

# CHAPTER 3

---

## Pragmas

This chapter contains description of *BERT77* pragma language. A pragma is simply a special comment placed in your FORTRAN program. Pragmas provide information or directives to *BERT77*. Information in pragmas is used to assist *BERT77* during the conversion process. Quite often information known by the user, but not explicitly expressed in the program can be helpful to *BERT77*. Directives in pragmas can be used to force *BERT* to do specific things. Because *BERT* in general makes "conservative" (safe) decisions, there may be times when *BERT* can not determine a data relationship that is known by the user.

Pragmas also allows *BERT* to work with special cases. If *BERT* can not determine if a portion of your program is concurrent due to some special programming technique, then it will be possible to convert this portion with the use of pragmas. Otherwise, these cases would be a "dead end" and require code modifications before *BERT* would make them parallel.

This chapter describes the *BERT* pragmas or directives. It is also suggested that you read in Chapter 1: *Parallelization Model and Methods*. The *Introduction* sections contains starting knowledge. The section *How to Use Pragmas* presents the most common cases when pragmas are helpful. The section *How to Write Pragmas* explains how to write complex pragmas correctly. Sections *Syntax of Pragmas* and *Explanation of Pragmas* include formal pragmas description and detailed explanations. Finally the section *Example of Parallelization* is a parallelization tutorial, explaining how to parallelize a real program interacting with *BERT*.

## Pragmas and Automatic Analysis

There is a trade off between automatic concurrency detection and pragma additions by the user. At one end of the spectrum is the desire to provide totally automatic conversion of a FORTRAN program without the need for the user to supply any information. This goal, while highly desirable, is difficult to achieve and requires a large amount of work. At the other end of the spectrum is the need for the user to provide a complete description of their program with *BERT* pragmas. While this would allow *BERT* to provide very optimal conversions, it places an additional (and unacceptable) burden on the user. The best result lies somewhere in between, where a

large percentage of concurrent behavior of the program is detected automatically and "difficult" cases are left for the user to describe using pragmas. It turns out that automatic detection of the difficult cases is a large percentage of the work. That is, the last ten percent of the fully automatic detection represents ninety percent of the work.

Future versions of *BERT* will have improved detection algorithms in addition to pragmas. Pragmas allow *BERT* to be useful "right now" for almost any FORTRAN program.

## Introduction

*BERT77* allows the use of pragmas to facilitate the conversion process. Pragmas are used to give *BERT77* "directives" about the program structure and data dependencies.

Pragmas take the form of FORTRAN comments. The general format is:

```
C$BERT pragma
```

where *pragma* is a specific pragma. In this way, the original source code can remain FORTRAN 77 compliant.

Pragmas can be inserted into the source program in one of two ways.

1. The user can add pragmas
2. *BERT77* can add pragmas

In the current version, the pragmas added by *BERT77* are never seen by the user unless the `-pio[e] [f]` option is used with the `bert` command. **This option is important if you wish to add your own pragmas. BERT will attempt to write the correct pragma which can then be edited by the user and placed in the original program.**

General information and an introduction to pragmas follows in this section. More detailed pragma descriptions can be found later in this chapter.

## Pragma Example

The user can force a program fragment to be executed in parallel. Two kinds of parallel execution are possible: DO loops and segments. Consider how two independent program segments can be described with pragmas:

```
CALL S1(X, Y, N)
CALL S2(A, B, N)

SUBROUTINE S1(X, Y, N)
  REAL X(100), Y
  INTEGER N
  Y = 0.
  DO I = 1, N
    Y = Y + X(I)
  ENDDO
END

SUBROUTINE S2(X, Y, N)
```

```

REAL X(100), Y
INTEGER N
Y = 1.
DO I = 1, N
    Y = Y * X(I)
ENDDO
END

```

Here the subroutine calls S1 and S2 are concurrent, but *BERT77*, in automatic, mode neglects this parallelism if it does not consider it efficient. If you want to force parallel execution of the two subroutine calls, you can use the following pragmas.

```

C$BERT PARBEGIN
    CALL S1(X, Y, N)
    CALL S2(A, B, N)
C$BERT PAREND

```

PARBEGIN starts a group of independent instructions. PAREND ends this group. All instructions in this group will be executed in parallel. If S1 and S2 are not present in the source code (e.g. they were compiled previously and placed in the user's library), then data used by the modules **MUST** be described in pragmas in the following manner.

```

C$BERT PARBEGIN
C$BERT DATA IN = {N, X(1: N)} OUT = {Y}
    CALL S1(X, Y, N)
C$BERT DATA IN = {N, A(1: N)} OUT = {B}
    CALL S2(A, B, N)
C$BERT PAREND

```

Here IN are all data values that are used by the line following the DATA pragma - either explicitly or implicitly. OUT are all data values that are produced by the corresponding line. A(1: N) means A(1), . . . , A(N) (N should be greater or equal to 1). Note, that the order of the data is important -similar to READ statements. Therefore, description IN = {A(1: N), N} is incorrect. If an IN or OUT set is not described by a pragma, it will be determined automatically by *BERT*. (See Chapter 1: *Parallelization Model and Methods* for more information.)

Another example is when you want to force parallel DO loop execution.

```

DO I = 1, N
    CALL S(A(I), DBLE(I)/DBLE(N))
ENDDO
...
SUBROUTINE S(X, Z)
    X = Z*Z
END

```

If *BERT* does not think that parallelization of the loop is useful and you think it will be, you can forced this loop to be executed in parallel by adding the C\$BERT PARDO (parallel DO loop) pragma. For the example above:

```

DO I = 1, N
C$BERT PARDO

```

```

        CALL S(A(I), DBLE(I)/DBLE(N))
    ENDDO
    ...
    SUBROUTINE S(X, Z)
        X = Z*Z
    END

```

This pragma will force the loop to be executed in parallel.

## How to Use Pragmas

Basic information and typical examples of parallelization directives can be found in this section.

### When to Use Pragmas

*When are pragmas needed?*

These are two different cases and each should be considered carefully. The first case is when a program is too sophisticated for automatic parallelization. It may be possible that *BERT* can not determine if a variable is recursive (depends on each loop cycle). In this case the user can force *BERT* to parallelize the loop anyway - presumably because the user knows the variable is not recursive. The second case is when there is a need to force a concurrent section to execute in parallel. *BERT* may have determined incorrectly that a concurrent section should be performed on a single processor, perhaps due to the incorrect estimation of a loop boundary. Pragmas allow all *BERT* decision to be overruled. Please be aware that in some cases improper use of pragmas may provide inefficient and incorrect results.

*What is described in pragmas?*

For the most part, pragmas describe how data handling and parallelizations should be performed. The IN and OUT sets described in a pragma are used by *BERT* so the correct data may be transmitted to worker processors.

*How complex pragmas can be?*

Pragmas can range from very simple to complex. The required complexity depends on the program.

### Selection of Parallel Segments

The simplest use of pragmas is selection of which concurrent portion must be performed in parallel.

Example:

```

        DO I = 1, N
            DO J = 1, N
    C$BERT PARDO
            ...
        ENDDO
    ENDDO

```

ENDDO

Here inner (J) loop will be executed as parallel, and outer (I) loop as sequential.

Another example shows forcing of parallel execution of two subroutine calls. Note, that in this case DO loop inside subroutine S will be executed sequentially, because it is nested to PARBEGIN/PAREND construction.

Example:

```
C$BERT PARBEGIN
      CALL S(A)
      CALL S(B)
C$BERT PAREND
      ...
      SUBROUTINE S(X)
      ...
      DO I = 1, N
      ...
      ENDDO
```

Simple pragmas can also be used to prohibit automatic parallelization of certain parts of programs.

Example:

```
      DO I = 1, N
C$BERT SEQDO
      ...
      ENDDO
```

Here pragma SEQDO (sequential DO loop) prohibits parallel execution of DO loop.

To avoid automatic parallelization also pragmas PAR=ON | OFF can be used (see the section on PAR pragmas).

## Avoiding Recursion (Parallelization Inhibitors)

The most common reason for not parallelizing a DO loop do to recursive usage of data. Recursive data are intermediate data produced by a loop iteration that is then used by following iteration(s). When *BERT* detects recursion, it gives diagnostics: "possibly recursive variable (array)". "Possibly" means that sometimes it may not "true" recursion and therefore the loop may be concurrent. When *BERT* is not sure that there is no recursion, it assumes "the worst case".

At various times *BERT* will detect "false" recursion - due to the worst case assumptions. Below are the most common cases of "false" recursion.

### "False" Recursion

Example:

```
DO I = 1, N
  IF(B .GT. 0) X = 1.
  IF(B .LE. 0) X = -1.
```

```

        ... = X
    ENDDO

```

Here *BERT* "does not know" that on every loop iteration *x* is **always** computed before it is used (the if statements make *BERT* think *x* is conditionally computed). *BERT* "thinks" that this can be the case when value of *x* is not assigned on some iteration. This kind of recursion can be overridden with a PARD0 pragma.

Another "false" recursion may appear when array indices has a complex form.

Example:

```

    DO I = 1, N
        A(I) = A(I+K) + F(B(I))
    ENDDO

```

Here *BERT* does not evaluate *I+K* and does not know, if old or new values of array *A* are used for computation. In the case when  $K \geq 0$  there is no recursion. Here again simple PARD0 pragma can be applied, but it is strongly recommended to describe the data using an IN field.

```

C$BERT PARD0 IN={A(I+K), B(I)}

```

**IMPORTANT:** If a single or generated data object is described in corresponding pragma field, All data objects of this kind must be described in the pragma. To do this it is recommended to run *BERT* with the "-pio" option and note the correct pragma patterns generated for your program.

One more example illustrates the following problem. *BERT* assumes that a variable value assigned within loop can be used after the assignment. When the value is not assigned on every iteration, *BERT* does not know, what physical processor has generated the correct value.

```

    DO I = 1, N
        IF(B .GT. 0) THEN
            X = ...
            ... = X
        ENDIF
    ENDDO

```

If value of *X* is not used after the DO loop, it can be parallelized by the pragma PARD0.

## Inductions

Consider the following loop:

```

    DO I = 1, N
        K = K + 1
        ... = K
        K = K + 2
    ENDDO

```

This loop can be parallelized with pragma

```
C$BERT PARDO INDUC={ (K, 3) }
```

Here value of variable K is increased in every iteration by constant step equal to 3. This is example of induction variable. *BERT* does not determine it automatically, because it is changed more than once in DO loop. It can be described by INDUC field in pragma. (K, 3) means: "induction variable K increased by 3 each iteration".

If an induction variable is increased inside a subroutine/function call, *BERT* also "needs help" though the addition of a pragma similar to the one above.

## Reductions

Consider the following example.

```
X = 1
DO I = 1, N
...
    X = -X
ENDDO
```

Here X changes its sign every iteration. It can be described with the pragma:

```
REDUC = { (X, X=SIGN(X, X*#), X = 1) }
```

This pragma means that reduction variable (X) is computed through partial results computed in different processors (#) by formula :

```
new value = SIGN(old value, old value * partial result)
```

Initial value of partial result is 1.

For more details see section "REDUC\_LIST Pragma".

## Number of DO Loop Iterations

When *BERT* estimates parallelization efficiency, it counts execution times. The precision of this estimation depends on how well it is possible to estimate the number of loop iterations. When the number of iterations cannot be computed explicitly, *BERT* makes assumptions. Nevertheless, in certain cases there is no reliable points to "start from". Then the directive indicating the number of iterations is useful like the following:

```
DO I = 1, N
C$BERT DO REPEATED(5000)
    X = X + ...
ENDDO
```

Here *BERT* gets information that this DO loop is expected to have 5000 iterations and can make further decisions itself. This directive is also useful when *BERT* over or under estimates the actual number of loop iterations. Under estimation may prevent parallelization while over estimation may create inefficient parallelizations.

## Unknown Function Calls

To solve the problem of time estimation for unknown function calls, the DATA pragma may be added. To account for the time of the unknown function, a TIME

field has been added to the DATA pragma. This field allows you to tell *BERT* the time required to call this function. The form of the TIME field is as follows:

```
TIME={time_value [scale]}.
```

where: `time_value` is integer constant and `scale` is `s` (seconds), `ms` (microseconds) or `ns` (nanoseconds). The default is `s` (seconds).

For example:

```
c$BERT data time={12345 ms} IN = ...
```

indicates to *BERT* the execution time of an instruction following this pragma is 0.012345 seconds. The time is for one pass of the instruction (or function call), not the total in the program. The user must determine this execution time.

To solve the problem of dependency analysis, the `-call` option can be used when invoking *BERT*. This problem arises when *BERT* must assume the unknown function can call any function in the program or may contain STOP and file instructions. Therefore, a loop that contains an unknown function call may have a large unresolvable set of dependencies. See the BERT 77 User Guide for more information.

## Decomposing DO Loops

Normally *BERT* decides itself, how many processors to use for certain parallel DO loop. It is also possible to "tell" *BERT* how many processors to use. For example:

```
DO I = 1, N
C$BERT PARDO (N / 2)
...
ENDDO
```

With this directive *BERT* will use  $N/2$  workers for the loop.

## How to Write Pragmas

Hints on pragmas construction are given in this section.

### Defining IN and OUT Sets

The IN set must contain all data values that are used for computations upon entering a FORTRAN line. The OUT set must contain all data values that are produced by the corresponding FORTRAN line and used in further computations. Further information on IN and OUT sets can be found in Chapter 1. For example:



```

PROGRAM ABSTRACT
COMMON X(100), Y(100), Z, N

CALL S
PRINT *,(Y(I), I = 1, N)
END

SUBROUTINE S
COMMON X(100), Y(100), Z, N
DO I = 2, N, 2
    IF(X(I) .GT. 0.) THEN
        Z = X(I) + X(I-1)
        Y(I) = Z * Z
    ENDIF
ENDDO
END

```

Here a valid data description for the statement calling module S is:

```
C$BERT DATA IN = {N, X(1: N), Y(2: N: 2)} OUT = {Y(2: N: 2)}
```

Where  $(Y(2: N: 2))$  is an array section and means  $Y(2), Y(4), Y(6), \dots, Y(N)$ . One may ask, "Why did array Y appear in the IN set?". The reason is that elements of Y are not necessarily computed by module S because of the IF statement. Therefore, sometimes these elements contain 'new' values and sometimes 'old' values. The IN set contains data that are actually transferred from the master to the worker. The OUT set contains data that are actually transferred from the worker to the master. If elements of array Y are not loaded to the worker, then elements which do not contain 'new' data will have incorrect values that overwrite correct values after transfer from the worker to the master. The general rule is:

**ALWAYS ADD VARIABLES FROM THE 'OUT' SET TO THE 'IN' SET IF YOU ARE NOT SURE THAT THEY ALWAYS GET ASSIGNED NEW VALUES IN PARALLEL SEGMENTS OF THE PROGRAM.**

Another question to ask is: "Why is the variable z not present in the OUT set?". Adding it to the OUT set is not an error, but its value is not used after module S is called, so it is local. Reducing the amount of data in the IN and OUT sets will increase performance of parallel program because communication can be minimized. In any case:

**REMOVE LOCAL VARIABLES FROM THE 'OUT' SET ONLY IF YOU ARE ABSOLUTELY SURE THAT THEIR VALUES ARE NOT USED IN FURTHER COMPUTATIONS. REMEMBER THAT THIS PROPERTY MAY DISAPPEAR AFTER MODIFICATION OF THE PROGRAM.**

Note, that variable z did not appear in the IN set. The reason is that the value of z on entering module S is not used for computations, as it is ALWAYS set to new value before use.

**EXCLUDE VARIABLES FROM THE 'IN' SET ONLY WHEN YOU ARE SURE THAT THEY ALWAYS GET NEW VALUES IN PARALLEL SEGMENTS BEFORE USE.**

## DO Loops Inside PARBEGIN/PAREND

As mentioned above, all statements within PARBEGIN/PAREND are executed in parallel. This leads to the following limitation:

**NO EXPLICIT OR IMPLICIT GOTO'S ARE ALLOWED INSIDE PARBEGIN/PAREND GROUP.**

For example, IF-ENDIF statements are not allowed, because they contain implicit branches. Although, the logical IF is allowed if it has no GOTO, because it is one FORTRAN statement. One exclusion is made from this rule. *BERT* regards an explicit DO loop as one FORTRAN statement. Hence, the following example is possible.

```
C$BERT PARBEGIN
DO I = 1, N
    Y = Y + X(I)
ENDDO
DO I = 1, N
    B = B * A(I)
ENDDO
C$BERT PAREND
```

## Parallel DO Loops

DO loops can be considered as a collection of consecutive iterations. When all iterations are independent, they can be executed in parallel, and the DO loop is called concurrent. Actual parallel execution, as decided by *BERT* depends upon the efficiency of the loop. Concurrent DO loops are the most important part of a parallel programs. Concurrent DO loops can be described as following:

```
DO I = 2, N, 2
C$BERT PARDO IN = {X(I-1: I), Y(I)} OUT = {Y(I)}
    IF(X(I) .GT. 0.) THEN
        Z = X(I) + X(I-1)
        Y(I) = Z * Z
    ENDIF
ENDDO
```

Note that adding a PARDO pragma will tell *BERT* to make the loop parallel regardless of efficiency or data dependencies. IN and OUT sets in the PARDO pragma describe data for one iteration. A PARDO can be an element of PARBEGIN/PAREND group like the following:

```
C$BERT PARBEGIN
CALL S1
DO I = 1, N
C$BERT PARDO
    ...
ENDDO
C$BERT PAREND
```

## Reduction and Induction Variables

Quite often DO loop iterations are not parallel because an associative operation like summation, addition, MAX, etc. is performed on the data. In this case, it is possible to compute partial results in parallel and then apply a given operation. These operations are called *reduction constructions* and variables - reduction variables. Summation and multiplication are usually detected by *BERT* v.1.03 automatically. An example with maximal value search is as follows.

```

VALMAX = F(X(1))
DO I = 2, N
C$BERT PARDO REDUC = {(VALMAX, VALMAX = MAX(VALMAX, #),
# = VALMAX)}
      IF(X(I). GT. VALMAX) VALMAX = F(X(I))
ENDDO

```

Where the '#' character denotes partial result. Description of reduction consists of three parts:

- 1) The first part is the name of the reduction variable (here VALMAX).
- 2) The second part is the operation performed on the partial result.
- 3) The third part is the initialization of the partial result on processors executing the parallel program.

Note: When using floating point arithmetic, the result may differ after applying parallel reduction constructions. This difference is caused by roundoff hardware errors. Changing the order of operations may produce different results. For the great majority of cases, these errors only influence the least significant digits. When the result must be exactly the same, reductions on floating point should not be used. This can be important particularly when you use unstable algorithms. The following program illustrates roundoff errors if run both in sequential and parallel modes.

```

program dx
double precision result,f,x,step,func
integer t1,t2
step=1./1000000.d0
result=0.d0
do 1 i=1,1000000
  x=(i-.5d0)*step
  f=func(x)
  result=result+f*step
1 continue
print *, ' result=', result
stop 0
end
double precision function func(x)
double precision x
func=dsin(x)*dcos(1.d0-x)
return
end

```

Another case appears when a variable is changed in every iteration by a constant value. This case makes it possible to compute the variable via a loop parameter.

```
DO I = 1, N
  C$BERT PARDO  INDUC = {(J, -1)}
  ...
  J = J - 1
ENDDO
```

The description of induction variable consists of two parts: variable name and step. Below are two more examples of pragma usage; PARTIAL CASCADE SUM and CYCLIC REDUCTION are based on "Designing Efficient Algorithms for Parallel Computers" by Michael J. Quinn (McGraw-Hill Book Co, 1987). Real speedup can be achieved when instead of using array elements, function calls are used. These examples should be regarded as general examples.

```
C***** PARTIAL CASCADE SUM *****
DO I = 0, N-1
  C$BERT PARDO
    X(I) = D(I)
  ENDDO
DO I = 0, LOG(N)-1
  DO J = 2**I+1, N-1
  C$BERT PARDO IN={X(J), X(J - 2**I)}; OUT = {X(J)};
    X(J) = X(J) + X(J - 2**I)
  ENDDO
ENDDO
C ***** CYCLIC REDUCTION *****
DO I = 0, LOG(N) -1
  DO J = 2**I + 1, N-1
  C$BERT PARDO IN = {A(J), A(J - 2**I), D(J)}; OUT = {A(J), X(J), D
(J)};
    IF (I .GE. 1) THEN
      T = A(J - 2**I)
      A(J) = A(J) + T
    ENDIF
    T = D(J-2**I)
    D(J) = A(J) * T + D(J)
    X(J) = D(J)
  ENDDO
ENDDO
```

## Syntax Of Pragmas

The syntax (grammar) for all of the pragmas is contained in this section.

```
pragma_statement: C$BERT pragma
pragma:
    block
    or
    data
    or
    do_desc
    or
    switch
switch:
    PAR=ON
```

---

	or	PAR=OFF
block:		PARBEGIN
	or	PAREND
data:		DATA [ data_list; ] ...
do_desc:		do_spec [ data_list; ] ...
	or	DO [ repeats ]
do_spec:		SEQDO [ ( decomposition ) ] [ repeats ]
	or	PARD0 [ ( decomposition ) ] [ repeats ]
decomposition:		FORTTRAN_expression
repeats:		REPEATED ( integer_expression )
data_list:		in_list
	or	out_list
	or	reduction_list
	or	induction_list
in_list:		IN = { var_list }
out_list:		OUT = { var_list }
reduction_list:		REDUC = { reduc_list }
induction_list:		INDUC = { induc_list }
var_list:		[ element [, element ] ... ]
reduc_list:		[ r_element [, r_element ] ... ]
induc_list:		[ i_element [, i_element ] ... ]
r_element:		( element, reduction, initialization )
i_element:		( name, step )
reduction:		#_assignment
initialization:		#_assignment
step:		FORTTRAN_expression
#_assignment:		FORTTRAN_assignmentnet where instead of scalar name can be used symbol '#'
element:		local_name
	or	global_name
	or	array_section
	or	special_name
local_name:		FORTTRAN_scalar_variable_name
global_name:		/ common_block_name
		[. invariant_extension] /
	or	<module_name>
invariant_extension:		FORTTRAN_integer
array_section:		array_name ( sections_list )
sections_list:		section [, section ] ...
section:		low [ : up [ : step ] ] [   overlap ]
low:		extended_expression
high:		extended_expression
step:		extended_expression
overlap:		extended_expression
extended_expression:		FORTTRAN_expression where instead of scalar name can be used one of the following:

```

                                BERT_$
                                name_BERT
                                name_BERT_LOW
                                name_BERT_UP
special name:                  BERT_STOP
                                or   BERT_FILE
                                or   BERT_GOTO

```

## Explanation of Pragmas

The individual pragmas are listed with specific notes and examples.

### Description

Pragmas are special directives for controlling parallelization. They allow you to describe parallel constructions supported by *BERT*. Pragmas can be used when *BERT* fails to parallelize some program segments automatically and for development of new parallel programs.

Pragmas have the form of FORTRAN comment lines. The first position of each pragma line must contain the symbol 'C'. If comment starts with 'C\$BERT' then it means that this line is pragma line. Otherwise *BERT* considers this line as a FORTRAN comment. When pragma contains more than one line, it can be continued on subsequent comment lines also starting with 'C'. A '&' symbol at the end of pragma line means that it will be continued. For example,

```

C$BERT    < lines          &
C          containing      &
C          pragma >

```

No difference in upper and lower case is made in pragmas. All blanks and Tab symbols after C\$BERT are ignored.

### Limitations

Pragmas may not contain text or bit constants and substrings.

### Pragma Listing

Pragma contains of two fields:      <keyword> <data> ,

where: <data> field may be empty and <keyword> is the pragma type.

The following pragma types are allowed:

```

PARDO
SEQDO
DO
PARBEGIN
PAREND
DATA
PAR=ON
PAR=OFF

```

## PAR Pragma

Pragma `PAR=OFF` is used for switching automatic parallelization off. This pragma has no effect on parallel constructions described by other pragmas. Pragma `PAR=ON` switches automatic parallelization on. The scope of these pragmas are all subsequent lines of FORTRAN source until another `PAR=ON|OFF` pragma or the end of the module is encountered.

Example:

```

      SUBROUTINE SUB1
C$BERT PAR=OFF
      ...
      END

```

## PARBEGIN, PAREND, and DATA Pragmas

These pragmas are used for concurrent execution of instruction sequences on different workers. Pragma `PARBEGIN` starts a sequence of concurrent instructions. Pragma `PAREND` terminates this sequence. Pragma `DATA` is used for describing the data which should be sent to corresponding worker and data which should be sent back to the master after its execution.

Example:

```

      PROGRAM
      ...
C$BERT PARBEGIN
C
C$BERT DATA <VAR_LIST for SUB1>
      CALL SUB1
C
C$BERT DATA <VAR_LIST for SUB2>
      CALL SUB2
C
C$BERT DATA <VAR_LIST for SUB3>
      CALL SUB3
C
C$BERT PAREND
C
      ...
      END

```

In this example three instructions will be executed in parallel. `VAR_LIST`s are described in below.

## PARDO Pragma

Pragma `PARDO` forces DO loop iterations to be executed in parallel. This pragma has the following format:

```
PARDO [ (number_of_workers) ] [ REPEATED( iterations ) ] VAR_LIST
```

where "*number\_of\_workers*" is the number of parts to split the DO loop. All these parts are executed independently. "*iterations*" is the expected number of DO loop iterations.

## SEQDO Pragma

Pragma SEQDO forces DO loop iterations to be executed sequentially. This pragma has the following format:

```
SEQDO [(number_of_workers)][REPEATED(iterations)] [VAR_LIST]
```

where "*number\_of\_workers*" is ignored. "*iterations*" is the expected number of DO loop iterations. This pragma can be used to override automatic parallelization of a DO loop.

## DO Pragma

Pragma DO can be used only to instruct *BERT* how many iterations are expected to be executed in the DO loop. This pragma has the following format:

```
DO REPEATED(iterations)
```

"*iterations*" is the expected number of DO loop iterations.

## VAR\_LIST Pragma

VAR\_LIST may contain IN\_list, OUT\_list, REDUC\_list and INDUC\_list. These lists has the following form:

```
ID_FIELD={ variable description, variable description, ... }
```

## IN\_LIST and OUT\_LIST Pragmas

IN\_list and OUT\_list have the same syntax. The IN\_list contains data which are sent to the worker before execution of the corresponding program segment. OUT\_list contains the data which are received by the master from the worker after execution of corresponding program segment. IN and OUT lists may contain the following variables.

Scalar variable example:

```
INTEGER B
...
C$BERT IN={ ..., b, }
...
```

Global variable (COMMON BLOCK). [referred as '/name/'] example

```
COMMON /ABC/ A, B, C
...
C$BERT OUT={ ..., /ABC/, ... }
```

which means transfer of the whole COMMON BLOCK ABC. If the COMMON BLOCK is invariant (i.e. its elements have the same length in every module) then it is possible to transfer it in smaller parts. Parts of invariant common blocks can be transferred using a number placed after a '.', like /ABC/.3.

```
C$BERT OUT={ ..., /ABC/.3, ... }
```

Here /ABC/.3 stands for variable C from COMMON BLOCK given above.



When a function or subroutine called from a given module uses SAVE variables, its possible to transfer these variables. An Example of static variables from other modules:

```
C$BERT PARBEGIN
...
C$BERT DATA OUT={ <SUB> }
CALL SUB
...
C$BERT PAREND
...
C
SUBROUTINE SUB
SAVE Z,Y
...
END
```

## Array Sections Pragma

Array sections, in the style of FORTRAN 90 can be expressed as:

```
C$BERT PARDO ... IN = { A(N1:N2:-1) }
```

A DO loop parameter can be used for array indexing as follows:

```
DO I = 1, 100
C$BERT PARDO ... OUT = { A(I,1:10) }
```

In this example, the worker which performs DO loop iterations returns values A(I,1:10). Let's consider the following example:

```
DO I = 1, 100
C$BERT PARDO IN = { B(I-1: I+1) }
```

Here a worker gets values B(I-1), B(I), B(I+1) for each loop iteration. Usually each worker performs more than one iteration. Lower and upper bounds of parts of DO loop are designated as: parameter\_BERT\_LOW and parameter\_BERT\_UP correspondingly.

```
C$BERT PARDO IN = { B(I_BERT_LOW-1: I_BERT_UP+1) }
```

This pragma generates nearly three times less data transfers than the previous one. Let's explore an example:

```
DO I = 2, 21
B(I-1) = ...
B(I) = ...
ENDDO
```

In the given DO loop, elements B(2:20) are computed twice. When we have two workers the first one computes B(1:11) and the second (B(11:21)). The following pragma may produce incorrect results:

```
C$BERT PARDO(2) ... OUT = { B(I_BERT_LOW - 1: I_BERT_UP) }
```

Termination order of workers is not determined, hence value B(11) may be produced by first worker and overwritten by second or vice versa. But the correct value of B(11) when loop is finished is the value produced by the second worker.

To avoid the problem an overlap can be used. An overlap is the number of last elements of array section which are not transferred if worker is not last. The last worker transfers the whole array section. The correct pragma is as follows:

```
C$BERT PARDO(2) ... OUT = { B(I_BERT_LOW - 1: I_BERT_IP| 1) }
```

Now the first worker returns B(1: 10) (B(11) is not transferred), the second worker returns B(11: 21).

#### GENERAL RULE:

VALUES OF ALL SCALAR VARIABLES FOR PARDO ARE RETURNED ONLY FROM THE LAST WORKER (OUT) AND SENT TO ALL WORKERS (IN). THE SAME RULE IS APPLIED FOR SINGLE ARRAY ELEMENTS WHICH DO NOT DEPEND EXPLICITLY ON A DO LOOP PARAMETER. ARRAY ELEMENTS WHICH EXPLICITLY DEPEND ON DO LOOP PARAMETERS ARE TRANSFERRED FROM WORKERS WHEN OVERLAP IS ENCOUNTERED.

#### Example:

```
C$BERT PARDO ... OUT = { A(1: 10| 10) }
```

Here all elements are transferred only from the last worker.

#### A NOTE OF CAUTION:

IF SOME VARIABLE VALUE IS RETURNED BY A WORKER BEFORE ANOTHER WORKER IS LOADED, THE LATER WORKER WILL RECEIVE THE NEW VALUE, OTHERWISE IT WILL RECEIVE OLD VALUE.

#### Incorrect example:

```

N=1
DO I = 1, N
C$BERT PARDO IN = {N, ... } OUT = {N, ... }
N = 2
...
ENDDO

```

Here the first worker receives N = 1, all other workers may receive either N = 1 or N = 2.

## REDUC\_LIST Pragma

REDUC\_list contains a description of additional actions for PARDO processing. REDUC\_list has the following format:

```
... (desc_of_reduc), (desc_of_reduc) ...
```

*desc\_of\_reduc* contains of three fields: *name\_of\_var*, *action*, *init*

*name\_of\_var* is variable description, either scalar or array section, *action* and *init* are FORTRAN statements.

When a variable in a DO loop is computed by repeating an associative operation (like '+', '\*', MIN, .OR., etc.), it is possible to parallelize the DO loop the following

way. Workers compute intermediate results, and the associative operation is performed on these intermediate results.

*name\_of\_var* is computed variable,

*action* is associative operation,

*init* is initialization of intermediate results.

The following rule is used. Name of intermediate result is '#'. Cantor

Example:

```

      DO I = 1, N
C$BERT PARD0  IN = { DELTA_MIN, DELTA_MAX, ... } OUT = { ... } &
C REDUC = { ( DELTA_MIN, DELTA_MIN = MIN(DELTA_MIN, #) ),      &
C           ( DELTA_MAX, DELTA_MAX = MAX(DELTA_MAX, #) ),      &
C           ( DELTA_SUM, DELTA_SUM = DELTA_SUM + #, # = 0. ) }
      ...
      IF(DELTA_MIN .GT. DELTA) THEN
        DELTA_MIN = DELTA
      ENDIF
      IF(DELTA_MAX .LT. DELTA) THEN
        DELTA_MAX = DELTA
      ENDIF
      DELTA_SUM = DELTA_SUM + DELTA
      ...
    ENDDO

```

If the computed variable is an array element or an array section then indices are placed only in the *name\_of\_var* field.

Example:

```

C$BERT PARD0 REDUC = { ( C(2:6), C = C * #, # = 1. ) ,      &
C                      ( C(1),   C = C + #, # = 0. ) }

```

## INDUC\_LIST Pragma

INDUC\_list contains a description of induction variables. INDUC\_list has the following format:

```
... (desc_of_induc), (desc_of_induc) ...
```

*desc\_of\_induc* contains of two fields: *name\_of\_var*, *step*

*name\_of\_var* is variable name, must be a scalar, *step* is FORTRAN expression.

When value of scalar variable is increased or decreased by constant value on each loop iteration, the variable is called an induction.

Example:

```

      DO I = 1, N
C$BERT PARD0 INDUC = { ( X, DX ) }
      ...
      X = X + DX
      Z(I) = X
    ENDDO

```

An induction variable value cannot be used for array indexation in VAR\_LISTs. Nevertheless, it is possible to use derived names: 'name\_BERT\_LOW' - value of induction variable 'name' on entering worker and 'name\_BERT\_UP' - value of induction variable 'name' on leaving worker.

Example:

```

      DO I = 1, N
C$BERT PARD0 OUT = { A(JK_LOW: JK_UP - 1), B(JK_LOW+1: JK_UP)} &
C      INDUC = { (JK, 1) }
      ...
      A(JK) = ...      ! variable JK used before change
      JK = JK + 1
      B(JK) = ...      ! variable JK used after change
ENDDO

```

Note. The length of name of parallel DO loop parameter or reduction/induction variables may not exceed 22 symbols.

## Example of Parallelization

In this section an example parallelization is presented. User interaction with *BERT* during parallelization is given here. Although example does not cover all possibilities, it illustrates main points. See the *BERT Users Guide* for more information.

The following example is taken from an image processing test suite.

```

      PROGRAM TEST60B
C
C      PROGRAM TEST60B
C      PROGRAM MEDIAN
C      RANKING TIME TO REPLACE EACH PIXEL OF AN N * N ARRAY WITH
C      THE MEDIAN VALUE OF IT'S 3*# NEIGHBORHOOD
C
C      TYPE  A = 512**2    BYTE IMAGE ARRAYS
C           B = 1024**2
C           C = 4096**2
C           D = 15000**2
C
C      PARAMETER( IXX=1024, IYY=1024)
C      PARAMETER( IKXX=3, IKYY=3)
C
C
C*
C      INTEGER MAX,I,J,M,N
C      INTEGER IM(1024,1024),IO(1024,1024),SRTBUF(9)
C      INTEGER IM(IXX,IYY),IO(IXX,IYY),SRTBUF(9)
C*
C      INTEGER CPU
C      REAL WALL,T1,T2,T3
C      LOGICAL TIMER
C*

```

```

        INTEGER SEED,K,P
C
C      SETUP
C
        WRITE(6,3)
3      FORMAT(2X," PROGRAM TEST60B MEDIAN 3*3 BASE")
        IX=IXX
        IY=IYY
        N=IXX
        IAREA=IX*IY
        IKX=IKXX
        IKY=IKYY
        IKAREA=IKX*IKY
C
C*
C Initializing some constants
C
C*
        MAX =IX
        T2 = 0
        SEED = 37
        K = 69
C*
C Initialize Memory
C
        DO 50 J = 1,MAX
            DO 50 I = 1,MAX
                P = (K*SEED + 1)
                SEED = P - (P/255)*255
                IM(I,J) = SEED + 1
50      CONTINUE
C*
C Printing out the resultant image
C
        WRITE(6,160) ((IM(I,J), J = 16,40), I = 16,40)
160      FORMAT(25I5)
C*****
C      START TEST
C*****
C
C      PROGRAM TEST60B
C      PROGRAM MEDIAN
C      RANKING TIME TO REPLACE EACH PIXEL OF AN N * N ARRAY WITH
C      THE MEDIAN VALUE OF IT'S 3*# NEIGHBORHOOD
        WRITE(6,3)
C
C      T1=SECOND(0.0)
C
C
C      CALL CALC(IX,IY,IKX,IKY,IM,SRTBUF,IO)
C
C
C      CALL IRTC
C

```

```

C      T2=SECOND()
      T3=T2-T1
      WRITE(6,928) T3
928   FORMAT(2X," CPU TIME = ", F15.8," SEC. ")
C
C*
C Ignore this code, it simply convinces the compiler to
C generate the
C the object code that carries out the benchmark by making the
C compiler
C think we might need the results for output. This code will
C never
C execute since the IF condition will always be false
C
      IF (I.GT.MAX+10) THEN
        DO 900 J = 1,MAX
          DO 900 I = 1,MAX
            WRITE(6,1000) IM(I,J)
900    CONTINUE
      END IF
1000  FORMAT(' YO',L1)
C*
C Report Runtimes
C*
      END
      SUBROUTINE CALC(IX,IY,IKX,IKY,IM,SRTBUF,IO)
C
      PARAMETER( IXX=1024, IYY=1024)
      PARAMETER( IKXX=3, IKYY=3)
C
C
C*
      INTEGER MAX,I,J,M,N
C      INTEGER IM(1024,1024),IO(1024,1024),SRTBUF(9)
      INTEGER IM(IXX,IYY),IO(IXX,IYY),SRTBUF(9)
C
C*****
C      START TEST
C*****
C
C
C      CALL SECOND(T1)
C*
C*
C Generate Median Filtered Image
C
      MAX=IX
      DO 200 J = 2,MAX-1
        DO 200 I = 2,MAX-1
          SRTBUF(1) = IM(I-1,J-1)
          SRTBUF(2) = IM(I-1,J)
          SRTBUF(3) = IM(I-1,J+1)

```

```

SRTBUF(4) = IM(I,J-1)
SRTBUF(5) = IM(I,J)
SRTBUF(6) = IM(I,J+1)
SRTBUF(7) = IM(I+1,J-1)
SRTBUF(8) = IM(I+1,J)
SRTBUF(9) = IM(I+1,J+1)
DO 100 L = 1,5
  DO 100 M = L+1,9
    IF (SRTBUF(M).LT.SRTBUF(L)) THEN
      TEMP = SRTBUF(L)
      SRTBUF(L) = SRTBUF(M)
      SRTBUF(M) = TEMP
    END IF
  CONTINUE
  IO(I,J) = SRTBUF(5)
CONTINUE
C    CALL SECOND(T2)
C
RETURN
END

```

After processing with *BERT* one may find out that the most of computations are in subroutine *CALC*. Information on the largest DO loop follows.

```

test60b.f:  LINE: 124
-----
ACTION:
  Processing DO loop.
NOTE:   Possibly recursive variable: 'temp'
NOTE:   Possible recursion in array 'srtbuf'
RESULT:
  Loop is not concurrent
-----
DO LOOP BODY
  IN = {max, temp, im(2-1: max-1+1, j_BERT_LOW-1: j_BERT_UP+1),
srtbuf(1: 9)}
  OUT = {temp, srtbuf(1: 9| 9), io(2: max-1, j)}
-----
ESTIMATIONS:
  Execution time: 1.565745 seconds  repeated 1022.0 times  (99.9%)
  Overhead time:  0.002244 (seconds)
  Dataflow speedup: 6.93046 times with 8 processors  (85.5129%)
  Estimated number of iterations : 1022
NOTE:   Efficiency of DO loop depends on boundary values
Parallelization is estimated as: HIGH EFFICIENT

```

Parallelization of this DO loop is estimated as high efficient. But there are two recursions not allowing *BERT* to parallelize it. One may discover that these recursions are false recursions (see *Avoiding Recursion* at the beginning of this chapter.) First, variable *TEMP* is suspected to be recursive because it is impossible for *BERT* to know if this value needed after the DO loop. Fortunately, it is not needed after the DO loop is therefore false recursion. The same is true about array *SRTBUF*. Do loop can be parallelized with following pragma:

```
      MAX=IX  
      DO 200 J = 2,MAX-1  
C$BERT PARD0 OUT = {io(2: max-1, j)}  
      DO 200 I = 2,MAX-1
```

The OUT field here is presented to disable warnings.

This pragma was generated by running *BERT* with the `-pio[el]f` option. The pragma produced by *BERT* was copied to the original program. The original program was then processed by *BERT*. The variables descriptions for `TEMP` and `SRTBUF` were simply deleted.