  &
                          + A(j+k,k_vec(3))*d34 + A(j+k,k_vec(4))*d44
        C5(j+k) = C5(j+k) + A(j+k,k_vec(1))*d15 + A(j+k,k_vec(2))*d25  &
                          + A(j+k,k_vec(3))*d35 + A(j+k,k_vec(4))*d45
      end do
  end do
end do
```

QMC=Chem : 1st hot spot

Dense matrix $\times$ sparse vector:

```
do kao=1,kmax2,4
  ...
  !DIR$ VECTOR ALIGNED
  do j=1,$IRP_ALIGN/4
    C2(j+k) = C2(j+k)                                    &
      + A(j+k,k_vec(1))*d12 + A(j+k,k_vec(2))*d22  &
      + A(j+k,k_vec(3))*d32 + A(j+k,k_vec(4))*d42

    C3(j+k) = C3(j+k)                                    &
      + A(j+k,k_vec(1))*d13 + A(j+k,k_vec(2))*d23  &
      + A(j+k,k_vec(3))*d33 + A(j+k,k_vec(4))*d43
  end do
  ...
end do
```

QMC=Chem : 1st hot spot

Efficiency of the matrix products (AVX2):

```
483c:       c4 21 7c 10 34 17       vmovups (%rdi,%r10,1),%ymm14
4842:       c4 22 05 b8 04 17       vfmadd231ps (%rdi,%r10,1),%ymm15,
4848:       c4 62 4d a8 36          vfmadd213ps (%rsi),%ymm6,%ymm14
484d:       c4 22 5d b8 04 1f       vfmadd231ps (%rdi,%r11,1),%ymm4,%
4853:       c4 22 55 b8 34 1f       vfmadd231ps (%rdi,%r11,1),%ymm5,%
4859:       c4 22 6d b8 04 27       vfmadd231ps (%rdi,%r12,1),%ymm2,%
485f:       c5 fc 10 94 24 60 01    vmovups 0x160(%rsp),%ymm2
4866:       00 00
4868:       c4 22 7d b8 34 27       vfmadd231ps (%rdi,%r12,1),%ymm0,%
486e:       c4 22 6d b8 04 0f       vfmadd231ps (%rdi,%r9,1),%ymm2,%y
4874:       c5 fc 10 12             vmovups (%rdx),%ymm2
4878:       c4 22 15 b8 34 0f       vfmadd231ps (%rdi,%r9,1),%ymm13,%
487e:       c5 7c 11 01             vmovups %ymm8,(%rcx)
4882:       c5 7c 10 84 24 40 01    vmovups 0x140(%rsp),%ymm8
```

QMC=Chem : 1st hot spot

Static analysis (MAQAO):

- No peel/tail loop. 100% vectorized code.
- AVX : 16 flops/cycle (100% peak). Up to 64% of the peak measured on Sandy Bridge in 2011 (L3-bound)
- AVX2: 20 flops/cycle (62% peak). Load units are a bottleneck

QMC=Chem : 2nd hot spot

- Matrix inversion
- Up to 5×5: $N!$ algorithm (hard-coded)
- Shermann-Morrisson updates written with templates (see code)

Summary

- IRP Programming: Method to simplify the development of large codes
- IRPF90: A DSL for IRP programming in Fortran, enabling performance

IRPF90: a programming environment for high performance computing
ArXiv e-prints, [cs.SE](0909.5012v1)