



Crescent Bay Software Corporation

VAST-F/toOpenMP

Automatic Parallelizer for Fortran

May 2010

Version 2.7

Preface

Document Number V96041: VAST-F/toOpenMP User's Guide

This is a guide to the use of VAST-F/toOpenMP. VAST-F/toOpenMP is an automatic parallelizer for Fortran programs on SMP systems. This manual is designed to give Fortran programmers an understanding of VAST-F/toOpenMP's capabilities and effective use.

Revision Record

Edition	Date	Description
2.1	2/01	First release of document in this form.
2.2	8/03	Corrections and updates.
2.3	11/03	Corrections and updates; addition of inlining information.
2.4	9/04	Minor corrections and reformatting.
2.5a	2/05	Added description of vifort driver.
2.5b	11/09	Added description of -H option.
2.6	11/09	Added descriptions of -du and UNROLL and LSTVAL directives.
2.7	5/10	Added description of the -vnoexpand_includes option.

Notices

Copyright (C) 2001, 2010 Crescent Bay Software Corp. No unauthorized use or duplication is permitted. All rights reserved.

VAST is a registered trademark of Crescent Bay Software Corp.

Crescent Bay Software Corp., 10950 Washington Boulevard Suite 230,
Culver City CA 90232 USA. Fax: (310)-836-7313 Phone: (310)-836-5183.

Web: <http://www.crescentbaysoftware.com>.

Table of Contents

Preface	i
1. Introduction	8
Overview	8
Automatic Parallelization	8
Expectations	8
Optimizations	9
Files.....	9
User interaction - Automatic Parallelization	9
How to use this guide.....	10
Optional features	10
Potential Error Detection.....	10
2. Invoking VAST-F/toOpenMP	11
Driver command Line.....	11
vastfomp command Line	11
VAST-F/toOpenMP Switches	12
Transformation control with -e (enable) and -d (disable)	13
Additional transformation switches: -g and -f.....	15
VAST-F/toOpenMP Options.....	15
3. User directives	17
Overview	17
VAST directive format.....	17
VAST directive summary	18
Transformation directives	19
NOCONCUR/CONCUR.....	19
SKIP.....	19
CNCALL	19

NOASSOC/ASSOC.....	19
INNER.....	19
SELECT.....	20
UNROLL.....	20
LSTVAL.....	20
Data dependency directives.....	20
NOSYNC/SYNC.....	20
NOEQVCHK/EQVCHK.....	21
PERMUTATION.....	21
RELATION.....	21
Listing directives.....	21
NOLIST/LIST.....	21
SWITCH directive.....	21
Inlining directives.....	22
Specifying directives on invocation.....	22
4. VAST listing	23
Overview.....	23
Input source listing.....	23
Diagnostic messages.....	24
Optimization inhibition messages.....	24
Informative messages.....	24
Error messages.....	25
Output source listing.....	25
Summaries.....	25
Listing control switches.....	25
Listing page length.....	27
5. Parallelizing Calculations	28
Overview.....	28
Creating and executing parallel regions.....	28
Private or shared.....	28
Conditional parallelization.....	30
Inner loops.....	31
Reductions.....	31
Parallelizing loops with external calls.....	32
Parallel cases.....	32

Extending parallel regions	34
Parallel array syntax	35
6. Loop optimizations	37
Overview	37
Loop fusion	37
Loop rerolling	38
Loop peeling	38
IF loops to DO loops	39
DO WHILE to DO	40
Loop interchange.....	40
7. Data dependency analysis	42
Overview	42
Ambiguous subscript resolution.....	42
Data dependency directives.....	43
NOSYNC -- declaring non-recursion	43
NOEQVCHK -- non-recursion in equivalences	43
RELATION -- specifying relationship between variables	44
PERMUTATION -- declaring safe indirect addressing	45
Scalar dependencies	45
Carry-around scalars	45
Equivalenced scalars.....	46
8. Interprocedural Analysis	47
Parallel calls	47
Examples.....	48
Performance benefit	49
Error detection with IPA	49
9. Inline expansion	51
Overview	51
Introduction	51
Automatic inline expansion.....	52
Automatic inlining criteria	52
Avoiding inlining.....	52
Automatic inlining level.....	53

Explicit inline expansion	53
Source code access.....	53
Inline expansion directives	53
AUTOEXPAND directive	53
Explicit mode	54
EXPAND directive	54
NEXPAND directive.....	54
SEARCH directive.....	55
Where to get the source code.....	55
Same file	55
Explicitly named file.....	55
Implicitly named file.....	55
Possible problems.....	56
Separate compilation	56
Code size	56
Debugging.....	56
Compilation rate	56
Nested expansion	57
Analysis inhibitors	57
Expansion inhibitors.....	57
Inhibitors specific to automatic expansion mode.....	57
Inline expansion abilities.....	58
Naming conventions.....	58
Common blocks.....	58
Arguments	59
Unique names.....	59
Labels.....	59
Returns and return values	59
Inlining ENTRYs	59
Cleanup of inlined code	59
Example.....	60
Expansion of SAVE and DATA	61
Inline expansion user messages	62
Inline expansion summary	62
User messages.....	62
10. Output formatting	64
Option switches	64
Output format parameters	68
Label renumbering.....	68

Label alignment.....	68
Indentation.....	69
Inline comments.....	70
Continuation lines.....	70
Output line length.....	70
FORMAT and DATA statements.....	71
Example.....	71
11. Diagnostic Messages	73
Overview.....	73
Data dependency conflicts.....	73
Translation diagnostics.....	75
Statement types.....	75
Branches.....	76
External references.....	77
DO statement.....	77
Miscellaneous.....	78
Warnings.....	78
Potential Errors.....	78
Obsolescent Features.....	79
Syntax errors.....	79
Internal errors.....	79
Directive errors.....	79
Notes.....	79
12. Further Information	81
Crescent Bay Software and VAST.....	81
Questions or comments.....	82
Index	83

1. Introduction

Overview

This document describes the features and options of the VAST-F/toOpenMP parallel processing product. VAST-F/toOpenMP is a software tool that optimizes Fortran programs for execution on SMP parallel systems by inserting OpenMP directives and restructuring loops. VAST-F/toOpenMP optimizes Fortran 77, Fortran 90, and Fortran 95 programs.

Automatic Parallelization

VAST-F/toOpenMP allows you to automatically adapt existing codes to use the multiple processors of an SMP system. You can just run your Fortran code through the system and automatically optimize the loop nests. The automatic parallelization of Fortran from existing programs is a very useful tool, but it is important to point out that additional tuning of the generated code will generally be helpful in getting full performance from the system. This can include adding some assertion directives which may help the automatic parallelization.

Expectations

How fast should you expect your application to run on a parallel system? For automatic parallelization, this can vary greatly from one application to the next. You may see very little speedup, or you may see speedup that scales with the number of processors in your system, or anything inbetween. Applications that parallelize well generally spend most of their CPU time in nested loops. Most applications that do not parallelize

well as they are originally written can be restructured to achieve good speedups on parallel systems, and can be improved by using either hand-inserted OpenMP directives or VAST assertion directives, and in some cases hand-restructuring important loop nests.

Optimizations

VAST-F/toOpenMP's optimizations include:

- Parallelization of loops containing reduction operations (such as global sum functions).
- Parallelizing multiple array dimensions at the same time.
- Restructuring loops to allow parallel execution.

The following sections cover how to run the product, the available options and switches, and more detailed technical information.

Files

When VAST-F/toOpenMP processes a Fortran program, it creates two files. One is a listing of the input program with diagnostic comments added to tell which loops were not optimized and why. The other is an enhanced version of the input Fortran program containing directives and restructured code. This file is ready for compilation by an OpenMP-compliant Fortran compiler.

User interaction – Automatic Parallelization

VAST-F/toOpenMP is intended for use primarily as an automatic tool; at a minimum, you need to know only how to invoke it (see section 2). However, because of the complexity of the transformations involved and the unavailability of some important data at compile time, the added optimizations VAST performs may not significantly decrease the input program's execution time, when you are relying on automatic parallelization.

If the execution time has not decreased, enable VAST's listing and look at the diagnostic messages; referring to the relevant User's Guide sections, you may be able to improve the optimization by switching on or off certain transformations or default assumptions, inserting directives, or

minor code modifications. In some cases, you may be rewarded with dramatically improved performance for a relatively small effort.

When using automatic parallelization you would normally take a “bottom up” approach, by examining the most time-consuming loop nests and making sure they were being parallelized.

How to use this guide

Section 2 describes how to invoke VAST-F/toOpenMP. If you want to use VAST as a strictly automatic tool, you can skip the remainder of the guide beyond section 2. Switches and options are described in section 3.

Sections 4 and 5 discuss communication with VAST -- VAST’s listing and messages, and ways to guide VAST’s action (user directives).

Of interest for automatic parallelization, concepts and rules of VAST’s optimization techniques are discussed in sections 6 through 9. Examples in these sections illustrate optimizable and unoptimizable loops.

Optional features

Most VAST-F/toOpenMP features are turned on by default, but some are not. A few of the more important ones are presented briefly to make sure you are aware of them.

Potential Error Detection

To detect potential errors, use the `-pr` switch.

2. Invoking VAST-F/toOpenMP

Driver command Line

If you are using VAST-F/ToOpenMP in “performance” mode, as opposed to source translator mode, you can simplify compilation by using the `vifort` driver. This acts like the `ifort` compiler, but invokes VAST before compilation. The driver is invoked by the command:

```
vifort [ifort_options] [-Wv,VAST_options] input1  
[...inputn]
```

Vifort invoked with no options will display a man page.

Example:

```
vifort -c -Wv,-ei mysub.f
```

This produces `mysub.o`, by first running VAST-F/ToOpenMP on `mysub.f` (specifying VAST option `-ei`), then compiling the resulting translated source to `mysub.o`. The translated source is discarded.

vastfomp command Line

VAST-F/toOpenMP is executed directly by the command:

```
vastfomp [-o output] [-l listing] [options]  
input1 [...inputn]
```

output = compilable output file. The default output file name is the input file name prefixed by V.

listing = VAST listing file. A null parameter indicates the listing should go to the terminal. Unless this parameter is used, no listing is produced.

options = VAST options and switches. Switches (on/off toggles) are passed with lower case parameter names. Options (numbers or names) are passed with upper case parameter names. All switches and option names are a single letter. See following sections for more information on switches and options.

input1...inputn = Fortran source input files.

VAST invoked with no arguments will print a short usage summary.

VAST invoked with the '-v' parameter will print out the version number.

Example 1:

To run the Fortran source file `crunch.f` through VAST:

```
vastfomp crunch.f
```

The optimized output is sent to `Vcrunch.f`, and the listing to standard output.

Example 2:

To run `sub.f` through VAST, save the VAST listing in file `sub.lst`, and the translated code in `sub.v.f`:

```
vastfomp -l sub.lst -o sub.v.f sub.f
```

VAST-F/toOpenMP Switches

VAST allows the following switches (which pass a string of alphanumeric switches to toggle) on invocation:

Control of transformations (-e and -d, and -g and -f invocation parameters):

[-e switches] [-d switches]

-e Transformation options to enable. (below)

-d Transformation options to disable. (below)

[-g switches] [-f switches]

-g Additional transformation options to enable. (below)

-f Additional transformation options to disable. (below)

Control of listing (-p and -q invocation parameters):

[-p switches] [-q switches]

-p Listing options to enable. (Section 4)

-q Listing options to disable. (Section 4)

Control of output formatting (-r and -n invocation parameters):

[-r switches] [-n switches]

-r Output formatting options to enable. (Section 10)

-n Output formatting options to disable. (Section 10)

Switches specifying global actions can be passed to VAST in the processor invocation command (as described above) or via the SWITCH source directive.

Transformation control with -e (enable) and -d (disable)

The table below shows the switches that affect the transformation of the input program.

Transformation control switches

Switch	Description	Default
a	Allow associative transformations.	on
c	Parallelize loops (Concurrency).	on
d	Don't ignore potential data dependencies.	on
e	Examine EQUIVALENCEs for data dependency.	on
f	Allow free format input.	off
h	Allow detection of parallel cases.	on
i	Parallelize inner loops	off
k	Treat D in Column 1 as a comment (OFF: D=blank).	on
l	Transform IF loops to DO loops.	on
n	Skip all optimizations and transformations.	off
u	Save last values of promoted scalars and indexes.	on
x	Create optimized source file.	on
0	Do threshold testing on parallel regions.	on
1	Convert array syntax to DO loops	on
5	Enable interprocedural analysis	off
7	Enable automatic inline expansion	off
8	Search for inlined routines within the current source file	off

As an example, `-del` causes EQUIVALENCE statements not to be examined for data dependency analysis, and IF loops not to be converted to DO loops. Note that some switches duplicate or overlap the functions of directives. For example, the `-dd` switch is equivalent to the NODEPCHK directive with file scope (`CVD$F NODEPCHK`).

Notes on transformation switches:

a: Permit associative transformations. `-da` is equivalent to the `NOASSOC` directive with file scope.

c: Do concurrency (parallelization) analysis. `-dc` is equivalent to the `NOCONCUR` directive with file scope.

d: Don't ignore potential data dependencies. If you want to tell VAST-F/toOpenMP that all potential dependencies in your program are not actual dependencies, use the `-dd` switch (this can cause wrong answers if your assertion is incorrect). For more information, see section 7.

e: Examine `EQUIVALENCE` statements for data dependency.

h: Enable parallel case optimization. To disable the automatic recognition of parallel cases for situations where code regions are completely independent of each other, use the `-dh` switch.

k: Treat `D` in column 1 as a comment character. If this switch is off, a `D` in column one is treated as a blank. This switch provides compatibility with a debugging feature of some compilers.

l: Transform `IF` loops to `DO` loops.

n: Do not do any optimizations or transformations. Like a global `SKIP` directive.

u: By default VAST-F/toOpenMP performs analysis to determine if scalar variables that hold changing values in a loop may be used following execution of the loop, i.e. their last values must be retained. In some cases (particularly if the scalar is an array element) VAST is too conservative and saves the last value unnecessarily. `-du` globally suppresses last-value saving.

x: Create an optimized source output file. This switch may be turned off if the diagnostic listing only is wanted. Turning this switch off may speed compile time and reduce disk space used. The setting of this switch does not affect the listing of the transformed source in the listing file.

0: Disabling this option will suppress the `IF` clause (threshold test) generated for some parallel loops/regions.

1: Convert array syntax to `DO` loops.

Note that some switches correspond to directives and some do not. Those that correspond to directives (`a`, `c`, `d`, `e`) may be toggled (via the `SWITCH` directive) more than once within a routine (although the preferred method is to use the corresponding directive directly). Those which do not can have only one valid setting for any one routine; if they are set more than once within a routine, only the last setting is used.

The `8` and `x` switches are valid only in the invocation command.

Additional transformation switches: -g and -f

Additional transformation control switches are enabled by `-g` and disabled by `-f`. The available switches are:

e: input is free format. Default for `.f` extension: OFF. Default for `.f90` extension: ON.

g: Parallelize loops with private arrays, where potential first values are a problem. See section on parallelization. Default: OFF.

j: Expand INCLUDE files and constant parameters. Default: OFF.

u: Allow 132-column fixed format input. Default: OFF.

6: Allow inline expansion of routines with SAVE statements.

VAST-F/toOpenMP Options

The options can have various parameters, including:

routines = names of Fortran subroutines/functions, separated by commas.

filenames = names of Unix file names, separated by commas.

nnn =integer constant

The available general options are:

-C *routines*

Names of concurrently-callable external routines.

-D *directive[:routine,routine...]*

Directive to process at invocation.

-F *filename*

File name to divert command line processing to. The `-F` parameter allows redirection of command line input to a file. This is useful when many options are specified (e.g. `-D` parameters), or to insure a uniform set of invocation options over many invocations.

-H *path*

Path on which to search for include source files and module information files.

-On

Specify optimization level n , where n is either 0 or 3. Default level is 3. At optimization level 0, no optimizations are performed except for inserting OpenMP directives.

-Ppage *nnn*

Page length (lines) for non-terminal listing (use with `-qt`) (default 66).

-Vnoexpand_includes

Don't expand the source for include files in the output source file (leave the include statement as-is).

-Z *output format parameter=value*

Output formatting parameters. See section 10 for allowable parameters.

Options that apply specifically to inline expansion (for more detail, see section 9):

-I*outines*

Names of subprograms to inline when references to them are encountered.

-J*n*

Depth of nested inlining to perform (default is 1).

-M*nnn*

Limit on source lines of auto-inlined routines (default is 50).

-N*outines*

Subprograms not to be inlined.

-S*ilenames*

Files in which to search for source of subprograms to be inlined.

-Y*outines*

Names of subprograms to inline (along with all subprograms they call, directly and indirectly) when references to them are encountered (i.e. nested inlining).

3. User directives

Overview

You may sometimes have information about the structure of a program and about its data that is unavailable to VAST-F/toOpenMP through inspection of individual program units. For this reason, a way for you to guide VAST is supplied via user directives. User directives are treated as comments by Fortran compilers, thus preserving code transportability.

VAST directive format

VAST-F user directives have `CVD$` (fixed format) or `!VD$` (free format) in the first four columns of a line. Following the `$` is an optional scope parameter. `F` stands for "file" (meaning the directive applies until the end of all input files), `R` stands for "routine" (directive applies until the end of the current routine), and `L` for "loop" (directive applies to the next loop encountered). A blank following the `$` is equivalent to `L`. Some directives ignore the scope parameter.

Directives affecting `IF` loops must have `R` or `F` scope; directives with `L` scope apply only to `DO` loops.

The body of the directive begins after one or more blanks. Many directives can be preceded by `NO`, thus effecting the reverse operation.

<code>CVD\$</code>	<code>NODEPCHK</code>	<i>(Ignore potential data dependencies in the next loop.)</i>
<code>CVD\$R</code>	<code>SKIP</code>	<i>(Turn off transformation in the rest of this routine.)</i>

VAST directive summary

The full set of directives is summarized in the table below. The "scope" entry is either I for "immediate," meaning that the directive applies immediately; L, meaning that it applies to the next loop; R, meaning that it applies to the whole routine; or LRF, which means that any of the loop, routine, or file options can be used to control the scope.

A short description of each of these directives follows the table. In addition, the more important directives are discussed in detail at the appropriate points in the sections on optimization.

VAST-F/toOpenMP directives

Directive	Function	Default	Scope
SKIP/ NOSKIP	Disable/reenable transformations	NOSKIP	LRF
NOSYNC/ SYNC	Do/don't ignore potential overlap of array sections.	SYNC	LRF
NOCONCUR/ CONCUR	Disable/enable parallelization.	CONCUR	LRF
SELECT	Specify desired optimization mode for loop.	none	L
CNCALL	Allow concurrent calls in loop.	n/a	LRF
SWITCH	Pass new global switches.	n/a	I
NOASSOC/ ASSOC	Don't/do perform associative transformations.	ASSOC	LRF
NOUNROLL/ UNROLL	Don't/do unroll loops.	UNROLL	LRF
NOLSTVAL/ LSTVAL	Don't/do save last values of temporaries.	LSTVAL	LRF
NOEQVCHK/ EQVCHK	Don't/do check EQUIVALENCES to see if they cause data dependencies.	EQVCHK	LRF
PERMUTATION	Pass list of integer arrays that have no repeated values.	n/a	R
RELATION	Specify relationship between two simple variables.	n/a	R
NOLIST/ LIST	Turn off/on listing.	LIST	I
COUNT	Supply iteration count for loop.	n/a	I
ITERATIONS	Supply iteration count for classes of loops.	n/a	R

Transformation directives

These directives are used to change the way VAST-F transforms a loop.

NOCONCUR/CONCUR

NOCONCUR disables conversion of loops to concurrent (parallel) form. CONCUR serves only to toggle back from NOCONCUR; it does not force conversion (see SELECT). The `-dc` switch is equivalent to NOCONCUR with file scope. NOCONCUR is a subset of SKIP.

SKIP

SKIP causes VAST-F to avoid transformation for the directed loop or routine. This is the directive to use if you want VAST-F/toOpenMP to leave a loop untransformed. NOCONCUR is a subset of SKIP.

CNCALL

CNCALL asserts that any subroutines called in a loop have no recursive side effects, and can be called concurrently by separate iterations of the loop.

NOASSOC/ASSOC

By default, VAST-F transforms certain constructs into vector or concurrent versions in which the order of operations may be different than the original (they have been associatively transformed). Because of the way numbers are internally represented in computers, this operation reordering may result in answers that differ slightly from the scalar original.

The NOASSOC directive disables all associative transformations, including generating reductions (such as sum or dot product of arrays), and operation reordering when minimizing dependent regions.

The `-d a` switch is equivalent to NOASSOC with file scope.

INNER

By default VAST-F/toOpenMP parallelizes only loops that are *not* innermost loops. In some cases, for example when it is known the DO trip count is very large, it may be desirable to parallelize an inner loop or loops. This can be enabled either via the command line option `-ei` or in the source via the INNER directive.

SELECT

When you want to override VAST's default choice heuristics, you can use the `SELECT` directive to indicate a preference for how a loop is optimized (or not). The `SELECT` directive takes a parenthesized argument that indicates the desired mode; for VAST-F/toOpenMP, the only allowable argument is `CONCUR`, to indicate that this loop should be parallelized if possible. (To suppress parallelization, use the `NOCONCUR` directive. Note that `CONCUR` and `SELECT (CONCUR)` have different meanings; `CONCUR` merely toggles back from `NOCONCUR`, and specifies that VAST should resume auto-parallelization; `SELECT (CONCUR)` specifies that a particular loop should be parallelized if possible.)

UNROLL

VAST-F/toOpenMP in general does not unroll loops, but in certain circumstances may completely unroll inner loops with small constant iteration counts. To suppress this transformation use the `UNROLL` directive.

LSTVAL

Use `NOLSTVAL` to suppress saving of last values of loop-dependent scalars. (See explanation under option switch `-du`.)

Data dependency directives

These directives are used to help VAST-F decide whether data dependency conflicts actually exist in a loop. If you know that an operation is not recursive, you can supply one of these directives to inform VAST-F. (Data dependency directives are discussed further in a later chapter.)

NOSYNC/SYNC

When elements of an array are modified within a loop, VAST-F must determine the exact storage relationship of these elements to all other references to the array in the loop. This must be done to ensure that the references do not overlap, and thus can safely be executed in parallel. When the relationships cannot be determined, VAST issues a potential dependency diagnostic. The `NOSYNC` directive asserts that all such potentially recursive relationships are in fact not recursive. It does not, however, force the optimization of operations that are unambiguously recursive. The `SYNC` directive is used only to toggle back to the default state.

NOEQVCHK/EQVCHK

NOEQVCHK directs VAST-F to ignore relationships between variables caused by EQUIVALENCE statements, when examining the data dependencies in a loop. The `-d e` switch is equivalent to NOEQVCHK with file scope.

PERMUTATION

The PERMUTATION directive declares an integer array to have no repeated values. This is useful when the integer array is used as a subscript for another array (indirect addressing). If it is known that the integer array is used merely to permute the elements of the subscripted array, then it can often be determined that no feedback exists with that array reference.

RELATION

The RELATION directive advises VAST-F that a specified relationship exists between two integer variables or between an integer variable and an integer constant. This information may be useful to VAST-F in resolving otherwise ambiguous array relationships.

Listing directives

NOLIST/LIST

Listing of the input source can be selectively suppressed with the NOLIST/LIST directive pair. If NOLIST (or the `-q 1` switch) is in force when the END statement is encountered, the rest of the listing (messages, translated source, summaries) is suppressed unless specifically enabled via `-p` switches.

SWITCH directive

You can set (or change) global switches with the SWITCH directive. SWITCH directives may be inserted into the source program. The format is:

```
CVD$ SWITCH[, -e ss][, -d tt]  
      [, -g uu][, -f vv]  
      [, -p ll][, -q kk]  
      [, -r zz][, -n xx]  
      [, output format_parameters]  
      [, -P nn], -C routines]
```

```
[,-I routines][,-N routines]  
[,-Y routines]
```

corresponding to the invocation switch parameters described in the previous chapter.

Inlining directives

For AUTOEXPAND, EXPAND and NEXPAND, please see section 9.

Specifying directives on invocation

The `-D` invocation parameter can be used to specify directives without inserting them in the actual input source code. The format is:

```
-D directive[:routine,routine,...]
```

Where `directive` is any VAST-F/toOpenMP directive, and `routine` is a routine in the input source to which the directive is to be applied. If no routine names are supplied, the directive applies to the entire input source. Multiple `-D` parameters can be supplied. Optionally, `-D` parameters can be placed in a file and invoked with the `-F` command line parameter.

4. VAST listing

Overview

Optimizing Fortran loops for parallel execution is a complex task, and to do the best possible job, VAST-F/toOpenMP may require some assistance from the user. Two paths of communication are available for this purpose: (1) VAST informs you of the actions it takes on the program (which loops were optimized, which loops were not optimized and the reasons for their rejection); (2) you can pass information and commands to VAST via directives inserted into the program (see section 2), or via global switches on the VAST invocation command (see section 2).

The full VAST listing consists of four parts: a listing of the input source with a graph of the loop structure showing the disposition of each loop; a block of diagnostic messages; a listing of the output transformed source; and summaries of loops and overall statistics. Any part of the listing can be separately enabled or disabled via the listing switches shown in the table below. An example of a full listing is also given below.

Input source listing

The input source lines are numbered on the listing. Source lines coming from `INCLUDES` are numbered as well. These line numbers are used in the messages, output source listing, and summaries. The codes used for the loop disposition graph are listed below:

Code	Meaning
A	No action requested. (Optimization turned off.)
D	Data dependent. (Parallelizing loop could give wrong answers.)
E	Deleted. (Results of the loop are never used.)
G	User parallel. (Explicit parallel directives used on this loop.)
H	Too short, not enough iterations.
M	Parallel case. (Two sections of code will be executed in parallel.)
N	Not chosen. (Loop is not optimized.)
P	Parallelized. (OpenMP parallel directive(s) inserted.)
R	Unrolled. (Several iterations will be calculated each pass.)
T	Translation problem. (No optimization for this loop.)
V	Optimized. (Superscalar optimizations performed on this loop.)
Z	Inner loop parallelized. (OpenMP parallel directive(s) inserted.)

Diagnostic messages

VAST's diagnostic messages appear in a group at the end of the source listing. Each message includes the line number and (if relevant) a variable name. These messages are VAST's means of notifying you of what problems it encountered in optimizing the loops in the source program. There are several different types of diagnostics.

Optimization inhibition messages

Translation Diagnostic. Describes a problem or potential problem in executing a loop in parallel. May prevent loop from being optimized.

Data Dependency Conflict. A real or potential feedback from one loop pass to the next prevents the safe use of vector or parallel operations. Potential feedback may result in generation of alternate versions of the loop. Otherwise feedback may prevent at least part of a loop from being optimized.

Informative messages

Warning Message. Some potentially troublesome input has been encountered.

Note Message. Tells about some opportunity or action on the input that might be of interest.

Error messages

Syntax Error. A construct that is not legal in Fortran has been encountered. No translation is done for this program unit.

Internal Error. An internal problem with VAST-F has been detected. No further processing is done for this subprogram; translation is attempted again for the next subprogram in the input file. Please report the error.

You can suppress each of these types of messages independently via the `-q` parameter on the VAST-F command line or on the `SWITCH` directive (see section 3). A later section a list of some of VAST-F's diagnostics, with further explanation of the messages, and tips on avoiding certain problems.

Output source listing

The output source code is listed following the messages. It shows the translated code. The numbers in the column at the right edge correspond to input source line numbers, to help in comparing the transformed source to the original source.

Summaries

Following the listing of the transformed source are two summary tables. One table (Loop Summary) summarizes the action taken for each loop in the routine. `%CD` is the percentage of code within the loop that is conditional and `%DP` is the percentage that is dependent. The final table (Event Summary) gives overall counts of errors, diagnostics, and loops transformed.

Listing control switches

The table below shows the switches that control the format of the listing file. These switches can be used either on the invocation (for example, `-q h`) or on the `SWITCH` directive (e.g., `CVD$ SWITCH, -q h`).

Listing control switches

Switch	Description	Default
b	List input line #s in columns 73-80 of output listing.	on
c	List data dependency conflict messages.	on
e	List event summary at end of routine.	on

f	List fatal error messages.	on
g	List translation diagnostics.	on
h	List input source lines.	on
i	List included lines.	on
l	Produce a listing.	on
n	List translated code.	on
p	List loop summary at end of routine.	on
r	Detect and list potential errors	off
t	Terminal listing: format output for 80 columns.	on
u	Show extent and disposition of loops in source.	on
w	List warning messages.	on
y	List syntax errors.	on

As an example, if you wanted to get a 132-column printer listing with no warning messages and no event summary, you would specify `-q t we`. If you wanted to get potential errors listed, you would use `-p r`.

Notes on the listing switches:

b: List corresponding input line numbers in columns 73-80 of the listing of the transformed source. This switch is valid only if the `n` switch is on. This listing feature is useful in relating transformed source lines to original source lines.

i: List lines that come from `INCLUDED` files. When this switch is on, source lines obtained from `INCLUDE` files are listed. They are identified by a dash following the line number. This switch is valid only if the `h` switch (list source lines) is on.

l: Produce a listing. `-q l` suppresses all parts of the listing (except the initial header, if the `t` switch is on). `-q l` is equivalent to the `NOLIST` directive.

r: VAST-F/toOpenMP can statically trace all uses and definitions of variables to look for potential programming errors. It generates messages on uses of variables that are undefined and definitions that are unused. This catches many programming errors, such as misspelled variable names, undefined arguments, missing, misplaced or redundant statements, and missing common declarations. This facility is turned on with the `-pr` switch; it is off by default.

t: Format the listing for a terminal. `-q t` results in a wide-format listing file, with printer control, pagination, and page headers, suitable for a 132-column line printer.

u: Show extent and disposition of loops in the input source listing.
"Draws" a bracket alongside each loop, indicating how VAST-F treated the loop (parallelized, too short, data dependent, and so forth).

Listing page length

If you want a paginated, wide format listing suitable for a line printer, use the `-qt` option. The page length defaults to 66 lines. To change the number of lines per page, use the `-Ppage` parameter. For example,

```
-Ppage 60
```

changes page length to 60 lines per page, when getting a paginated listing (`-qt`).

5. Parallelizing Calculations

Overview

Parallelization is the automatic distribution of loop iterations to multiple processors (or tasks). (This is also known as *concurrency analysis*.) VAST-F/toOpenMP parallelizes loops by inserting OpenMP parallelism directives.

Creating and executing parallel regions

When VAST selects a loop for parallelization, it attempts to combine it with as many other parallel loops as possible into a larger parallel region, in order to reduce overhead. There may be some redundant scalar code between parallel loops in a parallel region.

When a parallel region is identified, VAST inserts OpenMP directives to delineate it.

Private or shared

Variables that are local to a parallel region (do need to get values from before the region or pass values after the region) need to have a private copy created in each processor. Variables that are global to a parallel region (values come from outside the region, for instance from COMMON) must be shared with all other processors that are doing work in this parallel region.

When creating a parallel region, VAST designates as shared all variables that need to be shared among the tasks, and declares as private all variables that need to have a private copy for each processor. (Since the default is shared, these variables are not actually named on the directive.) This way, processors will be using the same addresses (passed in or from common) to access shared items, and private items will be allocated on each processors stack and will thus be different in each processor.

In some cases, the source code is restructured to avoid the use of certain variables that cannot be conveniently shared among tasks. In the loop below, *t* must have a copy in each processor, so it is declared private in the generated parallel region. *i* and *j* also must be private. The assignment of *k* is eliminated, as its values can be calculated as a function of the loop index, and its value is not needed after the loop.

```

subroutine example1
common a(5000), b(5000), d(5050), kk
k = kk
do i = 1, 5000
    k = k + kk
    t = sqrt(a(i)) + d(k)
    b(i) = t + 1.0/t
enddo
return
end

```

Translation:

```

!$OMP PARALLEL DO PRIVATE(t)
do i = 1, 5000
    t = sqrt(a(i)) + d(k+i*kk)
    b(i) = t + 1.0/t
end do

```

In the example below, *e* is an array that needs to have a private copy in each task. It is declared private in the parallel loop to accomplish this; *e* is not shared, but is called a private array.

```

common b(99,99), c(99,99), d(99,99)
real e(99)

```

```

do j = 1, m
    do i = 1, n
        e(i) = b(i,j) + c(i,j)
    enddo
    c(1,j) = 0.
    c(n,j) = 1.
    do i = 1, n
        c(i,j) = e(i)*c(i,j) - e(i)/d(i,j)
    enddo
enddo

```

This loop is parallel

"e" needs to be private

Translation:

```

!$OMP PARALLEL DO PRIVATE(e)
do j = 1, m
    do i = 1, n
        e(i) = b(i,j) + c(i,j)
    end do
    c(1,j) = 0.
    c(n,j) = 1.
    do i = 1, n
        c(i,j) = e(i)*c(i,j) - e(i)/d(i,j)
    end do
end do

```

Conditional parallelization

If a loop is suitable for parallelization except that it is potentially dependent, VAST may generate an IF-THEN block in the same way as for the threshold test. When evaluated at run time, this test determines whether the loop can execute correctly on multiple processors, or must be run on a single processor. For single and double-nested loops, this test is combined with the threshold test.

You can use the NOSYNC directive to assert that there is no overlap in the array references in the loop, and thus suppress the IF test and duplicated loop(s).

Inner loops

If the `INNER` directive or `-ei` option switch is enabled and no outer loop is available, inner loops will be analyzed for parallelization. By default, this switch is disabled. However, inner loops that clearly exceed the threshold value are automatically parallelized even if inner loops are not requested.

If there are some inner loops between large parallel loop nests, you may want to try using the `INNER` directive on them to parallelize them and expand the parallel region to include more of the code. As long as the parallel region will be started anyway, it is better to do as much useful work in parallel mode before returning to serial execution.

Reductions

VAST does not parallelize loops containing dependencies between loop iterations (data dependencies), except for certain reduction operations (for example, summation or dot product). Reduction operations are parallelized by giving each task a partial reduction to perform, and combining the partial results as each task finishes. The results are combined in a “critical region” where only one processor can execute at a time, indicated indirectly by the `REDUCTION` clause on the `PARALLEL` directive. This is done as in the example below:

```
subroutine reduct ( a, m, n, s )
  real*8 a(1000,1000)
  do 200 j = 1, m
  do 200 i = 1, n
    s = s + a(i,j)
  200 continue
  print *, s
end
```

Translation:

```
!$OMP PARALLEL DO IF (m*n .gt. 666) REDUCTION(+:s)
  do j = 1, m
    do i = 1, n
      s = s + a(i,j)
    end do
  end do
```

Parallelizing loops with external calls

VAST may parallelize loops containing subroutine calls and non-intrinsic function references if you insert a `CNCALL` directive (concurrent call) or use the `-C` invocation parameter. This asserts that any external routines referenced in the loop may safely be called in parallel (they don't modify data referenced in other iterations of the loop). The `-C` invocation parameter specifies names of specific routines that may be called in parallel wherever they are referenced. The directive below will allow the loop to be parallelized.

```
      subroutine cncallit ( a, x, n )
      real a(10000), x
cvd$  cncall
      do i = 1, n
          call subr ( a(i), x )
      enddo
      end
```

Translation:

```
!$OMP PARALLEL DO
      do i = 1, n
          call subr ( a(i), x )
      enddo
```

`CNCALL` automatically implies `INNER` if it is applied to an inner loop, as in the preceding example.

VAST assumes that all passed variables not explicitly defined in the loop are shared. You should check your potential concurrently calls routines to see that this is the case, and that they have no other dependencies that would lead to wrong results.

Parallel cases

When parallelism cannot be found within a loop nest, VAST attempts to find loops or loop nests that are completely independent of each other, and make them parallel cases. In addition, at times VAST will split up loops into sections and make each section a parallel case. In the example below, the first loop nest (202) is completely independent of the second loop (302). VAST generates OMP directives such that one processor will

execute the first nest, while a second processor will concurrently execute the second nest.

```
      DO 202 J = 1, M      Parallel case 1
        DO 201 I = 1, N
          A(I,J) = (A(I-1,J)+A(I,J-1))*0.5
201      CONTINUE
202      CONTINUE
      NK = N - K
      ML = M + L
      DO 302 J = 1, MJ    Parallel case 2
        DO 301 I = 1, NK
          C(I,J) = (C(I-1,J)+C(I,J-1))*0.5
301      CONTINUE
302      CONTINUE
```

Translation:

```
!$OMP PARALLEL PRIVATE(J, I, ML)
!$OMP SECTIONS
      DO 202 J = 1, M
        DO 201 I = 1, N
          A(I,J) = (A(I-1,J)+A(I,J-1))*0.5
201      CONTINUE
202      CONTINUE
!$OMP SECTION
      NK = N - K
      ML = M + L
      DO 302 J = 1, MJ
        DO 301 I = 1, NK
          C(I,J) = (C(I-1,J)+C(I,J-1))*0.5
301      CONTINUE
302      CONTINUE
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
```

Extending parallel regions

Transitioning between parallel execution and serial execution takes time, and much overhead can be saved if large amounts of parallel work can be parceled out in one parallel region. In general, it is preferable to have more than one loop nest in a parallel region.

VAST allows redundant code in parallel regions, as it is better to have all the parallel tasks execute the same thing than to have them end the old parallel region, have one processor execute the scalar code, and then start a new region.

Redundant code is allowed between loops and between an inner and outer loop. Only assignment statements are allowed, and dependencies involving scalar variables in the redundant code can prevent expanding the region.

A limit of five assignments are examined between parallel loops to see if they are suitable for redundant execution.

The following example contains serial statements between inner parallel loops; these statements can be executed redundantly.

```
DO I = 1, 999          Parallel loop
  A(I) = B(I) / C(I)
ENDDO
X = B(1)              Can be redundant
DO I = 1, 999        Parallel loop
  B(I) = C(I) * X
ENDDO
R = C(1)              Can be redundant
S = C(2)              Can be redundant
T = C(3)              Can be redundant
DO I = 1, 999
  C(I) = R + S * T
ENDDO
```

VAST-F/toOpenMP will move parallel regions outside of non-tightly-nested outer loops, so that the non-parallel loop will still be included in the parallel region.

```
DO J = 1, 22          Move parallel region outside here
  X = Y + C(J)        Execute redundantly
  DO I = 1, 999      This is the parallel loop
    A(I, J+1) = A(I, J) * B(I, J) + X
```

```
ENDDO
ENDDO
```

These types of optimizations can be combined, so that the parallel region can be moved outside an outer loop, and then combined with other loops in one large parallel region.

Only assignments between loops are allowed (CALLs, I/O, branches, and other statements are not handled), so the two loops in the example below cannot be put into the same parallel region.

```
DO 100 I = 1, 999
    A(I) = B(I) * C(I)
100 CONTINUE
    CALL SUB(Y)           Prevents extended region
DO 200 I = 1, 999
    B(I) = C(I) * X
200 CONTINUE
```

Finally, the following example shows an assignment which cannot be executed redundantly; the value of the shared variable x will be incorrect.

```
DO 100 I = 1, 999
    A(I) = B(I) * C(I) + X
100 CONTINUE
    X = X + SQRT(X)      Prevents extended region
DO 200 I = 1, 999
    B(I) = C(I) * X
200 CONTINUE
```

Parallel array syntax

If the `-e1` switch is enabled (the default), VAST-F/toOpenMP “scalarizes” array syntax and then parallelizes and optimizes the resulting loops. (To disable this feature, use VAST option `-d1`) For example, consider this routine:

```
subroutine arrsyn ( a, b, x )
real*8 a(400,400), b(400,400), x
x = sum(abs(a-b))
end
```

Translation:

```
doubleprecision d1
d1 = 0
!$OMP PARALLEL DO REDUCTION(+:d1)
do j2 = 1, 400
  do j1 = 1, 400
    d1 = d1 + abs(a(j1,j2)-b(j1,j2))
  end do
end do
x = d1
```

6. Loop optimizations

Overview

The transformations in this section all involve the manipulation of single-level loops and/or loop nests to reduce loop overhead or to expose more opportunities for the compiler's instruction optimizations.

Loop fusion

Adjacent loops with identical bounds can often be merged into a single loop. This reduces loop overhead and provides additional instructions for scheduling. There is a maximum of five loops and 50 total lines of fused code; this maximum can be increased with the `-G` parameter which will increase the total lines examined. For example, `-G 100` will examine 100 lines for fusion.

Example:

```
      DO 100 I = 1, N
          A(I) = B(I) + C(I)
100    CONTINUE
C
      DO 200 I = 1, N
          D(I) = B(I) - C(I)
200    CONTINUE
```

Translation:

```

DO 100 I = 1, N
    A(I) = B(I) + C(I)
    D(I) = B(I) - C(I)
100 CONTINUE

```

Loop rerolling

VAST-F/toOpenMP will take loops that have been hand unrolled and put them back into their original state. This may allow the compiler (or VAST-F/toOpenMP) to subsequently unroll the loops to a more optimal level for the target system.

Example:

```

DO 20 I = 1, N, 2
    A(I) = B(I) + C(I)
    A(I+1) = B(I+1) + C(I+1)
20 CONTINUE

```

Translation:

```

DO 20 I = 1, (N+1)/2*2
    A(I) = B(I) + C(I)
20 CONTINUE

```

Example:

```

DO 310 I = 1, 96, 5
    S = S + A(I)*B(I) + A(I+1)*B(I+1) +
*          A(I+2)*B(I+2) + A(I+3)*B(I+3) +
*          *A(I+4)*B(I+4)
310 CONTINUE

```

Translation:

```

DO 310 I = 1, 100          (Rerolled into one dot product.)
    S = S + A(I)*B(I)
310 CONTINUE

```

Loop peeling

VAST can "peel off" loop iterations into scalar statements outside the loop, leaving the bulk of the iterations free of extraneous variables.

Example:

```

im = n
do i = 1, n
    b(i) = a(i) - a(im)
    im = i
end do

```

Translation:

```

im = n
b(1) = a(1) - a(im)
do i = 2, n      (note the new start iteration)
    b(i) = a(i) - a(im)
end do

```

IF loops to DO loops

VAST-F/toOpenMP will turn IF loops into DO loops. This allows the compiler to optimize more easily certain loops, with the exposure of more regular looping structures. It may also create nested loop situations, which can be further optimized.

Example:

```

330  CONTINUE
      I = I + 1
      IF ( I .GT. N ) GO TO 340
      A(I) = 0.0
      GO TO 330
340  CONTINUE

```

Translation:

```

330  CONTINUE
      I = I + 1
      J1S = I
      IF ( N - J1S + 1 .GT. 0 ) THEN
          DO I = 1, N - J1S + 1
              A(J1S+I-1) = 0.0
          ENDDO
      ENDIF
340  CONTINUE

```

To be converted into a DO loop:

- The loop must have a single entrance and a single exit.
- The iteration count for the loop must be determinable at execution time before the loop is entered. The IF loop may contain other loops.

The `-q1` switch disables conversion of IF loops to DO loops. All directives (such as `NODEPCHK`) affecting IF loops must have routine or file.

DO WHILE to DO

DO WHILE statements can be converted to iterative DO loops when a fixed iteration count can be extracted from the loop.

Example:

```
DO WHILE ( I.GT.0 )
  A(I) = 0.
  I = I - 1
ENDDO
```

Translation:

```
DO I1X = 1, I
  A(I+1-I1X) = 0.
ENDDO
I = 0
```

Loop interchange

An outer loop is pushed inside an inner loop if the outer loop is a better candidate for optimization. Greatest preference is given to minimizing strides on arrays -- long strides can give poor cache performance. The outer loop may be pushed inside more than one inner loop.

Example:

```
DO 100 I = 1, 100
  DO 200 J = 1, 10
    A(I,J) = B(I,J)
  200 CONTINUE
100 CONTINUE
```

Translation:

```
DO J = 1, 10
  DO I = 1, 100
    A(I,J) = B(I,J)
```

ENDDO
ENDDO

7. Data dependency analysis

Overview

VAST-F/toOpenMP ensures that the optimized code gives the same answers as the original. For certain loops, parallel or vector execution would result (or could result) in incorrect answers. A loop in which results from one loop pass feed back into a future pass of the same loop is said to have a *data dependency conflict* and may not be completely optimized. (Such a loop is also said to be recursive or to contain recurrences.) In these cases, VAST detects the problem, reports it to the user, and leaves the loop in its original form.

VAST does extensive analysis of the arrays used in each loop nest, and examines the offset and stride of accesses to elements along each dimension of arrays that are both used and stored in a loop. If the uses and stores overlap on different iterations of a loop, or if they could possibly overlap, then there is a data dependency problem. In this case the order of execution of the iterations could change the results, and the order of execution of iterations where a loop is parallelized is not known, so the loop cannot be parallelized. Avoiding dependencies is important in getting the highest performance; this section describes some directives that VAST provides to help you do this.

Ambiguous subscript resolution

When it is not possible with information contained in the loop to determine the storage relationship between two references to an array, the situation is called *ambiguous subscripting* or *potential feedback*. In these

situations, VAST looks for statements outside of the loop that may provide information that clears up the ambiguity. In the loop below, VAST finds the assignment to N2, which makes it clear that N1 is never equal to N2, and recognizes that there is no feedback.

```
      N2 = N1 + 1
      DO 100 I=1,N                               (Optimized.)
100   A(I+1,N1) = A(I,N2)*B(I)
```

Data dependency directives

NOSYNC -- declaring non-recursion

The NOSYNC directive gives you the ability to direct VAST to ignore potential data dependencies in a loop. This capability should be used only when you know that no real recursion exists. When it detects a potential data dependency conflict, VAST issues a message asking you to apply this directive if the loop is in fact not recursive.

Let's look at an example of a situation where you may want to disable data dependency checking. In this loop, VAST cannot be sure that N1 does not equal N2 and thus rejects the loop:

```
      SUBROUTINE MOVE ( A, B, N, N1, N2 )
      REAL A(N,*), B(*)
      DO 3 I = 1,N                               (Not optimized.)
3     A(I+1,N1) = A(I,N2)+B(I)
```

If you know that N1 is never equal to N2, you can insert a directive as shown here:

```
CVD$  NOSYNC
      DO 4 I = 1,N                               (Optimized.)
4     A(I+1,N1) = A(I,N2)+B(I)
```

NOEQVCHK -- non-recursion in equivalences

It is very rare in real-world programs that recursion is hidden through the use of EQUIVALENCE statements. However, VAST must assume the worst and not optimize in situations where EQUIVALENCE statements could cause feedback. In programs where many of the variables are EQUIVALENCE d together, this can result in most of the loops being left unoptimized.

The NOEQVCHK directive is provided to tell VAST that EQUIVALENCE statements can be ignored for data dependency analysis. Use of this directive asserts that variables with different names do not overlap in

storage. (This is almost always the case, anyway.) Equivalence checking can be suppressed for the entire input file by the `-d e` switch on the VAST command line or the `SWITCH` directive.

In the example below, several local arrays have been equivalenced to a large array in common (perhaps to save space). If the arrays could overlap (for instance, if the value of the variable `N` was 1500 in the `DO 100` loop), then the `DO 100` loop could not be optimized. However, as we know the arrays do not overlap, the `NOEQVCHK` directive can be applied to the whole routine.

```
COMMON /BIG/ POOL(100000)
DIMENSION A(1),B(1),C(1)
EQUIVALENCE (POOL(1),A(1)),(POOL(1001),B(1)),
1 (POOL(2001),C(1))
CVD$R NOEQVCHK (Don't worry about equivalences.)
.
.
DO 100 I = 1, N
    A(I+IA) = B(I+IB) + C(I+IC)
100 CONTINUE
```

It is often better to recode the application to avoid `EQUIVALENCE` entirely.

RELATION -- specifying relationship between variables

You can use the `RELATION` directive to provide additional information to VAST about array subscript ranges, to help in determining if a loop is safe to optimize. The `RELATION` directive has the form:

```
CVD$ RELATION ( simple1 .rel. simple2 )
```

where `simple1` and `simple2` are simple integer variables (one of them can be an integer constant), and `rel` is one of the Fortran relational operators `GT`, `LT`, `GE`, `LE`, `EQ`, `NE`.

When VAST cannot otherwise determine whether the relationship between two uses of an array is recursive, it searches the `RELATIONS` supplied by the user for the current routine to see if they help.

`RELATION` directives are informative only and do not force any action.

`RELATION` directives apply for the whole program unit. If conflicting relations are given, the result is unpredictable. It is up to you to ensure that the relations specified are correct and consistent.

```
CVD$ RELATION ( J.GE.N )
. . .
DO 100 I = 1, N (If J .GE. N, no overlap.)
    A(I+J) = A(I) + B(I)
```

100 CONTINUE

The `RELATION` directive is provided for situations where you are unsure if the `NOSYNC` directive (a blanket assertion of non-recursion) is safe, or know it is not, but have some information about relative values of index variables.

PERMUTATION -- declaring safe indirect addressing

When an array with a vector-valued subscript appears on both sides of the equal sign in a loop, feedback is possible even if the subscript is identical. Feedback will occur if there are any repeated elements in the subscripting array.

Sometimes, indirect addressing is used because the elements of interest in an array are sparsely distributed; in this case an integer array is used to point at the elements that are really wanted, and there are no repeated elements in the integer array (see the example below).

This information can be passed to `VAST` through the `PERMUTATION` directive. `PERMUTATION` asserts that the named integer arrays contain no repeated elements (they serve merely to permute the elements of the arrays they indirectly address).

Syntax:

```
CVD$ PERMUTATION ( ia1, ia2, ... , ian )
```

`PERMUTATION` declares the integer arrays (`ia1`, and so forth) to have no repeated values for the entire routine.

```
CVD$ PERMUTATION (IPNT) (IPNT has no repeated values.)
```

```
...
```

```
DO 100 I = 1, N
```

```
    A(IPNT(I)) = A(IPNT(I)) + B(I)
```

```
100 CONTINUE
```

Scalar dependencies

Scalar variables are unchanging single locations in memory, such as a simple variable (X). Array references whose subscript values are invariant in a loop (and thus represent a single location through all passes of the loop) are called *array constants*. Array constants are treated similarly to simple scalar variables by `VAST`.

Scalar variables that are modified in a loop can sometimes inhibit optimization by causing data dependencies. Scalar variables that are not modified in the loop do not inhibit optimization.

Carry-around scalars

Scalars that may be used before they are defined in a loop are called carry-around scalars. They may or may not be recursive. Recursive carry-around scalars inhibit optimization. All references to these variables are collected in a scalar loop and split out from the rest of the calculation if possible.

```
DO 3313 I = 1,N           (Not optimized.)
  A(I) = S + 1/S         (S is carried around.)
  B(I) = C(I) - A(I) + S
3313 S = B(I) + D(I)
```

Equivalenced scalars

In some circumstances, scalars that are EQUIVALENCED may inhibit data dependency analysis. The NOEQVCHK directive may be used to allow such operations to optimize, if the equivalencing does not actually create recursion (it almost never does).

```
COMMON /BLOCK/ A(999)
EQUIVALENCE (S,A(100))
...
DO 67 I = M, N           (Not optimized.)
  S = B(I)**2
  A(I) = S + 1.0/S
67 CONTINUE
```

8. Interprocedural Analysis

Parallel calls

The benefits of interprocedural analysis for parallelism detection are:

- Parallelize loops containing calls, without needing user assertions.
- Don't need to inline everything. Keeps compiler time and code size down.
- Can handle loops that are too big for automatic inlining.
- Analyzes deep call trees.
- Automatically make independent calls into parallel cases.
- Process disjoint phases of the program simultaneously.
- Preserves modularity.

IPA is an extension of dataflow analysis across routine boundaries. When a subroutine/function call is encountered, the system tries to ascertain its impact on global data, such as:

- COMMON variables
- arguments
- EXTERNAL variables
- SAVE statements/static variables
- I/O
- nested calls

IPA features rapid compile rate (by spending time only if a payoff is imminent) and usability (by not demanding that the entire source be supplied by the user).

To invoke IPA for parallelism, use the `-e5` switch.

Examples

This section presents some small examples of parallelizing loops containing calls. For instance, in the code sequence below, VAST can perform the iterations of the `J` loop in parallel by analyzing `ABC` and `DEF`.

```
REAL X(100,100), Y(100,100), Z(100,100)
...
DO J = 1, 100
    CALL ABC (X(1,J),Y(1,J),100)
    CALL DEF (Y(1,J),Z(1,J),100)
END DO
...
SUBROUTINE ABC (A,B,N)
REAL A(N), B(N)
DO I = 1, N
    B(I) = A(I) + 1.0/A(I)
END DO
END
...
SUBROUTINE DEF (A,B,N)
REAL A(N), B(N)
DO I = 1, N
    B(I) = B(I) + SQRT(A(I))
END DO
END
```

Independent calls can be analyzed for parallelism, even if they are separated by some scalar statements, as in the example below. Here, VAST-F/toOpenMP will use one parallel case for the call to `SUBB` and a different one for the call to `SUBC` at the same time.

```
SUBROUTINE SUBA
REAL A(999),B(999),C(999),D(999),E(999)
...
```

```

        CALL SUBB ( A, B, C, M )    ! Parallel...
        N = M - 1
        CALL SUBC ( A, D, E, N )    ! ...case.
        ...
        SUBROUTINE SUBB ( X, Y, Z, LEN )
        REAL X(LEN), Y(LEN), Z(LEN)
        COMMON /BLOCK/ S(1000), T(1000)
        DO 100 I = 1, LEN
        Z(I) = X(I)/Y(I) - S(I)
100    CONTINUE
        END
        SUBROUTINE SUBC ( U, V, W, LEN )
        REAL U(LEN), V(LEN), W(LEN)
        COMMON /BLOCK/ S(1000), T(1000)
        DO 100 I = 1, LEN
        W(I) = U(I)*V(I) + T(I)
100    CONTINUE
        END

```

Performance benefit

IPA finds subprogram-level parallelism fairly often. However, these occurrences account for substantial runtime only in a small percentage of occurrences. However, IPA is completely automatic and this speedup may be the difference on an important code or benchmark. IPA does estimate the amount of work in the CALL tree, and avoids trivial cases.

In general, I/O in a routine will cause it not to be allowed in a parallel construct, and a message is generated to tell the user this. The user can designate I/O to certain units as not preventing parallelism. If COMMON is used for local storage, and this is detected, a warning message is generated to the user. Similarly, inconsistent COMMONS are detected and the user is warned.

Error detection with IPA

While the primary use of IPA is for high-level parallelization, the information collected by the IPA subsystem allows VAST-F to diagnose many real and potential problems, including:

- Mismatch in number of arguments
- Mismatch in type of arguments
- Passing a scalar to an array
- Passing a lower-order array to a higher-order array
- Passing a constant to a variable that is modified
- Passing a DO loop index to a variable that is modified

These errors are warning messages.

9. Inline expansion

Overview

This section describes the inline routine expansion subsystem of VAST-F. Inline expansion is not invoked by default; you must turn it on explicitly. To get automatic inlining from the same file, you can use the `-e78` or `-J1` options on the command line.

Introduction

Programs can often receive a performance benefit from the expansion of the bodies of certain subroutines and functions into the loops that call them. This allows the calling loop as well as the body of the called routine to be optimized. Application codes sometimes have small external functions that are called from inside many loops; these functions are good candidates for inline expansion. Here is a small example of inline expansion:

Original:

```
DO 100 I = 1, N
    A(I) = CALC (A(I), X+B(I), 2.0)
100 CONTINUE
...
END
FUNCTION CALC (A,B,C)
CALC = A + SQRT( B**2 + C**2 )
```

```

IF ( CALC.LT.0 ) CALC = ABS ( B + C )
END

```

Expanding function CALC in line:

```

DO 100 I = 1, N
    TEMP1X = X + B(I)
    CALC1X = A(I) + SQRT( TEMP1X**2 +
1      2.0**2 )
    IF ( CALC1X.LT.0 ) CALC1X =
1      ABS ( TEMP1X + 2.0 )
    A(I) = CALC1X
100 CONTINUE

```

Inline expansion reduces subroutine calling overhead. It also allows scalar optimizations such as invariant code relocation and common subexpression analysis to extend across the body of the subroutine as well as the body of the calling loop. Even more importantly, inline expansion may enable the parallelization of an outer loop that contains the call.

There are two modes of inlining: automatic and explicit. These modes can be requested by directive or by command line option.

Automatic inline expansion

The objective of automatic inlining is to get rid of "leaves" of the calling tree. There is no programmer intervention needed other than requesting inlining on the command line.

Automatic inlining criteria

When automatic inlining is enabled (via the `-e7` switch, the `AUTOEXPAND` directive or or the `-J` parameter) VAST expands every called subroutine or function which has less than a threshold number of lines of code (default is 50, but the user can change the threshold with the `-M` parameter), no calls inside the expanded routine (no nesting with automatic expansion), and no expansion inhibitors (commons must match, etc.).

Avoiding inlining

If there are routines you want to explicitly exempt from automatic inlining, you can use the `-N` parameter on the command line to specify them.

Automatic inlining level

You can change the level of automatic inlining with the `-J` parameter. This allows you to specify the level of routines to be inlined automatically from the bottom of the calling tree. The default when automatic inlining is requested is one level (routines that do not call anything else). Using `-J1` is equivalent to using `-e7`. Specifying `-J2` requests automatic inlining of routines at the bottom two levels of the calling tree, if they met the other criteria as well. As `-J` implies looking first in the same file, if your code is distributed with one routine in each file you might want to use `-d8` as well after the `-J` switch to save the compile time spent looking in the same file for other routines.

Explicit inline expansion

In explicit inlining, the user lists the routines to be expanded, or directs expansion in the line following a directive. The routines may be listed on the command line when VAST is invoked (`-I`), or passed on directives such as `EXPAND`.

Nested expansion can be requested with explicit inlining. The `NEXPAND` directive or `-Y` parameter will expand the indicated routines and all routines they call, leaving no external references.

Source code access

VAST can search for inlinable source code in various places:

- same file
- different file
- naming convention

When expanding `call routine`, VAST will by default look for file `routine.f`.

The `SEARCH` directive or `-S` parameter informs VAST where to look for source modules. A directory or file can be specified to point VAST at further source files for inlining.

Inline expansion directives

AUTOEXPAND directive

The `AUTOEXPAND` directive is used to invoke automatic routine expansion.

Format:

```
CVD$ AUTOEXPAND
```

You can use F, R, or L scopes on this directive. NOAUTOEXPAND cancels the action.

The AUTOEXPAND directive with file scope is equivalent to the -e7 switch. See below for a list of inhibitors to automatic inlining.

Explicit mode

In explicit mode, routines listed on a directive (see following section), or on the -I or -Y invocation parameters, are expanded without regard for the automatic mode criteria. If no list is given on the directive, it directs expansion of the single CALL or all function references in the immediately following statement. This reference need not be inside of a loop. See below for a list of inhibitors.

EXPAND directive

The EXPAND directive is supplied for explicit routine expansion. The format is:

```
CVD$ EXPAND [ (routine1[, routine2[, ...]]) ]
```

Where *routine1*, *routine2*, ... is a list of routines to expand in this routine. If a list is not supplied, expand the next statement. Scope is ignored on the EXPAND directive.

```
CVD$ EXPAND ( CALC )
      ...
      DO 100 I = 1, N
          A(I) = CALC( A(I), B(I)+1., N )
      100 CONTINUE
      ...
      END
```

You may also supply a list of routines to expand on the command line, via the -I option.

NEXPAND directive

The NEXPAND directive operates in the same manner as the EXPAND directive, except that the named routines are expanded in a nested manner until no calls remain. In this way, you can expand a whole sub-tree of subroutine and function calls. On the command line, you can use the -Y switch.

```
CVD$ NEXPAND [ ( list ) ]
```

SEARCH directive

You can tell VAST where to look for the routines to expand, via the `SEARCH` directive or `-S` option on the command line. The `SEARCH` directive has routine scope by default. It may be used with file scope (`CVD$F`) before the first instance of expansion in the input stream.

Directive format:

```
CVD$ SEARCH ( filename [ ,filename ... ] )
```

Invocation parameter format:

```
... -S filename [ ,filename ... ]
```

where `filename` is a character string that identifies a file in which to search for the routines to be expanded. (Note that this directive does not identify which routines are to be expanded, just where to look for routines that have been so identified by directive or by automatic criteria.)

If `filename` is the special entry `*.f` then, for example, the routine `xyz` will be looked for in file `xyz.f`. (This is the default search method.)

Where to get the source code

In order to expand a routine, VAST needs to know where to find its source. The source location depends on programming style and on the operating system user interface.

Same file

Called routines can be searched for in the same file as the calling routine (stacked input). This necessitates an initial pass by VAST through the entire input file (and files `INCLUDED` therein) to build a directory of the program units in the input file. The switch `-e8` enables this initial pass, which by default is not done.

Explicitly named file

You can supply (via the `SEARCH` or `-S` invocation option directive, described previously) the name of a file in which to search for a particular called routine.

Implicitly named file

Fortran programs are frequently stored such that each routine of the program resides in a separate file with a canonical name (for example, the name of the routine followed by `.f`). This is the default search method.

Possible problems

Separate compilation

Fortran allows program units to be compiled separately and linked together. Because different program units may be compiled at different times, it is not possible to make inline expansion completely foolproof without resorting to "programming environments" which place restrictions on the user. Even if routines are initially compiled from the same input file, an object of a modified version of a routine could be supplied at a later link.

For example, suppose program A has subroutine B, which calls subroutine C. Let's say subroutine C is expanded into subroutine B. Later, we decide to change the calculation in subroutine C, and so we edit it; rather than recompiling the entire program, we just recompile C and link it with the previously compiled routines. Our changes to C are not present in the actual code executed because C is no longer called from B (it was expanded).

Thus, you must be involved in the routine expansion process at least to the point of knowing which routines must be recompiled when a change is made. For this reason, VAST generates both a warning message for every expansion and an expansion event table so that it is clear what has been expanded.

Code size

A problem that may result from inline expansion is larger code size; if too many routines are expanded, or the expanded routines are too large, the size of the compiled code may reach unacceptable proportions. This potential problem can be controlled through judicious application of this transformation.

Debugging

Inline code expansion can complicate user run-time debugging; if the program fails in an expanded section of code, the error is reported in a different routine than the one it originally appeared in.

VAST records the original line number of the routine invocation on all the expanded lines.

Compilation rate

Inline expansion may result in two passes over the entire program, and longer compile times, depending on how much code is brought in line.

Nested expansion

Nested expansion will be done only if specified by user directives or with the `-Y` option, or if the `-J` parameter is used with level greater than one. Nested routines will not be expanded in default autoexpansion mode, thus reducing the possibility of code size mushrooming.

If you want nested routines to be expanded, you must explicitly specify each of them in the chain in an `EXPAND` directive or `-I` parameter, or specify the top routine in the call chain in an `NEXPAND` directive or `-Y` parameter.

Analysis inhibitors

There are several ways in which analysis may be inhibited, thereby prohibiting expansion. An appropriate message informs you of a failed expansion. A full list of messages appears at the end of this section.

Expansion inhibitors

- The routine to be expanded cannot be located.
- Syntax errors are found in the expansion routine.
- The arguments used in the calling sequence do not match the arguments in the expansion routine. (See next section for discussion).
- There is a conflict between common blocks of the calling routine and the expansion routine. (See next section for rules.)
- The routine to be expanded contains `NAMELIST`.
- The routine to be expanded contains `SAVE` statements. (Can be overridden by the `-g6` switch -- see below.)
- The routine to be expanded contains `DATA` statements for local variables, whose value is changed in the routine. (Can be overridden by the `-g6` switch -- see below.)
- A function is being expanded in a `DO WHILE` statement or an `ELSE IF` statement.
- A function name referenced in the expansion routine conflicts with a non-function name used in the calling routine.

Inhibitors specific to automatic expansion mode

In automatic mode, all calls to routines that meet the following criteria are expanded:

- The routine to be expanded has less than the maximum allowed number of non-comment lines. (The default is 50. This number can be changed via the `-M` parameter on the invocation.)
- The routine to be expanded does not call any other external routines. (If the `-J` option is used, it can expand nested references if they are within the number specified from the bottom of the call tree).
- There are no inhibitors to the expansion (common blocks that do not agree, and so forth).

If these parameters are unsatisfactory, you can always resort to explicit mode. The objective of automatic mode is to catch small, simple frequently-called external functions. More demanding cases must be explicitly requested.

Note that although called automatic, this mode still requires you to enable an option or insert a directive. An informational message is issued for each expansion action. This is to remind you that routines that have been expanded into need to be recompiled each time the expanded routine is changed.

Inline expansion abilities

This section presents some of the operations that are performed during inline expansion of subroutines and functions.

Naming conventions

Where necessary, VAST renames all variables and parameters in expanded program units in the following manner. The first four characters of the variable are combined with an integer number and the suffix `X` to create a unique name. For example, the local variable `I` could become `I1X`, and the next reference to `I` in another expanded routine could become `I2X`.

In this way, VAST keeps a relationship between the original user name and the inlined name; this makes the generated code much more readable.

Common blocks

All common blocks that are used in both routines must agree. `COMMON` blocks can be just in the caller or just in the callee. Common blocks that appear only in the expanded routine are added to the calling routine. If present in both, the names of objects in the common do not have to match, but sizes of corresponding objects must match. The caller common may be a superset of the callee common (the sizes must match, up to the end of the callee common).

Arguments

Dummy parameters are replaced with their corresponding actual arguments. In the case of expressions passed as actual arguments, VAST will create a temporary variable to hold the expression and use the temporary in each of the places the dummy argument appeared in the called routine.

The number of actual and dummy arguments must match. However, array elements can be passed to scalars, higher dimensioned arrays can be passed to lower dimensioned arrays, and single dimensioned arrays can be passed to multiple dimensioned arrays.

Unique names

Local variables used in the expanded routine are checked against the identifiers defined in the caller, and made unique. If already unique, they are left alone.

Similarly, constant parameter names (from `PARAMETER` statements) are examined and changed if necessary to avoid conflict with any name from the calling program. If identical in name and value with a constant parameter in the calling program, then no change is made.

Labels

All labels used in the called routine (`FORMATs`, `CONTINUEs`, `DOs`, and so forth) are changed so that there is no conflict with labels in the caller.

Returns and return values

`RETURN` statements are changed into branches to a new label in the caller that represents the end of the called routine. If it is an alternate `RETURN` statement, the branch corresponding to that `RETURN` is directed to the specified label.

In addition, references to the function name as a variable in an expanded function are replaced with another name. (Calls to the original function may still exist unexpanded.)

Inlining ENTRYs

VAST inlines calls to `ENTRY` points themselves. If both the subroutine itself and an entry inside of it are called, they can both be inlined.

Cleanup of inlined code

When constants are passed to routines, there are often opportunities to simplify the inlined code. VAST uses global constant propagation and global expression simplification to eliminate dead code resulting from

expansion constants. This can result in dramatic reduction in the size of the expanded routine if many constants are passed.

Example

The code segment below shows two-dimensional arrays passed to one-dimensional arrays, and significant cleanup of the expanded code (dead code elimination) due to the constants passed as arguments.

```

call daxpy(n,t,a(k+1,k),1,a(k+1,j),1)
.
.
.

subroutine daxpy(n,da,dx,incx,dy,incy)
double precision dx(1),dy(1),da
integer i,incx,incy,ix,iy,m,mp1,n
if(n.le.0)return
if (da .eq. 0.0d0) return
if(incx.eq.1.and.incy.eq.1)go to 20 (dead code, as
incx = 1 both incx and
one) iy = 1 are passed as
if(incx.lt.0)ix = (-n+1)*incx + 1
if(incy.lt.0)iy = (-n+1)*incy + 1
do 10 i = 1,n
dy(iy) = dy(iy) + da*dx(ix)
ix = ix + incx
iy = iy + incy
10 continue
return
20 continue
do 30 i = 1,n (this survives)
dy(i) = dy(i) + da*dx(i)
30 continue
return
end

```

Translation:

.

```

      .
      .
C***** Code Expanded From Routine:  DAXPY
      IF (N .LE. 0) GO TO 77001          (return)
      IF (T .EQ. 0.0D0) GO TO 77001    (loop 10 deleted)
      DO I = 1, N
          A(I+K,J) = A(I+K,J) + T*A(I+K,K)  (2d to 1d array)
      ENDDO
77001 CONTINUE
C***** End of Code Expanded From Routine:  DAXPY
      .
      .
      .

```

Expansion of SAVE and DATA

In some cases it is desirable to inline a routine which has `SAVE`d local data or `DATA`d items which get stored into. Without the `-g6` switch specified, such a routine would be rejected for inlining with one of the following messages:

```
SAVE stmt. inhibits expansion, change to COMMON stmt.
```

or

```
Expan. inhibitor - DATA stmt. implies saving of local var.
```

If it is desired to expand such a routine, the routine itself must be rewritten. In order to retain the correct values of the `DATA` or `SAVE` variables in the inlined instances of the routine, a `COMMON` block and possibly a `BLOCK DATA` must be created for these items. In most cases VAST has the ability to do this automatically.

To force this type of expansion, in addition to the expansion switches `-g6` must also be specified and the routine (or entry) to be expanded must appear explicitly in the `-I` or `-Y` parameters.

Care must be taken when doing this type of expansion since either the routine must be inlined at *all* the places it is called in the program or the changed routine must be the version which is linked to the program. If an unchanged version of the routine is accidentally linked to the program, wrong answers will result.

For example:

```
vastfomp -e78 -g6 -I xp1,xp2 bigsource.f
```

would cause `xp1` and `xp2` to be "forcibly" expanded wherever they are called in `bigsource.f`, and also whichever routines contain the entries `xp1` or `xp2` will be changed so as to be incompatible with the original version.

Inline expansion user messages

All inline expansion messages appear as warnings. VAST does not expand any routine that has caused the generation of one of the following messages, except for the `ROUTINE EXPANDED` message.

Inline expansion summary

VAST notifies you of any routines that have been expanded and also informs you as to why a particular routine was not expanded. If a listing was requested (`-p1`), a summary of the routines and all the locations where a routine was or was not expanded is displayed.

User messages

SOURCE FOR ROUTINE NOT FOUND

The expansion routine cannot be located, as specified by the `SEARCH` directive, `-S` option or by the default method.

SOURCE FOR ROUTINE NOT FOUND IN INPUT FILE

The input file has been defined as the location for expansion routines/functions, either by default or via switch. The routine/function is not found in the input file.

EXPANSION ROUTINE IS TOO BIG FOR AUTOMATIC EXPANSION

The routine has more than the maximum allowed number of non-comment lines (default 50; changeable via the `INLMAX` invocation parameter); use explicit expansion mode to expand.

EMBEDDED CALL STATEMENT ENCOUNTERED WHILE EXPANDING

The expansion routine references another subroutine; use explicit expansion mode to expand.

SYNTAX ERROR ENCOUNTERED IN EXPANSION ROUTINE

The expansion routine has a syntax error and will not be expanded.

ARGUMENT MISMATCH BETWEEN CALL AND EXPANSION ROUTINE

The arguments specified in the calling sequence of a subroutine or in a function reference do not match with arguments of the

expansion routine/function. For example, this could result from a mismatch of data types.

EXCEEDED MAXIMUM NUMBER OF EXPANDED ROUTINES

The maximum number of routines/functions VAST will expand has been exceeded. Currently this number is 600.

COMMON BLOCK MISMATCH BETWEEN CALLING AND EXPANSION ROUTINE

Common blocks found in calling and expansion routine/function do not agree, or a COMMON variable in the expansion routine is used as a local variable in the calling routine. The COMMON in question is listed with this message.

FUNCTION IN EXPANDED ROUTINE CONFLICTS WITH NON-FUNCTION

A function found in the expanded routine is in conflict with a non-function element in the calling routine. The function name in question is listed with this message.

ROUTINE EXPANDED

This warning message will be issued whenever a routine/function has been expanded.

10. Output formatting

This section describes the output formatting feature of VAST/toOpenMP. You can select from many options and switches to get your code reformatted in the way you prefer.

Output formatting options fall into two categories: switches (single features that can be turned on or off) and parameters (numerical values). Output formatting switches and parameters may be specified either on the VAST/toOpenMP invocation line or on the SWITCH directive.

Option switches

The switches in the table 9.1 can be specified by using the `-r` and `-n` parameters on the VAST/toOpenMP invocation line or on the SWITCH directive.

Table 9.1 -- Output formatting switches

Switch	Description	Default
a	Place inline comments above statement, if they do not fit	on
b	Put space after a comma in a subscript (e.g. <code>A(X, Y)</code>).	off
c	Put space after a comma in a list (e.g. <code>CALL X (A, B, C)</code>).	on
d	Indent first/last lines of DO loops.	off

e	Put spaces around equal sign (=).	on
f	Move format statements before END statement.	off
g	Put spaces around subscript parentheses.	off
h	Force labels to be on CONTINUE and FORMAT stmts. only	on
i	Indent first/last lines of IF blocks..	off
j	Put spaces around the ** and // operators.	off
k	Put spaces around .AND., .OR., .EQV.and .NEQV. only.	off
l	Convert Fortran output to lower case.	off
m	Squeeze two-line statements into one by ignoring spacing rules.	on
n	Put spaces around non-subscript parentheses.	off
o	Put spaces around all logical operators.	off
p	Put spaces around + and - .	on
q	Spacing around logical operators (see notes).	on
r	Generate comments listing external references.	off
s	Follow spacing rules for operators inside parentheses.	off
t	Put spaces around * and /.	off
u	Add keywords UNIT= and FMT=.	off
v	Put spaces around = in keyword lists.	off
w	Indent first/last lines of WHERE blocks	off
x	RENUMB=100:10 , FORMAT=900:10 , TDYON=R	off
y	Reserved.	--
z	Create output file	on
1	Terminate DO loops with ENDDO.	on
2	Align equals signs for assignment statements in the same block.	off

Notes on output formatting switches:

- a. The column in which the inline comment begins is determined by the length of the comment and by the ILCCOL parameter described later in this chapter. If the original Fortran line is not reformatted, this switch may still have an effect if the ILCCOL parameter is set to a column less than the column in which the original inline comment begins. This switch has no effect on inline comments that remain inline.

- c. Lists are any collection of variable and array names, such as those found in the argument lists of SUBROUTINE and FUNCTION statements, CALLs to subroutines or functions, and the input/output list of READ, PRINT, and WRITE statements.
- f. This switch causes all of the FORMAT statements to be collected at the end of the module. If this switch is off (the default), the FORMAT statements remain where they were originally. If this switch is on, renumbering will automatically give FORMAT statements the highest labels, because they will now be at the end of the routine. This situation may change if the `FORMAT=` parameter, described later in this section, is used. (Indentation and spacing of FORMAT statements is never changed.)
- h. Fortran allows statement numbers on any statement, but the only statement numbers necessary are those used as targets for branches, terminal statements of a DO loop, and FORMAT statements. Branch targets and terminal statements of a DO loop can be converted to CONTINUE statements, and numbered accordingly. With this switch on (the default), this is done automatically.
- k. Put spaces around `.AND.`, `.OR.`, `.EQV.`, and `.NEQV.` only. The default is off. For IF statements, the `k`, `o`, and `q` options are interrelated. Only one may be on at a time. The `o` switch is: put spaces around all logical operators. Default is off. The `q` switch is: put spaces around all logical operators when alone in an IF statement, else around `.AND.`, `.OR.`, `.EQV.` and `.NEQV.` only.

Example:

```
LOGICAL L1, L2
IF (A(I).GT.B(I)) A(I) = B(I)
IF (C(I).GT.D(I).AND..NOT.L1) C(I) = D(I)
IF (L1 .AND. .NOT.L2) L2 = .TRUE.
```

This example becomes, with the `k` switch on and the `o` and `q` switches off:

```
LOGICAL L1, L2
IF (A(I).GT.B(I)) A(I) = B(I)
IF (C(I).GT.D(I) .AND. .NOT.L1) C(I) = D(I)
IF (L1 .AND. .NOT.L2) L2 = .TRUE.
```

with the `o` switch on and the `k` and `q` switches off:

```
LOGICAL L1, L2
```

```

IF (A(I) .GT. B(I)) A(I) = B(I)
IF (C(I) .GT. D(I) .AND. .NOT. L1) C(I) = D(I)
IF (L1 .AND. .NOT. L2) L2 = .TRUE.

```

and with the q switch on and the k and o switches off:

```

LOGICAL L1, L2
IF (A(I) .GT. B(I)) A(I) = B(I)
IF (C(I) .GT. D(I) .AND. .NOT. L1) C(I) = D(I)
IF (L1 .AND. .NOT. L2) L2 = .TRUE.

```

- l.** This switch causes VAST/toOpenMP to convert all of the executable part of the code to lowercase. VAST/toOpenMP's declarations will be generated in lowercase as well. The user's declarations will be copied in the same case as they were in the input program. (Character constants and comments are also not converted.)
- m.** Sometimes, strict adherence to the output formatting spacing rules yields two-line statements that would be more readable as one line. By ignoring some of the spacing rules, the statement is made to fit on one line. If you want this automatic squeezing to be turned off, use `-n m`.
- n.** Non-subscript parentheses occur in `IF`, `CALL`, `READ`, `WRITE`, `SUBROUTINE`, `FUNCTION`, and other statements.
- o.** See explanation under switch `k`.
- q.** See explanation under switch `k`.
- r.** Generate a block of comments that lists the subroutines and external functions called by the current program unit (including inlined routines). This block of comments is inserted into the translated program unit preceding the first executable statement. Routines are listed in first-occurrence order. If no subroutines or external functions are called, a comment to that effect is inserted.
- s.** This switch uses the settings of the `j`, `p`, and `t` switches to determine spacing inside of parentheses. If the switch is off, as is the default, then no spacing will take place.
- u.** Fortran allows the keywords `UNIT=` and `FMT=` in input and output statements. Most codes, however, use the positional notation. If desired, the keywords will be added and formatted in the same manner as any other keywords in the input or output statement.

Note that if the original statement contains a `UNIT=` or `FMT=` keyword, and the switch is off, the keywords will still remain. This switch does not remove any keywords from the original statement.

- x. A shorthand switch for invoking `RENUMB=100:10`, `FORMAT=900:10`, `-r R`. Causes renumbering of labels and insertion of comment block summarizing externals.
- z. Create an optimized source file. This switch may be turned off if the diagnostic listing only is wanted. Turning this switch off may speed compile time and reduce disk space used. The setting of this switch does not affect the listing of the transformed source in the listing file.

Output format parameters

The following output format parameters can appear on the VAST/toOpenMP command line or on the SWITCH directive. On the VAST command line, these parameters are preceded by the `-z` flag (e.g. `-Z RENUM=100:10`)

Label renumbering

`RENUMB=m:n`

This parameter controls the renumbering of the statement labels within a routine. If only the `RENUMB=` part appears without a number following the `=`, then the statement labels are renumbered starting at 100 and increasing by 10 for each successive label.

If the parameter is given as `RENUMB=m`, where `m` is any number (less than 99999), then the increment defaults to 10. If the parameter is given as `RENUMB=:n`, then the starting number defaults to 100. The forms `RENUMB=m:` and `RENUMB=:` are not allowed. Use `RENUMB=n` or `RENUMB=`, respectively.

`FORMAT=m:n`

This parameter works exactly like the `RENUMB` parameter, but only on `FORMAT` statements. If a different numbering scheme is desired for `FORMAT` statements than for other statements, this parameter is used. If `FORMAT=` is used alone, the default is a starting number of 900 and an increment of 10. The forms `FORMAT=m` and `FORMAT=:n` are also valid as described in the `RENUMB=m:n` section above.

Label alignment

`LABELS=n:l`

This parameter controls placement of statement labels within the statement label field. For the labels to be left-justified, `l` should be set to the character `L`. For the labels to be right-justified, `l` should be set to the character `R`. The column in which the labels are justified is given by `n`, which must be a number from 0 to 5. If `n` is 0, no alignment of statement labels occurs. The default is to right-justify labels in column 5 (`5:R`).

Fortran 90 construct names are treated as part of the statement they apply to, and indented accordingly, with the remainder of the statement following after a single space.

Indentation

There are five indentation parameters:

`INDDO=n`

`INDIF=n`

`INDWH=n`

`INDCN=n`

`INDAL=n`

where `n` is the number of spaces to indent. In each case, `n` must be a number between 0 and 10, inclusive.

`INDDO` is for `DO` blocks, `INDIF` for `IF` blocks, `INDWH` for `WHERE` blocks, and `INDCN` is the number of spaces to indent continued lines. `INDAL` sets all the other parameters to `n` (`INDDO`, `INDIF`, `INDWH`, and `INDCN`). The default for all of these parameters is 3.

For a `DO` block, the `DO` statement and the terminal statement of the `DO` are placed at the same indentation level, and statements between them occur at the new indentation level. For an `IF` block, the `IF` statement itself, any `ELSE IF` or `ELSE` statements, and the `ENDIF` statement all occur at the same indentation level, and statements between them occur at the new indentation level.

Continued lines are indented `n` spaces from the initial line, where `n` is given by the `INDCN` parameter.

`LSTCOL=n`

This parameter gives the last column available for indentation, that is, no statement will be started after this column. The default is 31.

Indentation is preserved up to the column limit given by LSTCOL, and no statement begins after column n. Using the defaults gives eight separate block levels, each indented by three spaces from the previous level.

Inline comments

ILCCOL=n

This parameter aligns inline comments starting at column n. (For fixed-format output, if n is less than 15 or greater than 66, no alignment is done.) Column n is the column in which the inline comment delimiter will appear.

The default is to align inline comments at column 50 (ILCCOL=50). To turn off alignment of the inline comments, use ILCCOL=0. If an inline comment will fit on the original line, even after reformatting, but will not start at the column used for alignment, it will be placed on the original line so that the last character of the comment will be in the last position of the line. If an inline comment does not fit on the same line as before, a comment will be inserted above the line (or below if the A switch is off), and the comment will be aligned to the appropriate column if possible.

Continuation lines

CONCHR=*

For fixed-format output, this parameter determines what character to use for continued lines. By default, the successive numbers 1,2,3,...8,9 are used. The tenth continued line uses the decimal point and then the cycle begins over again with 1. If the CONCHR= parameter is used, then the character following the = is used as the continuation character. If that character is 0, the default is used, because 0 is not a valid continuation character. (This parameter applies to fixed format source only.) A statement which must be continued on another line may be broken at any point, except within a name or a constant.

Output line length

Zolen=n

For free format output, this parameter allows specification of the output line length (up to 132 columns).

FORMAT and DATA statements

The output formatting does not change the indentation or spacing inside FORMAT and DATA statements. (However, FORMAT statements may optionally be renumbered and/or moved to the end of the routine.)

Example

Here is an example of VAST/toOpenMP formatting a Fortran 77 routine. This example shows the use of several switches, but you can rely on the defaults as well.

Here is the input:

```
CVD$ SWITCH,-rfjnostuv2,renumb=10:5
      SUBROUTINE TIDYX1
      REAL A(100),B(100),C(100),D(100),E(100)
      REAL AA(100,100),BB(100,100),
1     CC(100,100),DD(100,100)
      DO 111J=1, 100
      A(J)=SQRT(B(J))
111   CONTINUE
9999  FORMAT (3F12.8)
      DO 132 J=1,100
      S = B(J)-D(J)*E(J)      ! note alignment
      B(J)=D(J)+E(J)/S**2    ! of inline comments.
      IF (A(J).GT.B(J).AND.C(J).LT.D(J)) GO TO 100
      C(J)=B(J)/C(J)
      GO TO 19
100   CONTINUE
      C(J)=C(J)/B(J)
19    CONTINUE
      DO 2 I =1,100
      A(J)=AA(I,J)
      AA(I,J)=A(J)+B(I)/BB(I,J)
2     CC(I,J)=DD(I,J)
      WRITE(6,9999,ERR=154) A(J),B(J),C(J)
```

```

132 CONTINUE
154 RETURN
    END

```

And this is the cleaned-up version:

```

SUBROUTINE TIDYX1
REAL A(100),B(100),C(100),D(100),E(100)
REAL AA(100,100),BB(100,100),
1  CC(100,100),DD(100,100)
DO J = 1, 100
    A(J) = SQRT ( B(J) )
END DO
DO J = 1, 100
    S      = B(J) - D(J) * E(J)      ! note alignment...
    B(J) = D(J) + E(J) / S ** 2    ! of inline comments.
    IF (A(J) .LE. B(J) .OR. C(J) .GE. D(J)) THEN
        C(J) = B(J) / C(J)
    ELSE
        C(J) = C(J) / B(J)
    ENDIF
DO I = 1, 100
    A(J)      = AA(I, J)
    AA(I, J) = A(J) + B(I) / BB(I, J)
    CC(I, J) = DD(I, J)
END DO
WRITE ( UNIT=6, FMT=15, ERR=10 ) A(J), B(J), C(J)
END DO
10 CONTINUE
RETURN
15 FORMAT (3F12.8)
END

```

11. Diagnostic Messages

Overview

This section shows selected VAST-F/toOpenMP messages, grouped by category. Most of the messages described here have to do analysis of loops for parallel execution.

Data dependency conflicts

Data dependency messages are always followed by the name of the variable which is causing the problem. If outer loop translation is being attempted, the label and index of the loop causing the problem is given as well.

"Feedback of array elements"

```
DO 4 I=1,N
4 A(I+1) = A(I) + B(I)
```

Feedback of results makes the loop recursive and thus unsafe to translate to partition.

"Feedback of scalar value from one loop pass to another"

```
DO 112 I = 1,N
A(I) = A(I) + SCA
112 SCA = A(I)/C(J)
```

The variable SCA is used in the first line of the DO loop to set A(I), and then is set to a function of A(I) in the last line. This creates feedback of

elements of A from one loop pass to the next, which prevents optimization. SCA is called a "carry-around" scalar, as it carries a value around to the next pass of the loop.

"Potential feedback of array elements -- use directive if ok"

```
DO 3 I=1,N
3 A(I+J) = A(I) + B(I)
```

It is not clear whether there is feedback between the two uses of A in this loop or not (it depends on the value of J). Loops of this kind that the user is sure are safe can be translated by putting the directive CVD\$ NOSYNC in front of the loop. Here is another case where this message would result:

```
DO 1 I=1,N
1 B(I) = B( IB(I) )+A(I)
```

B(IB(I)) is a gathered array, and as the values in IB(I) are unknown, it may conflict with the assignment of elements of B on the left side of the equal sign. If the pattern of B(IB(I)) is known not to overlap B(I), then the NOSYNC directive should be used.

As a convenience, the -dd option switch or NOSYNC directive with routine or global scope may be used instead of loop-by-loop directives.

"Multiple store conflict"

```
DO 100 I = M,N
A(I) = B(I)
100 IF ( C(I) .GT. 0 ) A(I-1) = C(I)
```

Stores into overlapping sections of the same array must be done in the same order as in the scalar loop.

"Potential multiple store conflict -- use directive if ok"

```
DO 100 I=1,N
A(I+J)=B(I)
100 A(I+K)=C(I)
```

The loop above has a potential overlap between the two stores into A; usually in these situations there is no real overlap between the two sections of A and the NOSYNC directive should be used to allow the loop to translate.

"Feedback of array elements (equivalenced arrays)"

Actual feedback between arrays equivalenced together.

"Potential feedback (equivalenced arrays) -- use directive if ok"

```
COMMON /BLOCK/ A(999)
EQUIVALENCE (S,A(100))
...
```

```

DO 67 I = M, N
    S = B(I)**2
    A(I) = S + 1.0/S
67 CONTINUE

```

Either two arrays or an array and a scalar are equivalenced, and their storage relationship in the loop cannot be determined. Use the NOSYNC or NOEQVCHK directives or the -dd or -de switches to allow translation, if there is in fact no recursion.

"Equivalence of scalars prevents translation - use directive if ok"

```

COMMON / BLOCK / A(99)
EQUIVALENCE (A(1),S), (A(2),T)
...
DO 68 J = M, N                                (Not translated.)
    S = B(I)**2
    T = C(I)**2
    D(I) = SQRT(S+T)
68 CONTINUE

```

Two scalars are in the same equivalence class and at least one is modified in the loop. The NOSYNC or NOEQVCHK directives, or the -dd or -de switches, may be used to allow translation, if in fact there is no recursion.

"Too many data dependency problems"

The loop has exceeded the maximum number of data dependency conflicts allowed (10). Translation of the loop is abandoned at this point to avoid printing out further messages.

Translation diagnostics

Translation diagnostics point out constructs that prevent a loop from being translated.

Statement types

These messages complain about types of statements that prevent translation.

"ASSIGN prevents loop translation"

```

DO 210 I = 1, N
    IF ( A(I) .GT. 0 ) ASSIGN 225 TO TOGO
210 CONTINUE

```

"ASSIGNed GOTO prevents loop translation"

```
      DO 220 I = 1, N
          GOTO TOGO
220   CONTINUE
225   CONTINUE
```

"Computed GOTO prevents loop translation"

```
      DO 230 I = 1, N
          GO TO ( 231, 232, 233 ) IGO(I)
231   B(I) = 0
232   B(I) = B(I) + A(I)
233   B(I) = B(I) * C(I)
230   CONTINUE
```

"I/O statements prevent loop translation"

```
      DO 240 I = 1, N
          WRITE ( IOUNIT ) A(I)*B(I)+C(I)
240   CONTINUE
```

"RETURN prevents loop translation"

```
      DO 250 I = 1, N
          IF ( X(I) .LT. 0 ) RETURN
          Y(I) = SQRT(X(I))
250   CONTINUE
```

"STOP prevents loop translation"

```
      DO 260 I = 1, N
          IF ( X(I) .LT. 0 ) STOP 99
          Y(I) = ALOG ( X(I) )
260   CONTINUE
```

Branches

The messages below relate to handling of conditional operations.

"Backward transfers prevent loop translation"

```
      DO 107 I=1,N
104   A(J) = SQRT(B(J-1) + D(I))
          J = J + 1
          IF (B(J) .GT. LIMIT) GO TO 104
107   C(I) = A(J-1)
```

The branch to label 104 is backward, not forward, and prevents translation. In some cases however, backward transfers may be converted into separate DO loops.

"Branches out of the loop prevent translation"

```
DO 108 I=1,N
  IF (A(I).LT.0) GO TO 777
108 B(I)=6.0
```

The label "777" is not in the loop.

"Branching too complex to translate this loop"

Because of limits on time and table space, conditional constructs with more than six simultaneously active conditions (i.e. IF-THENS, IF-GOTOS, etc.) cannot be optimized.

External references

The messages below deal with external references in loops.

"Subroutine call prevents loop translation"

```
DO 110 I = 1,N
  A(I) = SQRT(B(I)/C(I))
  CALL REVAMP(A(I),D(I))
110 D(I) = EXP(A(I)+X)
```

"Reference to function that has no array version"

```
DO 115 I=1,N
115 A(I) = MYFUNC(B(I))
```

References to non-intrinsic functions in a loop prevent translation of the loop.

DO statement

These messages relate to DO statements.

"DO statement parameters must be integer for array transl."

```
DO 100 W = 1.0001, 1000000.
  A(I) = W
100 CONTINUE
```

Non-integer variables or constants are not allowed as the loop index or in the start, end or increment fields for translation purposes to array syntax.

"User function references not allowed in iteration count"

```
DO 210 I = 1, NLEN(J,K)
210 A(I)=0.
```

In this example `NLEN` is an external user function. Such functions cannot appear in the iteration count for a `DO` loop (they cannot be in the start, end or increment fields of the `DO` statement) for the loop to be optimized. Statement functions are allowed, however.

Miscellaneous

These messages fall into none of the previous categories.

"Null loop body"

```
      DO 111 I=1,N
111  CONTINUE
```

Nothing in the loop. (The loop is not eliminated.)

"Character data type inhibits loop translation"

```
      DO 200 I = 1, N
          P(I)(1:2) = Q(I)(2:3)
200  CONTINUE
```

Use of character type data prevents a loop from being considered for translation.

Warnings

Potential Errors

These warning messages relate to potential errors in the input program. Use the switch `-pr` to get this kind of message for your code.

**** Variable used but never defined ****

A local variable is used in executable statements but is never defined.

"Variable defined but never used"

A local variable is defined in executable statements but is never used.

**** Variable used but not defined ****

A local variable is used when it is undefined, although it is defined elsewhere in the program unit.

"Variable defined but not used"

This definition of a local variable is not used, although the variable is used elsewhere in the program unit.

"Variable appears only in argument list"

A local variable appears only once, in the argument list of a subroutine call.

"Dead code"

Flags a section of code which, because of the program's flow of control, can never be executed.

Obsolescent Features

Additional warnings are generated for obsolescent features that are no longer preferred usage.

Syntax errors

There are a very large number of syntax error messages. These messages are for the most part self explanatory, and so are not repeated here.

Internal errors

Please report any VAST internal errors immediately to your support representative.

"Internal error detected (phase)-- please report"

Directive errors

These messages describe errors in the way directives to VAST-F/toOpenMP have been used.

"Unknown directive -- it is ignored"

CVD\$ SCALARIZE

"Switch input error"

CVD\$ SWITCH=-1

"Excess characters following directive"

CVD\$ SKIP THIS LOOP, PLEASE

In this case, the characters following SKIP are invalid.

Notes

Notes are not generated by default. They can be enabled with the `-po` switch.

"IF loop converted to DO-loop"

An IF loop has been converted to a DO loop. (The resulting DO loop may or may not be partitioned.)

12. Further Information

Crescent Bay Software and VAST

Crescent Bay Software was founded in 2003 by a group that has been together since the mid-1970s. (At Pacific-Sierra Research until 1988, when PSR was bought by Veridian Corp.)

We began work on the VAST (Vector and Array Syntax Translator) project in 1979, and VAST systems are in use at most of the supercomputer sites in the world. The VAST technology is used in a variety of ways in different systems, from an optimizing prepass of the compiler to a stand-alone translator.

In addition to **VAST-F/toOpenMP**, other VAST products include:

VAST-F/Parallel, automatic parallelization for Fortran. Includes compilation of OpenMP.

VAST-C/Parallel, automatic parallelization for the C language. Features the same base optimizer as the Fortran version of the product.

VAST-DPC, Data Parallel C compiler (also compiles C*). Brings the Data Parallel computing model to the ANSI C language.

VAST/77to90, Fortran 77 to Fortran 90 translator. Many people have found it to be a very useful tool for migrating old code to the new language.

VAST/AltiVec, providing explicit vectorization for Fortran and C/C++ on G4/G5 systems.

Questions or comments

If you have any questions or comments about VAST-F/toOpenMP, please do not hesitate to contact:

Crescent Bay Software

10950 Washington Boulevard, Suite 230

Culver City CA 90232

Phone: (310)-836-5183

Fax: (310)-836-7313

Index

ambiguous subscript resolution, 43
array constants, 45
array syntax, 14, 35
associative transformations, 14, 19
AUTOEXPAND directive, 54
automatic distribution of loop iterations, 28
automatic inlining, 52
backward transfers, 77
branches out of the loop, 77
carry-around scalars, 46
CNCALL, 19
CNCALL directive, 32
command line, 11
concurrent call, 32
continuation lines, 70
critical region, 31
-D invocation parameter, 22
data dependencies, 31, 42
Data Dependency Conflict, 24
data dependency directives, 43

data dependency messages, 73
diagnostic messages, 23, 24, 73
EQUIVALENCE, 14, 46
error detection, with IPA, 49
event summary, 25
EXPAND directive, 54
expansion of SAVE and DATA, 61
explicit inlining, 53
external references, 77
-F command line parameter, 16, 22
feedback of array elements, 73
files, 9
-G fusion parameter, 37
IF loops into DO loops, 39
INCLUDE files, 26
indentation, 69
inline comments, 70
inline expansion, 51
INNER directive, 31
inner loops, 31
input files, 12
input line numbers, 26
Internal Error, 25
interprocedural analysis, 47
IPA, 47
label alignment, 69
label renumbering, 68
last values, 14
listing, 12, 23
listing control switches, 25
listing, page length, 27
listing, wide format, 27
loop disposition graph, 23

- loop fusion, 37
- loop interchange, 40
- loop optimizations, 37
- loop rerolling, 38
- loop summary, 25
- loops containing subroutine calls, 32
- multiple store conflict, 74
- nested expansion, 57
- NEXPAND directive, 54
- NOASSOC, 19
- NOCONCUR, 14, 19
- NOEQVCHK, 21, 46
- NOEQVCHK directive, 44
- NOLIST, 21
- NOSYNC, 20, 43
- Note Message, 24
- optimizations, 9
- options, 12, 15
- output file, 11, 14
- output formatting, 13, 16, 64
- output formatting switches. *See* switches, output formatting
- parallel case optimization, 14
- parallel cases, 32
- parallel regions, 28
- parallelization, 28
- peeling, loop, 38
- PERMUTATION, 21
- PERMUTATION directive, 45
- potential error detection, 26
- potential errors, 78
- potential feedback, 43, 74
- private array, 29
- private variables, 28

recurrences, 42
reduction operations, 31
RELATION, 21
RELATION directive, 44
routine expansion, 51
-S parameter, 53
scalar dependencies, 45
SEARCH directive, 53, 55
shared variables, 28
SKIP, 19
summation, 31
SWITCH directive, 21
switches, 12, 21
switches, output formatting, 64
Syntax Error, 25
tidy, 64
transformation control, 13
Translation Diagnostic, 24
translation diagnostics, 75
unroll, 20
user directives, 17
VAST-F/Parallel, 8
Warning Message, 24