Appendix

PFA – Options

- **The PFA options fall into several categories:**

  - ***Parallelisation***
  - ***Directive control***

  Obsoleted, use LNO,IPA,INLINE,LIST,..... instead:

  - ***Scalar optimisations*** refer to the appendix in this presentation
  - ***Listing control*** refer to the appendix in this presentation
  - ***Language control*** refer to the appendix in this presentation
  - ***Filename control*** refer to the appendix in this presentation

- **Each PFA option has a long name, a short name and a default value**
- **Many PFA options have equivalent directives**
- **Complete info: pfa man page and Insight documents**

PFA – Directive Control

–directives=*list* **Specify which directives to accept***

list **consists of any combination of:**

A Accept assertions (*default*)
C Accepts Cray CDIR$ directives
be aware of the CDIR$ IVDEP directive**
K Accept C*$* directives (*default*)
S Accept C$ directives (*default*)
V Accept VAST (tm) CVD$ directives (*default*)

* Use -nodirectives to ignore ALL directives and assertions

** -SWP:strict_ivdep=ON eliminates a possible dependence between two memory references in the presence of an IVDEP directive (or pragma), only if the compiler can prove that the dependence is loop-carried. Default is ON.

PFA – Listing Control

–lines=*n*	Number of lines/page (n=0: paginate at subroutine boundaries)
–suppress=*list*	Suppress messages from a class specified in the list (e.g. warnings)
–listoptions=*list*	Specify the information to be included in listing file

list consists of any combination of:

C	Calling tree at the end of the program listing
K	Echo options used
L	Loop by loop optimization table *(default)*
N	Print program unit name when being processed
O	Annotated listing of original program *(default)*
P	Processing performance statistics
S	Summary of optimizations performed
T	Annotated listing of transformed program

PFA – Filename Control

–input=*file*	Specify file to be the input file (*default: option not preceded by –*)
–fortran=*file*	Specify the name of the intermediate Fortran file (*default: file.m*)
–list=*file*	Specify the name of the listing file (*default: file.l*)

PFA – Language Control

–assume=*list* **Make certain assumptions**

list consists of any combination of:

- E Allow equivalence variable to refer to the same memory location (*default*)
- L Save last scalar values in a loop (*default*)
- P Allows for parameter aliasing in subprogram

–save=*key* **Specify how to treat local variables**

key can take one of the following values:

- A Performs a lifetime analysis of local variables and *tries* to find those that require to be saved between calls. Generates the SAVE statement for such variables.
- M Do not generate SAVE statements (*default*)

–scan=*value* **Control the number of columns to be limited to *value* (default is 72)**

–syntax=key **Restrict to ANSI F77 (*key=A*) or VMS Fortran (*key=V*) (default is SGI Fortran)**

PFA – Most Frequently Used Options

Inlining and Interprocedural Analysis (IPA)

–inline=*list*	Specify the candidate routines for inlining (colon separated)
–inline_from_files=*list*	Specify the file(s) in which the routines are (colon separated)
–inline_depth=*value*	Specify the depth of inlining (to inline nested calls)
–ipa=*list*	Specify the candidate routines for IPA (colon separated)
–ipa_from_files=*list*	Specify the file(s) in which the routines are (colon separated)

Conditions that prevent inlining (or IPA):

- Dummy arguments do not match the actual arguments in number, type, shape, or size
- The calling program and called routine have conflicting declarations for the same COMMON block
- The calling program and the called routine have conflicting EQUIVALENCE statements
- The routine to be inlined has a SAVE, ENTRY, or NAMELIST statement
- The routine to be inlined has a DATA loaded variable
- The routine to be inlined is too long (the limit is about 600 lines)

Strongly Recommended to use –INLINE/–IPA to do this

PFA – Directive Control

–directives=*list* **Specify which directives to accept***

***list* consists of any combination of:**

- **A Accept assertions (*default*)**
- **C Accepts Cray CDIR$ directives**
  - be aware of the CDIR$ IVDEP directive**
- **K Accept C*$* directives (*default*)**
- **S Accept C$ directives (*default*)**
- **V Accept VAST (tm) CVD$ directives (*default*)**

* Use –nodirectives to ignore ALL directives and assertions

** –SWP:strict_ivdep=ON eliminates a possible dependence between two memory references in the presence of an IVDEP directive (or pragma), only if the compiler can prove that the dependence is loop-carried. Default is ON.

PFA – Scalar Optimisations

–roundoff=*level* **Control whether a reduction can run in parallel**

level **can take one of the following integer values:**

0 Suppresses any round–off changing transformations (*default*)
1 Same as level 0 (!)
2 Allows reductions (+,*,min,max) to run in parallel
3 Recognize REAL induction variables, perform memory management transforms (outer loop unrolling, blocking)

In general, it is recommended to use to flag to parallelize reduction operations

One should be aware though, that the numerical results can be different due to the different order of operations:

A = (B + C) + (D + E) =/= B+ (C + (D+E)) from a numerical point of view !!!

PFA – Scalar Optimisations

–optimize=*level* Specify the optimization level

level can take one of the following integer values:

0 Avoid converting loops to run in parallel
1 Parallel and enables loop interchange (no advanced data dependency)
2 "1" and advanced data dependency analysis (parallelization of reductions in combination with –roundoff=2)
3 "2" and additional advanced optimization techniques
4 "3" and code specialization (multiple copies of loops)
5 "4" and adjacent loop fusion, more advanced parallelization (*default*)

PFA – Scalar Optimisations

–scalaropt=*level* Specify the scalar optimization level

level can take one of the following integer values:

0 Perform no scalar optimizations
1 Simple transformations (dead code elemination, if–goto into if–then–else)
2 Loop unrolling, loop fusion (–optimize=5 needed) and some other optimizations
3 Enable memory management transformations, also –roundoff=3 needed (*default*)

PFA – Scalar Optimisations

–unroll=*value1* Unroll the innerloop "level" times (*default=4*)
–unroll2=*value2* Control how complex the unrolled loop can be

value2 is an arithmetic operation count, used to unroll loops that contains more calculations:

The number of array references and arithmetic operations in the loop are counted. The loop is unrolled until:

A. The number of iterations > value1 *or*

B. The operation count > value2

–cachesize=value Set the size of the cache in units of KByte (default: *system dependendent*)

Notes:

Need to set –optimize=4, or higher, to activate the –unroll/–unroll2 options

Don't expect miracles from setting a –cachesize value. Check for blocking!

Strongly Recommended to use LNO to do this

More pragmas

There is another category C pragmas that we list here:

Pragma	Scope
#pragma serial	next loop
#pragma concurrent	next loop
#pragma concurrent call	next loop
#pragma no side effects (name[name...])	program unit
#pragma arl(n)	selectable
#pragma set chunksize (n)	next loop
#pragma set numthreads (n)	next loop
#pragma set schedtype (type)	next loop
#pragma distinct (name,name[name...])	program unit
#pragma inline [here][routine][global] [(name[name..])]	selectable
#pragma ipa [here][routine][global] [(name[name..])]	selectable
#pragma padding (variable list)	program unit
#pragma storage order (variable list)	program unit

Many are also available as a compiler option and there are more pragmas than listed here (see Ch 4 of the PwrC UG .....)

Pragma serial

#pragma serial

Forces the loop, and enclosing loops, to run serially

```
 for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
#pragma serial
      for (k=0; k<N; k++)
         x[i][j][k] *= y[i][j];
      for (k=0; k<N; k++)
         x[i][j][k] += z[i][k];
     }
```

These three loops are now serial

This loop is still optimized

Pragma concurrent [call]

#pragma concurrent

Tell the compiler to ignore *assumed* dependencies

#pragma concurrent call

Inform the compiler that the function calls in the following loop are safe to be executed in parallel

Pragma arl

#pragma arl(n) *

Add information about possible memory aliases

n = 0   No assumption (worst–case)
1   There are no pointer self–references i.e. a pointer does not contain its own address (default)
2   Function arguments are distinct
3   Local pointer/arrays are distinct from global pointers/arrays
4   All pointers/arrays are distinct from each other

#pragma no side effects (name [, name ...])

Inform the compiler that the function "name"performs no I/O and only changes local variables. Hence, they are safe to parallelize
(and probably don't do a thing anyway...)

) *This can also be achieved with an option and you probably need it when auto–parallelising a code .....