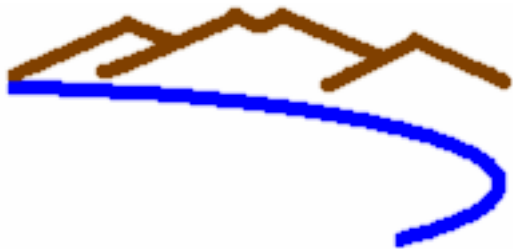


Crescent Bay Software

---

# VAST-F/Parallel

Automatic Parallelizer for Fortran



May 2005

Version 2.3

# Preface

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

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

### Revision Record

Edition	Date	Description
1.0	4/96	First release of document in this form.
1.1	5/96	Added summary information to introduction, driver documentation, discussion of static vs. dynamic reduction associativity, parallel region listing information, and minor corrections.
1.3	9/96	Added documentation of user-directed parallel directives.
1.4	11/97	Added information on expanded parallel region optimization.
2.0	8/98	OpenMP information added.
2.1	2/99	Minor updates.
2.2	11/03	Minor updates.
2.3	5/05	Minor updates.

### Notices

Copyright (C) 1996-2005, Crescent Bay Software Corp. No unauthorized use or duplication is permitted. All rights reserved.

*VAST is a registered trademark of Crescent Bay Software.*

Crescent Bay Software, 10950 Washington Blvd. Suite 230, Culver City 90232  
USA. Fax: (310)-836-7313 Phone: (310)-836-5183. Web:  
<http://www.crescentbaysoftware.com>.

# Table of Contents

<b>Preface</b>	<b>i</b>
<b>1. Introduction</b>	<b>1</b>
Overview .....	1
Automatic Parallelization .....	1
Expectations .....	1
OpenMP Directives .....	2
Superscalar Optimizations .....	2
Optimizations .....	2
Files .....	2
User interaction – Automatic Parallelization .....	2
User Interaction -- OpenMP .....	3
How to use this guide .....	3
Optional features .....	3
Inline expansion .....	4
Potential Error Detection .....	4
Automatic Parallelization and/or OpenMP .....	4
<b>2. Invoking VAST-F/Parallel</b>	<b>5</b>
pf90 Driver .....	5
Usage .....	5
File Extensions .....	6
pf90 Options .....	6
Example .....	7
Command Line .....	7
VAST-F/Parallel Switches .....	8
Transformation control with -e (enable) and -d (disable) .....	8
Additional transformation switches: -g and -f .....	10
VAST-F/Parallel Options .....	10
Processors .....	11

<b>3. User directives</b>	<b>12</b>
Overview .....	12
VAST directive format.....	12
VAST directive summary.....	13
Transformation directives .....	14
NOCONCUR/CONCUR .....	14
SKIP.....	14
CNCALL .....	14
NOASSOC/ASSOC.....	14
Data dependency directives.....	14
NOSYNC/SYNC .....	14
NOEQVCHK/EQVCHK .....	15
PERMUTATION.....	15
RELATION .....	15
Listing directives .....	15
NOLIST/LIST.....	15
SWITCH directive .....	15
Specifying directives on invocation .....	16
<b>4. VAST listing</b>	<b>17</b>
Overview .....	17
Input source listing.....	17
Diagnostic messages .....	18
Optimization inhibition messages.....	18
Informative messages.....	18
Error messages .....	18
Output source listing .....	19
Summaries.....	19
Listing control switches .....	19
Listing page length.....	20
<b>5. Parallelizing Calculations</b>	<b>21</b>
Overview .....	21
Creating and executing parallel regions .....	21
Private and Shared.....	21
Threshold test .....	24
Suppressing the threshold test.....	25
Changing the threshold value.....	25

Scheduling Loop Iterations .....	26
Dynamic Scheduling.....	26
Static Scheduling .....	26
How VAST Schedules Loops .....	26
Conditional parallelization .....	26
Inner loops.....	27
Reductions.....	27
Parallelizing loops with external calls.....	28
Parallel cases .....	29
Extending parallel regions.....	30
Parallel Outer Unroll .....	31
Parallel Array Syntax .....	31
<b>6. Loop Optimizations</b> .....	<b>33</b>
Overview .....	33
Loop fusion .....	33
Loop rerolling.....	34
Loop unrolling.....	34
Store postponement.....	36
Reduction unrolling.....	36
Loop peeling.....	37
IF loops to DO loops .....	37
DO WHILE to DO .....	38
Loop interchange.....	39
Outer loop unrolling.....	39
Loop collapse .....	40
<b>7. Data dependency analysis</b> .....	<b>41</b>
Overview .....	41
Ambiguous subscript resolution.....	41
Data dependency directives.....	42
NOSYNC -- declaring non-recursion .....	42
NOEQVCHK -- non-recursion in equivalences.....	42
RELATION -- specifying relationship between variables.....	43
PERMUTATION -- declaring safe indirect addressing.....	43
Scalar Dependencies .....	44
Carry-around scalars .....	44

Equivalenced scalars .....	44
<b>8. Inline expansion</b> .....	<b>46</b>
Overview .....	46
Introduction .....	46
Automatic inline expansion .....	47
Automatic inlining criteria .....	47
Avoiding inlining .....	47
Automatic inlining level .....	47
Explicit inline expansion .....	48
Source code access .....	48
Inline expansion directives .....	48
AUTOEXPAND directive .....	48
Explicit mode .....	48
EXPAND directive .....	49
NEXPAND directive .....	49
SEARCH directive .....	49
Where to get the source code .....	50
Same file .....	50
Explicitly named file .....	50
Implicitly named file .....	50
Possible problems .....	50
Separate compilation .....	50
Code size .....	51
Debugging .....	51
Compilation rate .....	51
Nested expansion .....	51
Analysis inhibitors .....	51
Expansion inhibitors .....	51
Inhibitors specific to automatic expansion mode .....	52
Inline expansion abilities .....	52
Naming conventions .....	52
Common blocks .....	53
Arguments .....	53
Unique names .....	53
Labels .....	53
Returns and return values .....	53
Inlining ENTRYs .....	53
Cleanup of inlined code .....	54
Example .....	54
Expansion of SAVE and DATA .....	56
Inline expansion user messages .....	56

Inline expansion summary .....	56
User messages .....	56
<b>9. Interprocedural Analysis</b>	<b>58</b>
Parallel Calls .....	58
Examples .....	59
Performance benefit .....	60
Error detection with IPA .....	60
<b>10. Diagnostic Messages</b>	<b>61</b>
Overview .....	61
Data dependency conflicts .....	61
Translation Diagnostics .....	63
Statement types .....	63
Branches .....	64
External references .....	65
DO statement .....	65
Miscellaneous .....	65
Warnings .....	66
Potential Errors .....	66
Obsolescent Features .....	66
Syntax errors .....	66
Internal Errors .....	67
Directive Errors .....	67
Notes .....	67
<b>11. OpenMP Support</b>	<b>68</b>
Overview .....	68
User responsibility .....	69
Important switches .....	69
Tradeoffs: OpenMP and automatic parallelization .....	69
PARALLEL and END PARALLEL directives .....	69
Shared and private .....	70
DO directive .....	71
Parallel sections .....	71
Critical sections .....	72
Barriers .....	72

Single .....	72
Master.....	73
Atomic.....	74
Ordered regions.....	74
Threadprivate .....	75
Run-time library routines .....	75
Execution Environment Routines .....	76
Lock Routines .....	76
OpenMP environment variables.....	76
Nested Parallelism.....	76
<b>12. Further Information</b>	<b>78</b>
Crescent Bay Software and VAST.....	78
Questions or Comments .....	78
<b>Index</b>	<b>79</b>

# 1. Introduction

---

## Overview

This document describes the features and options of the VAST-F/Parallel parallel processing product. VAST-F/Parallel is a software tool that optimizes Fortran programs for execution on SMP parallel systems. VAST-F/Parallel automatically distributes calculations in a program to parallel threads. It does this by examining the program and restructuring it to use a parallel thread library. Parallel code can be discovered automatically by VAST (“automatic parallelization”) or explicitly marked with OpenMP directives. VAST-F/Parallel optimizes Fortran 77, Fortran 90, and Fortran 95 programs.

### Automatic Parallelization

VAST-F/Parallel 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 OpenMP directives or VAST assertion directives.

## OpenMP Directives

VAST-F/Parallel processes the complete set of OpenMP directives, as defined in the *OpenMP Fortran Application Program Interface* (Version 1.0, October 1997). These directives allow you to specify parallelism at a higher level in the program.

## Superscalar Optimizations

VAST-F/Parallel can generate a reasonable degree of parallelism on many programs, depending on the algorithms used, coding style, size of arrays, and other factors. In addition to improving performance by using parallel threads, VAST-F/Parallel also provides a suite of high-level scalar optimizations that improve the performance of code in each thread.

---

## Optimizations

VAST-F/Parallel'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.
- Superscalar optimizations in combination with parallel optimizations.

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

---

## Files

When VAST-F/Parallel 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. This file is ready for compilation by a Fortran compiler.

Normally, VAST is used through the `p£90` driver, which combines execution of VAST with running the compiler, so you just compile as usual.

---

## User interaction – Automatic Parallelization

VAST-F/Parallel 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.

---

## User Interaction -- OpenMP

When using OpenMP directives, you would normally take a "top down" approach. This would involve examining the whole application and looking for parallel opportunities in the basic structure of the code at the highest level. To insert OpenMP directives, you must make sure that the section of code you are parallelizing does not have data dependencies that will cause wrong answers when executed in parallel. Threads cannot depend on values calculated in other threads without some synchronization being inserted. You must also make sure that the variables in the parallelized code have the correct scoping (private or shared).

OpenMP features are described in Section 11.

---

## How to use this guide

Section 2 describes how to invoke VAST-F/Parallel. 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.

Section 11 describes OpenMP support, for parallelism beyond automatic recognition.

---

## Optional features

Most VAST-F/Parallel 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.

## **Inline expansion**

To enable automatic expansion of calls to small subroutines and functions contained in the same input file, use `-e78` on the invocation. (See the chapter on Inlining for further details.)

## **Potential Error Detection**

To detect potential errors, use the `-p r` switch.

## **Automatic Parallelization and/or OpenMP**

By default, OpenMP directives are handled when they are encountered, and loops are automatically parallelized as well. If you want only OpenMP handling, use the `-dc` switch. If you want only automatic parallelization, use the `-f4` switch.

## 2. Invoking VAST-F/Parallel

---

### pf90 Driver

Normally, you will invoke VAST-F/Parallel via the VAST-F/Parallel driver. (This is named to correspond with the Fortran compiler that will be invoked – pf77 for Absoft f77 or for g77, pf90 for Absoft f90, plf for IBM xlf, etc. For simplicity we'll use pf90 in the text.) It's invoked just like your normal compiler, and you use all your normal compiler switches with it. The driver takes care of running the automatically parallelization precompilation phase, compiling the code, and linking with the threads library. For example,

```
pf90 myprog.f
```

Will create an a.out that is ready to run and will automatically create and use parallel threads.

### Usage

```
pf90 [<pf90_option>]... [<fcomp_option>]...  
      [-Wv,<v_option>[,<v_option>]...] [files]
```

where:

<pf90\_option> represents any pf90 driver option (see “pf90 Options” below).

<fcomp\_option> represents any Fortran compiler option. All normal compiler options can be used.

<v\_option> represents any Vast option except the -o option.

pf90 options and input files may appear in any order.

## File Extensions

1. filename with a .f90 suffix: free form F90 source file.
2. filename with a .s suffix: assembler source file.
3. filename with a .o suffix: object file.
4. filename with a .a suffix: archive file.
5. filename with a .f suffix: fixed f77 source file

## pf90 Options

- c** suppress the link editing phase of the compilation and do not remove any object files produced.
  - dryrun** display but do not execute cc internal commands. pf90 will still check for the existence of essential executable files.
  - keep** keep intermediate VAST files. The intermediate files produced are as follows. For an input file “file1.f” the intermediate file will be named “Vfile1.f”.
  - l<suffix>** search the library file lib<suffix>.a.
  - L<dir>** search directory <dir> for libraries prior to searching the default directories.
  - o<name>** name the final output file <name>. When used with -c, names the object file, otherwise names the link edited output file. (The default link edited output file is named a.out.) -o is ignored if the -vo option is used.
  - v** display pf90 internal commands as execution progresses.
  - vo** execute VAST pass only. Certain compiler options included on the command line may be ignored.
  - vn** execute Fortran compiler only (no VAST). VAST options included on the command line are ignored. Note that the -vo and -vn options are mutually exclusive.
  - w** suppress warning message
  - Wv** hand off arguments to pass VAST. VAST options must be separated by commas and may not contain embedded blank space characters.
- Examples:
- Wv,-da
  - Wv,-e78,-da
- Yvc,<path>** substitute for the default Fortran compile an alternate executable whose pathname is specified by <path>.
  - Yvv,<path>** substitute for the default VAST binary an alternate executable whose pathname is specified by <path>.

## Example

```
pf90 -o par.exe -Wv,-Pprocs8,-dc file1.f
```

Executes VAST with the number of processors specified to be 8, and no automatic parallelism (OpenMP only). Compiles the intermediate file Vfile.f using the Fortran compiler and invokes linker to produce the executable output file par.exe.

---

## Command Line

If you want to run the parallel preprocessor yourself rather than through the pf90 driver, VAST-F/Parallel is executed directly by the command:

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

*output* = compilable output file. The default output file name is the input file name prefixed by V. (Note that any module information files generated for Fortran 90 modules may be placed in a separate directory via the -L option; see below.

*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:

```
vastfp crunch.f
```

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

### Example 2:

To run sub.f through VAST, tell it to use 8 threads, save the VAST listing in file sub.lst, and the translated code in sub.v.f:

```
vastfp -l sub.lst -Pproc8 -o sub.v.f sub.f
```

---

## VAST-F/Parallel 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 5)

-q Listing options to disable. (Section 5)

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

*[-r switches] [-n switches]*

-r Output formatting options to enable. (Section 14)

-n Output formatting options to disable. (Section 14)

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
i	Parallelize inner loops	off
j	Enable kernel recognition (maxtix multiply, etc.)	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
p	Allow loop collapse.	on
x	Create optimized source file.	on
0	Do threshold testing on parallel regions.	on
1	Convert array syntax to DO loops	on
7	Automatically expand called routines inline.	off
8	Search input file for expandable routines.	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/Parallel 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.

**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.

**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.

**1:** Convert array syntax to `DO` loops.

**7:** Automatically expand inline the bodies of subroutines and functions that meet certain criteria. `-e7` is equivalent to the `AUTOEXPAND` directive with file scope. (See later section on inlining).

**8:** Search input file for expandable routines.

Note that some switches correspond to directives and some do not. Those that correspond to directives (`a`, `c`, `d`, `e`, `7`) 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**: Enable free-form input. (Default: ON for .f90 files, OFF for .f files.)

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

**s**: 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. Default: ON.

**u**: Allow 132-column fixed format input.

**6**: Allow expansion of routines with SAVED variables or DATAed variables that are stored into. See the section on inlining for more details. Default: OFF.

---

## **VAST-F/Parallel 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 options are:

### **[-C *routines*]**

Names of concurrently-callable external routines.

### **[-H *path*]**

Directory in which to search for INCLUDE files and for module information files.

### **[-I *routines*]**

Names of routines that should be expanded inline.

### **[-J *nnn*]**

Level to automatically expand from bottom of call tree (default 1).

### **[-L *path*]**

Directory in which to put any generated Fortran 90 module information files. Module information files (named <module\_name>.vo) are separate from object files, and are used by VAST to process USE statements in other subprograms that reference these modules.

**[-M *nnn*]**

Maximum threshold (code lines) for automatic inlining.

**[-N *routines*]**

List of routines not to inline.

**[-Ppage *nnn*]**

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

**[-Pprocs *n*]**

Number of processors to generate code for. See the next section.

**[-S *filenames*]**

Files in which to search for routines to be inlined.

**[-Y *routines*]**

Names of routines that should be expanded inline (including called routines; nested expansion).

**[-Z *output format parameter=value*]**

Output formatting parameters. See section 8 for allowable parameters

---

## Processors

In order to create an appropriate number of threads, the number of processors needs to be known. VAST-F/Parallel has a command line option (`-Pprocs`) to allow you to specify the number of processors you want to compile for.

Basically, VAST will create one thread for each processor. The syntax is:

```
-Pprocs N
```

Where N is an integer constant. This VAST-F/Parallel command line option allows the user to supply processor configuration information so that the generated code can be tailored to the hardware. For example, the command line:

```
vastfp -Pprocs 10 myfile.f
```

specifies that 10 threads should be created for parallel regions in “myfile.f”. If this option isn't specified at all, VAST-F/Parallel will supply a default number of processors. Currently the default is:

```
-Pprocs 2
```

*NOTE: on some systems, VAST-F/Parallel controls the maximum number of threads that can be run in parallel, based on the license you have purchased. If you need to run on more processors, then you need to upgrade your license to allow a larger number.*

# 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/Parallel 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.

*(NOTE: The directives in this section are used to aid in automatic parallelization. For OpenMP directives, where you explicitly control what will be done in parallel and what will be synchronized, see Section 11.)*

---

## 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.

CVD\$	NODEPCHK	<i>(Ignore potential data dependencies in the next loop.)</i>
CVD\$R	SKIP	<i>(Turn off transformation in the rest of this routine.)</i>
CVD\$F	LIST	<i>(Turn on listing for rest of file(s).)</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/Parallel 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
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
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
NOUNROLL/ UNROLL	Unroll loop.	UNROLL	LRF
AUTOEXPAND/ NOAUTOEXPAND	Automatically expand small routines inline.	NOAUTO	R
EXPAND	Expand particular routine(s).	n/a	I
NEXPAND	Nested expansion of particular routine(s).	n/a	I
SEARCH	Supply file/path location(s) for sources for inlined routines.	n/a	I

---

## 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/Parallel 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.

---

## 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 l` 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.

---

## 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/Parallel 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/Parallel 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:

<b>Code</b>	<b>Meaning</b>
<b>A</b>	No action requested. (Optimization turned off.)
<b>B</b>	Parallel case. (Two sections of code will be executed in parallel.)
<b>D</b>	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.
- I** Inlined. (Routine referenced on this line is expanded.)
- N** Not chosen. (Loop is not optimized.)
- P** Parallelized. (Iterations of the loop will be passed out to threads.)
- 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.)

---

## 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>b</b>	List input line #s in columns 73-80 of output listing.	on
<b>c</b>	List data dependency conflict messages.	on
<b>e</b>	List event summary at end of routine.	on
<b>f</b>	List fatal error messages.	on
<b>g</b>	List translation diagnostics.	on
<b>h</b>	List input source lines.	on
<b>i</b>	List included lines.	on
<b>l</b>	Produce a listing.	on
<b>n</b>	List translated code.	on
<b>p</b>	List loop summary at end of routine.	on
<b>r</b>	Detect and list potential errors	off
<b>t</b>	Terminal listing: format output for 80 columns.	on
<b>u</b>	Show extent and disposition of loops in source.	on
<b>w</b>	List warning messages.	on
<b>y</b>	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 `-qtwe`. If you wanted to get potential errors listed, you would use `-pr`.

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

**r:** VAST-F/Parallel 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. `-qt` 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 parallelizes loops by using a parallel threads library. Parallel threads are assigned different pieces of the calculation to complete.

---

## 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 creates a new parallel routine and inserts the code for the parallel region in it. A parallel call to the new routine (using the library function `vp_parallel`) is then inserted in place of the original code.

Each parallel thread executes the entire parallel region. When parallel loops or parallel cases are encountered in the parallel region, each parallel thread does only its portion of the work. Other calculations in the parallel region are performed redundantly in each thread.

---

## Private and 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 thread. Variables that are global to a parallel region (values come from outside the region, for instance from `COMMON`) must be shared with all other threads.

When creating a parallel thread routine for a parallel region, VAST passes as arguments (or includes COMMON references for) all variables that need to be shared among the tasks, and declares locally within the new routine all variables that need to have a private copy for each thread. This way, parallel threads will be using the same addresses (passed in or from common) to access shared items, and private items will be allocated on each threads stack and will thus be different in each thread.

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 thread, so it declared locally in the generated parallel routine. *i* and *j* also must be private and are also not in the parameter list. The value of *kk* is needed by all tasks, and so it is passed to the parallel routine and thereby shared.

```

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:

```

subroutine example1
common a(5000), b(5000), d(5050), kk
external _vp_example11
k = kk
call vp_pcall (_vp_example11, 1, k) Parallel call
return
end

subroutine _vp_example11 (vp_mythread,
1  vp_nmthreads, k) Pass initial value of k to all threads
common a, b, d, kk “kk” is shared by all threads
INTEGER k, kk
REAL a(5000), d(5050), b(5000)
INTEGER i, vp_mythread, vp_nmthreads, j3, j4, j5
REAL t “t” is private – each thread gets its own copy

```

```

j3 = (5000 + vp_nmthreads - 1)/vp_nmthreads
j4 = vp_mythread*j3 + 1
j5 = min(5000 - j4 + 1, j3)
do i = j4, j5 + j4 - 1   Parallel loop
    t = sqrt(a(i)) + d(k+i*kk)
    b(i) = t + 1.0/t
end do
return
end

```

In the example below, the array a must be shared by each thread, so when the parallel thread routine is created, a is passed to it.

```

real a(99,99)
do j = 1, m
    do i = 1, n
        a(i,j) = sqrt(b(i,j))
    enddo
enddo

```

Finally, in the example below, e is an array that needs to have a private copy in each task. It is declared locally in the parallel routine to accomplish this; e is not passed to the parallel routine. e is called a private array.

```

common b(99,99), c(99,99), d(99,99)
real e(99)
do j = 1, m   This loop is parallel
    do i = 1, n
        e(i) = b(i,j) + c(i,j)   "e" needs to be private
    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

```

---

## Threshold test

Parallelization can slow down execution if the loop contains insufficient work to compensate for the added overhead. If the loop nest iteration counts can be determined at compile time, VAST makes a decision whether or not to parallelize the loop. If the loop nest iteration counts cannot be determined at compile time, VAST inserts code to do a threshold test at run time: if it is computed at run time that the loop has a lot of work then the loop is done in parallel mode, otherwise the serial version is executed. VAST adjusts the threshold value based on the number and type of operations in each loop pass.

The threshold test is generated only for single and double-nested parallelized loops. The threshold test is not applied to loops containing more than one other loop; they are large enough that they are assumed to contain enough work to make parallelization pay off.

The default value of the threshold is 1000. It is a measure of the number of clock cycles of work needed in a parallel region to allow parallel execution to be worthwhile.

For double loops, the product of the inner and outer loop iteration counts is compared to the threshold value, which is automatically adjusted based on the amount of work in the loop; it is adjusted inversely proportional to the number of statements and operations in the loop.

If the inner loop iteration count depends on the outer loop, the square of the outer loop count is compared. For single loops, the iteration count is directly compared to the threshold value. The example below show a simple double-nested loop, and the threshold comparison that results. When the loop must be cut out into a new parallel routine, the call to the new routine is done conditionally based on the threshold test.

```
do j = 1, .m
  do i = 1, n
    a(i,j) = 0.0
  enddo
enddo
```

Translation: (With all unrolling turned off, for clarity using `-fx -U0`):

```
if (m*n .gt. 1000) then      Test the threshold
  call vp_pcall (_vp_simple1 , 3, m, n, a)
else                          Not big enough for parallel
  do j = 1, m      Scalar version
    do i = 1, n
      a(i,j) = 0.
    end do
  end do
```

```

endif
end
subroutine _vp_simple1 (vp_mythread,
1  vp_nmthreads, m, n, a)  Parallel thread routine
INTEGER m, n
DOUBLE PRECISION a(1000,1000)
INTEGER j, i, vp_mythread, vp_nmthreads, j1, j2
j3 = (m + vp_nmthreads - 1)/vp_nmthreads
j4 = vp_mythread*j3 + 1
j5 = min(m - j4 + 1, j3)
do j = j4, j5 + j4 - 1      Do this thread's chunk
  do i = 1, n
    a(i,j) = 0.
  end do
end do
return
end

```

## Suppressing the threshold test

Array dimensions are looked at to determine if the size of the array allows an iteration count that exceeds the threshold. Inner loops having an explicit iteration count that exceeds the threshold are automatically parallelized without needing to request translation of inners. A `SELECT (CONCURRENT)` directive suppresses the threshold test for the selected loop.

The threshold test can also be suppressed globally via the `-d0` switch. If you know that the threshold test is unnecessary for all the loops in a routine or program, because all the loops contain a large amount of work, this is a good way to reduce code size and runtime overhead.

## Changing the threshold value

You may want to change the threshold value depending on the system load of the system you generally run on. Use of parallel processors affects and is affected by the other jobs running on a system. If you generally run in dedicated mode, or on a lightly loaded system, then you will want a lower threshold value. If you generally run on a more heavily loaded system, then you will want a higher threshold value.

A different threshold value for a loop can be supplied with the `THRESHOLD` directive. You can supply a different maximum threshold value for the whole file via the `-T` parameter on the invocation or on the `SWITCH` directive.

---

## Scheduling loop iterations

VAST supports two methods of scheduling loop iterations for parallel execution: dynamic and static scheduling.

### Dynamic scheduling

In dynamic scheduling, the iterations of the loop are passed out to processors in chunks, with whichever processor is next available processing the next chunk. Dynamic scheduling is the most flexible, if you have varying amounts of computation in a loop nest. If one thread is taking a long time to complete its current chunk, other threads that have already finished their initial chunk will continue to process the rest of the loop. Dynamic scheduling helps with load balancing of the parallel threads.

The problem with dynamic scheduling is that checking to see what chunk is next takes additional overhead. Also, as the chunks that a processor gets may not be contiguous, dynamic scheduling may result in less efficient use of the caches of the parallel processors.

### Static scheduling

With static scheduling, the iterations of the loop are partitioned among the threads at the start of the calculation. Each loop completes its whole allocation, and then the calculation moves on. As there is no need for further interaction, static scheduling has less overhead. Also, static scheduling insures that all iterations a thread will process are consecutive, which may result in better use of the cache.

### How VAST Schedules Loops

By default, VAST uses static scheduling for most loops. Loops that contain concurrent calls (see the `CNCALL` directive or the `-C` parameter) are processed with dynamic scheduling, as they are assumed to have large (and varying) computational loads that can offset the additional dynamic overhead.

If you are having load balancing problems, you can try the `-el` switch, which will cause dynamic scheduling to be used for all loops. This might be useful if you have many large loops with lots of conditional calculations, where big calculations may be done on some iterations and not on others.

---

## 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 `-e i` option switch is enabled and no outer loop is available, inner loops will be analyzed for both parallelization and superscalar. 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 thread can execute at a time. This is done as in the example below:

```
subroutine reduct ( a )
  real*8 a(100000)
  do 200 i = 1, n
    s = s + a(i,j)
200  continue
  print *, s
end
```

Translation:

```
...
call vp_pcall (_vp_reduct1, 2, s, a)
print *, s
end
subroutine _vp_reduct1 (vp_mythread,
1  vp_nmthreads, s, a)
  REAL s
  DOUBLE PRECISION a(10000)
  INTEGER i, vp_mythread, vp_nmthreads, j3, j4, j5
  REAL s1
  s1 = 0
```

```

j3 = (10000 + vp_nmthreads - 1)/vp_nmthreads
j4 = vp_mythread*j3 + 1
j5 = min(10000 - j4 + 1, j3)
do i = j4, j5 + j4 - 1  Parallel loop
    s1 = s1 + a(i)      Sum into private scalar
end do
call vp_critical      Critical region start
s = s + s1           One thread at a time
call vp_endcritical  End of critical region
return
end

```

---

## 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:

```

subroutine cncallit ( a, x, n )
real a(10000), x
external _vp_cncallit1
call vp_pcall (_vp_cncallit1 , 3, n, x, a)
end
subroutine _vp_cncallit1 (vp_mythread,
1 vp_nmthreads, n, x, a)
INTEGER n
REAL x, a(10000)

```

```

    INTEGER i, vp_mythread, vp_nmthreads, j1, j2,
1    vp_ticket
    external vp_ticket
    call vp_setdo (%VAL(1), %VAL(n), %VAL(16))
    do while (vp_ticket(j1,j2) .ne. 0)
    do i = j1, j2
        call subr (a(i), x)
    end do
    end do
    return
end

```

CNCALL automatically implies INNER if it is applied to an inner loop, as in the preceding example. It also implies dynamic scheduling, as seen in the example by the calls to vp\_ticket.

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 execute them as parallel cases. In addition, at times VAST will split up loops into sections and execute each section as a parallel case. In the example below, the first loop nest (202) is completely independent of the second loop (302). VAST can create code so that one thread will execute the first nest, will a second thread 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

```

---

## 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 thread routine. 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 then 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/Parallel will move parallel regions outside of non-tightly-nested outer loops, so that the non-parallel loop will still be included in the parallel thread routine.

```

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 outer unroll

When an outer loop is unrolled inside of an inner loop, the unrolled loop that remains can still be parallelized.

```
do j = 1, 100    Loop will be parallelized and unrolled
    do i = 1, n
        a(i,j) = b(i,j) * c(i)
    enddo
enddo
```

---

## Parallel array syntax

VAST-F/Parallel “scalarizes” array syntax and then parallelizes and optimizes the resulting loops. 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: (The two loops resulting from the array syntax are collapsed into one long loop).

```

subroutine arrsyn ( a, b, x )
integer j1, j2
real*8 a(400,400), b(400,400), x
external _vp_arrsyn1
integer j5, j6, j7
doubleprecision d1, d11
d1 = 0
call vp_pcall (_vp_arrsyn1, 3, d1, a, b)
x = d1
end
subroutine _vp_arrsyn1 (vp_mythread,
1  vp_nmthreads, d1, a, b)
DOUBLE PRECISION d1, a(400,400), b(400,400)
INTEGER j2, j1, vp_mythread, vp_nmthreads
DOUBLE PRECISION d11
d11 = 0
j5 = (160000 + vp_nmthreads - 1)/vp_nmthreads
j6 = vp_mythread*j5 + 1
j7 = min(160000 - j6 + 1, j5)
do j2 = j6, j7 + j6 - 1
    d11 = d11 + abs(a(j2,1)-b(j2,1))
end do
call vp_critical
d1 = d1 + d11
call vp_endcritical
return
end

```

# 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)
```

---

## Loop rerolling

VAST-F/Parallel will take loops that have been hand unrolled and put them back into their original state. This may allow the compiler (or VAST-F/Parallel) 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 unrolling

Loops can be unrolled automatically under various criteria, or under user direction with the UNROLL directive or -U command line switch.

For example, loops will be unrolled automatically when there is only one operation in the loop (thus not affording enough computation to overlap with the branch); two copies of the body of the loop will be executed in these cases.

Example:

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

Translation:

```
      J1S = IAND ( N, 1 )
C
      DO 100 I = 1, J1S
          C(I) = A(I) + B(I)
100  CONTINUE
C
      DO 101 I = J1S+1, N, 2
          C(I) = A(I) + B(I)
          C(I+1) = A(I+1) + B(I+1)
101  CONTINUE
```

Also, when the loop iteration count is 4 or less and the body of the loop is small, the loop will be completely unrolled and replaced with assignment statements.

Example:

```
      DO 100 J = 1, M
          DO 100 I = 1, 3
              A(I,J) = 0
100  CONTINUE
```

Translation:

```
      DO 100 J = 1, M
          A(1,J) = 0
          A(2,J) = 0
          A(3,J) = 0
100  CONTINUE
```

The `-U` command line option can be used to control the depth of unrolling. The numerical parameter supplied is used by VAST to determine the desired size of the unrolled loop in statements. For example, `-U16` indicates a desired size of 16 statements the loop; a loop with four assignments would be unrolled four times.

To allow you to control the unrolling process, the `UNROLL` directive is provided, as follows:

```
CVD$[ {L,R,F} ] [NO]UNROLL [ (nn) ]
```

When routine or file scope is specified, automatic unrolling of loops is enabled or disabled over that scope. The optional parameter, which must be a constant, specifies the threshold loop iteration count for automatic unrolling. Loops with an iteration count greater than this value will not be unrolled. By default, VAST-F/Parallel will automatically unroll loops of length 4 or less.

To force a loop to be explicitly unrolled, use the `UNROLL` directive with local scope immediately preceding the loop. In this case, the optional parameter is

taken as the number of times to unroll the loop. If a parameter is not supplied, VAST-F/Parallel uses an internally calculated function of the loop length, loop complexity, and default threshold.

Short inner loops are automatically unrolled if:

- The iteration count is constant, and below the threshold (default: 4)
- The iteration count times the number of statements in the loop is less than 32.
- The loop contains only assignment statements. No branches, I/O statements, or external references are allowed.
- The DO loop control parameters are integer.
- The last value of the loop index is not required after the loop is executed.

These restrictions do not apply to loops unrolled explicitly by directive. The only inhibitors in this case are assigned GOTOs and I/O keywords other than END=, ERR=, FMT=, and UNIT= . Outer loops may be unrolled by directive also.

## Store postponement

By default, array stores in unrolled loops are “postponed”:

```
DO 100 I = 1, 100
    C(I) = A(I) + B(I)
100 CONTINUE
```

Translation:

```
DO 101 I = 1, 100, 2
    C1 = A(I) + B(I)
    C2 = A(I+1) + B(I+1)
    C(I) = C1
    C(I+1) = C2
101 CONTINUE
```

Where the right-hand sides of all unrolled assignments corresponding to a single original assignment are computed and temporarily stored into scalars, and then the temp scalars stored into the actual array locations.

In some cases this allows the compiler to overlap more instructions. To disable this feature, use the `-fp` switch.

---

## Reduction unrolling

Reduction unrolling is a special case of inner loop unrolling. Multiple accumulator temporaries are created, to allow overlap of unrolled summation iterations. This transformation is under control of the associative option, as it changes the order of operations from the original.

Example:

```

do i = 1, n
  s = s + a(i)*b(i)
end do

```

Translation:

```

s1 = 0
s2 = 0
s3 = 0
s4 = 0
do i = 1, n, 4
  s1 = s1 + a(i)*b(i)
  s2 = s2 + a(i+1)*b(i+1)
  s3 = s3 + a(i+2)*b(i+2)
  s4 = s4 + a(i+2)*b(i+2)
end do
s = s + s1 + s2 + s3 + s4

```

---

## 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/Parallel 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
```

---

## Outer loop unrolling

This transformation allows outer loops to be unrolled inside of inner loops, providing more instructions to optimize in the inner loop without decreasing the iteration count of the inner loop. This optimization will only be done if the inner loop has only one operation in it or if it allows common expressions along the outer loop to be more easily eliminated.

For example, in the loop below  $C(I)$  can be fetched one time and used four times in the unrolled loop.

```
        DO 10 J = 1, 100  
            DO 20 I = 1, M  
                A(I,J) = B(I,J) + C(I)  
20        CONTINUE  
10       CONTINUE
```

Translation:

```
        DO J=1,100,4
```

```

        DO 20 I = 1, M
            C1X = C(I)
            A(I,J) = B(I,J) + C1X
            A(I,1+J) = B(I,1+J) + C1X
            A(I,2+J) = B(I,2+J) + C1X
            A(I,3+J) = B(I,3+J) + C1X
20      CONTINUE
      ENDDO

```

---

## Loop collapse

Loop nests that traverse all of the inner dimensions of the arrays in the loop can be automatically collapsed into single loops with larger iteration counts.

Collapse criteria:

- The loops must be tightly nested, with one loop index per array dimension.
- The loop bounds must be identical to the array bounds for the first N-1 dimensions, where N is the number of dimensions to be collapsed.
- All the array references that are indexed by the loops must conform, that is, have the same subscripting.

In the example below, all three dimensions are collapsed. The loops do not have to occur in the order of the subscripts.

```

      SUBROUTINE NEST ( L,M,N,A,B )
      DIMENSION A(L,M,N) , B(L,M,N)
      DO 100 K = 2, N-1
        DO 100 J = 1, M
          DO 100 I = 1, L
            A(I,J,K) = B(I,J,K)
          100 CONTINUE

```

Translation:

```

      ...
      DO 100 K = 1, L*M*(N-2)
        A(K,1,2) = B(K,1,2)
      100 CONTINUE

```

# 7. Data dependency analysis

---

## Overview

VAST-F/Parallel 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. 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 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 inlineable 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
ix = 1 both incx and incy
iy = 1 are passed as one)
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 SAVEed local data or DATAed 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:

```
vast2 -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.

# 9. 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/Parallel will use one thread to call SUBB and a different thread to call 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.

# 10. Diagnostic Messages

---

## Overview

This section shows selected VAST-F/Parallel 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 NEQVCHK 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/Parallel 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.)

# 11. OpenMP Support

---

## Overview

There are situations where automatic parallelization cannot find all available parallelism. For these situations, VAST provides user-directed parallelism through OpenMP directives. VAST supports the OpenMP interface, as defined in the *OpenMP Fortran Application Program Interface* (Version 2.0, November 2000). This specification is available from website [www.openmp.org](http://www.openmp.org), and should be used as the reference for OpenMP. (As of this writing most of the 2.0 interface is supported, with the exception of some features of thread-private variables.)

This chapter provides some discussion and examples of the use of OpenMP features and VAST's implementation of OpenMP, and is intended to support (not replace) the official OpenMP Specification.

OpenMP is an attempt to create “standard” set of directives for shared-memory parallel systems. While not a formal international standard, OpenMP has a number of adherents in the high-performance industry. It is of benefit to application programmers who wish to insert automatic parallelization in their codes to have a portable way to spell “Parallel DO”.

OpenMP contains a large set of directives and options. It allows dynamic range of parallel regions; “orphaned” directives can be placed in routines called from within parallel regions. This makes parallelization of larger sections of the program more tractable. The OpenMP API also specifies environment variables and run-time routines in support of the main directive suite.

Note that VAST-specific directives (start with **CVD\$**) are generally used to make general assertions and control automatic parallelization. These directives are not used for Open MP. The OpenMP directives start with **C\$OMP** or **!\$OMP** or **\*\$OMP** and are used to explicitly take control of what is to be parallelized.

---

## User responsibility

You must be careful when using OpenMP directives, because you have assumed control of the safety checks that the automatic parallelization normally does for you. You must make sure that the parallel calculations you request are truly independent of each other; for instance, no thread can depend on a calculation done in a different thread without appropriate synchronization. The variables in your parallel calculation must be scoped appropriately: you must determine whether each variable should be private to each thread or shared between the threads, and declare the variables appropriately.

---

## Important switches

By default, VAST-F/Parallel will automatically parallel loops and also create parallel code for OpenMP directives that it encounters. You may wish to have VAST-F/Parallel do only one or the other. This can be accomplished with these switches:

**-dc**: use this if you want only OpenMP processing, and no automatic parallelization.

**-f4**: use this to turn off OpenMP processing. OpenMP directives will be ignored.

---

## Tradeoffs: OpenMP and automatic parallelization.

If it will work on a code, automatic parallelization is a better choice. The compiler can choose the best loop/method for that particular platform. (Different platforms can have parallel overheads that are different, for example...)

If completely automatic parallelization is not parallelization enough of your calculation, you should try to use !VD\$ NOSYNC directives, to help the automatic parallelization.

Only sophisticated users should attempt to apply OpenMP directives to the source. Parallelization is confusing, and OpenMP has many options. To insert OpenMP directives you must have a good understanding of data dependencies and data scoping rules.

OpenMP allows larger parallel regions, and bigger parallel scope. Some applications may not have much parallelization available on the the loop nest level (where automatic parallelization excels), but may have significant parallel opportunities at a high level. For these applications, OpenMP may offer the best performance.

---

## PARALLEL and END PARALLEL directives

Parallel and End Parallel OpenMP directives delimit the parallel region. There are a number of possible clauses on the PARALLEL directives:

**PRIVATE:** Variables on the PRIVATE list have their own values in each parallel thread.

**SHARED:** Variables on the SHARED list shared by the parallel threads; all the threads see them at the same memory location.

**DEFAULT:** This just effects what is shared and private.

**FIRSTPRIVATE:** Variables are copied from shared versions to local versions. Each thread's private copy is initialize to have the same value as the variable prior to the parallel region..

**REDUCTION:** Specifies the operator and variable involved in a reduction. Local contributions to the reduction value are added to the final result in a locked region after the loop.

**IF:** The IF clause is changed into a Fortran block IF. The entire parallel region is duplicated in parallel and non-parallel forms. One branch of the block IF will wind up calling the new parallel routine, the other will have non-parallel code.

**COPYIN:** Copy THREADPRIVATE commons into the new parallel routine. Code at top of parallel routine moves data from master version of common block to the thread private version.

Here is an example of an OpenMP parallel region:

```
subroutine asdf(a,b,c,d)
  real a(1000),b(1000),c(1000),d(1000)
C$OMP PARALLEL IF(N.GE.1000)
C$OMP DO
  do i = 1,n
    a(i) = b(i) * c(i)
  enddo
C$OMP DO
  do i = 1,n+1
    d(i) = a(i) + c(i)
  enddo
C$OMP END PARALLEL
  print *, d
end
```

---

## Shared and private

All variables in a parallel region are either private or shared. In using OpenMP directives, you need to determine what category each of your variables fall into. By default, variables referenced in the user-directed parallel region are assumed to be shared, with the notable exception of the index variable of the do loops

inside the parallel region. Thus, variables that need to have different values in each iteration of the parallel loop need to be indicated with the private clause.

Example:

```
c$omp parallel do private(t,x) shared(a,b,c,y)
do i = 1, n
    t = a(i) + sqrt(b(i))
    c(i) = t + 1/t + myfunc(t, x, y)
enddo
```

---

## DO directive

The `do` directive is used to indicate that the immediately following loop is to be executed in parallel mode. The `parallel` directive and `do` directive can be combined as a `parallel do` directive. The `enddo` directive is optional.

The `DO` directive can have `PRIVATE`, `FIRSTPRIVATE`, and `REDUCTION` directives. There are handled same way as "PARALLEL" directive. Additional clauses available are:

**LASTPRIVATE:** Last value of private variable should be stored into shared counterpart.

**ORDERED:** ORDERED clause may cause the whole loop to be done sequentially.

**SCHEDULE:** SCHEDULE clause specifies how to allocate work to threads. (STATIC, DYNAMIC, GUIDED, RUNTIME). VAST uses STATIC scheduling by default.

Example:

```
!$omp parallel do private(t) reduction(+:s)
do i = 1, n
    t = myfunc(a(i)) + atan(b(i))
    s = s + t + 1/t
enddo
```

---

## Parallel sections

Parallel sections are disjoint pieces of code that can be processed at the same time. Each section will be assigned to a different thread. `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses are allowed and are handled as described for `PARALLEL`. A `SECTIONS` directive starts of the parallel construct, the parallel cases are separated by a `SECTION` directive, and the parallel construct is terminated by an `END SECTIONS` directive. The `SECTIONS` directive can be combined with the `parallel` directive.

Example:

```
c$omp parallel sections
```

```
c$omp section
    call my_code1 ( x, y )
c$omp section
    call my_code2 ( a, b, c )
c$omp section
    call my_code3 ( d, f(5), 2.0 )
c$omp end sections
c$omp end parallel
```

---

## Critical sections

Syntax:

```
c$omp critical
c$omp end critical
```

Only one processor is allowed in a critical section at one time. If one processor is in the critical section, any other processors wishing to enter the region must wait.

Example:

```
c$omp parallel do private (x1)
    do i = 1, n
        x1 = a(i) + myfunc(b(i), 2.0)
    c$omp critical
        x = func2 ( x, x1 )
    c$omp end critical
enddo
```

---

## Barriers

Syntax:

```
c$omp barrier
```

When a barrier is executed, any processor will wait at that point until all processors have arrived at that point. This can be useful to synchronize the processors at important points in the calculation.

---

## Single

The `SINGLE` directive and `END SINGLE` directives are used to delimit a region that only one processor is allowed to execute. Any other processors that show up wait at the end of the single region until the one processor is done executing it. In the example below, the single region is used compute  $c(m)$  before it is used in the loop below.

```

        subroutine show_single (a,b,c,m)
        real a(1000),b(1000),c(1000)
C$OMP PARALLEL
C$OMP DO
        do i = 1,1000
            a(i) = b(i) * b(i)
        enddo
C$OMP SINGLE
        c(m) = func3(sqrt(c(m)))
C$OMP END SINGLE
C$OMP DO
        do i = 1,1000
            a(i) = a(i) * c(i)/c(m)
        enddo
C$OMP END PARALLEL
        end

```

---

## Master

The MASTER and END MASTER directives are used to delimit a region where only the “master” processor is to execute. In the example below, the master region is used to do some I/O in the middle of the parallel region.

```

        subroutine test_master(a,b,c,s)
        real a(5000),b(5000,5000),c(5000),d(5000)
C$OMP PARALLEL
C$OMP DO REDUCTION(+:S)
        do j = 1,m
            do i = 1,n
                b(i,j) = a(i)**2
                s = s + b(i,j)
            enddo
        enddo
C$OMP BARRIER
C$OMP MASTER
        print *, s
C$OMP END MASTER
C$OMP DO

```

```

do i = 1,n
    d(i) = a(i)**2 + s
enddo
C$OMP END PARALLEL
end

```

---

## Atomic

ATOMIC is used to control access to a memory location; this location will be updated “atomically” (only one processor at a time can be doing an update to that location). It is analogous to a small critical region.

Example:

```

c$omp parallel do private (x1)
    do i = 1, n
        x1 = a(i) + myfunc(b(i), 2.0)
    c$omp atomic
        y(ibin(i)) = y(ibin(i)) + x1
    enddo

```

---

## Ordered regions

ORDERED regions are used to demarcate sections of the code that must be executed in exactly the same order as the original scalar code. Ordered regions are in general very inefficient and should be avoided.

```

c$omp parallel do ordered private (x)
    do j = 1, m
        do i = 1, n
            a(i,j) = sqrt(b(i,j)) + cos(c(i,j))
        enddo
        call p1 ( a(1,j), n )
    c$omp ordered
        do i = 1, n
            a(i+1,j+1) = a(i,j) + x*.0000001
        enddo
    c$omp endordered
        call p2 ( a(1,j), n )
    enddo

```

---

## Threadprivate

THREADPRIVATE is used for providing a separate copy of a common block or variable for each thread. It is useful when the parallel region has a dynamic range of a large block of routines.

Example:

```
      subroutine t02
      parameter ( nn = 50, mm = 25 )
      common /c2d/ a(nn,mm), b(nn,mm), n, m
      real*8 a, b, c, d,e
      common / tp2 / e(nn)
c$omp threadprivate (/tp2/)
c$omp parallel do
  do j = 1, m
    do i = 1, n
      e(i) = i*.01
    enddo
    call t02a ( j )
  enddo
  return
end
      subroutine t02a ( j )
      parameter ( nn = 50, mm = 25 )
      common /c2d/ a(nn,mm), b(nn,mm), n, m
      real*8 a, b, e
      common / tp2 / e(nn)
c$omp threadprivate (/tp2/)
  do i = 1, n
    a(i,j) = b(i,j) + e(i)
  enddo
  return
end
```

---

## Run-time library routines

The OpenMP API defines a number of subsidiary functions.

## Execution Environment Routines

OMP\_SET\_NUM\_THREADS(*n*)

OMP\_GET\_NUM\_THREADS

OMP\_GET\_MAX\_THREADS

OMP\_GET\_THREAD\_NUM

OMP\_GET\_NUM\_PROCS

OMP\_IN\_PARALLEL

OMP\_SET\_DYNAMIC

OMP\_GET\_DYNAMIC

OMP\_SET\_NESTED

OMP\_GET\_NESTED

## Lock Routines

These routines can be used for locked regions.

OMP\_INIT\_LOCK

OMP\_DESTROY\_LOCK

OMP\_SET\_LOCK

OMP\_UNSET\_LOCK

OMP\_TEST\_LOCK

---

## OpenMP environment variables

The OpenMP API defines four environment variables:

OMP\_NUM\_THREADS

OMP\_DYNAMIC

OMP\_NESTED

OMP\_SCHEDULE

First three are used by the run-time library to initialize the system. Environment variables may be overridden with OMP\_SET function calls.

Schedule type can be one of: { STATIC, DYNAMIC, GUIDED }

---

## Nested Parallelism

The OpenMP API allows parallel regions to be nested. Currently VAST-F/PARALLEL actually uses all available threads on the first (outermost) parallel region, and just uses the current thread for any subsequent (inner) parallel regions.

# 12. Further Information

---

## Crescent Bay Software and VAST

Our group (as the High-Performance Compiler Optimization Group at Pacific-Sierra Research) 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, as well as many desktop and embedded systems. 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/Parallel**, other VAST end-user products are available:

**VAST-F/toOpenMP**, translator into OpenMP. Automatically converts Fortran programs to use OpenMP directives. Provides an excellent start in porting codes to take advantage of OpenMP. Can be used with any compiler that supports OpenMP (including VAST-F/Parallel).

**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.

---

## Questions or Comments

If you have any questions or comments about VAST-F/Parallel, please do not hesitate to contact us – contact information is in the preface of this document.

# Index

## A

ambiguous subscript resolution, 41  
array constants, 44  
array syntax, 9, 31  
associative, 36  
associative transformations, 9, 14  
ATOMIC, 74  
AUTOEXPAND directive, 48  
automatic distribution of loop iterations, 21  
automatic inlining, 47

## B

backward transfers, 64  
barrier, 72  
branches out of the loop, 64

## C

**C\$OMP directives**, 68  
carry-around scalars, 44  
CNCALL, 14  
CNCALL directive, 28  
command line, 7

concurrent call, 28  
critical region, 27  
critical section, 72

## **D**

-D invocation parameter, 16  
data dependencies, 27, 41  
Data Dependency Conflict, 18  
data dependency directives, 42  
data dependency messages, 61  
diagnostic messages, 17, 18, 61  
DO directive, 71  
driver, 5  
dynamic scheduling, 26

## **E**

EQUIVALENCE, 9, 44  
error detection, with IPA, 60  
event summary, 19  
EXPAND directive, 49  
expansion of SAVE and DATA, 56  
explicit inlining, 48  
external references, 65

## **F**

-F command line parameter, 16  
feedback of array elements, 61  
files, 2  
FIRSTPRIVATE, 71

## **G**

-G fusion parameter, 33

## **I**

IF loops into DO loops, 37  
INCLUDE files, 20  
inline expansion, 9, 46

INNER directive, 27  
inner loop unrolling, 34  
inner loops, 27  
input files, 7  
input line numbers, 20  
Internal Error, 18  
interprocedural analysis, 58  
IPA, 58

## **L**

LASTPRIVATE, 71  
listing, 7, 17  
listing control switches, 19  
listing, page length, 20  
listing, wide format, 20  
load balancing, 26  
Loop Collapse, 40  
loop disposition graph, 17  
loop fusion, 33  
loop interchange, 39  
loop optimizations, 33  
loop rerolling, 34  
loop summary, 19  
loop unrolling, 34  
loops containing subroutine calls, 28

## **M**

MASTER, 73  
module information files, 10  
Module information files, 10  
multiple store conflict, 62

## **N**

nested expansion, 51  
nested parallelism, 77  
NEXPAND directive, 49

NOASSOC, 14  
NOCONCUR, 9, 14  
NOEQVCHK, 15, 44  
NOEQVCHK directive, 42  
NOLIST, 15  
NOSYNC, 14, 42  
Note Message, 18

## **O**

OpenMP, 2, 68  
OpenMP environment variables, 76  
optimizations, 2  
options, 7, 10  
ORDERED regions, 74  
outer loop unrolling, 39  
output file, 7, 9  
output formatting, 8, 11

## **P**

Parallel and End Parallel OpenMP directives, 69  
parallel case optimization, 10  
parallel cases, 29  
parallel regions, 21  
Parallel sections, 71  
parallelization, 21  
peeling, loop, 37  
PERMUTATION, 15  
PERMUTATION directive, 44  
pfor driver, 5  
potential error detection, 20  
potential errors, 66  
potential feedback, 41, 61  
-Pprocs, 11  
private, 70  
private array, 23

private variables, 21

processors, 11

## **R**

recurrences, 41

REDUCTION, 71

reduction operations, 27

RELATION, 15

RELATION directive, 43

routine expansion, 46

## **S**

-S parameter, 48

scalar dependencies, 44

SEARCH directive, 48, 49

shared, 70

shared variables, 21

SINGLE directive, 72

SKIP, 14

static scheduling, 26

store postponement, 36

summation, 27

SWITCH directive, 15

switches, 7, 8, 15

Syntax Error, 18

## **T**

THREADPRIVATE, 75

threads, default number, 11

THRESHOLD directive, 25

threshold test, 24

transformation control, 8

Translation Diagnostic, 18

translation diagnostics, 63

## **U**

UNROLL directive, 34, 35

unrolling, outer loops, 39  
unrolling, reductions, 36  
user directives, 12  
user-directed parallelism, 68

## **V**

VAST-F/Parallel, 1  
vp\_ticket, 29

## **W**

Warning Message, 18