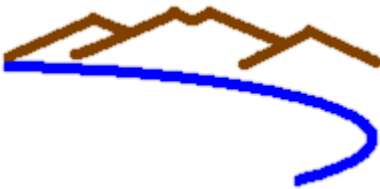


Crescent Bay Software

---

# VAST-C/Parallel

Automatic Parallelizer and Optimizer for ANSI C



March 2007

Version 1.4

# Preface

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

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

### Revision Record

Edition	Date	Description
1.0	3/97	First release of document in this form.
1.1	1/04	Minor updates
1.2	11/05	Minor updates
1.3	10/06	Updated option interface description, other minor updates, corrections, formatting.
1.4	3/07	Updated driver and interface details and typographical uniformity.

Copyright (C) 1997,2007, Crescent Bay Software Corporation. No unauthorized use or duplication is permitted. All rights reserved.

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

Crescent Bay Software Corporation, 10950 Washington Blvd. #230 Culver City, California 90232 USA. Fax: (310)-836-7313 Phone: (310)-835-5183.

Email: [info4@crescentbaysoftware.com](mailto:info4@crescentbaysoftware.com). Web: <http://www.crescentbaysoftware.com>.

# Table of Contents

<b>Preface</b>	<b>i</b>
<b>1. Introduction</b>	<b>1</b>
Overview .....	1
Optimizations .....	1
Files.....	2
User interaction .....	2
How to use this guide.....	2
Optional features.....	3
<b>2. Invoking VAST-C/Parallel</b>	<b>4</b>
pcc Driver .....	4
Usage .....	4
File Extensions .....	5
pcc Options .....	5
Example.....	6
Assertion levels.....	6
Notes on assertion levels.....	6
VAST-C/Parallel Options .....	7
<b>3. User pragmas</b>	<b>9</b>
Overview .....	9
Pragma format .....	9
VAST pragma summary.....	10
Transformation pragmas.....	11
noconcur / concur.....	11
skip / noskip .....	11
cncall .....	11
noassoc / assoc.....	11

select(concurrent).....	11
Data dependency pragmas .....	12
nosync / sync .....	12
permutation .....	12
relation .....	12
disjoint .....	12
<b>4. VAST listing</b> .....	<b>13</b>
Overview .....	13
Diagnostic messages .....	13
Optimization inhibition messages .....	13
Informative messages .....	14
Error messages .....	14
Summaries.....	14
Listing control switches.....	14
<b>5. Optimization overview</b> .....	<b>16</b>
Pointers and array references .....	16
Understanding loop variables .....	17
Loop types .....	17
for loops.....	17
while loops.....	18
do loops.....	18
Allowed statements.....	18
Iteration count.....	19
Structures.....	20
Assignment operators.....	20
Translation diagnostics on loops.....	21
<b>6. Loop Optimizations</b> .....	<b>22</b>
Overview .....	22
Loop fusion .....	22
Loop re-rolling.....	23
Loop unrolling.....	23
while to for .....	25
Common subexpression enhancement.....	26
Scalar division removal.....	26

Loop interchange for cache optimization .....	26
Loop nest collapse .....	27
Multiple levels of indirection.....	28
Outer loop unrolling.....	28
<b>7. Parallelization</b> .....	<b>29</b>
Overview .....	29
Selection criteria for parallelization.....	29
Controlling concurrency.....	30
concur/noconcur pragmas.....	30
threshold pragma .....	30
skip pragma .....	30
select(concurrent) pragma.....	30
Creating and executing parallel regions .....	30
Private versus shared.....	31
Threshold test.....	32
Suppressing the threshold test.....	33
Changing the threshold value.....	33
Scheduling Loop Iterations .....	34
Dynamic Scheduling.....	34
Static Scheduling .....	34
How VAST Schedules Loops.....	34
Conditional parallelization .....	34
Inner loops.....	35
Reductions.....	35
Parallelizing loops with external calls.....	35
Parallel regions .....	36
Parallel cases .....	36
Extending parallel regions .....	37
Parallel outer unroll .....	37
<b>8. Data dependency analysis</b> .....	<b>38</b>
Overview .....	38
Data dependency examples .....	38
Function arguments .....	39
Pointers and data dependency .....	40

Potential dependency run-time testing .....	41
Ambiguous subscript resolution .....	42
Data dependency pragmas .....	42
nosync pragma .....	42
disjoint pragma.....	43
relation - specifying relations between variables.....	43
permutation -- declaring safe indirect addressing .....	44
Data dependency conflict messages .....	44
<b>9. Decision processes</b>	<b>47</b>
Allowable conditional constructs.....	47
if statements: .....	47
Conditional and unconditional forward transfers: .....	47
Conditional operator: .....	48
if...else statements: .....	48
Two types of conditions .....	49
Removal of invariant IFs .....	50
Loop-index-dependent conditions.....	50
Messages about conditionals .....	51
<b>10. Idiom Recognition</b>	<b>53</b>
Nested loop idiom recognition.....	53
Matrix multiply .....	53
Vector-matrix multiply.....	54
Rank one update .....	54
<b>11. Functions</b>	<b>56</b>
Standard C library functions .....	56
Split-out function references.....	56
Inline expansion .....	57
Message about external references .....	57
<b>12. Inline expansion</b>	<b>58</b>
Introduction .....	58
Inline expansion pragmas .....	59
autoexpand pragma .....	59
Explicit mode .....	59
expand pragma .....	59

nexpand pragma .....	60
search pragma .....	60
Where to get the code .....	60
Same file .....	60
Explicitly named file.....	61
Implicitly named file.....	61
Possible problems.....	61
Separate compilation .....	61
Code size .....	61
Debugging.....	61
Compilation rate .....	61
Analysis inhibitors .....	62
Expansion inhibitors.....	62
Inhibitors specific to automatic expansion mode.....	62
Inline expansion user messages .....	62
Inline expansion summary .....	62
User messages.....	63
<b>13. Further Information</b> .....	<b>64</b>
Crescent Bay Software and VAST.....	64
Questions or Comments.....	64
<b>Index</b> .....	<b>65</b>

# 1. Introduction

---

## Overview

This document describes the features and options of VAST-C/Parallel, a software tool that optimizes C programs for execution on SMP parallel systems. VAST-C/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.

VAST-C/Parallel allows you to automatically adapt existing codes to use the multiple processors of an SMP system. The automatic parallelization of C 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.

VAST-C/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-C/Parallel also provides a suite of high-level scalar optimizations that improve the performance of code in each thread.

---

## Optimizations

VAST-C/Parallel's optimizations include:

- Full analysis of large loops nests, with parallelization at the outermost levels.
- Creation of large parallel regions to reduce parallel overhead..
- Restructuring of loop nests to allow parallel execution.
- Parallelization of loops containing reduction operations (such as global sum functions).

- 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-C/Parallel processes a C program, it creates two files. One (optional) is a diagnostic listing comments added to tell which loops were not optimized and why. The other is an enhanced version of the input C program containing restructured loops and calls to thread library routines. This file is ready for compilation by a C compiler.

Normally, VAST-C/Parallel is used through the `pcc` driver, which combines invocation of VAST-C with running the compiler and linking with the threads library. With the driver, you just compile as usual and get a parallel version of the code.

---

## User interaction

VAST-C/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-C/Parallel performs may not significantly decrease the input program's execution time.

If the execution time has not decreased, enable VAST-C/Parallel'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 pragmas, or minor code modifications. In some cases, you may be rewarded with dramatically improved performance for a relatively small effort.

At a minimum, you should be familiar with the assertion levels (`-vassertion_level`), which are discussed in the next chapter. These can be very important to performance on some C codes.

---

## How to use this guide

Section 2 describes how to invoke VAST-C/Parallel. If you want to use VAST-C/Parallel 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-C/Parallel -- VAST-C/Parallel's listing and messages, and ways to guide VAST-C/Parallel's action (user directives or pragmas).

Concepts and rules of VAST-C/Parallel's optimization techniques are discussed in the remaining sections. Examples in these sections illustrate optimizable and unoptimizable loops.

---

## Optional features

Most VAST-C/Parallel features are turned on by default, but some are not. To enable automatic expansion of calls to small functions, use `-Vinline` on the invocation. (See the chapter on Inlining for further details.)

To specify an assertion level, use the `-Vassertion_level` switch. For almost all codes you can safely specify the switch `-Vassertion_level13`; fairly often, the code will get significantly more parallelization (and more performance).

## 2. Invoking VAST-C/Parallel

---

### pcc Driver

Normally, you will invoke VAST-C/Parallel with the pcc driver. It is used 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,

```
pcc myprog.c
```

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

### Usage

```
pcc [<pcc_option>]... [<cc_option>]...  
    [-Wv,<v_option>[,<v_option>]...] [files]
```

where:

- <pcc\_option> represents any pcc driver option (see “pcc Options” below).
- <cc\_option> represents any C compiler option. All normal compiler options can be used.
- <v\_option> represents any VAST-C option.

pcc options and input files may appear in any order.

## File Extensions

1. filename with a `.c` suffix: C source file.
2. filename with a `.i` suffix: preprocessed C source file.
3. filename with a `.s` suffix: assembler source file.
4. filename with a `.o` suffix: object file.
5. filename with a `.a` suffix: archive file.

## pcc Options

- c** suppress the link editing phase of the compilation and do not remove any object files produced.
- dryrun** display but do not execute pcc internal commands. pcc will still check for the existence of essential executable files.
- keep** Keep intermediate VAST files. These files are preprocessed C source files. The intermediate files produced are as follows. For an input file `file1.c` the intermediate file will be named `file1.int.c`.
- 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 pcc internal commands as execution progresses.
- vo** Execute VAST pass only. Certain compiler options included on the command line may be ignored. Transformed source file is put in `<input_filename>.int.c`.
- vn** Execute C compiler only. VAST-C options included on the command line are ignored. Note that the `-vo` and `-vn` options are mutually exclusive.
- w** Suppress warning messages.
- Wv** Hand off arguments to pass to VAST-C. VAST-C options must be separated by commas and may not contain embedded blank space characters.  
  
Examples:  
`-Wv, -Vnosync`  
`-Wv, -Vnoassoc`
- Yvc, <path>** Substitute for `cc` an alternate executable whose pathname is specified by `<path>`.

`-Yvv, <path>` Substitute for `vastce` an alternate executable whose pathname is specified by `<path>`.

## Example

```
pcc -o par.exe -Wv, -Vthreads4, -Vassertion_level3 file1.c
```

Executes `vastcp` with the number of processors (technically the number of threads) specified to be 4, and assertion level 3. Compiles the intermediate file `file.int.c` using `cc` and invokes the linker to produce the executable output file `par.exe`.

---

## Assertion levels

The table below shows data dependency assertion levels that affect the optimization and parallelization of the input program; these are controlled with `-Vassertion_level`. Use of this option is heavily encouraged! Use the highest level that you know will work for your program – you will be rewarded with higher performance. In real programs, it is very rare for objects with different names to overlap; a little assistance from you in the form of a `-Vassertion_level` option can go a long way in improving the optimization of your program.

### Level Description

- 0 Assume Nothing. (Default)**
- 1 Arguments to functions do not overlap.**
- 2 Pointers do not overlap with non-pointers.**
- 3 Pointers with different names do not overlap.**
- 4 All overlap is explicit.**

Higher assertion levels include all lower levels. As an example, `-Vassertion_level3` causes VAST-C to assume that pointers with different names do not overlap, pointers do not overlap with non-pointers, and arguments to functions do not overlap.

### Notes on assertion levels

*Level 0:* Assume nothing. This is the default. At this level VAST-C must assume that any pointer may overlap with any other variable, and that arguments to a function may also overlap. This assumption hinders the ability of VAST-C to determine whether or not a data dependency exists in the presence of pointer variables and function arguments. If a data dependency may exist, VAST-C cannot parallelize the loop.

```
func ( a, b )
int *a, *b;
{
    for ( i = 0; i < n; i++ )
        *a++ = *b++;
```

```
}
```

Above, VAST-C must assume that pointers *a* and *b* may overlap in memory.

**Level 1.** -- Assume that any pointer may overlap with another pointer or nonpointer, except that arguments to a function will not overlap with each other. VAST-C will assume, in the above example, that *a* and *b* do not overlap in memory.

**Level 2.** -- In addition to level 0 and 1 assumptions, also assume that pointers will not overlap with non-pointers.

```
for ( i = 0; i < n; i++ )
    *p++ = a[i];
```

Above, VAST-C can assume that pointer *p* does not overlap with array *a*.

**Level 3.** -- In addition to level 0, 1, and 2 assumptions, assume that pointers with different names do not overlap.

```
for ( i = 0; i < n; i++ )
    *p++ = *q++;
```

Above, VAST-C may assume that pointer *p* does not overlap with pointer *q*.

**Level 4.** -- In addition to level 0, 1, 2 and 3 assumptions, assume that no overlap exists unless it is explicit. For example, *\*p* may not overlap with *\*(p+n)*.

---

## VAST-C/Parallel Options

All the options are listed below, with a brief discussion. The remainder of this document discusses their use in more detail.

### **-Vassertion\_level***n*

General disambiguation assertion. *n* indicates strength of assertion:

- 1 = arguments don't overlap,
- 2 = pointers don't overlap with non-pointers,
- 3 = pointers with different names don't overlap,
- 4 = no overlap exists unless explicit.

Higher levels include lower levels.

### **-Vinline**{*aaaa,bbbb*}

Request automatic inlining of small "leaf" procedures. Alternatively, specify inlining of specific functions.

### **-Vnoargumentoverlap**

Assert that arguments to functions don't overlap. Equivalent to **-Vassertion\_level1**.

**-Vnoassoc**

Demand that optimizations that could change the last few bits of the result (relative to the scalar original results) not be done. (No associative transformations). Can seriously degrade performance in some cases.

**-Vnosync**

Assume potential dependencies are not real dependencies. May improve performance by allowing more loops to vectorize.

**-Vnopointeroverlap**

Assert that pointers in vectorizable loops don't overlap. It is very helpful to make this assertion, if you can. This is equivalent to **-Vassertion\_level3**.

**-Vskip**

Skip all transformations. There are some non-vector optimizations that VAST does -- this should turn off everything. You can enable transformations for specific loops with the **#pragma vd\_ noskip** pragma.

**-Vthreads*n***

Specify how many threads (processors) to generate code for. (Default 2.)

**-Vparallel\_threshold*nnnn***

Specify a minimum amount of work necessary for a code region to be executed in parallel. The units are arbitrary, though they could be thought of as roughly equivalent to machine clocks. Default 1000.

**-Vconcurrent\_call*aaaaa,bbbb***

Specify functions that can be called from parallel loops.

**-Vnoinline*aaaa,bbbb***

Specify specific functions not to inline.

**-Vinner**

Allow parallelization of innermost loops.

**-Vparcase**

Allow parallelization of parallel cases.

**-Vautoinline\_maxlines*nn***

Specify an upper threshold for size of function to auto-inline, in number of source lines.

**-Vautoinline\_nest*n***

Specify a maximum function call depth to inline. **-Vinline** is equivalent to **-Vautoinline\_nest1**.

# 3. User pragmas

---

## Overview

You may sometimes have information about the structure of a program and about its data that is unavailable to VAST-C through inspection of individual program units. For this reason, a way for you to guide VAST-C is supplied via user pragmas (also called user directives). User pragmas are passed via the `#pragma` statement provided in ANSI C.

(For user-directed parallelism pragmas, where you explicitly control what will be done in parallel and what will be synchronized, see Section 11.)

---

## Pragma format

VAST-C pragmas have the format:

```
#pragma vd_ <directive>
#pragma vd_l <directive>
#pragma vd_r <directive>
#pragma vd_f <directive>
```

The `#pragma vd_` flags this as a pragma to be used by VAST\_C. Following is an optional scope parameter. **f** stands for "file" (meaning the directive applies until the end of the input file), **r** stands for "routine" (directive applies until the end of the current function), and **l** for "loop" (directive applies to the next loop encountered). A blank is equivalent to **l**. Some directives ignore the scope parameter.

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

Examples of pragmas:

```
#pragma vd_ nosync (Ignore potential data dependencies in the following loop.)
```

```
#pragma vd_r noconcur (Turn off parallelization for the rest of this function.)
```

```
#pragma vd_f noassoc (Do no associative transformations for the rest of the file.)
```

---

## VAST pragma summary

The full set of pragmas is summarized in the table below. The "scope" entry is either **i** for "immediate," meaning that the pragma applies immediately; **l**, meaning that it applies to the next loop; **r**, meaning that it applies to the whole routine (function); 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 pragmas follows the table. In addition, the more important pragmas are discussed in detail at the appropriate points in the sections on optimization.

<i>VAST-C/Parallel pragmas</i>			
Pragma	Function	Default	Scope
<b>skip/</b> <b>noskip</b>	Disable/reenable transformations	noskip	lrf
<b>nosync/</b> <b>sync</b>	Do/don't ignore potential overlap of array sections.	sync	lrf
<b>noconcur/</b> <b>concur</b>	Disable/enable parallelization.	concur	lrf
<b>cncall</b>	Allow concurrent calls in loop.	n/a	lrf
<b>noassoc/</b> <b>assoc</b>	Don't/do perform associative transformations.	assoc	lrf
<b>permutation</b>	Pass list of integer arrays that have no repeated values.	n/a	r
<b>relation</b>	Specify relationship between two simple variables.	n/a	r
<b>count</b>	Supply iteration count for loop.	n/a	i
<b>iterations</b>	Supply iteration count for classes of loops.	n/a	r
<b>nounroll/</b> <b>unroll</b>	Unroll loop.	unroll	lrf

<b>autoexpand/</b> <b>noautoexpand</b>	Automatically expand small routines inline.	noauto	r
<b>expand</b>	Expand particular routine(s).	n/a	i
<b>nexpand</b>	Nested expansion of particular routine(s).	n/a	i
<b>search</b>	Supply file/path location(s) for sources for inlined routines.	n/a	i

---

## Transformation pragmas

These pragmas are used to change the way VAST-C 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 / noskip**

`skip` causes VAST-C to avoid transformation for the directed loop or routine. This is the pragma to use if you want VAST-C/Parallel to leave a loop unoptimized (no parallel or superscalar optimizations). `noconcur` is a subset of `skip`.

### **cncall**

`cncall` asserts that any functions 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-C transforms certain constructs into optimized 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` pragma disables all associative transformations, including generating reductions (such as sum or dot product of arrays), and operation reordering when minimizing dependent regions.

The `-vnoassoc` command line option is equivalent to `noassoc` with file scope.

### **select (concurrent)**

Use this pragma to select a particular loop in a loop nest for parallelization.

---

## Data dependency pragmas

These pragmas are used to help VAST-C 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 pragmas to inform VAST-C. (Data dependency pragmas are discussed in more detail in a later chapter.)

### **`nosync / sync`**

When elements of an array are modified within a loop, VAST-C 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` pragma 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` pragma is used only to toggle back to the default state.

### **`permutation`**

The `permutation` pragma 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` pragma advises VAST-C 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-C in resolving otherwise ambiguous array relationships.

### **`disjoint`**

The `disjoint` pragma is used to tell VAST-C that pointers do not overlap. This can be very useful for codes that make heavy use of pointers. (Even better is the C99 keyword `restrict`.)

# 4. VAST listing

---

## Overview

Optimizing C loops for parallel execution is a complex task, and to do the best possible job, VAST-C/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 pragmas 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 three parts: 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.

---

## 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 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. (Consider using a -L assertion level or a NOSYNC pragma if you know the loop is actually safe to parallelize.)

## 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 C has been encountered. No translation is done for this program unit.

*Internal Error.* An internal problem with VAST-C 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-C command line or on the SWITCH pragma (see section 3). A later section a list of some of VAST-C's diagnostics, with further explanation of the messages, and tips on avoiding certain problems.

---

## 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 pragma (e.g., #pragma vd\_ switch, -q h).

Listing control switches

Switch	Description	Default
<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>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 `-q t w e`. If you wanted to get potential errors listed, you would use `-p r`.

#### Notes on the listing switches:

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

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

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

# 5. Optimization overview

This section presents some general features of VAST-C's optimization.

---

## Pointers and array references

A pointer is a variable that contains the address of another variable. The use of pointers can lead to more compact and efficient coding, but also can create programs which are more difficult to analyze. The use of pointers makes it especially difficult to determine whether or not data dependencies exist. For the most part, pointers are equivalent to array indexing and, viewed as such, the existence of overlap can be determined.

The availability of pointers in C gives the programmer a great amount of flexibility. Arrays of data can be referenced in many different ways. VAST-C recognizes and understands these different uses. Below are three equivalent examples, all of which VAST-C optimizes.

```
float *a, *b, x;

/* loop 1 */
for ( i = 0; i < n; i++ )
    *(a+i) = *(b+i) + x;

/* loop 2 */
for ( i = 0; i < n; i++ )
    a[i] = b[i] + x;

/* loop 3 */
i = 0;
```

```
while ( i++ < n )
    *a++ = *b++ + x;
```

Figure 6.1 Equivalent loops.

Pointers can be parallelized in the above examples (in the same manner as arrays) if we can determine that a and b do not overlap. This is done either through an invocation option to VAST-C, a user `pragma`, program analysis by VAST-C, or a runtime test generated by VAST-C. A full discussion on pointers and their effect on data dependency analysis can be found in the "Data Dependency Analysis" section of this guide.

---

## Understanding loop variables

Understanding loop optimization is made possible by understanding the optimizable and non-optimizable uses of the variables in a loop. Every variable in an optimizable loop can be categorized as one of three things: scalar, index, or vector. A scalar is a single value which does not change through all the iterations of the loop. An index is an integer quantity which is incremented by a constant amount each pass through the loop. A vector is a range of memory locations, with a constant skip or stride between consecutive elements.

Here is an example of each of these:

Loop:

```
for ( i=0; i < n; i++ )
{
    *(a+j) = x + b[i];
    j += 2;
}
```

Classification:

i,j: index variables  
a,b: vector  
x : scalar

Later sections in this document examine each of these kinds of variables in detail, and explore how their different uses can make a loop optimizable or not.

---

## Loop types

for, while, and do loops are considered for optimization.

### for loops

VAST-C analyzes entire nests of for loops (including converted if loops) for possible optimization.

```

for ( i = 0; i < n; i ++ )
    for ( j = 0; j < m; j++ )
        a[i][j] = b[i][j] * c[i][j];

```

## while loops

VAST-C analyzes while loops. while loops must have a fixed iteration count to qualify for optimization.

```

while ( i < n )
{
    *a++ = *b++ ;
    i++;
}

```

(The iteration count is n-i.)

## do loops

Similar to while loops, VAST-C analyzes do loops. Again the loop must have a fixed iteration count to qualify for optimization.

```

do
{
    *a++ = *b++;
    i++;
} while ( i < n );

```

(The iteration count is max(1,n-i).)

---

## Allowed statements

Statements which may appear in an optimizable loop are listed below.

Expressions

```

a[i] = b[i] + c[i];

```

Compound statements

```

{ ... }

```

Selection statements

```

if, else, switch, case, default

```

Iteration

```

while, do...while, for

```

Jump

```

goto, continue, break

```

---

## Iteration count

For many of its optimizations, it helps VAST-C to have some idea of the iteration counts of loops in the program. Very short loops are often better off translated in a different way than loops with many iterations. In choosing concurrent and scalar loops, knowing approximate values for the iteration counts is extremely useful.

If the iteration count of a loop is constant, VAST-C understands it already. In cases where the iteration count is a variable, VAST-C looks at the declared dimensions of arrays used in the loop in order to determine what the maximum iteration count could be.

If the iteration count is variable and cannot be determined from the information in the routine until execution time, but you know the approximate number of iterations, you can use the count or iterations pragma to provide this information to VAST-C. The count pragma has the format:

```
#pragma vd_ count ( n1 )
```

The iterations pragma has the format:

```
#pragma vd_ iterations(var1=n1[,var2=n2,...])
```

where

n1, n2...

The assumed iteration count values. These do not have to be exact as they are only used as guidelines by VAST-C.

var1, var2...

The indices of a loop with the given assumed iteration count values. The count pragma can be used at the file (f), routine (r), or at the local (l) level. The iterations pragma may only be used at the routine level. A f count(0) or a noiterations pragma returns VAST-C to its normal iteration count processing.

```
optim6 ( a, b, n )
float *a, *b;
int n;
{
  int i;
#pragma vd_ count (3)
  for (i=0; i<n; i++)    (Not optimized, iteration count is too small.)
    *(a+i) = *(b+i);
}
```

Below is an example with a small iteration count on the inner (fast) dimension.

```
optim7 ( a, b, c, l, m )
float a[100][3], b[100][3], c[100][3];
int l, m;
{
```

```

int i, j;

#pragma vd_ iterations ( j=3, i=100 )
for ( i=0; i < m; i++ )
    for ( j=0; j < l; j++ )
        a[i][j] = b[i][j] + c[i][j];

```

---

## Structures

VAST-C can optimize loops that contain structure references. Optimization can occur both across structures and within structures, at the same time. A loop across structures can be parallelized.

```

typedef struct tag { int xaxis[1000],
                    yaxis[1000]; } Cartesian;

Cartesian box1, *pbox1;
int slope[1000];
pbox1 = &box1;
for ( i = 0; i < 1000; i++ )
    slope[i] = ( pbox1->yaxis[i+1] - box1.yaxis[i] ) /
               ( pbox1->xaxis[i+1] - box1.xaxis[i] );

```

The above example shows the use of elements within a structure, including the use of structure pointers. Below is an example of a loop using elements across structures.

```

struct grade { float math, english,
              history; } students[100];

math_average = 0;
for ( i = 0; i < 100; i++ )
    math_average += students[i].math;
math_average /= 100.0;

```

---

## Assignment operators

VAST-C fully understands the use of all assignment operators, multiple assignments, and nested assignments and will optimize loops which contain such constructs.

```

for ( i =0; i < 100; i++ {
    a[i] = b[i] = (float) 100;
    c[i] = ( b[i] = b[i] + 3 ) + a[i];
    d[i] *= c[i];
}

```

---

## Translation diagnostics on loops

Some of the more common translation diagnostics that may be issued for loops are shown in this section.

### *unable to get constant length from loop*

```
while ( *a > 0 )
    *b++ = sqrt(*a++) ;
```

VAST-C issues this diagnostic when it cannot convert a for, while, or do loop to a form that has a fixed iteration count known before the loop is executed.

### *user function references not allowed in iteration count*

```
for ( i=0; i < nlen(j,k); i++ )
    a[i] = 0.
```

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

### *iteration count too short*

```
for ( i=0; i < 2; i++ )
    a[i] = b[i];
```

When loops have an explicit iteration count which is less than a set cut-off amount they are not parallelized as they will probably run faster in scalar mode.

### *array dimensions indicate iteration count is too short*

```
float a[2], b[2];
for ( i=0; i < n; i++ )
    a[i] = b[i];
```

The declared size of an array dimension may limit the maximum possible iteration count to less than the cut-off (see above).

# 6. Loop Optimizations

---

## Overview

This section describes some of the features that VAST-C provides for loop optimization. As most of the time is spent in loops, optimizing them can be very important to final performance.

---

## Loop fusion

VAST-C combines consecutive loops that have no statements between them, the same iteration count, and that will give the same answers when merged. This optimization helps expose common subexpressions and reduces overhead. There is a default maximum of five loops and 50 total lines of fused code, as it would be unwise to create loops that were too large; the compiler might run out of registers. You can use the `-G` option to raise the amount of fusion.

Below is an example of two loops where common subexpression analysis will eliminate repeated operations. `c[i]` will only need to be loaded only and `c[i]*.5` will only need to be calculated once.

```
for ( i=0; i < 100; i++ )
    a[i] = b[i] + c[i]*.5;
for ( i=0; i < 100; i++ )
    e[i] = d[i] + c[i]*.5;
```

Below is the fused form of the loops.

```
for ( i=0; i < 100; i++ ) {
    a[i] = b[i] + c[i]*.5;
    e[i] = d[i] + c[i]*.5;
}
```

---

## Loop re-rolling

Optimizing compilers for some high-speed scalar computers shuffle the order of the instructions they produce in order to overlap operations as much as possible. A common technique used to aid such compilers is to "unroll" short loops to give the compiler more flexibility in reordering instructions within a single loop pass. For example, a simple dot product

```
for ( i=0; i < n; i++ )
    s += a[i];
```

might be unrolled like this:

```
for ( i=0; i < n-3; i +=4 )
    s += a[i] + a[i+1] + a[i+2] + a[i+3];
```

The second loop performs the same function as the first, but it does four iterations per pass instead of one. The unrolling technique is generally undesirable on vector machines, since it decreases vector length and increases indexing overhead. In some cases, VAST-C can detect unrolled loops and "re-roll" them into their original form.

A re-rollable loop must have an explicit constant non-unit increment specified on the for statement. It must contain no data dependencies. An unrolled summation or dot product must add each operand to the reduction scalar in index order. The number of assignments in an unrolled assignment must be the same as the increment on the for loop, and each assignment must do the next computation in index order. Below is an example of a loop that VAST-C can re-roll.

```
for ( i = 0; i < 997; i +=3 )  (Rerolled into one vector add.)
{
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}
```

Translation:

```
for ( i = 0; i < 1000; i++ )
{
    a[i] = b[i] + c[i];
}
```

---

## Loop unrolling

Loop unrolling is of benefit for loops whose scalar optimization is inhibited because they have too few operations per pass. Loop unrolling reduces the percentage of time spent in loop overhead, and provides more instructions for the compiler to overlap in each pass of the loop.

VAST-C can unroll automatically or a pragma may be used to force it to unroll. Automatic unrolling is done when VAST-C finds a case it believes will benefit from this treatment; you may request unrolling for other loops with the unroll pragma.

```
#pragma unroll
for ( i = 0; i < n; i++ )
    d[i+1] = a[i] + b[i]*c[i];
```

Translation:

```
j1s = n % 4;                                     (Unrolled 4 times.)

for ( i=0; i < j1s; i++ )                       (Cleanup loop)
    d[i+1] = a[i] + b[i]*c[i];

for ( i=j1s; i < n; i+=4 ) (If N<4, this loop not done.)
{
    d[i+1] = a[i] + b[i]*c[i];
    d[i+2] = a[i+1] + b[i+1]*c[i+1];
    d[i+3] = a[i+2] + b[i+2]*c[i+2];
    d[i+4] = a[i+3] + b[i+3]*c[i+3];
}
```

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.

```
for ( j=0; j<m; j++ ) {
    for(i=0; i<3; i++) *(a+i+j*100) = 0.0;
}
```

Translation:

```
for ( j=0; j < m; j++ )
{
    a[j*100] = 0.0;
    a[1+j*100] = 0.0;
    a[2+j*100] = 0.0;
}
```

To allow you control over the unrolling process, VAST-C provides the unroll pragma, as follows:

```
#pragma vd_ {l,r,f} unroll [(number_of_times)]
```

where `number_of_times` is the number of times to unroll the body of the loop (the number of copies of the statements in the loop that appear in the new, unrolled, loop.) If no argument is supplied, VAST-C calculates the unrolling depth based on default parameters. A value of zero as the `number_of_times`

field turns off unrolling. The `nounroll` pragma can also be used to turned off unrolling.

Explicit unrolling is applied at an early stage in VAST-C and thus has relatively few restrictions.

Loops are automatically unrolled whenever certain system-specific conditions are met. Automatic unrolling's goal is to create loops with an "ideal" size, i.e., that are large enough to get maximal instruction overlap. The goal size is system-specific, and can be based on number of statements, number of array references, number of operations, and/or other factors. Loops are automatically unrolled if:

They consist solely of assignment statements. No branches or external references are allowed.

There are not too many statements in the loop. The maximum number is half the unrolling factor.

The last value of the for index is not required after the loop is executed.

The number of times that a loop is unrolled depends on the number of statements in the loop. VAST-C tries to produce a loop that is unrolled a power of two times and contains a maximum of eight statements.

---

## while to for

while loops are converted to iterative for loops when a fixed iteration count can be extracted from the loop. This may allow further optimization.

```
while(i > 0) {
    *(a+i-1) = 0.0;
    i -= 1;
}
```

Translation:

```
ilx = i;
if( ilx > 0 )
{
    i2x = ilx > 0 ? ilx & 1 : 0;
    if( i2x == 1 )
        a[ilx-1-1] = 0.0;
    for( i=i2x + 1; i <= ilx; i += 2 )
    {
        a[ilx-1-i] = 0.0;
        a[ilx-2-i] = 0.0;
    }
}
```

---

## Common subexpression enhancement

Common expressions within a loop can be collected and computed only once, even when the expression must be permuted to expose the commonality.

As the answers may be slightly different due to the finite word size, this optimization is only done when associative transformations are allowed. VAST-C rewrites the expression with additional parentheses to make the common expression apparent to the compiler.

```
for ( i=0; i<n; i++)
{
    *(a+i) = *(b+i) * *(c+i) * *(d+i);
    *(e+i) = *(c+i) * *(d+i) * *(f+i);
}
```

Translation:

```
for ( i=0; i < n; i++ )
{
    a[i] = (c[i] * d[i]) * b[i];
    e[i] = (c[i] * d[i]) * f[i];
}
```

---

## Scalar division removal

VAST-C converts division by scalar to multiplication by the reciprocal, with the reciprocal calculated outside the loop. This transformation will only be done when associative transformations are allowed by the user, as the answers may be slightly different.

This transformation is only done for floating point division.

```
for(i=0; i<n; i++) *(a+i) = x + *(b+i)/s;
```

Translation:

```
float r1s;
r1s = 1. / s;
for ( i=0; i<n; i++ )
    a[i] = x + b[i] * r1s;
```

---

## Loop interchange for cache optimization

On systems with cache memory, consecutive accesses to array elements are more efficient than accesses to array elements with gaps between them. In the example below, the loops are interchanged to move the loop that has unit stride and a larger iteration count to the inside.

```

for(i=0; i<100; i++) {
    for(j=0; j<10; j++) a[j][i] = b[j][i];
}

```

Translation:

```

for( j=0; j < 10; j++ )    (now the outer loop)
{
    for( i=0; i < 100; i++ ) (now the inner loop)
        a[j][i] = b[j][i];
}

```

---

## Loop nest collapse

Most computers process long vectors more efficiently than short vectors. A common technique for increasing average vector length relies on the conventional storage pattern of C arrays. Using this technique, the last N dimensions of an array can be treated as a singly-dimensioned array whose length is the product of the sizes of the N dimensions. Thus an array declared `a[5][7]` can be treated as if it had been declared `a[1][5*7]`.

Under certain restrictive conditions, VAST-C uses this technique to automatically collapse loop nests into a single loop whose iteration count is the product of the iteration counts of the uncollapsed loops. This transformation can be disabled by the `-d p` option switch.

Collapse criteria:

- The loops must be tightly nested, and there must be one loop index per optimized dimension.
- The loop bounds must be identical to the array bounds.
- There must be no recursion in the loops.
- There must be no use of vector indexes outside of array subscripts.
- All vector array references in the loop must conform, that is, the last N subscripts of each reference must be identical to the last N subscripts of all the other vector array references, where N is the nesting depth. Each of the N subscripts must be indexed by one and only one of the loop indexes, with a stride of one. The last N-1 dimensions of the declarations of these arrays must also conform.

```

float a[12][10], b[12][10];
for ( i = 0; i < 12; i++ )
    for ( j = 0; j < 10; j++ )
        a[i][j] = b[i][j] * b[i][j];

```

Here is the collapsed form of the loop:

```
float a[12][10], b[12][10];
for ( j = 0; j < 120; j++ )
    a[0][j] = b[0][j] * b[0][j];
```

---

## Multiple levels of indirection

When VAST-C encounters pointers of multiple levels of indirection, e.g. `**p`, VAST-C must analyze these constructs in a different manner than a multiply dimensioned array. Collapsing cannot work with a pointer such as `**p`, since the first level of indirection merely points at a strip of memory elsewhere. Unlike the previous example of multiply dimensioned arrays, VAST-C cannot take advantage of memory storage since there may not be contiguous memory across levels of indirection.

```
func( a, b )
int **a, **b;
...
for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ )
        (*(a+i)+j) *= (*(b+i)+j);
```

---

## 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 is especially useful when 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.

```
for(j=0; j<100; j++) {
    for(i=0; i<m; i++) a[j][i] = b[j][i] + c[i];
}
```

Translation:

```
for( j=1; j <= 100; j += 4 ) {
    for( i=0; i < m; i++ ){
        a[j-1][i] = b[j-1][i] + c[i];
        a[j][i] = b[j][i] + c[i];
        a[1+j][i] = b[1+j][i] + c[i];
        a[2+j][i] = b[2+j][i] + c[i];
    }
}
```

# 7. Parallelization

---

## Overview

*Parallelization* is the automatic distribution of loop iterations to multiple processors (or tasks). (This is also known as *concurrency analysis*.) VAST-C/Parallel parallelizes loops by using a parallel threads library. Parallel threads are assigned different pieces of the calculation to complete.

Parallelization is the automatic distribution of loop iterations or parallel cases to multiple processors (or tasks). (This is also known as *concurrentization*.) VAST-C parallelizes loops by “outlining” them to new separate functions containing control code to distribute specific iterations to separate processors, and then invoking this new parallel function via a threads library interface called from the original function.

Parallelization can be disabled by the `noconcur` or `skip` pragmas or the `-Vskip` command line option. Parallelization can be re-enabled by the `concur` or `noskip` pragmas.

---

## Selection criteria for parallelization

To pick the best loop in a nest for parallelization, VAST-C uses these criteria:

- Loop iteration count
- Presence of data dependence
- Amount of work within the loop
- Nesting level of the loop

In general, greater iteration count, more work, outer nesting level, and absence of dependence are favored.

---

## Controlling concurrency

There is a significant amount of overhead involved in executing a loop concurrently, and if there is not a reasonable amount of work to be done, the loop is often better left in vector or scalar mode.

### **concur/noconcur pragmas**

You can enable or disable parallelization via the `concur` and `noconcur` pragmas. By default, parallelization is off. Parallelization may be enabled or disabled independently of vectorization. Parallelization may also be enabled or disabled by the `-e c` or `-d c` option switches. NOTE: When faced with several loops in a nest, use “`select(concurrent)`” (see below) to choose the loop you want parallelized, not the `concur` or `noconcur` pragmas.

### **threshold pragma**

The threshold pragma is used to pass an estimate for the amount of work that will be required in a loop before concurrent execution will pay off. The default for this value is currently around 800 clock periods. You may desire to adjust this value based on the load factor of the system(s) you run on.

### **skip pragma**

The skip pragma causes VAST-C to do no optimization of the indicated loop -- both superscalar optimization and concurrency are disabled.

### **select ( concurrent ) pragma**

You can use the `select(concurrent)` pragma to select a different loop in the nest for parallelization, when there are several candidates.

---

## Creating and executing parallel regions

When VAST-C/Parallel 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-C/Parallel creates a new parallel routine and inserts the code for the parallel region in it. A parallel call to the new routine (using the VAST 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 versus 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) must be *shared* with all other threads.

When creating a parallel thread routine for a parallel region, VAST passes as arguments 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 to access shared items, and private items will be allocated on each thread's 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 is 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. The variable *k* is eliminated altogether – it is replaced by *I+kk*.

```
for ( j=0; j<1000; j++ )
  for ( i=0; i<1000; i++ )
  {
    k = i +kk;
    t = sqrt(a[j][i]);
    b[j][k] = t + 1.0/t;
  }
```

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.

```
float a[99][99];
for ( j=0; j<99; j++ ) (parallel loop)
  for ( i=0; i<99; i++ )
    a[j][i] = b[j][i] * b[j][i] +x;
```

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.

```
float b[99][99], c[99][99], d[99][99];
void f (int n, int m)
{
```

```

float e[99];
for ( j=0; j<m; j++ ) (parallel loop)
{
    for ( i=0; i<n; i++ )
        e[i] = b[j][i] + c[j][i];    "e" needs to be private
    c[j][1] = 0.;
    c[j][n] = 1.;
    for ( i=0; i<n; i++ )
        c[j][i] = e[i]*b[j][i] - d[j][i]/e[i];
    }
}

```

---

## 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-C/Parallel makes a decision whether or not to parallelize the loop. If the loop nest iteration counts cannot be determined at compile time, VAST-C/Parallel 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-C/Parallel 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.

```

for ( j=0; j<m; j++ )
    for ( i=0; i<n; i++ )

```

```
a[j][i] = 0.0
```

Translation:

```
if ( m * n > 1000 )
{
    (call parallel version)
}
else
{
    for ( j= 0; j < m; j++ )
    {
        for ( i=0; i < n; i++ )
            a[j][i] = 0.0;
    }
}
```

For double loops, the product of the inner and outer loop iteration counts is compared to a threshold value. The threshold value is inversely proportional to the number of statements and operations in the loop. The default maximum value of the threshold is 1000 clock periods.

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.

## 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)** pragma suppresses the threshold test for the selected loop.

## 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** pragma. You can supply a different maximum threshold value for the whole file via the **-Vparallel\_threshold** parameter on the command line.

---

## Scheduling Loop Iterations

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

---

## Conditional parallelization

If a loop is suitable for parallelization except that it is potentially dependent, VAST-C/Parallel may generate an `if` 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` pragma 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` pragma or `-vinner` command line option 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` pragma 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-C/Parallel 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. (Note that `noassoc` suppresses parallelization of reductions.)

```
for ( j=0; j<m; j++ )
  for ( i=0; i<n; i++ )
  {
    s += a[j][i]      Reduction
  }
```

---

## Parallelizing loops with external calls

VAST-C may parallelize loops containing user function references if you insert a `cncall` pragma or use the `-vconcurrent_call` 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 invocation parameter specifies names of specific routines that may be called in parallel wherever they are referenced.

The `cncall` pragma in the example below will allow the iterations of the loop to be done in parallel -- the user is saying that function `crunch` does not have any bad side effects. Nothing in `crunch` should depend on the order the calls to `crunch` are made.

```
#pragma vd_ cncall
for ( i=0; i<n; i++ )
  b[i] = crunch ( a[i], &x );  Concurrent Call
```

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

VAST-C assumes that all variables not explicitly defined in the loop are shared; in the example above, `x` will not be specified as local, which may cause incorrect results if `x` is modified inside `crunch`.

---

## Parallel regions

In order to get the lowest overhead, VAST-C will combine parallel regions and/or move them outside of surrounding loops. In this way the overhead of starting up parallel activities is reduced. The entire parallel region is split out into a new parallel function. In the example below, the parallel region will be moved to start outside of the outer `j` loop -- the parallel processors will do the `j` loop redundantly. This is more efficient than restarting all processors for each iteration of `j`. In the generated code, the `j` loop will be included in the created parallel function.

```
#pragma vd_r inner
...
for ( j=0; j<m; j++ )
{
  for ( i=0; i<n; i++ )
    a[j][i] = b[j][i]+a[j-1][i]*0.5;
  for ( i=0; i<n2; i++ )
    c[j][i] = (a[j][i]+c[j-1][i])*0.5;
}
```

---

## Parallel cases

When parallelism cannot be found within a loop nest, VAST-C/Parallel 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-C/Parallel will split up loops into sections and execute each section as a parallel case. In the example below, the first loop nest is completely independent of the second loop. VAST-C/Parallel can create code so that one thread will execute the first nest, will a second thread will concurrently execute the second nest.

```
for ( j=0; j<m; j++ )
  for ( i=0; i<n; i++ )
    a[j][i] = (a[j][i-1]+a[j-1][i])*0.5;
nk = n - k;
ml = m + l;
for ( j=0; j<nk; j++ )
  for ( i=0; i<ml; i++ )
```

```
c[j][i] = (c[j][i-1]+c[j-1][i])*0.5;
```

---

## 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-C/Parallel 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.

VAST-C/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. 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.

---

## 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
```

# 8. Data dependency analysis

---

## Overview

VAST-C/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-C/Parallel detects the problem, reports it to the user, and leaves the loop in its original form.

VAST-C/Parallel 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 pragmas that VAST-C/Parallel provides to help you do this.

---

## Data dependency examples

In the loop shown below, the reference to `a[i-2]` at the top of the loop conflicts with the store into `a[i]` at the bottom. VAST-C prints a message to this effect and does not optimize the loop.

```
for ( i=3; i <= 100; i++ )    (Not optimized)
{
    t = a[i-1] + a[i-2];
```

```

    b[i] = t + 3.0 + a[i];
    a[i] = sqrt(b[i]) - 5.0;
}

```

VAST-C uses information from other array dimensions as part of its analysis where possible. The loop in the following example is optimized because VAST-C can see that the second dimension indexes of the a references can never be equal (since n is not equal to n+1) and thus there is no recursion (The references to a are to two totally different column vectors - they do not share data).

```

for (i=2; i<= 100; i++ ) (Optimized.)
    a[n][i] = a[n+1][i-1]*b[i] + c[i];

```

---

## Function arguments

In ANSI C it is possible for function arguments to overlap in memory. While this is rarely the case, lacking further information, VAST-C must take the safe road and avoid optimizing loops which involve function arguments appearing on both the left and right sides of an assignment operator.

```

func( pa, pb )
int *pa, *pb;
{
    .
    .
    .
    for ( i = 0; i < 100; i++ )
        *(pa+i) = *(pb+i);
    .
    .
    .
}

```

If, in the above example, pa and pb overlap in memory in a recursive way, then vectorization of the loop would result in incorrect results.

```

int *a;
.
.
func( a, a-1 );

```

As the example above shows, the arguments overlap, so in the function func the loop is recursive. As we said, it is very rare for recurrence to exist due to function arguments. For this reason, the assertion level switch is provided to inform VAST-C that arguments never overlap. If you had specified an assertion level of 1 or higher (-Vassertion\_level1), the loop in func could be optimized.

---

## Pointers and data dependency

Pointers are a powerful language feature and, while they can produce more compact and efficient code, they can also present difficulties for data dependency analysis.

```
for ( i = 0; i < n; i++ )
    *pa++ = *pb++ + *pc++;
```

If there is no other information, VAST-C cannot determine whether overlap exists in such cases as the above since `pa`, `pb`, `pc` may overlap. In such a simple case, VAST-C may generate both vector and scalar versions together with a runtime test to determine whether these pointers overlap. As you might suspect, the testing code can mushroom quickly if the number of pointers involved becomes too large. In fact, it would be possible for the if test to outweigh the benefit of the vector code. A simple and efficient way of handling the problem of potential overlap of pointers is provided via pragmas and optimization levels. The `disjoint` pragma provides the user with a way to inform VAST-C of the characteristics of given pointers.

```
#pragma vd_ disjoint ( pa, pb, pc )
for ( i = 0; i < n; i++ )
    *pa++ = *pb++ + *pc++;          (Vectorized.)
```

Using the `disjoint` pragma here informs VAST-C that `pa`, `pb` and `pc` are disjoint - they do not overlap. Given this information, the loop will be optimized.

A more convenient way for you to help VAST-C with pointers is provided with the assertion level switch. This switch informs VAST-C of the 'style' in which the program has been coded.

Generally, the programmer knows whether or not they have used pointers in an overlapping manner. They may know that only pointers of the same name overlap thereby allowing VAST-C to automatically optimize loops that contain pointers to different names. In the previous example, if you had specified an assertion level of 3 or higher (`-vassertion_level3`), then VAST-C would know that `pa` does not overlap with any pointers that are not of the same name. Therefore, `pa` does not overlap with `pb` or `pc`.

In some instances, a programmer may use multiple references of the same pointer in a loop, yet recurrences do not exist. For these cases assertion level 4 is provided. At this level only explicitly coded feedback hinders optimization.

```
for ( i = 0; i < n; i++ )
{
    x = *pa++;
    *pa = x;
}
```

It may not be obvious, but feedback is explicitly coded in the above example, and the loop would not be optimized at any assertion level. We can see that the contents of the location pointed at by `pa` are being stored into the location of `pa+1`. Perhaps more obvious would be the equivalent:

```

for ( i = 0; i < n; i++ )
    *(pa+i+1) = *(pa+i);

```

In addition to the disjoint pragma and the assertion level switch, the `nosync` pragma may always be used to inform VAST-C that no data dependency exists.

---

## Potential dependency run-time testing

When it is unclear whether a loop is data-dependent or not, because of variables in array subscripts, VAST-C will generate two versions of the loop together with a run-time test. If the loop is not data-dependent, a parallel version of the loop will execute; otherwise the scalar version.

```

dadp02( a, b, ip1, n )
int ip1, n;
float a[], b[];
{
    int i;
    for ( i = 0; i < n; i++ )
        a[ip1+i] = a[i] + b[i]; (Potentially dependent.)
}

```

Translation:

```

if ( abs( ip1 ) >= n || ip1 == 0 )
{
    (parallel version)
}
else
{
    for ( i=0; i <= n - 1; i += 1 )
        a[ip1+i] = a[i] + b[i];
}

```

It is possible for array constants to conflict with vector array references. Because of this, the following cannot be safely optimized ( $j$  may be between 2 and  $n$ ) unless an if statement is placed around the translated code to ensure that this is not the case. VAST-C also handles loops of this kind by generating alternate code.

```

dadp03 ( a, b, n, j )
float *a, *b;
for ( i = 1; i < n; i++ ) (Cannot be optimized unconditionally)
    *(a+i) = *(a+j) - *(b+i);

```

Translation:

```
if ( j < 1 || n - 1 < j )
{
#pragma vdir nodep
    for ( i=0; i < n - 1; i++ )
        a[1+i] = a[j] - b[1+i];
}
else
{
    for ( i=1; i <= n - 1; i += 1 )
        *(a + i) = *(a + j) - *(b + i);
}
```

---

## Ambiguous subscript resolution

When it is not possible with information contained in the loop to determine the relationship between two references to an array, the situation is called ambiguous subscripting or potential dependency. In these situations, VAST-C looks for statements outside of the loop that may provide information which clears up the ambiguity. In the loop below, VAST-C finds the assignment to `n2`, which makes it clear that `n1` is never equal to `n2`, and thus there is no feedback.

```
n2 = n1 + 1;
for ( i = 0; i < n; i++ )           (Optimized.)
    a[n1][i+1] = a[n2][i] * b[i];
```

In the example below, the stores into pointer variables outside the loop are analyzed to determine that there is no dependency.

```
pa = &a;
pb = pa + 100;
for ( i = 0; i < 100; i++ ) (Optimized.)
    *pa++ = *pb++ + x;
```

---

## Data dependency pragmas

There are a number of different pragmas that allow you to help VAST-C optimize more code. These pragma tell the system information about the variables in the loop, so that better data dependency decisions can be made.

### **nosync pragma**

The `nosync` pragma asserts that no overlap exists in the loop, and thus no synchronization is needed. This is the pragma to use for potentially concurrent loops that have hidden relationships between variables. This capability should be used only when you know that no real overlap exists. When it detects

potential feedback, VAST-C issues a message asking you to apply this pragma 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. If you know that the variable `k` has a value greater than 999, then there is no overlap between the references to array `a` in the loop below, and you can use the `nosync` pragma to tell VAST-C that the loop can be optimized.

```
#pragma vd_ nosync
for ( i = 0; i < 1000; i++ )
    a[i+k] = a[i] + b[i] ;
```

## disjoint pragma

You can use the `disjoint` pragma to inform VAST-C that certain pointers are disjoint, that is, the memory ranges they point to do not overlap.

Syntax:

```
#pragma vd_ disjoint [(pa,pb,...,pn)]
```

`disjoint` states that the pointers listed do not overlap with each other or any other variables. If no list of pointers is given, all pointers are assumed to be disjoint.

```
#pragma vd_ disjoint ( pa, pb )
for ( i = 0; i < n; i++ )    (Vectorized.)
    *pa++ = *pb++;
```

Since `pa` and `pb` are pointers and may overlap, it would be unsafe to optimize the above loop without the `disjoint` pragma. The `nosync` pragma may also be used in this case.

## relation - specifying relations between variables

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

```
#pragma vd_ 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 `>`, `<`, `>=`, `<=`, `==`, `!=`, with the normal C meanings.

When VAST-C cannot otherwise figure out 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` pragmas are informative only and do not force any action.

`relation` pragmas 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.

```
#pragma vd_ relation ( j >= n )
...
for ( i = 0; i < n; i++ )    (If j >= n, no overlap.)
    a[i+j] = a[i] + b[i];    (Vectorized.)
```

The `relation` pragma is provided for situations where you are unsure if the `nodepch` pragma (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 an assignment operator 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 (e.g., the example below).

This information can be passed to VAST-C through the `permutation` pragma. `permutation` asserts that the named integer arrays contain no repeated elements (i.e., they serve merely to permute the elements of the arrays they indirectly address).

Syntax:

```
#pragma vd_ permutation ( ia1, ia2, ... , ian )
```

`permutation` declares the integer arrays (`ia1`, etc.) to have no repeated values for the entire routine.

```
#pragma vd_ permutation ( ipnt )    (ipnt has no repeated values.)
...
for ( i =0; i < 100; i++ )
    a[ipnt[i]] = a[ipnt[i]] + b[i];
```

---

## **Data dependency conflict messages**

Data dependency messages are always followed by the name of the variable which is causing the problem and index of the loop causing the problem is given as well.

### *feedback of array elements*

```
for ( i =0; i < 100; i++ )
    a[i+3] = a[i] + b[i];
```

Feedback of results makes the operation recursive and thus unsafe to optimize.

### *feedback of scalar value from one loop pass to another*

```
for ( i=0; i < 100; i++ )
```

```

{
    a[i] = a[i] + s;
    s = a[i]/c[j];
}

```

The variable `s` is used in the first line of the 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. `s` 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 pragma if ok***

```

void test(a,b,c)
float *a,*b,*c;
...
for (i=0; i<n; i++)
    *a++ = *b++ + *c++ ;

```

The use of passed-in pointers makes it unclear whether the ranges of locations taken on by `b` and `c` overlap with that of `a`. If you know that they don't overlap use, for example, the assertion level option on invocation (at least `-Vassertion_level1` in this case) to advise VAST-C.

```

for ( i =0; i < 100; i++ )
    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 you are sure are safe can be optimized by inserting the `nosync` pragma in front of the loop. Here is another case where this message would result:

```

for ( i=0; i < 100; i++ )
    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` pragma should be used.

As a convenience, the `-Vnosync` option switch or `nosync` pragma with routine or global scope may be used instead of loop-by-loop pragmas.

***must synchronize to preserve order of accesses***

```

#pragma vd_ nodepchk
for ( j=0; j<m; j++ )
    for ( i=0; i<n; i++ )
        a[j][i] = a[j+k][i] + b[j][i];

```

In order to be executable in parallel, a loop must not update elements of any array that may be read by other iterations of the loop. (Note that this is more restrictive than recursion in the vectorization sense.) The `nosync` pragma is provided for cases where the overlap is ambiguous in the source code, as in the example above.

### *multiple store conflict*

```
for ( i=1; i <= 100; i++ )
{
    if ( c[i] > 0 ) a[i-1] = c[i];
    a[i] = b[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 pragma if ok*

```
for ( i =0; i < 100; i++ )
{
    a[i+j] = b[i];
    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 pragma should be used to allow the loop to optimize.

### *too many data dependency problems*

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

# 9. Decision processes

This section describes the varieties of conditional statements in loops and how VAST-C deals with them.

---

## Allowable conditional constructs

VAST-C can analyze any combination of conditional assignments, conditional and unconditional forward branching and ifs. Because of compilation-speed restrictions, there is a limit of six simultaneously active conditions. VAST-C vectorizes the following conditional constructs -

### if statements:

Form:

```
if (expression) [{ ... }];
```

Example:

```
for( i=0; i < n; i++ )  
    if ( b[i] < c[i]-x )a[i] = sqrt(d[i]);
```

### Conditional and unconditional forward transfers:

Form:

```
if (expression) goto label;  
and  
goto label;  
(where label is lower down in the same loop)
```

Example:

```
for( i=0; i < n; i++ )  
{
```

```

        if ( a[i] > 0.0 ) goto next;
        b[i] = c[i];
next:    e[i] = 0.0;
        d[i] = b[i] - 6.1;
    }

```

## Conditional operator:

Form:

```
(boolean expression) ? return1 : return2;
```

Example:

```

for( i=0; i < 100; i++ )
    a[i] = b[i] > c[i] ? b[i] : c[i];

```

## if...else statements:

Form:

```

if (expression)
{
    .
    .
    .
}
else if (expression) (optional)
{
    .
    .
}
else (optional)
{
    .
    .
}

```

Example:

```

for ( i=0; i < n; i++ )
{
    if ( a[i] == b[i] )
    {
        a[i] = 2.0*b[i];
        e[i] = e[i]*e[i];
    }
}

```

```

else
{
    a[i] = 0.0;
    e[i] = d[i];
}
}

```

There is no nesting limit to ifs. Forward transfers do not have to be properly nested; VAST-C converts all forward transfers into structured if blocks. Loops containing backward transfers are not parallelized.

---

## Two types of conditions

There are two basic types of conditions, as illustrated in the loops below. The first loop shows a loop independent conditional assignment:

```

for (i=0; i < 100; i++ )
{
    b[i] += 1.0;
    if ( n ) a[i] = b[i] + 2.0;
}

```

The if clause ( n ) does not depend on the loop index at all; it remains the same for all iterations of the loop. Thus, we know whether or not n is equal to 0 before the loop is executed.

The other kind of condition is shown in this loop:

```

for ( i=0; i < 100; i++ )          (Vectorized)
    if ( c[i] < 0 ) d[i] = e[i] + f[i];

```

It is called loop dependent because the if clause ( c[i] < 0 ) might change with each loop pass. Sometimes we want to do the calculation (add e[i] to f[i]) and sometimes we do not.

The %CD field in the loop summary of the VAST-C listing summarizes what percentage of the operations in the translated loop are loop-dependent conditional. A value close to 100 in this field means that the loop is almost completely conditional, and may run slower than the scalar original if the condition is sparse.

For vector loops, you should examine loops with large values in the %CD field to make sure that the calculation in those loops is usually executed. If the calculation is skipped over by conditional statements most of the time, the loop will probably run slower when vectorized, and a novector pragma should be used to turn off vectorization. The %CD field is not important for parallel loops.

---

## Removal of invariant IFs

Invariant IFs are removed from loop nests completely. Code is replicated so that the test is made once outside of the loop and then alternate versions of the loop are executed based on the result of the test.

In removing invariant IFs, VAST-C limits the amount of additional code generated: no more than three additional versions and/or 100 additional lines of code are added.

```
for (i=0; i < 100; i++ )
{
    b[i] = 1.0;
    if ( n ) a[i] = b[i] + 2.0;
    c[i]= a[i] * .5;
}
```

Translation:

```
if ( n )
{
    for (i=0; i < 100; i++ )
    {
        b[i] = 1.0;
        a[i] = b[i] + 2.0;
        c[i]= a[i] * .5;
    }
}
else
{
    for (i=0; i < 100; i++ )
    {
        b[i] = 1.0;
        c[i]= a[i] * .5;
    }
}
```

---

## Loop-index-dependent conditions

The loop below shows a loop-index-dependent conditional assignment. VAST-C eliminates such conditions by adjusting the limits of the vector operation.

```
for ( i=0; i < 100; i++ )
    if ( i != j ) a[i] = b[i] + c[i];
```

Translation:

```

if ( j >= 0 && j <= 99 )
{
    for ( i=0; i < j; i++ )
        a[i] = b[i] + c[i];
    for ( i=0; i < 99-j; i++ )
        a[1+j+i] = b[1+j+i] + c[1+j+i];
}
else
{
    for ( i=0; i < 100; i++ )
        a[i] = b[i] + c[i];
}

```

The if (j >= 0...) test determines whether j is in the range of the loop index. If so, the operation is performed up to the j-1 element and then from the j+1 element to n. If j is not in the range of the loop index, the operation is performed on all elements. A loop-index-dependent condition is of the general form:

```
if ( index_expression rel invariant_expression )
```

where:

rel represents a comparison. The possible values are ==, !=, >, >=, <, or <=.

invariant\_expression represents a mathematical expression, constant, or variable.

index\_expression is as defined in previous section.

---

## Messages about conditionals

The messages below relate to handling of conditional operations.

*backward transfers are not optimizable*

```

j = 1;
for ( i=0; i < 100; i++ )
{
top:
    a[j] = b[j] + d[i];
    j++;
    if ( b[j] > limit ) goto top;
    c[i] = a[j];
}

```

The branch to label top is backward, not forward, and prevents optimization. This message appears only if VAST-C was not able to convert the backward branch into a for loop.

*branches out of the loop prevent optimization*

```
for ( i=0; i < 100; i++ )
{
    a[i] = b[i] + c[i];
    if ( a[i] < 0 ) break;
    b[i] = 6.0;
}
```

*branches are too complex to optimize*

Because of limits on time and memory, conditional constructs with more than six simultaneously active conditions (i.e. ifs, cases, etc.) are not analyzed.

# 10. Idiom Recognition

---

## Nested loop idiom recognition

VAST-C recognizes certain frequently occurring higher-level constructs as complete units and replaces them with calls to optimized operations in the system math library. This optimization can be suppressed via the `-d j` switch.

### Matrix multiply

VAST recognizes matrix multiplies in a very general way. They can be:

- Inside outer loops
- With or without initialization
- Independent of loop order
- Independent of subscript order (transposed matrices)
- With or without temporary scalar
- Have non-unit strides
- Scaled by complicated expressions
- Surrounded by other computations in the loop

Example:

```
float a[100][100], b[100][100], c[100][100];
int i,j,k;
for ( i=0; i<100; i++ )
    for ( j=0; j<100; j++ )
        for ( k=0; k<100; k++ )
            a[i][j] += b[i][k] * c[k][j];
```

Translation:

```
void SGEMM();
SGEMM(100, 100, 100, 1., b, 100, 1, c,
      100, 1, 1., a, 100, 1);
```

A more complicated example using the same declarations:

```
for ( i=0; i<50; i++ )
    for ( j=0; j<100; j=j+5 )
        for ( k=0; k<100; k++ )
            a[2*i][j] += b[2*i][k]
                * (3./2.) * c[k][j+1];
```

Translation:

```
void SGEMM();
SGEMM(50, 20, 100, (float )(3. / 2.),
      b, 200, 1, c[0][1], 100, 5, 1., a,
      200, 5);
```

## Vector-matrix multiply

A call to `sgemv` (single) or `dgemv` (double) is generated when a vector-matrix multiply is encountered.

```
float a[100], b[100][100], c[100][100];
int i,j,k;
for ( i=0; i<100; i++ )
    for ( k=0; k<100; k++ )
        a[i] = a[i] + b[i][k] * c[k][n];
```

Translation:

```
void SGEMV();
.
.
.
SGEMV(100, 100, 1., b, 100, 1,
      c[0][n], 100, 1., a, 1);
```

## Rank one update

A call to `sger` (single) or to `dger` (double) is generated when a rank one update is encountered.

```
float a[45], b[38], c[35][68];
int i,k;
```

```
float s;  
for ( i=0; i<25; i++ )  
    for ( k=0; k<66; k++ )  
        c[i][k] = c[i][k] +s*a[i]*b[k];
```

Translation:

```
void SGER();  
SGER(25, 66, s, a, 1, b, 1, c, 68, 1);
```

# 11. Functions

This section discusses the handling of loops containing function references. Unless otherwise stated, function references inhibit optimization.

---

## Standard C library functions

References to standard library functions do not inhibit optimization.

```
for( i=0; i < 100; i++ ) (Optimizable)
    a[i] = sin(sqrt(b[i]+c[i]));
```

For example, the following are optimizable functions ( if the user includes the proper library ): abs, sqrt, exp, log, log10, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, and tanh.

---

## Split-out function references

When it is known that a user function has no recursive side effects, the user may direct VAST-C to split out function references from otherwise optimizable loops with the split pragma. The function will be called in its own loop.

```
#pragma vd_ split    ( Says that splitting is okay )
for ( i=0; i < 100; i++ )
{
    *x = a[i] + b[i];
    fnsbal ( c[i], x ); ( Will be split from the loop )
    d[i] = e[i] + *x;
}
```

---

## Inline expansion

Rather than splitting a loop (see above), a better tactic may be to direct VAST-C to pull the external function inline. See the section of this document on Inline Expansion for a full description.

---

## Message about external references

References to non-library functions in a loop prevent optimization, unless they are inlined or split. If you reference such a function in a loop, you will get the following translation diagnostic:

*user function reference prevents optimization*

```
for ( i=0; i < 100; i++ )
    a[i] = myfunc(b[i]);
```

# 12. Inline expansion

This section describes the inline function expansion feature of VAST-C.

---

## Introduction

Programs can often receive a performance benefit from the expansion of the bodies of certain functions into the loops which 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 repeatedly; these functions are good candidates for inline expansion. Here is a small example of inline expansion:

Original:

```
{
    ...
    for ( i=0; i < n; i++ )
        a[i] = calc( a[i], x+b[i], 2.0 );
    ...
}
float calc( a,b,c )
float a, b, c;
{
    float t;
    t = a + sqrt( b*b + c*c );
    if ( t < 0 ) t = -( b + c );
    return t;
}
```

Expanding function calc inline:

```
for ( i=0; i < n; i++ )
{
    float calc1x,t,blx;
    blx = x + b[i];
    t = a[i] + sqrt( blx*blx + 2.0*2.0 );
    if ( t < 0 ) t = -( blx + 2.0 );
    calc1x = t;
    a[i] = calc1x;
}
```

Inline expansion reduces function calling overhead. It also allows scalar optimizations such as invariant code relocation and common subexpression analysis to extend across the body of the function as well as the body of the calling loop.

---

## Inline expansion pragmas

### **autoexpand pragma**

The autoexpand pragma is used to invoke automatic routine expansion.

Format:

```
#pragma vd_ autoexpand
```

The f, r, or l scopes can be used. noautoexpand cancels the action. The autoexpand pragma with global scope is equivalent to the -vinline switch. If you want several levels of expansion from the bottom of the calling tree, use the -vautoinline\_nest option to indicate how many levels of nested inlining you request. -vinline requests only function at the lowest level be inlined, and is thus similar to -vautoinline\_nest1.

If you want to exempt some functions from autoexpansion, you can use the -vnoinline invocation parameter. The functions listed will not be expanded.

### **Explicit mode**

In explicit mode, routines listed on a pragma are expanded without regard for the automatic mode criteria. If no list is given on the pragma, it directs expansion of all function references in the immediately following statement. This reference need not be inside of a loop.

### **expand pragma**

The expand pragma is supplied for explicit routine expansion.

Format:

```
#pragma vd_ expand [(list)]
```

Where list is a list of functions to expand in this routine. If (list) is not supplied, expand the next statement.

Scope is ignored on the expand pragma.

Example of pragma use:

```
#pragma vd_ expand (calc)
...
for ( i=0; i < n; i++ )
    a[i] = calc( a[i], b[i]+1, n );
...
```

The `-vinline list` switch on the invocation line performs the same function as this pragma, but applies to the entire file. The list is the same as above, but must not be null.

## **nexpand pragma**

Expand the indicated function and all functions it calls. The `-Y` invocation parameter allows you to request nested expansion from the command line.

## **search pragma**

You can tell VAST-C where to look for the routines to expand with the search pragma.

Format:

```
#pragma vd_ search (filenames)
```

Where filenames is a list of files (separated by commas) in which to search for the routines to be expanded.

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

---

## **Where to get the code**

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

### **Same file**

Referenced functions can be searched for in the same file as the calling function ("stacked input"). This necessitates an initial pass by VAST-C through the entire input file (and files included therein) to build a directory of the program units in the input file. The switch `-e 8` enables this initial pass, which by default is not done.

## Explicitly named file

You can supply (via the search pragma) the name of a file in which to search for a particular called routine.

## Implicitly named file

In Unix-based systems, C programs are frequently stored such that each routine of the program resides in a separate file with a canonical name (e.g. the name of the function followed by .c). This is the default search method.

---

# Possible problems

## Separate compilation

C 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 completely "fool proof" inline expansion. 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 function B which references function C. Let's say function C is expanded into function B. Later, we decide to change the calculation in function 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 functions must be recompiled when a change is made. For this reason, VAST-C 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 code mushrooming; 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 bombs in an expanded section of code, the error is reported in a different routine than the one it originally appeared in.

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

## Compilation rate

Any scheme to analyze more than one program unit at a time will lead to significantly slower compilation rates, and inline expansion is no exception. It

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

---

## 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 function to be expanded can not be located.
- Syntax errors are found in the expansion function.

### Inhibitors specific to automatic expansion mode

In automatic mode, all references to functions that meet the following criteria are expanded.

- The function to be expanded has less than 50 lines. This threshold is controllable using the `-Mnnn` switch where `nnn` is the number of lines.
- The routine to be expanded does not call any other external functions that will not also be expanded.
- There are no inhibitors to the expansion.

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

Note that although called automatic, this mode still requires you to insert a pragma (`autoexpand`) or use a switch (`-vinline`) when invoking VAST-C. An informational message is issued for each expansion action. This is to remind you that functions that have been expanded into need to be recompiled each time the expanded function is changed.

---

## Inline expansion user messages

All inline expansion messages appear as warnings. VAST-C does not expand any function that has caused the generation of one of the following messages, except for the function expanded message.

### Inline expansion summary

VAST-C notifies you of any functions that have been expanded and also informs you as to why a particular function was not expanded. A summary of the routines and all the locations a function was or was not expanded is displayed.

## User messages

### *expansion function not found*

The expansion function can not be located, as specified by the search pragma or by the default location.

### *function not found in input file*

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

### *expansion function is too big for automatic expansion*

The routine has more than the maximum number of lines (default is 50); increase maximum line default or use explicit expansion mode to expand.

### *external functions found while expanding*

The expansion routine references another function beyond the limit for nested expansion; use explicit expansion mode to expand, or increase the -J parameter.

### *syntax error encountered in expansion routine*

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

### *exceeded maximum number of expanded functions*

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

### *function expanded*

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

# 13. Further Information

---

## Crescent Bay Software and VAST

Crescent Bay Software exclusively produces compiler tools and compiler technology for high-performance optimization, with a particular emphasis on all forms of parallelism.

We began work (as the Compiler Tools group of Pacific-Sierra Research) 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. VAST technology is used in a variety of ways in different systems, from an optimizing pass of the compiler to a stand-alone translator.

In addition to **VAST-C/Parallel**, the VAST system includes:

**VAST-F/Parallel**, automatic parallelization for the Fortran language.

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

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

**VAST-F/toOpenMP**, Fortran to OpenMP translator. Somewhat like VAST-F/Parallel, except the resulting translated source, containing inserted OpenMP directives is intended to be retained permanently.

---

## Questions or Comments

If you have any questions or comments about VAST-C/Parallel, please do not hesitate to contact us. (See preface for contact information.)

# Index

## **%**

%CD field, 49

## **A**

ambiguous subscripting, 42

assertion level, 40

assertion levels, 6

assignment operators, 20

associative transformations, 11

autoexpand directive, 59

automatic distribution of loop iterations, 29

## **B**

branches out of the loop, 52

## **C**

CNCALL, 11

cncall directive, 35

collapse loops, 27

common expressions, 26

concur directive, 29

concurrency, 29

conditional statements, 47

critical region, 35

## **D**

data dependencies, 35, 38  
Data Dependency Conflict, 14  
data dependency directives, 42  
data dependency messages, 44  
diagnostic messages, 13  
disjoint directive, 40, 43  
driver, 4  
dynamic scheduling, 34

## **E**

event summary, 14  
expand directive, 59  
expansion of functions, 58

## **F**

feedback of array elements, 44  
feedback of scalar value, 44  
files, 2  
for loops, 17  
function arguments, 39  
function inlining, 58  
function references, 35  
function with no side effects, 56

## **I**

idiom recognition, 53  
INCLUDE files, 15  
index, 17  
indirect addressing, 44  
inline expansion, 58  
inlining switch, -Vinline, 7  
INNER directive, 35  
inner loops, 35  
Internal Error, 14  
invariant conditionals, 50  
iteration count, 19

## **L**

library functions, 56  
listing, 13  
listing control switches, 14  
listing, wide format, 15  
loop collapse, 27  
loop fusion, 22  
loop independent conditional, 49  
loop re-rolling, 23  
loop summary, 14

## **M**

matrix multiply, 53  
messages about conditionals, 51  
messages, dependency, 44  
multiple store conflict, 46

## **N**

nested expansion, 60  
noargumentoverlap switch, -Vnoargumentoverlap, 7  
NOASSOC, 11  
noassoc switch, -Vnoassoc, 8  
NOCONCUR, 11  
noconcur directive, 30  
nopointeroverlap switch, -Vnopointeroverlap, 8  
NOSYNC, 12  
nosync switch, -Vnosync, 8  
Note Message, 14

## **O**

optimizations, 1  
outer loop unrolling, 28

## **P**

parallel cases, 36  
parallel regions, 30, 36  
parallelization, 29  
pcc driver, 4

PERMUTATION, 12  
permutation directive, 44  
pointers, 16, 28, 43  
pointers, and data dependence, 40  
potential dependency, 42  
potential feedback, 45  
private array, 31  
private variables, 31

## **R**

recurrences, 38  
reduction operations, 35  
RELATION, 12  
relation directive, 43  
run-time test, 41

## **S**

scalar, 17  
scalar division, 26  
search directive, 60  
shared variables, 31  
SKIP, 11  
skip directive, 30  
skip switch, -Vskip, 8  
split directive, 56  
statements, 18  
static scheduling, 34  
structures, 20  
summation, 35  
synchronize to preserve order, 45  
Syntax Error, 14

## **T**

threshold directive, 30  
THRESHOLD directive, 33  
threshold test, 32  
Translation Diagnostic, 13

translation diagnostics, 21

## **U**

unrolling, 23

unrolling, outer loops, 28

user function reference, 57

## **V**

vector, 17

vector-matrix multiply, 54

## **W**

Warning Message, 14