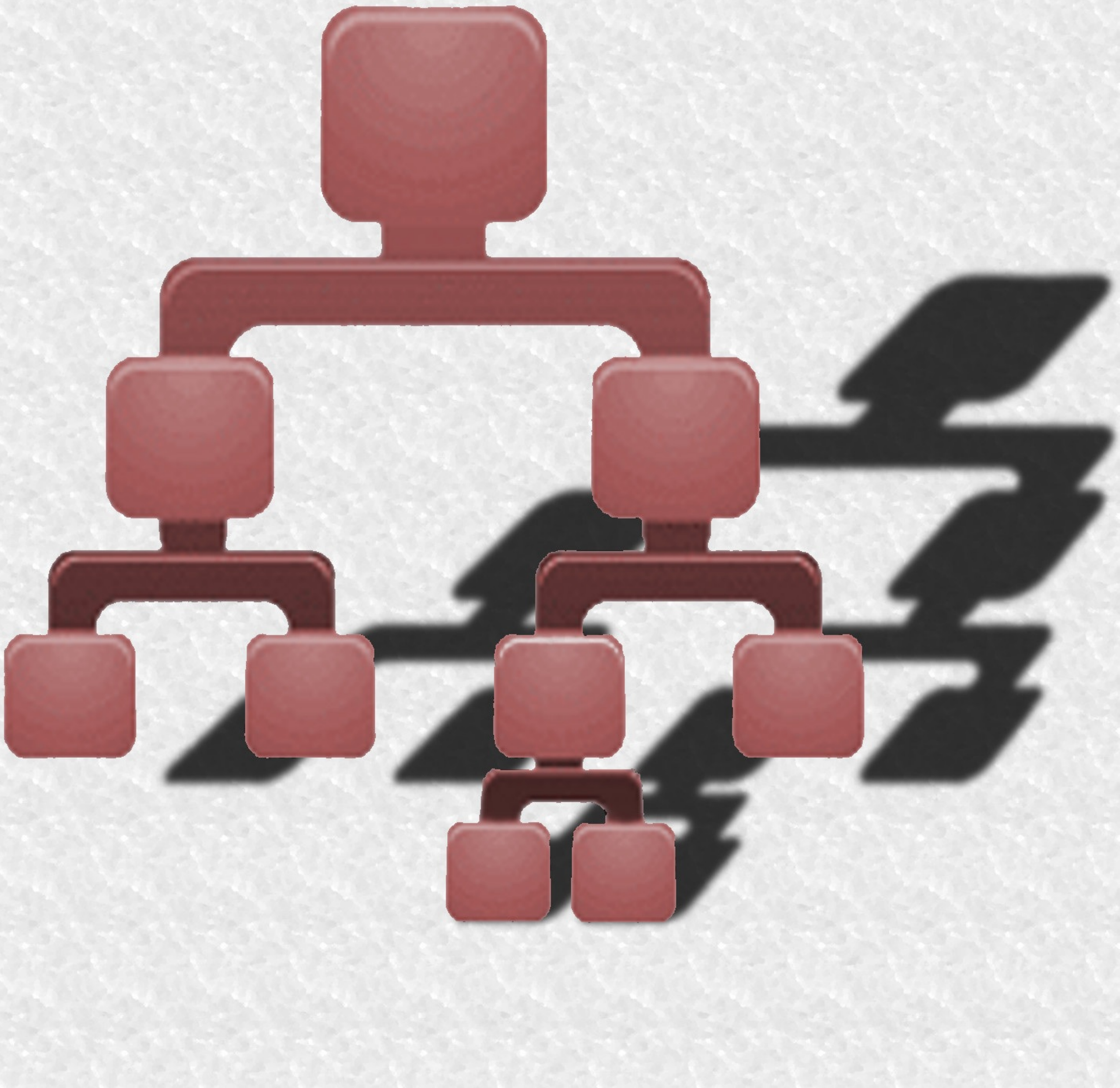
IRPF90



Anthony Scemama

Table of Contents

IRPF90

IRPF90 is a Fortran code generator. Schematically, the programmer only writes computation kernels, and IRPF90 generates the "glue code" that will link all these kernels together to produce the expected result, handling all the relationships between the variables. In this way, even large codes can still be under control.

IRPF90 (1.7.5) is free software under the GPL licence available on GitLab.

Introduction

Today, large scientific codes in Fortran are difficult to maintain. The complexity of the programs comes from the dependencies between the different entities of the code. As the entities become more and more coupled, the program becomes more and more difficult to maintain and to debug.

If the programmer wants to keep the code under control, he has to be aware of all the consequences of a modification of the source code on all possible execution paths. When the code was written by multiple developers and when the code is large (hundred thousands of lines), this becomes extremely difficult for the programmer. However, the machine can handle easily such a complexity by handling all the dependencies between the variables, as in a Makefile.

IRPF90 is a Fortran code generator. Schematically, the programmer only writes computation kernels, and IRPF90 generates the "glue code" that will link all these kernels together to produce the expected result, handling all the relationships between the variables. In this way, even large codes can still be under control.

The Implicit Reference to Parameters programming strategy

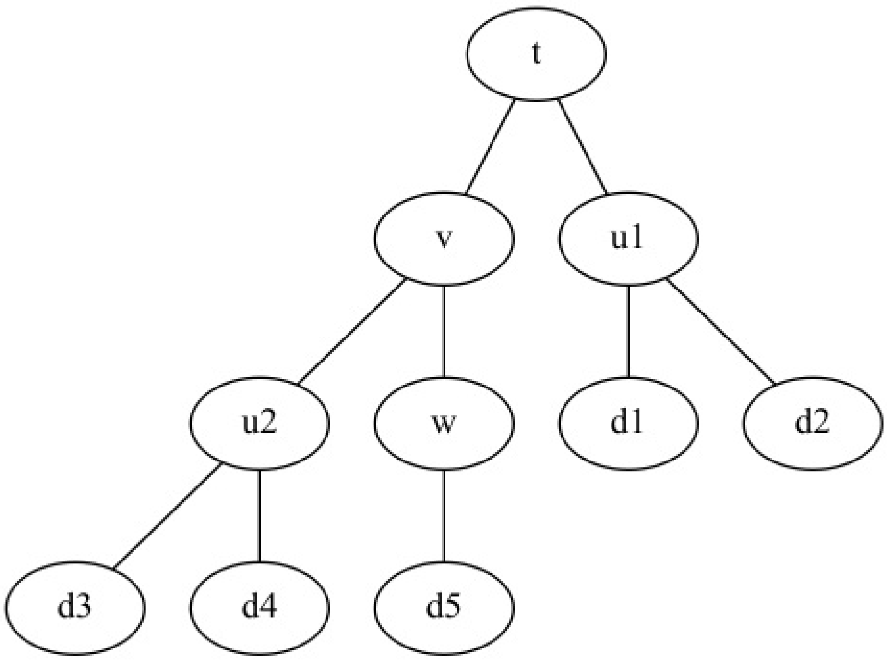
The Implicit Reference to Parameters (IRP) method was first introduced by François Colonna in the paper IRP programming : an efficient way to reduce inter-module coupling. Here, we only give a practical overview of the main ideas, but the reader is encouraged to read the original paper.

A scientific program (or sub-program) is a complicated function of its data. One can represent the program as a tree whose root is the output and whose leaves are the data. The nodes are the intermediate variables, and the edges represent the *needs/needed_by* relationships.

Let us consider a program which computes `t( u(d1,d2), v(u(d3,d4), w(d5)) )` with

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

This program can be represented with the following tree:



Writing the program in Fortran would require the programmer to have this tree in mind:

```
program compute_t
    implicit none

    integer :: d1, d2, d3, d4 d5  ! Input data
    integer :: u1, u2, v, w, t    ! Computed entities

    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
```

This way of programming is imperative, which is the natural way to write Fortran : the programmer tells the machine how its internal state will change by giving step-by-step instructions. If the instructions are not given in the proper order, the program is wrong. Therefore, at each line the programmer has to be aware of the full state of the program, which results from the *needs/needed_by* relationships of the variables. Imperative programming explores the tree from the leaves to the root.

The same program can be written using the functional programming paradigm. Instead of telling the machine *what to do*, we can express *what we want*. Considering the program this way explores the tree from the root to the leaves.

```
program compute_t
    implicit none

    integer :: d1, d2, d3, d4 d5        ! Input data
    integer, external :: u, u, v, w, t  ! Functions

    call read_data(d1,d2,d3,d4,d5)

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

end program
```

Now, the *needs/needed_by* relationships between the entities are expressed by calling function `t`. The programmer doesn't handle any more the order in which the instructions will be executed : we don't known which one `u(d3,d4)` and `w(d5)` will be executed first. However, the global knowledge of the tree is still required to write this program.

In order to get rid of the *global knowledge* of the tree, we will transform it into *local knowledge*, which is much easier to handle. For each entity, we will only express the other 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

It appears now that the arguments of the functions are not *variables* but *parameters*. In that case, we can put the parameters inside the functions, as they will always be the same.

```
program compute_t
    implicit none
    integer, external :: t
    write(*,*), "t=", t()
end program
```

```
integer function t()
    implicit none
    integer, external :: u1, v
    t = u1() + v() + 4
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 v()
    implicit none
    integer, external :: u2, w
    v = u2() + w() + 2
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

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

```
subroutine read_data(d1,d2,d3,d4,d5)
    implicit none
    integer, intent(out) :: d1,d2,d3,d4,d5
    print *,  'd1,d2,d3,d4,d5 ?'
    read (*,*) d1,d2,d3,d4,d5
end
```

Now, the program automatically builds the tree and explores it. The programmer doesn't have to handle the execution of the code any more. However, there are a few aspects that can be improved. First, we have to write many empty parentheses `()` which make the code less readable. Secondly, we have to declare the return type of these functions every time we use them. Finally there is a major drawback: here, the data (`d1` ... `d5`) is read three times because there is no way to know that it has already been read. These last points can all be easily addressed. Indeed, if a function is a pure function (with no side effects), calling the function with the same values as arguments will always return the same value. In our program, the functions have no arguments, so we only need to build once the return value and cache it for subsequent calls. This mechanism is known as memoization.

For each node we write a *builder*, which is a subroutine that builds a *valid* value of an entity (according to the equations given at the beginning of this section).

```fortran
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
```

Then, we write a *provider* for each entity. A provider is a subroutine with no input arguments whose role is to prepare a valid value of an entity. It calls the providers of the needed entities, calls the builder of the desired entity, saves the computed value in a cache and then marks the quantity as built. The next calls to the provider will return the cached value.

Before writing the providers, we need to create a *global* variable for each node of the tree, as well as a flag to mark it as built. For convenience, we shall put all of them in a Fortran module `nodes` :

```
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
```

And the main program is just

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

The rules are simple:

1. Each entity has only one builder and only one provider
2. The arguments of the builder are the values of the needed entities.
3. Calling a provider always guarantees that the entity of interest is *valid* after the provider has been called

Applying rigorously these rules makes the development of large codes as easy as for smaller codes.

Introduction to IRPF90

As we have seen in the previous section, the IRP method is very powerful, but it requires a lot of discipline. IRPF90 is a tool that will write all the boilerplate IRP code for you, keeping your source code clear. It will also write Makefiles, documentation man pages, introduce compiler directives for code optimization, etc...

IRPF90 Basics

Let us rewrite the same code as in the previous section, but in the IRPF90 framework.

First, we create a file named `uvwt.irp.f` :

```
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 :: x,y
  fu = x+y+1
end function
```

This file contains usual Fortran statements, as well as new keywords. In Fortran there are subroutines and functions, and IRPF90 introduces *Providers*. If an entity is declared with a `BEGIN_PROVIDER ... END_PROVIDER` block, then it is an IRP entity and it will behave as a global variable in the whole program. All the provided entities are not supposed to be modified outside of their providers. The main point is that the provider will always be called automatically before the variable is used. The programmer doesn't know when and where the provider will be called.

Let us now introduce a provider for coupled data. Here, the input data will be read from the standard input in a given order, so it is convenient to provide them all at once in file `input.irp.f` :

```
 BEGIN_PROVIDER [ integer, d1 ]
&BEGIN_PROVIDER [ integer, d2 ]
&BEGIN_PROVIDER [ integer, d3 ]
&BEGIN_PROVIDER [ integer, d4 ]
&BEGIN_PROVIDER [ integer, d5 ]

  print *,  'd1, d2, d3, d4, d5?'
  read(*,*) d1, d2, d3, d4, d5

END_PROVIDER
```

Now, we can write the main function in the `irp_example1.irp.f` file:

```
program irp_example1
  implicit none
  print *, 't = ', t
end
```

To compile the program, we will have to set up the IRPF90 environment:

```
$ ls
input.irp.f  irp_example1.irp.f  uvwt.irp.f

$ irpf90 --init
$ ls
input.irp.f  irp_example1.irp.f  IRPF90_man  IRPF90_temp  Makefile
```

The created `IRPF90_temp` directory contains temporary files for the compiling step: the generated Fortran files, as well as the corresponding `.mod` and `.o` files. `IRPF90_man` contains the generated man pages that document the code, and a `Makefile` was created :

```
IRPF90 = irpf90  #-a -d
FC     = gfortran -ffree-line-length-none
FCFLAGS= -O2

SRC=
OBJ=
LIB=

include irpf90.make

irpf90.make: $(filter-out IRPF90_temp/%, $(wildcard */*.irp.f)) \
             $(wildcard *.irp.f) $(wildcard *.inc.f) Makefile
        $(IRPF90)
```

To build the test program, simply run `make` . The `Makefile` includes the `irpf90.make` file which does not exist, but there is a rule to create it by calling IRPF90. IRPF90 analyzes the code present in all the `*.irp.f` files of the current directory. The list of IRP entities is created in a first pass, then a second pass analyzes the dependencies between the entities. From all this information, it creates the Fortran code that will call the providers of each entity before it is used. As the dependencies between the entities are known the `irpf90.make` file, containing all the dependencies between the files, can be written.

Once IRPF90 has created the `irpf90.make` file, it can be included and the Fortran files can be compiled. As the `irpf90.make` file depends on all the `*.irp.f` files of the current directory, each modification or creation of an `*.irp.f` file will force IRPF90 to run before compiling. To summarize, you almost never need to write anything in the Makefiles. You just need to write `*.irp.f` files and run `make` .

```
$ make
Makefile:9: irpf90.make: No such file or directory
irpf90
IRPF90_temp/irp_example1.irp.module.F90
IRPF90_temp/irp_example1.irp.F90
IRPF90_temp/uvwt.irp.module.F90
IRPF90_temp/uvwt.irp.F90
IRPF90_temp/input.irp.module.F90
IRPF90_temp/input.irp.F90
Warning: Variable u1 is not documented
Warning: Variable u2 is not documented
Warning: Variable t is not documented
Warning: Variable w is not documented
Warning: Variable v is not documented
Warning: Variable d1 is not documented
Warning: Subroutine irp_example1 is not documented
Warning: Subroutine fu is not documented
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -O2 -c IRPF90_ter
gfortran -ffree-line-length-none -I IRPF90_temp/  -o irp_example1 I
```

Array entities

An array is considered *valid* when all of its values are valid. The dimensions of an array entity can be IRP entities, constants or intervals.

```
BEGIN_PROVIDER [ integer, fact_max ]
  fact_max = 10
END_PROVIDER

BEGIN_PROVIDER [ double precision, fact, (0:fact_max) ]
  implicit none
  integer :: i
  fact(0) = 1.d0
  do i=1,fact_max
    fact(i) = fact(i-1)*dble(i)
  end do
END_PROVIDER
```

In this example, as the array `fact` depends on its dimensioning variable `fact_max` , `fact_max` is provided first. Then, the `fact` array is allocated with the required dimensions, and then the code inside the provider is executed. Note that if the `fact` array is not used in the program, it will never be allocated.

Freeing entities

It is possible to free memory by using the `FREE` keyword.

```
BEGIN_PROVIDER [ double precision, table2, (size(table1,1)) ]
  implicit none
  table2(:) = 2.d0 * table1(:)
  FREE table1
END_PROVIDER
```

When `table1` is freed, the entity `table1` is marked as *non-valid*, such that if it is needed again, it will be reallocated and rebuilt.

When applying the `FREE` keyword to scalar entities, those are just marked as non-built.

Forcing to provide entities

The `PROVIDE` keyword forces to provide an entity, even if it is not needed.

In this example,

```
subroutine s()
  implicit none
  PROVIDE u v
end
```

`u` and `v` will be provided before entering in the scope of subroutine `s`.

This second example forces to re-provide the `random_x` entity at every loop cycle (version >= 1.5.0):

```
do i=1,N
  PROVIDE random_x
  print *, random_x
  FREE random_x
end do
```

Automatic documentation

Inside each provider, subroutine and function it is recommended to write a few lines to explain what it does. The documentation is written inside a `BEGIN_DOC ... END_DOC` block.

```
BEGIN_PROVIDER [ double precision, fact, (0:fact_max) ]
  implicit none

  BEGIN_DOC
!  Computes an array of fact(n)
  END_DOC

  integer :: i
  fact(0) = 1.d0
  do i=1,fact_max
    fact(i) = fact(i-1)*dble(i)
  end do
END_PROVIDER
```

When `irpf90` runs, a warning will be printed if the documentation block is absent. A file named `irpf90_entities` is created, where each line corresponds to one IRP entity and gives:

- the name of the file in which it is defined
- the Fortran type
- the name of the IRP entity
- the dimensions if the entity is an array

```
input.irp.f    : integer                        :: d1
input.irp.f    : integer                        :: d3
input.irp.f    : integer                        :: d5
fact.irp.f     : integer                        :: fact_max
uvwt.irp.f     : integer                        :: t
uvwt.irp.f     : integer                        :: u1
uvwt.irp.f     : integer                        :: u2
uvwt.irp.f     : integer                        :: w
```

This file is very useful for scripting. For instance,

```
$ # Get the file in which fact_max is defined
$ awk '/:: fact_max/ { print $1 }' irpf90_entities
fact.irp.f

$ # Get the names of all double precision IRP entities
$ INTS=$(awk '/integer  / { print $5 }' irpf90_entities)
$ echo $INTS
d1 d2 d3 d4 d5 fact_max t u1 u2 v w
```

Another very useful tool is the `irpman` command:

```
$ irpman <irp_entity>
```

This opens a man page for the desired IRP entity containing its description (given in the `BEGIN_DOC ... END_DOC` blocks), the file in which it is defined, which other entities are needed to build it, and which other entities need the current entity. It also gives an *Instability factor*, which is an estimate measure of how dangerous it can be to modify the IRP entity.

Here is the man page displayed for the `v` entity:

```
IRPF90 entities(l)          v         IRPF90 entities(l)

Declaration
       integer   :: v

Description
       v(x) = x+y+2

File
       uvwt.irp.f

Needs
       u2
       w

Needed by
       t

Instability factor
       25.0 %

IRPF90 entities             v         IRPF90 entities(l)
```

To activate tab completion in Bash, you can source the `irpman` exectuable itself

```
$ source $(which irpman)
```

Now, pressing tab on the command line after irpman gives the list of all IRP entities:

```
$ irpman <TAB><TAB>
d1            d4            fact_max      irp_example2  u2
d2            d5            fu            t             v
d3            fact          irp_example1  u1            w

$ irpman fa<TAB>
$ irpman fact<TAB><TAB>
fact      fact_max
```

Helping features

Assertions

Assertions are boolean expressions that must be true, to check the runtime behavior of the program. Assertions can be introduced with `ASSERT` keyword:

```
BEGIN_PROVIDER [ integer, u2 ]
  call compute_u(d3,d4,u2)
  ASSERT (u2 < d3)
END_PROVIDER
```

In this particular example, if `u2 < u3` nothing happens. If `u2 >= u3` , then the program fails:

```
Stack trace:            0
-------------------------
provide_t
provide_v
provide_u2
u2
-------------------------
u2: Assert failed:
 file: uvwt.irp.f, line: 23
(u2 < d3)
u2 =            8
d3 =            3

STOP 1
```

Assertions are activated by using `irpf90 -a` . If the `-a` option is not present, all the assertions are discarded.

Templates

Templates is a very useful feature of many languages. IRPF90 provides a simple way to write templates to generate similar providers and functions. The template is defined in the `BEGIN_TEMPLATE ... END_TEMPLATE` block. The first section of the block contains the template code, in which *template variables* are used prefixed with a dollar sign. Then the `SUBST` keyword defines the template variables to substitute, and multiple *substitution definition* lines are given. The substitution definitions are separated by two semi-colons (`;;`), and within a substitution definition the variable substitutions are separated by one semi-colon (`;`).

```
BEGIN_TEMPLATE

  BEGIN_PROVIDER [ $type , $name ]
   call find_in_input('$name', $name)
  END_PROVIDER

  logical function $name_is_zero()
    $name_is_zero = ($name == 0)
  end function

SUBST [ type, name ]

  integer    ;   size_tab1 ;;
  integer    ;   size_tab2 ;;
  real       ;   distance  ;;
  real       ;   x         ;;
  real       ;   y         ;;
  real       ;   z         ;;

END_TEMPLATE
```

In this example, `type` and `name` are the template variables, referenced as `$type` and `$name` in the first block. Six providers and functions will be generated :

- replacing `$type` with `integer` and `name` with `size_tab1`
- replacing `$type` with `integer` and `name` with `size_tab2`
- replacing `$type` with `real` and `name` with `distance`

- replacing `$type` with `real` and `name` with `x`
- replacing `$type` with `real` and `name` with `y`
- replacing `$type` with `real` and `name` with `z`

Augmented assignment operators

These patterns are very frequent in scientific applications:

- `a = a + b`
- `a = a * b`

If `a` has a very explicit name, this pattern can give:

```
my_very_explicit_name(dim1,dim2,dim3) =  my_very_explicit_name(dim1
  + b*c - d
```

Such constructs are not optimal:

- The name of the variable is long, so the line has to be split and the code is less readable
- The programmer is likely to make a typo by typing twice a very long variable name. This is likely to be caught by the compiler.
- When the programmer modifies a dimension in the left member, he has to modify it accordingly in the right member. Such errors will not be caught by the compiler.

Augmented assignment operators cure these problems by allowing the programmer to write:

```
my_very_explicit_name(dim1,dim2,dim3) +=  b*c - d
```

IRPF90 introduces three operators: `+=` , `-=` , and `*=` . Divisions could not be added since `/=` already means "not equal". To divide using an augmented assignment operator, `*= 1. /` can be used to multiply by the inverse.

Embedded shell scripts

When a programmer writes code, the input comes from the keyboard. With IRPF90 it is possible to define sections where the input is not the keyboard but it comes from the output of script that will be executed at compile time. This is achieved with `BEGIN_SHELL ... END_SHELL` blocks. Any scripting language can be used.

This example will use Bash to generate code that will print the date when the program was compiled:

```
program test
  BEGIN_SHELL [ /bin/bash ]
cat << EOF | sed 's/\(.*\)/echo "\1\"/g'
    print *, 'Compiled by `whoami` on `date`'
    print *, '$PWD'
    print *, '$(hostname)'
EOF
  END_SHELL
end
```

```
$ ./test
 Compiled by scemama on Wed Feb 4 22:27:46 CET 2015
 /tmp/irpf90_test
 laptop
```

Another example generates 100 functions with Python:

```
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
```

Conditional compilation

In IRPF90, the C preprocessor can't really be used, as the produced Fortran files may not have everything in the same order as the `*.irp.f` files. Instead, IRPF90 provides the `IRP_IF ... IRP_ELSE ... IRP_ENDIF` keywords to enable conditional compilation.

```
IRP_IF new_feature

  print *, 'New feature'
  call new_feature()

IRP_ELSE

  print *, 'Old feature'
  call old_feature()

IRP_ENDIF
```

To generate the program with the old feature, just run `irpf90` as usual. If you want to activate the new feature instead, use `irpf90 -Dnew_feature`. Multiple `-D` options can be given in the command line

Integration in Vim

When running `irpf90`, two files are created for the interaction with `vim`:

- the `$HOME/.vim/syntax/irpf90.vim` file
- a `tags` file in the current directory

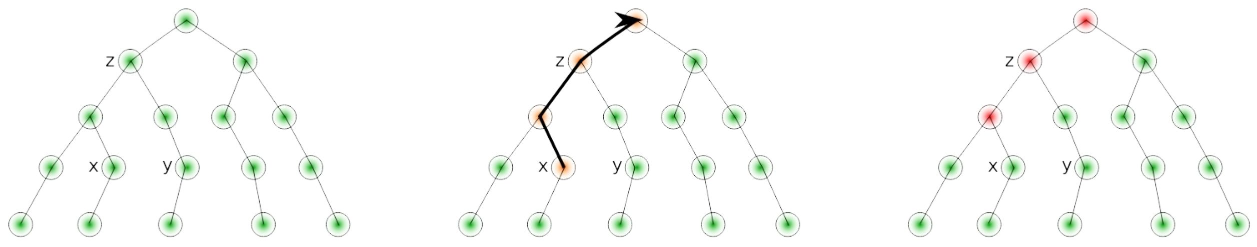
The first file is a syntax file for syntax highlighting. It extends the standard Fortran file to color the additional keywords of IRPF90. It also adds two features : hitting `K` when the cursor is on the name of an IRP entity displays its man page, and hitting `=` on a group lines selected with `<Shift>-V` auto-indents the code. However, auto-indentation is to be used outside of `BEGIN_SHELL ... END_SHELL` blocks, especially for embedded Python scripts.

The `tags` file is similar to the file created with the `ctags` utility when programming in C. The presence of this file allows `vim` to jump automatically on the definitions of providers, functions and subroutines. For instance, inside `vim`, `:tag u1` jumps to the provider of `u1`. Another option is to place your cursor on an IRP entity somewhere where it is used and hit `<CTRL>-]` to jump on its definition. To come back where you were, hit `<CTRL>-T`.

Modifying entities outside of providers

Touch

IRPF90 guarantees that an entity will not be re-provided if its value is already computed and valid. If the value of an entity is modified by a side effect outside of its provider, all the entities which are parent of the modified entity should be invalidated. This is done using the `TOUCH` keyword.



(a) (b) (c)

In this example, all the entities have been provided (figure (a)). If the user requests the value of `z` , it will be fetched from the memory. Then, the value of `x` is modified by a side effect, so the tree is not valid any more. Using `TOUCH x` (figure (b)), all the parents are invalidated, but the value of `x` is set as valid (figure (c)). Requesting `z` now will give the correct value of `z` taking account of the modification of `x` and re-providing only what is necessary between `x` and `z` .

Iterative processes

An iterative process at iteration `n` is characterized by : `A(n+1)` needs `C(n)` needs `B(n)` needs `A(n)` needs `C(n-1)` etc...

As in IRPF90 the dependence tree is static, we will want to write the iterative process as: `A` needs `B` needs `C` needs `A` ... But this implies a cyclic dependency. In that case, we will violate the rule that an entity is created only by one provider, and we will allow `C` to be modified from outside of its provider, such that it only depends on `C0` , its initial guess value. The provider of `C` will then only describe its initialization by copying its initial guess value `C0` .

```
BEGIN_PROVIDER [ double precision, C ]
  C = C0
END_PROVIDER
```

The converged value `A_converged` of `A` will be written as follows:

```
BEGIN_PROVIDER [ double precision, A_converged ]
  logical :: iteration_condition
  do while (iteration_condition)
    ! Modify the value of C depending on
    ! the current value of A
    C = f(A)
    TOUCH C
  enddo
  A_converged = A
END_PROVIDER
```

Soft touch

When IRPF90 encounters the `TOUCH` keyword, it produces the following Fortran code:

```
! >>> TOUCH x
 call touch_x
! <<< END TOUCH
  if (.not.a_is_built) then
    call provide_a
  endif
  if (.not.b_is_built) then
    call provide_b
  endif
```

After a `TOUCH` statement, all the entities in the current variable scope are provided again to ensure that the program will be correct. This can sometimes lead to providing entities that will not be needed later, especially in the cases where the `TOUCH` statement is the last statement of a provider.

The `SOFT_TOUCH` statement has the same effect as `TOUCH`, except that the entities in the current scope are not re-provided. It can be used as an optimization of the `TOUCH` when all the other entities are no more used in the current subroutine, function or provider. However, it has to be used with caution.

Debugging

Displaying the exploration of the tree

New users of IRPF90 who are experienced Fortran programmers like to display the exploration of the tree when they execute their first programs. The `-d` option of `irpf90` prints a message when the program enters or exits a function/provider/subroutine.

In the `uvwt` example, the output is:

```
$ ./irp_example1
 d1
1
 d2
2
 d3
3
 d4
4
 d5
5
 t =           26
```

Activating the `-d` option gives the following output:

```
$ ./irp_example1
0 : -> provide_t
0 :  -> provide_u1
0 :   -> provide_d1
0 :    -> d1
 d1
1
 d2
2
 d3
```

```
3
 d4
4
 d5
5
0 :     <- d1  6.889999999999999E-004
0 :    <- provide_d1  8.070000000000000E-004
0 :    -> u1
0 :     -> fu
0 :     <- fu  9.999999999996990E-007
0 :    <- u1  2.900000000000038E-005
0 :   <- provide_u1  8.999999999999998E-004
0 :   -> provide_v
0 :    -> provide_w
0 :     -> w
0 :     <- w  1.000000000000133E-006
0 :    <- provide_w  2.700000000000011E-005
0 :    -> provide_u2
0 :     -> u2
0 :      -> fu
0 :      <- fu  1.000000000000133E-006
0 :     <- u2  2.015000000000000E-003
0 :    <- provide_u2  2.026000000000000E-003
0 :    -> v
0 :    <- v  0.000000000000000E+000
0 :   <- provide_v  2.089000000000000E-003
0 :   -> t
0 :   <- t  0.000000000000000E+000
0 : <- provide_t  3.033000000000000E-003
0 : -> irp_example1
 t =           26
0 : <- irp_example1  0.000000000000000E+000
```

The floating point numbers given in the output are the CPU times, and the integer on the left of each line is the thread ID.

Compiler errors

When the Fortran compiler fails, it reports an error in the Fortran code. This error is difficult for us to track because IRPF90 generated the Fortran and we need to be able to do the mapping from the Fortran compiler's error to the error in the `*.irp.f` file. To achieve this goal, the generated Fortran code has comments at the end of the lines which correspond to the file names and line numbers of the original `*.irp.f` file.

Let us introduce an error in the IRPF90 code (a missing closing parenthesis)

```
BEGIN_PROVIDER [ integer, u2 ]
  implicit none
  BEGIN_DOC
! This is u2 = u(d3,d4)
  END_DOC
  integer :: fu
  u2 = fu(d3,d4
END_PROVIDER
```

The generated Fortran code in the `IRPF90_temp/uvwt.irp.F90` file is

```
subroutine bld_u2
  use uvwt_mod
  use input_mod
  implicit none                      ! uvwt.irp.f:  35
  character*(2) :: irp_here = 'u2'   ! uvwt.irp.f:  34
  integer :: fu                      ! uvwt.irp.f:  39
  u2 = fu(d3,d4                      ! uvwt.irp.f:  40
end subroutine bld_u2
```

Running `make` produces this error (with the Intel Fortran compiler)

```
ifort -I IRPF90_temp/  -O2  -c IRPF90_temp/uvwt.irp.F90 -o IRPF90_t
IRPF90_temp/uvwt.irp.F90(71): error #5082: Syntax error, found END-
  u2 = fu(d3,d4                      ! uvwt.irp.f:  40
----------------------------------------------------^
compilation aborted for IRPF90_temp/uvwt.irp.F90 (code 1)
make: *** [IRPF90_temp/uvwt.irp.o] Error 1
```

IRP_here

You can remark the presence of the `irp_here` variable in the generated `bld_u2` generated subroutine. Every subroutine, function or provider has a string local variable named `irp_here`, which contains the name of the current context. This variable is very helpful for users to print debug/error messages:

```
print *, irp_here//' : a = ', a
```

Tracing memory allocations

When the memory used by a program becomes too large, one would like to find the IRP entities that may be responsible. The `-m` option will display a message in the standard output when an array for an IRP entity is allocated or deallocated (using the `FREE` keyword).

Here is a real-world example (taken from the Quantum package IRPF90 code):

```
          10 Allocating ci_electronic_energy(N_states_diag)
          10 Allocating ci_eigenvectors(N_det,N_states_diag)
          10 Allocating ci_eigenvectors_s2(N_states_diag)
      128260 Allocating psi_det(N_int,2,psi_det_size)
...
 Deallocating ci_eigenvectors
       30600 Allocating ci_eigenvectors(N_det,N_states_diag)
        6120 Allocating det_connections(N_con_int,N_det)
```

The integer at the beginning of the line is the number of elements in the array.

Debugging an embedded script

Embedded shell scripts may be difficult to debug. The `--preprocess` option helps the programmer to check the files IRPF90 will produce after executing the scripts. For example, consider the file named `test.irp.f` :

```
program test
  BEGIN_SHELL [ /bin/bash ]
cat << EOF
    print *, 'Compiled by `whoami` on `date`'
    print *, '$PWD'
    print *, '$(hostname)'
EOF
  END_SHELL
end
```

The following command displays the produced Fortran file:

```
$ irpf90 --preprocess test.irp.f
program irp_program
 call test
 call irp_finalize_1624498827()
end program
subroutine test
  character*(4) :: irp_here = 'test'
    print *, 'Compiled by scemama on Mon Mar  9 18:41:50 CET 2015'
    print *, '/dev/shm/tmp'
    print *, 'lpqlx139'
end
```

Debugging `TOUCH` statements

`TOUCH` statements are particularly dangerous because they violate the principle that one IRP entity can only be built by its builder, which can only be called by its provider. To see what will be invalidated by a `TOUCH` statement can be useful to understand the consequences of a dangerous modification. The `-t` option displays which IRP entities will be invalidated :

```
$ irpf90 -t psi_coef
Touching psi_coef invalidates the following entities:
- ci_electronic_energy
- ci_energy
- coef_hf_selector
- exc_degree_per_selectors
- n_det_generators
- n_det_selectors
- one_body_dm_mo
- psi_average_norm_contrib
- psi_det_sorted
- psi_generators
- psi_selectors
- s2_values
```

Makefile configuration

User configuration

When `irpf90 --init` is run, a standard Makefile is created:

```
IRPF90 = irpf90  #-a -d
FC     = gfortran -ffree-line-length-none
FCFLAGS= -O2

SRC=
OBJ=
LIB=

include irpf90.make

irpf90.make: $(filter-out IRPF90_temp/%, $(wildcard */*.irp.f)) \
             $(wildcard *.irp.f) $(wildcard *.inc.f) Makefile
	$(IRPF90)
```

The `IRPF90` variable contains the IRPF90 invocation. All the options of IRPF90 should be given in this line. To include some additional directories to be read by IRPF90, the `-I` option should be used.

The `FC` variable describes the Fortran compiler to use. As IRPF90 adds comments at the end of the lines to express the correspondence between the Fortran generated files and the IRPF90 source files, the lines are too long for the default options of gfortran. This explains why `-ffree-line-length-none` is inserted by default. If the Intel Fortran compiler is used, this option should be removed. The `FC` variable should contain the invocation of the Fortran compiler which is common to compiling Fortran files and to link the project. For example, the `-openmp` option of the Intel compiler should be placed on this line as it should be mentioned to compile the Fortran files, but it is also required at the link stage.

`FCFLAGS` contains the flags that should be present at compile time but not at link time. The code optimization options should appear here.

It is possible to add some files to the project that will not be seen by IRPF90, but need to be compiled and linked to the project. For example, a Fortran file containing a "black box" subroutine could be taken from another project, and your IRPF90 program could call this subroutine. To do that, the names of these Fortran source files should be added to the `SRC` variable, and the names of the corresponding object files to the `OBJ` variable. Note that it is also possible to add some C source files and objects, but then the corresponding compiling rules would have to be added to the `Makefile` .

External libraries may be added to the `LIB` variable.

Auto-generated configuration

The targets are generated by IRPF90 and are written in the `irpf90.make` file. The name of each main program is a target. The `all` target builds all the programs.

The `clean` target removes all the object files. The `veryclean` target removes also the `IRPF90_temp/` and `IRPF90_man/` directories as well as the `irpf90.make` , `irpf90_entities` and `tags` files.

If the `USE_IRPF90_A` environment variable is set, then an archive `irpf90.a` containing everything except the main functions is created. This archive is linked with every target program. If this environment variable is not set, all object files are linked to create the executables.

IRPF90 creates Fortran modules that will store the cached data. These modules only contain data but no subroutines nor functions. This makes them very fast to compile. All the subroutines and functions are written in other files, and they use the modules. In this way, all the files containing subroutines and functions depend only on module files. The greatest advantage is that all of the files containing subroutines and functions, which are the longest to compile, are independent of each other. This makes the parallel builds very efficient using `make -j <n>` where `<n>` is the number of threads. Another advantage is that the `irpf90.a` library can be linked and used in any other program *without* requiring any `.mod` file. Module `.mod` files are to be avoided in libraries as they are compiler-specific (Intel Fortran `.mod` files are not compatible with gfortran and *vice versa*).

HPC-oriented functionalities

Profiling

IRPF90 includes a profiler that will measure the number of CPU cycles spent in all the providers. At the end of the run, it will display for each entity the total number of cycles, the average number of cycles (with an error bar), the total time and the average time for each entity.

Here is an example taken from a real application:

```
                         N.Calls       Tot Cycles         Avg Cycl
-------------------------------------------------------------------
...
ci_energy                     6.      1662765.       277127.+/-  5904
coef_hf_selector              7.     13009101.      1858443.+/- 60566
davidson_criterion            1.         1736.         1736.+/-
davidson_sze_max              1.           18.           18.+/-
det_connections               1.      6945057.      6945057.+/-
diag_algorithm                6.        15253.         2542.+/-    24
do_pt2_end                    1.       233928.       233928.+/-
elec_alpha_num                1.       751170.       751170.+/-
exc_degree_per_selectors      7.       209402.        29915.+/-  1082
expected_s2                   1.       240961.       240961.+/-
ezfio_filename                1.       386883.       386883.+/-
...
```

Codelet generation

When optimizing for performance, it is common to write a simple codelet that will just benchmark one provider. IRPF90 can write this codelet for you:

```
$ irpf90 --codelet <NAME>[:<PRECONDITION>]:<NMAX>
```

- `NAME` : Name of the IRP entity whose provider is to test
- `PRECONDITION` : A space-separated list of other entities to provide before running the benchmark
- `NMAX` : Number of repetitions to improve the accuracy.

Here is an example of the `uvwt` example.

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

This will generate the `codelet_v.irp.f` in which `t` is provided before the benchmark is run, and `v` will be built 100000 times:

```
program codelet_v
  implicit none
  integer :: i
  double precision :: ticks_0, ticks_1, cpu_0, cpu_1
  integer, parameter :: irp_imax = 100000

  PROVIDE t
  call provide_v
  double precision :: irp_rdtsc

  call cpu_time(cpu_0)
  ticks_0 = irp_rdtsc()
  do i=1,irp_imax
    call bld_v
  enddo
  ticks_1 = irp_rdtsc()
  call cpu_time(cpu_1)
  print *, 'v'
  print *, '-----------'
  print *, 'Cycles:'
  print *,  (ticks_1-ticks_0)/dble(irp_imax)
  print *, 'Seconds:'
  print *,  (cpu_1-cpu_0)/dble(irp_imax)
end
```

Now a new main program has been generated, it can be built using `make`. When the run is finished, the number of CPU cycles and the time in seconds is given for one execution of the provider:

```
Cycles:
  17.6698700000000
Seconds:
 7.740000000000000E-009
```

Optimizing branches

IRPF90 tries to provide the entities as soon as possible to avoid putting unnecessary `if` statements inside the loops.

When a branch occurs with an `if` condition, if an entity is needed in *all* the branches it can safely be provided before the `if` statement. This IRPF90 code

```
if (condition) then
  print *, 'True', A
else
  print *, 'False', A
endif
```

generates the Fortran code

```
if (.not. a_is_built) then
  call provide_a
endif
...
if (condition) then
  print *, 'True', A
else
  print *, 'False', A
endif
```

If the IRP entity is not needed in all branches, it will be provided only inside those branches. The IRPF90 code

```
if (condition) then
  print *, 'True'
else
  print *, 'False', A
endif
```

generates the Fortran code

```
if (condition) then
  print *, 'True'
else
  if (.not. a_is_built) then
    call provide_a
  endif
  print *, 'False', A
endif
```

This can be avoided by using the `PROVIDE` statement before entering in the `if` statement.

```
PROVIDE A
if (condition) then
  print *, 'True'
else
  print *, 'False', A
endif
```

generates the Fortran code

```
if (.not. a_is_built) then
  call provide_a
endif
...
! PROVIDE A
if (condition) then
  print *, 'True'
else
  print *, 'False', A
endif
```

This behavior can generate inefficient code if there is an `if` statement inside a loop with some entities provided not in all the branches. A command-line option `--checkopt` will check where there are entities provided inside loops, and print messages as:

```
Optimization: test.irp.f line 16
  PROVIDE a
```

Array alignment

Array alignment is necessary to get performance on x86 architectures. Indeed, vector instructions (SSE,AVX,AVX-512) require the data to be aligned on a 16-, 32- or 64-byte boundary. With the Intel compiler, it is possible to give the compiler a directive to align an array on a given boundary:

```
!DIR$ ATTRIBUTES ALIGN : 32 :: X
```

Doing this will force the first element of array `X` to have an address which is a multiple of 256 bits. Using aligned arrays for one-dimensional array will remove the peeling loops produced by the compiler when producing and auto-vectorized binary.

For two-dimensional arrays, it is possible to have all columns aligned if the array is aligned and the length of a column is a multiple of the alignment.

IRPF90 can set the alignment directive for all the IRP entities that are arrays using a command-line argument:

```
irpf90 --align=32
```

will use a 32 byte alignment for every array entity, but it will also replace in the code all the `$IRP_ALIGN` patterns with `32`. In this way, it is possible to make a code which is valid for all kind of array alignments.

Let's create a function that will calculate the length of the leading dimension such that it is a multiple of the alignment:

```
integer function align_double(i)
  implicit none
  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
```

We can now create a matrix with all columns aligned, using the `!DIR$ VECTOR ALIGNED` directive safely.

```
 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) ]
  implcit none
  integer :: i,j
  do j=1,n
   !DIR$ VECTOR ALIGNED
   do i=1,n_aligned
     ! do stuff to create Matrix(i,j)
   enddo
  enddo
END_PROVIDER
```

Variable substitution

It is possible to create a binary executable specifically tuned for one input file. The option `--substitute` replaces the variables present in loop ranges and `if` conditions by those given in the command line. Doing this gives much more information to the Fortran compiler and typically up 5-10% of performance can be gained with such a strategy.

For example, consider this piece of code:

```
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
```

We can replace the variables `lmax` , `nmax` and `choice1` by their input value using

```
$ irpf90 -s lmax:100 -s nmax:48 -s choice1:.True.
```

This will generate the following fortran code:

```
if (.True.) then
  !DIR$ VECTOR ALIGNED
  do i=1,100
    call do_stuff
  enddo
else
  !DIR$ VECTOR ALIGNED
  do i=1,48
    call do_something_else
  enddo
endif
```

The `if (.True.)` statement can be interperted by the Fortran compiler. It will then remove the `else` branch that will never be taken, and remove the `if` test. For the loop which will run, the compiler knows exactly how many loop cycles will be performed, and it can take the right decisions for loop unrolling and vectorization strategies.

Inlining providers

For each IRP entity, a provider and a builder function are created. The provider always calls the builder. The `--inline builders` forces to inline the builders in the providers.

When an IRP entity `A` is used, the following code is generated

```
if (.not.a_is_built) then
  call provide_a
endif
```

If the `--inline providers` option is present, there will be a directive in the generated Fortran code to force the inlining of the `call provide` statement.

To inline both providers and builders, use `ifpr90 --inline all`.

OpenMP

OpenMP is straightforward to use with IRPF90 for simple loops. Trouble may arrive when entities are provided in OpenMP blocks such that two threads may be providing the same entities simultaneously.

To avoid such situations, an error message is displayed if an entity is not provided by thread zero. A common solution to this problem is to explicitly provide the needed entities before entering in the OpenMP section.

Another possibility is to use `irpf90 --openmp`. In that case, all the providers become automatically thread-safe using one OpenMP lock per IRP entity.

Coarray Fortran support

When the `--coarray` option is given, all the entities are co-arrays in the CoArray Fortran (CAF) language extension, defined as `[*]`. Therefore, it is possible for any process image to access the IRP entities of all the other images.

Let us first create convenient providers to cache the values of the `num_images` and `image_id` functions that will be called very often.

```
 BEGIN_PROVIDER [ integer, n_images ]
&BEGIN_PROVIDER [ integer, image_id ]
 implicit none
 BEGIN_DOC
 ! CAF internals
 END_DOC
 n_images = num_images()
 image_id = this_image()
END_PROVIDER
```

Now, we create an array that will be different on each image:

```
BEGIN_PROVIDER [ real, X, (10) ]
 implicit none
 BEGIN_DOC
 ! X(i) = image_id x i
 END_DOC
 integer  :: i
 do i=1,size(X)
   X(i) = real(image_id * i)
 enddo
END_PROVIDER
```

In the main program, you will want to print the value of `X` of images 1 and 2. Only the first image will print, so this will imply an `if` statement as

```
if (this_image() == 1) then
  print *, X
endif
```

The problem is that `X` will need to be provided *only* if the `image_id` is equal to one. Here, we will have to force to provide `X`, whatever the value of `this_image`.

```
program caf_test
 implicit none

 PROVIDE X
 if (image_id == 1) then
   print *, 'This image:'
   print *, X
   print *, 'Image 2:'
   print *, X[2]
 endif
end
```

In the `Makefile`, set

```
IRPF90 = irpf90 --coarray
FC      = ifort -coarray
```

Build the program and the output will give:

```
$ ./caf_main
 This image:
   1.000000       2.000000       3.000000       4.000000       5.00
   6.000000       7.000000       8.000000       9.000000       10.0
 Image 2:
   2.000000       4.000000       6.000000       8.000000       10.0
   12.00000       14.00000       16.00000       18.00000       20.0
```

Tutorial : A molecular dynamics code

Molecular dynamics models the movement of atoms according to their initial positions and velocities. In this tutorial, we will write a molecular dynamics program to illustrate how to use IRPF90. This program will read the force field parameters from an input file, as well as the initial positions of the atoms. After each little displacement of the atoms according to their velocities, the new set of coordinates will be printed into an output file such that a video animation can easily be produced with an external tool.

Here is the list of what we will have to code:

- The potential energy of a couple of atoms (Lennard-Jones potential). This will will be a very simple introduction to IRPF90.
- The potential and kinetic energy of system of *N* atoms. We will have to create arrays dimensioned by other IRP entities.
- The acceleration of the particles using finite differences for the calculation of derivatives. This part will introduce the `TOUCH` keyword.
- The Verlet algorithm to make everything move.

The first thing you will have to do is download IRPF90 from the web site: http://irpf90.ups-tlse.fr

Physical Parameters

For all this tutorial, we will use Argon atoms with the following parameters:

- mass : 39.948 g/mol
- epsilon : 0.0661 j/mol
- sigma : 0.3345 nm

The atom coordinates are given in nanometers.

Prepare the working environment

Create a new directory for the project. Inside this directory, initialize the IRPF90 environment using:

```
$ irpf90 --init
```

Two directories were created

```
$ ls
IRPF90_man  IRPF90_temp  Makefile
```

and a Makefile containing default parameters for the gfortran compiler

```
IRPF90 = irpf90  #-a -d
FC     = gfortran
FCFLAGS= -O2 -ffree-line-length-none

SRC=
OBJ=
LIB=

include irpf90.make

irpf90.make: $(filter-out IRPF90_temp/%, $(wildcard */*.irp.f)) $(w
        $(IRPF90)
```

In the Makefile, activate the asserts and the debug options by uncommenting `-a` and `-d` in the definition of the `IRPF90` variable.

The Lennard-Jones potential

Exercise

Write a program which computes the Lennard-Jones potential :

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

The user will be asked for the values of the Lennard-Jones parameters `sigma_lj` and `epsilon_lj`, as well as the `interatomic_distance`.

Create the main program in a file named `test.irp.f`, and the providers in a file named `potential.irp.f`. You don't need to modify the `Makefile`.

To compile the program, run

```
$ make
Makefile:9: irpf90.make: No such file or directory
irpf90  -a -d
IRPF90_temp/potential.irp.module.F90
IRPF90_temp/potential.irp.F90
IRPF90_temp/test.irp.module.F90
IRPF90_temp/test.irp.F90
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_ter
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_ter
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_ter
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_ter
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_ter
gfortran -I IRPF90_temp/  -O2 -ffree-line-length-none -c IRPF90_ter
gfortran -I IRPF90_temp/  -o test IRPF90_temp/test.irp.o IRPF90_ter
```

The warning `Makefile:9: irpf90.make: No such file or directory` can be ignored: the missing `irpf90.make` will be created by applying the rule in the `Makefile` that calls IRPF90.

A binary file named `test` will be created.

```
$ ls
irpf90_entities  IRPF90_man   Makefile        tags  test.irp.f
irpf90.make      IRPF90_temp  potential.irp.f  test
```

Expected Output

```
$ ./test
           0 : -> provide_v_lj
           0 :  -> provide_epsilon_lj
           0 :   -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
0.3345
           0 :   <- 0="" epsilon_lj="" 3.63000000000000041E-004=""
           0 :   -> interatomic_distance
 Inter-atomic distance?
0.3
           0 :   <- 0="" interatomic_distance="" 1.7049999999999999
           0 :  <- 0="" v_lj="" 0.0000000000000000="" :="" <-="" pr
  0.46819241808782769
           0 :
```

Solution

File `test.irp.f`

```
program test
  implicit none
  BEGIN_DOC
  ! Test program
  END_DOC
  print *,  V_lj
end
```

File `potential.irp.f`

```
BEGIN_PROVIDER [ double precision, V_lj ]
  implicit none
  BEGIN_DOC
  ! Lennard Jones potential energy.
  END_DOC
  double precision               :: sigma_over_r
  sigma_over_r = sigma_lj / interatomic_distance
  V_lj = 4.d0 * epsilon_lj * ( sigma_over_r**12 - sigma_over_r**6
END_PROVIDER

 BEGIN_PROVIDER [ double precision, epsilon_lj ]
&BEGIN_PROVIDER [ double precision, sigma_lj   ]
  implicit none
  BEGIN_DOC
  ! Parameters of the Lennard-Jones potential
  END_DOC
  print *, 'Epsilon?'
  read(*,*) epsilon_lj
  ASSERT (epsilon_lj > 0.)

  print *, 'Sigma?'
  read(*,*) sigma_lj
  ASSERT (sigma_lj > 0.)
END_PROVIDER

BEGIN_PROVIDER [ double precision, interatomic_distance ]
  implicit none
  BEGIN_DOC
  ! Distance between the atoms
  END_DOC
  print *, 'Inter-atomic distance?'
  read (*,*) interatomic_distance
  ASSERT (interatomic_distance >= 0.)
END_PROVIDER
```

Describing the atoms

Exercise

In the same directory, create a program which reads in the standard input:

- The number of atoms
- For each atom: the mass and the *x*, *y*, *z* coordinates

The program will the print the matrix of the distances between atom pairs.

You will have to create :

- A provider for `Natoms`, the number of atoms
- A provider for `coord` and `mass`, the atom coordinates and mass. These are arrays with dimensions `(3,Natoms)` and `(Natoms)` respectively.
- A provider for `distance`, the distance matrix. Its dimension is `(Natoms,Natoms)`.

You can check that your code is well documented using the `irpman` command:

```
$ irpman coord
IRPF90 entities(l)                   coord                  IRPF9

Declaration
       double precision, allocatable :: coord  (3,Natoms)
       double precision, allocatable :: mass   (Natoms)

Description
       Atomic data, input in atomic units.

File
       atoms.irp.f

Needs
       natoms

Needed by
       distance

Instability factor
        50.0 %

IRPF90 entities                      coord                  IRPF9
```

Expected output

```
$ ./test2
           0 : -> provide_distance
           0 :  -> provide_natoms
           0 :   -> natoms
 Number of atoms?
3
           0 :   <- 0="" natoms="" 1.59999999999999986E-004="" :=""
           0 :   -> coord
 For each atom: x, y, z, mass?
0. 0. 0. 40.
1. 2. 3. 10.
-1. 0. 2. 20.
           0 :   <- 0="" coord="" 2.03999999999999754E-004="" :=""
           0 :  <- 0="" distance="" 1.99999999999983177E-006="" :='
   0.0000000000000000        3.7416573867739413        2.23606797
   3.7416573867739413        0.0000000000000000        3.0000000000
   2.2360679774997898        3.0000000000000000        0.0000000000
           0 :
```

Solution

File `test2.irp.f`

```
program test2
  implicit none
  BEGIN_DOC
  ! Second test : distance matrix
  END_DOC
  integer                        :: i
  do i=1,Natoms
    print *, distance(1:3,i)
  enddo
end program
```

File `atoms.irp.f`

```
BEGIN_PROVIDER [ integer, Natoms ]
```

```
  implicit none
  BEGIN_DOC
  ! Number of atoms
  END_DOC
  print *, 'Number of atoms?'
  read(*,*) Natoms
  ASSERT (Natoms > 0)
END_PROVIDER

 BEGIN_PROVIDER [ double precision, coord, (3,Natoms) ]
&BEGIN_PROVIDER [ double precision, mass , (Natoms)   ]
  implicit none
  BEGIN_DOC
  ! Atomic data, input in atomic units.
  END_DOC
  print *, 'For each atom: x, y, z, mass?'
  integer                         :: i,j  ! <-- Variables can be dec
                                          !     anywhere
  do i=1,Natoms
    read(*,*) (coord(j,i), j=1,3), mass(i)
    ASSERT (mass(i) > 0.d0)
  enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, distance, (Natoms,Natoms) ]
  implicit none
  BEGIN_DOC
  ! distance : Distance matrix
  END_DOC
  integer                         :: i,j,k
  do i=1,Natoms
    do j=1,Natoms
      distance(j,i) = 0.d0
      do k=1,3
        distance(j,i) += (coord(k,i)-coord(k,j))**2  ! <-- Note the
                                                     !     operato
      enddo
      distance(j,i) = dsqrt(distance(j,i))
    enddo
  enddo
```

```
END_PROVIDER
```

Potential for multiple particles

Exercise

Change the provider of `V_lj` of the first program. Now, instead of computing the Lennard-Jones potential of a single inter-atomic distance *r*, you will compute the total potential energy which is the sum of the potential energies due to each pair of atoms:

$$V_{\rm LJ} = \sum_{i=1}^{\tt Natoms} \sum_{j>i}^{\tt Natoms} V(r_{ij})$$

The dependencies have changed now, as your new version of `V_lj` needs the previously defined distance matrix `distance`. You can now run again the first program.

Expected output

```
$ ./test
           0 : -> provide_v_lj
           0 :  -> provide_epsilon_lj
           0 :   -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
0.3345
           0 :   <- 0="" epsilon_lj="" 3.06000000000000022E-003=""
           0 :   -> natoms
 Number of atoms?
3
           0 :   <- 0="" natoms="" 0.0000000000000000="" :="" <-=""
           0 :   -> provide_coord
           0 :    -> coord
 For each atom: x, y, z, mass?
0 0 0 10
0 0 .3 20
.1 .2 -.3 15
           0 :    <- 0="" coord="" 0.0000000000000000="" :="" <-=""
           0 :   <- 0="" distance="" 0.0000000000000000="" :="" <-=
           0 :  <- 0="" v_lj="" 0.0000000000000000="" :="" <-="" pr
  0.39685690695535791
           0 :
```

Solution

File `potential.irp.f`

```
BEGIN_PROVIDER [ double precision, V ]
  implicit none
  BEGIN_DOC
  ! Potential energy.
  END_DOC
  V = V_lj
END_PROVIDER

BEGIN_PROVIDER [ double precision, V_lj ]
```

```
  implicit none
  BEGIN_DOC
  ! Lennard Jones potential energy.
  END_DOC
  integer                        :: i,j
  double precision               :: sigma_over_r
  V_lj = 0.d0
  do i=1,Natoms
    do j=i+1,Natoms
      ASSERT (distance(j,i) > 0.d0)   ! <-- Avoid a possible divisi
      sigma_over_r = sigma_lj / distance(j,i)
      V_lj += sigma_over_r**12 - sigma_over_r**6
    enddo
  enddo
  V_lj *= 4.d0 * epsilon_lj  ! <-- Note the *= operator
END_PROVIDER

 BEGIN_PROVIDER [ double precision, epsilon_lj ]
&BEGIN_PROVIDER [ double precision, sigma_lj   ]
  implicit none
  BEGIN_DOC
  ! Parameters of the Lennard-Jones potential
  END_DOC
  print *, 'Epsilon?'
  read(*,*) epsilon_lj
  ASSERT (epsilon_lj > 0.)

  print *, 'Sigma?'
  read(*,*) sigma_lj
  ASSERT (sigma_lj > 0.)
END_PROVIDER
```

Computing the total energy

Exercise

Write a program which prints the total energy of the system.

$$E_{\text{tot}} = T + V$$

`V` is the potential (Lennard-Jones here) and `T` is the kinetic energy

$$T = \frac{1}{2} \sum_{i=1}^{\texttt{Natoms}} m_i v_i^2$$

Write the providers for the kinetic energy and for the total energy. All the velocities will be chosen to be initialized equal to zero in the `velocities` provider. Remember you already have the provider for the masses of the atoms.

Expected output

```
$ ./test3
           0 : -> provide_e_tot
           0 :  -> provide_t
           0 :   -> provide_velocity2
           0 :    -> provide_velocity
           0 :     -> provide_natoms
           0 :      -> natoms
 Number of atoms?
3
           0 :      <- 0="" natoms="" 1.58999999999999853E-004=""
           0 :     <- 0="" velocity="" 5.99999999999992900E-006=""
           0 :    <- 0="" velocity2="" 5.00000000000022995E-006=""
           0 :    -> coord
 For each atom: x, y, z, mass?
0 0 0 10
0 0 .3 20
.1 .2 -.3 15
           0 :    <- 0="" coord="" 1.97999999999999825E-004="" :="
           0 :   <- 0="" t="" 9.99999999999699046E-007="" :="" <-='
           0 :   -> provide_v_lj
           0 :    -> provide_epsilon_lj
           0 :     -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
.3345
           0 :     <- 0="" epsilon_lj="" 2.07999999999999852E-004='
           0 :     -> distance
           0 :     <- 0="" distance="" 2.00000000000026545E-006=""
           0 :    <- 0="" v_lj="" 1.00000000000013273E-006="" :=""
           0 :   <- 0="" v="" 1.00000000000013273E-006="" :="" <-='
           0 :  <- 0="" e_tot="" 1.00000000000013273E-006="" :=""
  0.39685690695535791
           0 :
```

Solution

File `test3.irp.f`

```
program test3
  implicit none
  BEGIN_DOC
  ! Prints the total energy
  END_DOC
  print *,  E_tot
end program
```

File `energy.irp.f`

```
BEGIN_PROVIDER [ double precision, E_tot ]
  implicit none
  BEGIN_DOC
  ! Total energy of the system
  END_DOC
  E_tot = T + V
END_PROVIDER
```

File `velocity.irp.f`

```
BEGIN_PROVIDER [ double precision, T ]
  implicit none
  BEGIN_DOC
  ! Kinetic energy per atom
  END_DOC
  T = 0.d0
  integer                        :: i
  do i=1,Natoms
    T += mass(i) * velocity2(i)
  enddo
  T *= 0.5d0
END_PROVIDER

BEGIN_PROVIDER [ double precision, velocity2, (Natoms) ]
  implicit none
  BEGIN_DOC
  ! Square of the norm of the velocity per atom
  END_DOC
  integer                        :: i, k
  do i=1,Natoms
    velocity2(i) = 0.d0
    do k=1,3
      velocity2(i) += velocity(k,i)*velocity(k,i)
    enddo
  enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, velocity, (3,Natoms) ]
  implicit none
  BEGIN_DOC
  ! Velocity vector per atom
  END_DOC
  integer                        :: i, k
  do i=1,Natoms
    do k=1,3
      velocity(k,i) = 0.d0
    enddo
  enddo
END_PROVIDER
```

Computing the acceleration

Exercise

The acceleration vector is given by

$$\{a_{x_i}\} = \left\{ -\frac{1}{m_i} \frac{\partial V}{\partial x_i} \right\}$$

where *x_i* is the *x* coordinate of atom *i* (an element of the `coord` array). Write the provider for `V_grad_numeric`, the finite-difference approximation of the derivative of the potential with respect to the coordinates:

$$\frac{\partial V}{\partial x_i} \sim \frac{V(x_i + \Delta x_i) - V(x_i - \Delta x_i)}{2\Delta x_i}$$

It will be necessary to use the `TOUCH` keyword.

The computation of the acceleration should not depend directly on the method used to compute the gradient, so we will use `V_grad` in the provider for the `acceleration`. `V_grad` will be a simple copy of `V_grad_numeric`.

Expected output

```
$ ./test4
           0 : -> provide_acceleration
           0 :  -> provide_natoms
           0 :   -> natoms
 Number of atoms?
3
           0 :   <- 0="" natoms="" 1.68999999999999879E-004="" :=""
           0 :   -> coord
 For each atom: x, y, z, mass?
0 0 0 10
0 0 .3 20
.1 .2 -.3 15
           0 :   <- 0="" coord="" 2.26999999999999771E-004="" :=""
           0 :   -> provide_v_grad_numeric
           0 :    -> provide_v
           0 :     -> provide_v_lj
           0 :      -> provide_epsilon_lj
           0 :       -> epsilon_lj
 Epsilon?
0.0661
 Sigma?
.3345
           0 :       <- 0="" epsilon_lj="" 1.60999999999999685E-00
           0 :       -> distance
           0 :       <- 0="" distance="" 2.00000000000026545E-006=
           0 :      <- 0="" v_lj="" 2.00000000000026545E-006="" :=
           0 :     <- 0="" v="" 1.00000000000013273E-006="" :="" <
           0 :     -> dstep
           0 :     <- 0="" dstep="" 9.99999999999699046E-007="" :=
           0 :     -> touch_coord
           0 :     <- 0="" touch_coord="" 1.00000000000013273E-006
           0 :      -> provide_v_lj
           0 :       -> provide_distance
           0 :        -> distance
           0 :        <- 0="" distance="" 1.00000000000013273E-006
           0 :       <- 0="" v_lj="" 9.99999999999265365E-007="" :
           0 :      <- 0="" v="" 9.99999999999265365E-007="" :="" 
           0 :     <- 0="" touch_coord="" 1.00000000000013273E-006
           0 :      -> provide_v_lj
           0 :       -> provide_distance
           0 :        -> distance
           0 :        <- 0="" distance="" 1.00000000000013273E-006
           0 :       <- 0="" v_lj="" 1.00000000000013273E-006="" :
           0 :      <- 0="" v="" 1.00000000000013273E-006="" :="" 
----8<--------------------------------------------------------------
           0 :      -> provide_v_lj
           0 :       -> provide_distance
           0 :        -> distance
           0 :        <- 0="" distance="" 1.00000000000013273E-006
           0 :       <- 0="" v_lj="" 1.00000000000013273E-006="" :
           0 :      <- 0="" v="" 1.00000000000013273E-006="" :="" 
           0 :     <- 0="" touch_coord="" 1.00000000000013273E-006
           0 :   <- 0="" v_grad="" 1.00000000000013273E-006="" :="
           0 :  <- 0="" acceleration="" 1.00000000000013273E-006="
 -1.21434697006317371E-003 -2.42873782740904431E-003  -2.885248388
```

```
   3.772257075318474 76E-004   7.5445143164765 1383E-004    1.442182447
   3.0659703664781545 7E-004   6.1322330940916 1033E-004   5.88995461163
            0 :
```

Solution

File `test4.irp.f`

```
ratiot4
  implicit none
  BEGIN_DOC
  ! Program testing the acceleration
  END_DOC
  integer                        :: i
  do i=1,Natoms
    print *, acceleration(:,i)
  enddo
end program
```

File `potential.irp.f`, add

```
BEGIN_PROVIDER [ double precision, dstep ]
  implicit none
  BEGIN_DOC
  ! Finite difference step
  END_DOC
  dstep = 1.d-4
END_PROVIDER

BEGIN_PROVIDER [ double precision, V_grad_numeric, (3,Natoms) ]
  implicit none
  BEGIN_DOC
  ! Numerical gradient of the potential
  END_DOC
  integer                        :: i, k
  do i=1,Natoms
    do k=1,3
```

```
      coord(k,i) += dstep   ! Move coordinate x_i to x_i + delta
      TOUCH coord mass      ! Tell IRPF90 that coord has been chang
      V_grad_numeric(k,i) = V      ! V is here V(x_i + delta)
      coord(k,i) -= 2.d0*dstep     ! Move coordinate x_i to x_i - d
      TOUCH coord mass      ! Tell IRPF90 that coord has been chang
      V_grad_numeric(k,i) -= V      ! V is here V(x_i - delta)
      V_grad_numeric(k,i) *= .5d0/dstep
      coord(k,i) += dstep   ! Put back x_i to its initial position
                            ! It is not necessary to re-touch coord
                            ! - at the next loop iteration it will
                            ! - out of the loop, it is soft-touched
    enddo
  enddo
  SOFT_TOUCH coord mass ! Does not re-provide the current entities.
                        ! not be re-computed. This reduced the CPU
                        ! dangerous.
END_PROVIDER

BEGIN_PROVIDER [ double precision, V_grad, (3,Natoms) ]
  implicit none
  BEGIN_DOC
  ! Gradient of the potential
  END_DOC
  integer                        :: i,k
  do i=1,Natoms
    do k=1,3
      V_grad(k,i) = V_grad_numeric(k,i)
    enddo
  enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, acceleration, (3,Natoms) ]
  implicit none
  BEGIN_DOC
  ! Acceleration = - grad(V)/m
  END_DOC
  integer                        :: i, k
  do i=1,Natoms
    do k=1,3
      acceleration(k,i) = -V_grad(k,i)/mass(i)
    enddo
  enddo
END_PROVIDER
```

Implementing the molecular dynamics

Exercise

The Verlet algorithm is the following

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \mathbf{v}^n \Delta t + \mathbf{a}^n \frac{\Delta t^2}{2}$$
$$\mathbf{v}^{n+1} = \mathbf{v}^n + \frac{1}{2}(\mathbf{a}^n + \mathbf{a}^{n+1})\Delta t$$

where *n* is the index of the current step, **r** is the position vector, **v** is the velocity vector, **a** is the acceleration vector and Δt is the time step.

Write a subroutine which implements the Verlet algorithm. To do this, at each iteration :

- Compute the coordinates at step *n*+1
- Compute the component of the velocity which depends on the position at step *n*
- `TOUCH` the coordinates and the velocities
- Add to the velocities the part which depends on step *n*+1
- `TOUCH` the velocities

For this exercise, remove the debug option in the `Makefile` .

Expected output

```
$ ./test5
 Number of atoms?
3
 For each atom: x, y, z, mass?
0 0 0 40
0 0 .5 40
.1 .2 -.5 40
   0.0000000000000000        0.0000000000000000        0.0000000000
   0.0000000000000000        0.0000000000000000       0.5000000000
  0.10000000000000001       0.20000000000000001      -0.5000000000
 Epsilon?
0.0661
 Sigma?
0.3345
 Nsteps?
1000
 Time step?
0.2
 -4.85173622655117529E-002 -9.70435723126635286E-002  0.1881931839
 -1.11022166416810172E-002 -2.22085304763539326E-002  0.6206434510
  0.15961957890929021       0.31925210279233324      -0.8088366350
```

Solution

File `test5.irp.f`

```
program test5
  implicit none
  BEGIN_DOC
  ! Program testing the verlet algorithm
  END_DOC
  integer                        :: i
  do i=1,Natoms
    print *,  coord(1:3,i)
  enddo
  call verlet
  do i=1,Natoms
    print *,  coord(1:3,i)
  enddo
end
```

File `verlet.irp.f`

```
BEGIN_PROVIDER [ integer, Nsteps ]
  implicit none
  BEGIN_DOC
  ! Number of steps for the dynamics
  END_DOC
  print *, 'Nsteps?'
  read(*,*) Nsteps
  ASSERT (Nsteps > 0)
END_PROVIDER

 BEGIN_PROVIDER [ double precision, tstep ]
&BEGIN_PROVIDER [ double precision, tstep2 ]
  implicit none
  BEGIN_DOC
  ! Time step for the dynamics
  END_DOC
  print *, 'Time step?'
 ASSERT (tstep > 0.)
 tstep2 = tstep*tstep
END_PROVIDER

subroutine verlet
```

```
   implicit none
   integer :: is, i, k
   do is=1,Nsteps
    call print_data(is)      ! A de-commenter pour l'exercice suivant
    do i=1,Natoms
     do k=1,3
      coord(k,i) += tstep*velocity(k,i) + 0.5*tstep2*acceleration(k,i
      velocity(k,i) += 0.5*tstep*acceleration(k,i)
     enddo
    enddo
    TOUCH coord velocity mass
    do i=1,Natoms
     do k=1,3
      velocity(k,i) += 0.5*tstep*acceleration(k,i)
     enddo
    enddo
    TOUCH velocity
   enddo
  end subroutine
```

Using scripts to generate specialized functions

In this example we write a Python script `power.py` that will generate specialized functions to calculate the `n` -th power of `x` .

```
#!/usr/bin/python

POWER_MAX = 20

def compute_x_prod(n,d):
  if n == 0:
    d[0] = None
    return d
  if n == 1:
    d[1] = None
    return d
  if n in d:
    return d
  m = n/2
  d = compute_x_prod(m,d)
  d[n] = None
  d[2*m] = None
  return d

def print_subroutine(n):
  keys = compute_x_prod(n,{}).keys()
  keys.sort()
  output = []
  print "double precision function power_%d(x1)"%n
  print "  double precision, intent(in) :: x1"
  print "  BEGIN_DOC"
  print "!  Fast computation of x**%d"%(n)
  print "  END_DOC"
  for i in range(1,len(keys)):
     output.append( "x%d"%keys[i] )
  if output != []:
    print " double precision :: "+', '.join(output)
  for i in range(1,len(keys)):
   ki = keys[i]
   ki1 = keys[i-1]
   if ki == 2*ki1:
     print " x%d"%ki + " = x%d * x%d"%(ki1,ki1)
   else:
     print " x%d"%ki + " = x%d * x1"%(ki1)
  print " power_%d = x%d"%(n,n)
  print "end"

for i in range(POWER_MAX):
  print_subroutine (i+1)
  print ''
```

Executing this script gives

```
double precision function power_1(x1)
 double precision, intent(in) :: x1
 BEGIN_DOC
!  Fast computation of x**1
 END_DOC
 power_1 = x1
end

double precision function power_2(x1)
 double precision, intent(in) :: x1
 BEGIN_DOC
!  Fast computation of x**2
 END_DOC
 double precision :: x2
 x2 = x1 * x1
 power_2 = x2
end

double precision function power_3(x1)
 double precision, intent(in) :: x1
 BEGIN_DOC
!  Fast computation of x**3
```

```
  END_DOC
  double precision :: x2, x3
  x2 = x1 * x1
  x3 = x2 * x1
  power_3 = x3
end

...


double precision function power_17(x1)
  double precision, intent(in) :: x1
  BEGIN_DOC
!   Fast computation of x**17
  END_DOC
  double precision :: x2, x4, x8, x16, x17
  x2 = x1 * x1
  x4 = x2 * x2
  x8 = x4 * x4
  x16 = x8 * x8
  x17 = x16 * x1
  power_17 = x17
end

double precision function power_18(x1)
  double precision, intent(in) :: x1
  BEGIN_DOC
!   Fast computation of x**18
  END_DOC
  double precision :: x2, x4, x8, x9, x18
  x2 = x1 * x1
  x4 = x2 * x2
  x8 = x4 * x4
  x9 = x8 * x1
  x18 = x9 * x9
  power_18 = x18
end

double precision function power_19(x1)
  double precision, intent(in) :: x1
```

```
 BEGIN_DOC
!  Fast computation of x**19
 END_DOC
 double precision :: x2, x4, x8, x9, x18, x19
 x2 = x1 * x1
 x4 = x2 * x2
 x8 = x4 * x4
 x9 = x8 * x1
 x18 = x9 * x9
 x19 = x18 * x1
 power_19 = x19
end

double precision function power_20(x1)
 double precision, intent(in) :: x1
 BEGIN_DOC
!  Fast computation of x**20
 END_DOC
 double precision :: x2, x4, x5, x10, x20
 x2 = x1 * x1
 x4 = x2 * x2
 x5 = x4 * x1
 x10 = x5 * x5
 x20 = x10 * x10
 power_20 = x20
end
```

Then, we create a `benchmark.irp.f` file that contains provider to compute the 20 first `n` -th power of `x` using the traditional `x**n` , and another provider that will use our specialized functions

```
BEGIN_PROVIDER [ double precision, x ]
  implicit none
  BEGIN_DOC
! Value of x
  END_DOC
  x = 1.2345d0
END_PROVIDER
```

```
BEGIN_PROVIDER [ double precision, x_p, (20) ]
  implicit none
  BEGIN_DOC
! array of x**p for 0 < p < 21 with the standard power functions
  END_DOC
  integer :: i
  do i=1,20
    x_p(i) = x**i
  enddo

END_PROVIDER

! Put the power.py script here for better inlining of the functions
BEGIN_SHELL [ /usr/bin/python ]
import power
END_SHELL

BEGIN_PROVIDER [ double precision, x_p_fast, (20) ]
  implicit none
  BEGIN_DOC
! array of x**p for 0 < p < 21 with the fast power functions
  END_DOC
  BEGIN_SHELL [ /bin/bash ]
  for i in {1..20}
  do
    echo "  double precision, external :: power_$i"
    echo "  !DIR$ FORCEINLINE"
    echo "  x_p_fast($i) = power_$i(x)"
  done
  END_SHELL

END_PROVIDER
```

We now Create a codelet for the `x_p` and the `x_p_fast` providers using

```
$ irpf90 -c      x_p:100000000
$ irpf90 -c x_p_fast:100000000
$ make
```

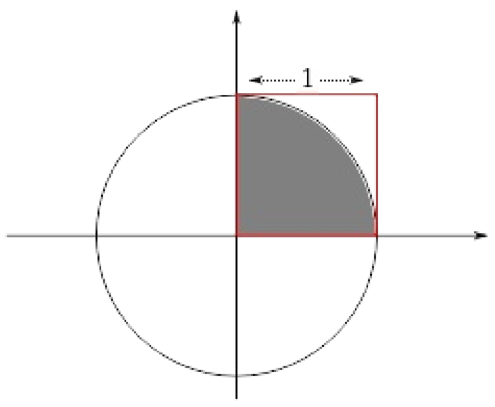
We easily see that we get a speedup of 12x with the specialized power routines:

```
$ ./codelet_x_p
 x_p
 -----------
 Cycles:
   261.97605379999999
 Seconds:
  1.13999589999999997E-007

$ ./codelet_x_p_fast
 x_p_fast
 -----------
 Cycles:
   21.707082620000001
 Seconds:
  9.47659000000000108E-009
```

CoArray Fortran example

Here, we want to calculate $$\pi$$ with a Monte-Carlo algorithm. Each image will compute its own Monte Carlo average, and the global average will be computed at the end.



The area inside a unit circle is $$\pi$$. The red square is the square containing all points with coordinates in the $$([0,1],[0,1])$$ range. The grey area represents the set of points that are in the $$([0,1],[0,1])$$ range and which are at a distance less than one to the center of the circle. For the computation of $$\pi$$ with a Monte Carlo algorithm, each sample will consist in drawing two uniform random numbers in the $$[0,1]$$ range (one for the $$x$$ coordinate and one for the $$y$$ coordinate). Is the distance of the point to the center is less than one, we increment a counter. Our estimate of $$\pi$$ will be $$4 N*{\rm inside} / N*{\rm total}$$.

Single core program

Let us first write a single core program. We write the providers in the `pi.irp.f` file. The `N_steps` entity defines the number of Monte-Carlo steps to compute the value of $$\pi$$ in a single process.

```
BEGIN_PROVIDER [ integer*8, N_steps ]
 implicit none
 BEGIN_DOC
 ! Total number of MC steps
 END_DOC
 N_steps = 0000000_8
END_PROVIDER
```

The `N_blocks` entity is the total number of independent calculations of $$\pi$$ one will do in a single process.

```
BEGIN_PROVIDER [ integer, N_blocks ]
 implicit none
 BEGIN_DOC
 ! Total number of blocks, each containing N_steps steps.
 END_DOC
 N_blocks = 100
END_PROVIDER
```

One will need to initialize the seed of the Fortran random number generator:

```
subroutine init_seed(i)
  implicit none
  integer, intent(in) :: i
  BEGIN_DOC
! Initializes the random seed with the current this_image()
  END_DOC
  integer :: seed(12)
  seed(:) = i
  call random_seed(put=seed)
end
```

`pi_block` is the Monte-Carlo evaluation of $$\pi$$ over `N_steps` .

```
BEGIN_PROVIDER [ double precision, pi_block ]
 implicit none
 BEGIN_DOC
 ! Value of pi computed over N_steps
 END_DOC

 integer                        :: i_step
 integer*8                      :: count_in
 double precision               :: x,y

 count_in = 0_8

 do i_step=1,N_steps
   call random_number(x)
   call random_number(y)
   if ( (x*x + y*y) <= 1.d0) then
     count_in += 1_8
   endif
 enddo
 pi_block = 4.d0*dble(count_in)/dble(N_steps)

END_PROVIDER
```

Let us now write the main program in `test_mono.irp.f`. It will print the running average and error bar of $$\pi$$ on the standard output. At the end of each loop cycle, the `pi_block` entity is freed, such that it will be freshly provided at the beginning of the next loop iteration.

```
program test_mono
  implicit none
  BEGIN_DOC
! Test the single core program
  END_DOC
  integer          :: i
  double precision :: pi_sum, pi_sum2, n
  double precision :: pi_average, pi_variance, pi_error

  call init_seed(1)

  pi_sum  = 0.d0
  pi_sum2 = 0.d0
  n = 0.d0
  do i=1,N_blocks
    PROVIDE pi_block
    n += 1.d0
    pi_sum  += pi_block
    pi_sum2 += pi_block*pi_block
    pi_average = pi_sum / n
    pi_variance = pi_sum2/n - pi_average**2
    pi_error = sqrt(pi_variance/(n-1.d0))
    print *,  pi_average, '+/-', pi_error
    FREE pi_block
  enddo
end
```

The output of this program gives:

```
$ ./test_mono
   3.14213880000000      +/-                      NaN
   3.14194260000000      +/-   1.961999945451743E-004
   3.14176573333333      +/-   2.100316592883950E-004
   3.14156600000000      +/-   2.488976753433852E-004
   3.14144144000000      +/-   2.295326231094694E-004
...
   3.14160352500000      +/-   5.354283981217712E-005
   3.14160270103093      +/-   5.299438256043764E-005
   3.14159646938776      +/-   5.281972755126230E-005
   3.14160685252525      +/-   5.330451270701332E-005
   3.14161012000000      +/-   5.286984052818029E-005
```

Parallel program

Now, we will write the prallel version of the program. First we will add to the `Makefile` the `--coarray` option to `irpf90` and the `-coarray` option to `ifort`.

We can now write the parallel main program. We use a temporary array that will fetch all the remote values of `pi_block`. After a synchronization directive (`SYNC ALL`), the master process can compute the running average and error bar, and print the result.

```
program test_caf
  implicit none
  BEGIN_DOC
! Test the single core program
  END_DOC

  integer          :: i,j
  double precision :: pi_sum, pi_sum2, n
  double precision :: pi_average, pi_variance, pi_error
  double precision, allocatable :: pi_block_local(:)

  allocate (pi_block_local(num_images()))

  call init_seed(11*this_image())
```

```
    pi_sum  = 0.d0
    pi_sum2 = 0.d0
    n = 0.d0
    do i=1,N_blocks
      PROVIDE pi_block
      do j=1,num_images()
        pi_block_local(j) = pi_block[j]
      enddo
      SYNC ALL
      if (this_image() == 1) then
        do j=1,num_images()
          n += 1.d0
          pi_sum  += pi_block_local(j)
          pi_sum2 += pi_block_local(j)*pi_block_local(j)
        enddo
        pi_average = pi_sum / n
        pi_variance = pi_sum2/n - pi_average**2
        pi_error = sqrt(pi_variance/(n-1.d0))
        print *,  pi_average, '+/-', pi_error
      endif
      FREE pi_block
    enddo

    deallocate(pi_block_local)
  end
```

Using 4 processes, the output of the program gives:

```
$ ./test_caf
   3.14237790000000      +/-  4.966975835348314E-004
   3.14234140000000      +/-  2.348424884354168E-004
   3.14224363333333      +/-  1.958218077039208E-004
   3.14210150000000      +/-  1.786455096347535E-004
   3.14193388000000      +/-  1.646706423564424E-004
...
   3.14161792187500      +/-  2.869715284522186E-005
   3.14161840309278      +/-  2.842813192882745E-005
   3.14161785918367      +/-  2.815996517045344E-005
   3.14161942727273      +/-  2.794672232344559E-005
   3.14162559000000      +/-  2.785054179473510E-005
```

One can see that the error bar is twice smaller than in the single core program. This reflects the fact that there are four times more samples.

Templated sort routine

This example generates 4 routines with the exact same algorithm.

- `insertion_sort` for `real` arrays
- `insertion_dsort` for `double precision` arrays
- `insertion_isort` for `integer` arrays
- `insertion_i8sort` for `integer*8` arrays

```
BEGIN_TEMPLATE

 subroutine insertion_$Xsort (x,iorder,isize)
  implicit none
  $type,intent(inout)    :: x(isize)
  integer,intent(inout)  :: iorder(isize)
  integer,intent(in)     :: isize
  $type                  :: xtmp
  integer                :: i, i0, j, jmax

  do i=1,isize
   xtmp = x(i)
   i0 = iorder(i)
   j = i-1
   do j=i-1,1,-1
    if ( x(j) > xtmp ) then
     x(j+1) = x(j)
     iorder(j+1) = iorder(j)
    else
     exit
    endif
   enddo
   x(j+1) = xtmp
   iorder(j+1) = i0
  enddo

 end

SUBST [ X, type ]

    ; real ;;
 d  ; double precision ;;
 i  ; integer ;;
 i8 ; integer*8 ;;
 i2 ; integer*2 ;;

END_TEMPLATE
```

Introspection

```
program get_doc

 integer :: iargc
 character*(32) :: arg
 integer :: i, j

 !----------
 ! Command : ./get_doc
 ! Prints the list of IRP entites
 !----------
 if (iargc() == 0) then
  print *, 'List of IRP entities'
  do j=1,size(entities)
   print *, entities(j)
  enddo
  return
 endif

 !----------
 ! Command : ./get_doc  titi  toto  momo
 ! Prints the documentation of IRP entities titi, toto and momo
 !----------
 do i=1,iargc()
   call getarg(i,arg)

!----------
! Python script executed at compile time that will find the name o
! IRP entities of the current program. If the name of an entity is
! command line, its documentation will be printed.
!----------

BEGIN_SHELL [ /usr/bin/python ]
import os
entities = []
for filename in os.listdir('.'):   # Loop over all file names
```

```
  if filename.endswith('.irp.f'):  #  If the name ends with .irp.f
    file = open(filename,'r')      #   we open it
    for line in file:              #   For each of its lines
      if line.strip().lower().startswith('begin_provider'):
                                   #    If the line starts with
                                   #     begin_provider (case inser
        name = line.split(',')[1].split(']')[0].strip()
                                   #    The line is split to extrac
                                   #     of the IRP entity
        entities.append(name)      #    And it is added to the 'ent
    file.close()                   #   We close the file

for e in entities:
  print "  if (arg == '%s') then"%(e,)
  print "    print *, %s_doc"%(e,)
  print "  endif"
END_SHELL
 enddo
end

!---------------
! Script that creates the providers.
!---------------

BEGIN_SHELL [ /usr/bin/python ]

import os
doc = {}
for filename in os.listdir('.'):
  if filename.endswith('.irp.f'):
    file = open(filename,'r')
    inside_doc = False
    for line in file:
      if line.strip().lower().startswith('begin_provider'):

        name = line.split(',')[1].split(']')[0].strip()

        doc[name] = ""
      elif line.strip().lower().startswith('begin_doc'):
        inside_doc = True
      elif line.strip().lower().startswith('end_doc'):

        inside_doc = False
      elif inside_doc:
        doc[name] += line[1:].strip()+" "

    file.close()

lenmax = 0
for e in doc.keys():
  lenmax = max(len(e),lenmax)

# We create here the provider of 'entities' which is the array of
# all the entities
# ------------------------------------------------------------------
print "BEGIN_PROVIDER [ character*(%d), entities, (%d) ]"%(lenmax,l
print " BEGIN_DOC"
print "! List of IRP entities"
print " END_DOC"
for i,e in enumerate(doc.keys()):
  print "entities(%d) = '%s'"%(i+1, e)
print "END_PROVIDER"

# We create the providers of each entity
# --------------------------------------
for e in doc.keys():
  print "BEGIN_PROVIDER [ character*(%d), %s_doc ]"%(len(doc[e]),e
  print " BEGIN_DOC"
  print "! Documentation of variable %s"%(e,)
  print " END_DOC"
  print " %s_doc = '%s'"%(e,doc[e])
  print "END_PROVIDER"

END_SHELL
```

Index of command line options

Usage

```
irpf90 [OPTION]
```

Options

`-C, --coarray` : All providers are coarrays

`-D, --define` : Defines a variable identified by the `IRP_IF` statements.

`-I, --include` : Include directory

`-a, --assert` : Activates ASSERT statements. If absent, remove ASSERT statements.

`-c, --codelet (<entity>:<NMAX>|<entity>:<precondition>:<NMAX>)` : Generate a codelet to profile a provider running NMAX times

`-d, --debug` : Activates debug. The name of the current subroutine/function/provider will be printed on the standard output when entering or exiting a routine, as well as the CPU time passed inside the routine.

`-g, --profile` : Activates the profiling of the code.

`-h, --help` : Print this help

`-i, --init` : Initialize current directory. Creates a default Makefile and the temporary working directories.

`-l, --align <N>` : Align arrays using compiler directives and sets the $IRP_ALIGN variable. For example, --align=32 aligns all arrays on a 32 byte boundary.

`-m, --memory` : Prints memory allocations/deallocations.

`-n, --inline (all|providers|builders)` : Force inlining of providers or builders

`-o, --checkopt` : Shows where optimization may be required

`-p, --preprocess` : Prints a preprocessed file to standard output. Useful for debugging files containing shell scripts.

`-r, --no_directives` : Ignore all `!DEC$` and `!DIR$` compiler directives

`-s, --substitute` : Substitute values in do loops for generating specific optimized code.

`-t, --touch` : Display which entities are touched when touching the variable given as an argument.

`-v, --version` : Prints version of irpf90

`-z, --openmp` : Has to be set for OpenMP codes

```
```